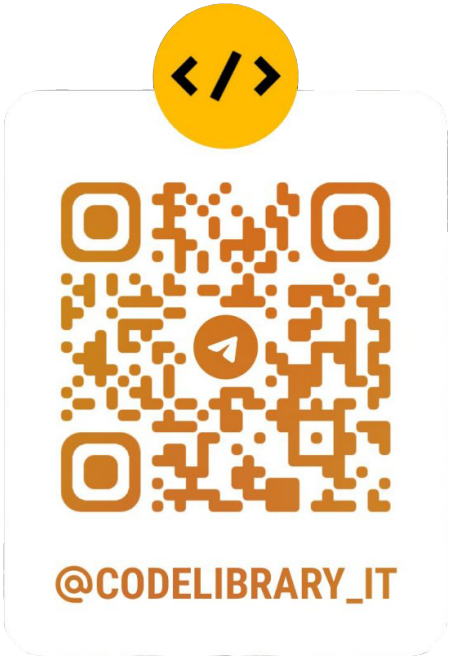
Ти Джей Краудер

Новые возможности JavaScript®

Как написать чистый код
по всем правилам современного языка

БОМБОРА
ИЗДАТЕЛЬСТВО

Мировой
компьютерный
бестселлер


@CODELIBRARY_IT

T. J. Crowder

The New Toys JavaScript®

Ти Джей Краудер

Новые возможности JavaScript®

Как написать чистый код по всем правилам современного языка

БОМБОРА
ИЗДАТЕЛЬСТВО
Москва 2023

УДК 004.43
ББК 32.973.26-018.1
К78

T.J. Crowder

JAVASCRIPT: THE NEW TOYS

Copyright © 2020 by Thomas Scott «T.J.» Crowder
All Rights Reserved. This translation published under license
with the original publisher John Wiley & Sons, Inc.

Краудер, Ти Джей.

К78 Новые возможности JavaScript : как написать чистый код по всем правилам современного языка / Ти Джей Краудер : [перевод с английского М. А. Райтман]. — Москва : Эксмо, 2023. — 640 с. — (Мировой компьютерный бестселлер).

ISBN 978-5-04-159515-9

Перед вами сборник правил написания кода на современном языке JavaScript. На наглядных примерах автор объясняет, как работают последние версии JS, какие приемы в нем можно использовать, чтобы сделать код коротким и чистым, а каких ошибок лучше избегать, чтобы не было багов.

УДК 004.43
ББК 32.973.26-018.1

ISBN 978-5-04-159515-9

© Райтман М. А., перевод на русский язык, 2023
© Оформление. ООО «Издательство «Эксмо», 2023

Венди и Джеймсу, которые встречались
допоздна и работали по выходным
с безграничной поддержкой и любовью.

ОГЛАВЛЕНИЕ

ОБ АВТОРЕ 20
О ТЕХНИЧЕСКОМ РЕДАКТОРЕ 21
О ТЕХНИЧЕСКОМ КОРРЕКТОРЕ 21
БЛАГОДАРНОСТИ 22
ВВЕДЕНИЕ 23
О чем эта книга? 23
Кому стоит читать эту книгу 25
Как пользоваться этой книгой 25

ГЛАВА 1. НОВЫЕ ВОЗМОЖНОСТИ В ES2015–ES2020 И ДАЛЕЕ 27

Определения, что есть что и терминология 28
Что такое Ecma? ECMAScript? TC39? 28
Что такое ES6? ES7? ES2015? ES2020? 28
«Движки» JavaScript, браузеры и др. 29
Что за «новые возможности»? 30
Как создаются новые возможности? 32
Кто здесь главный 32
Процесс 33
Вовлечение в процесс 35
Следите за новыми возможностями 36
Использование текущих функций во вчерашних условиях и разрабатываемых возможностей сегодня 37
Транспилирование примера с помощью Babel 38
Обзор главы 42

ГЛАВА 2. ОБЪЯВЛЕНИЯ БЛОЧНОЙ ОБЛАСТИ ВИДИМОСТИ ДЛЯ LET И CONST 45

Введение в `let` и `const` 46
Истинная блочная область видимости 46
Повторные объявления — это ошибка 47
Поднятие и временная мертвая зона 48
Новый вид глобальных переменных 51
Const: Константы в JavaScript 53
Основы `const` 53
Объекты, на которые ссылается const, по-прежнему изменяемы 54

Блочная область видимости в циклах 55
Проблема «замыканий в циклах» 56
Привязки: Как работают переменные, константы и другие идентификаторы 57
Циклы `while` и `do-while` 62
Последствия для производительности 63
Константа в блоках цикла 64
Константа в циклах `for-in` 65
От старых привычек к новым 66
Используйте `const` или `let` вместо `var` 66
Сохраняйте узкую область видимости переменных 66
Используйте блочную область видимости вместо встроенных анонимных функций 66

ГЛАВА 3. ФУНКЦИИ 69

Стрелочные функции и лексические `this`, `super` и т. д. 70
Синтаксис стрелочной функции 70
Стрелочные функции и лексические this 75
Стрелочные функции не могут быть конструкторами 75
Значения параметров по умолчанию 76
Значения по умолчанию — это выражения 78
Значения по умолчанию вычисляются в их собственной области видимости 79
Значения по умолчанию не увеличивают арность функции 81
Остаточные параметры 81
Висящие запятые в списках параметров и вызовах функций 83
Свойство имени функции 85
Объявления функций внутри блоков 86
Объявления функций внутри блоков: Стандартная семантика 88
Объявления функций внутри блоков: Устаревшая веб-семантика 90
От старых привычек к новым 92
Используйте стрелочные функции вместо различных обходных путей для этого значения 92
Используйте стрелочные функции для обратных вызовов, если не используете аргументы или this 93
Рассмотрите применение стрелочных функций и в других местах 93
Не используйте стрелочные функции, когда вызывающей стороне необходимо контролировать значение this 94
Используйте значения параметров по умолчанию, а не код, предоставляющий значения по умолчанию 95
Используйте остаточный параметр вместо ключевого слова arguments 95
Рассмотрите возможность использования висящих запятых, если это оправдано 95

ГЛАВА 4. КЛАССЫ 97

Что такое класс? 98
Представление нового синтаксиса класса 98
Добавление конструктора 100
Добавление свойств экземпляра 102
Добавление метода прототипа 102
Добавление статического метода 104
Добавление свойства-акцессора 105
Вычисляемые имена методов 107
Сравнение с устаревшим синтаксисом 107
Создание подклассов 110
Ключевое слово `super` 113
Написание конструкторов подклассов 114
Наследование свойств и методов прототипа суперкласса и доступ к ним 115
Наследование статических методов 119
Ключевое слово `super` в статических методах 120
Методы, возвращающие новые экземпляры 121
Создание подклассов для встроенных компонентов 126
Где доступен `super` 127
Отказ от `Object.prototype` 131
Синтаксис `new.target` 131
Объявления классов в сравнении с выражениями классов 135
Объявления классов 135
Выражения классов 136
Еще не все 137
От старых привычек к новым 137
Использование класса при создании функций конструктора 137

ГЛАВА 5. ОБЪЕКТЫ 139

Вычисляемые имена свойств 139
Стенография свойств 140
Получение и настройка прототипа объекта 141
Метод `Object.setPrototypeOf` 141
Свойство `__proto__` в браузерах 142
Буквальное указание имени свойства `__proto__` в браузерах 143
Синтаксис метода и применение `super` вне классов 143
Тип данных `Symbol` 146
Почему же символы? 147
Создание и использование символов 148
Символы не для конфиденциальности 149
Глобальные символы 150

Хорошо известные символы 154
Новые функции объектов 155
Метод Object.assign 155
Метод `Object.is` 156
Метод `Object.values` 157
Метод `Object.entries` 157
Метод `Object.fromEntries` 158
Функция `Object.getOwnPropertySymbols` 158
Метод `Object.getOwnPropertyDescriptors` 158
Метод `Symbol.toPrimitive` 159
Порядок свойств 161
Синтаксис расширения свойств 163
От старых привычек к новым 164
Использовать вычисляемый синтаксис при создании свойств с динамическими именами 164
Используйте сокращенный синтаксис при инициализации свойства из переменной с тем же именем 165
Используйте метод `Object.assign` вместо пользовательских функций «extend» или явного копирования всех свойств 165
Используйте синтаксис расширения при создании нового объекта на основе свойств существующего объекта 165
Используйте символ, чтобы избежать коллизии имен 165
Используйте методы `Object.getPrototypeOf/setPrototypeOf` вместо свойства `__proto__` 166
Используйте синтаксис метода для методов 166

ГЛАВА 6. ВОЗМОЖНОСТИ ИТЕРАЦИИ: ИТЕРИРУЕМЫЕ ОБЪЕКТЫ, ИТЕРАТОРЫ, ЦИКЛЫ FOR-OF, ИТЕРАТИВНЫЕ РАСШИРЕНИЯ, ГЕНЕРАТОРЫ 167

Итерируемые объекты, итераторы, циклы for-of, итерируемое расширение 167
Итераторы и итерируемые объекты 168
Цикл `for-of`: Неявное использование итератора 168
Явное использование итератора 170
Остановка итерации на ранней стадии 171
Прототип итератора объекта 172
Сделать что-либо итерируемым объектом 175
Итерируемые итераторы 179
Синтаксис итеративного расширения 181
Итераторы, цикл `for-of` и DOM 182
Функции-генераторы 184
Базовая функция-генератор, просто производящая значения 185
Использование функций-генераторов для создания итераторов 186
Функции-генераторы в качестве методов 188

Использование генератора напрямую 189
Потребление значений генераторами 189
Использование оператора return в функции-генераторе 193
Приоритет оператора `yield` 194
Методы `return` и `throw`: Завершение работы генератора 195
Остановка генератора или итеративного объекта: yield* 197
От старых привычек к новым 202
Используйте конструкции с итеративными элементами 202
Используйте возможности итеративных коллекций DOM 203
Используйте интерфейсы итераторов и итеративных объектов 203
Используйте синтаксис итеративного расширения в большинстве мест, где вы применяли Function.prototype.apply 203
Используйте генераторы 204

ГЛАВА 7. ДЕСТРУКТУРИЗАЦИЯ 205

Краткий обзор 205
Базовая деструктуризация объекта 206
Базовая (и итеративная) деструктуризация массива 209
Значения по умолчанию 211
Синтаксис Rest в шаблонах деструктуризации 213
Использование отличающихся имен 214
Вычисляемые имена свойств 216
Вложенная деструктуризация 216
Деструктуризация параметров 217
Деструктуризация в циклах 220
От старых привычек к новым 221
Используйте деструктуризацию при получении только некоторых свойств от объекта 221
Используйте деструктуризацию для объектов options 222

ГЛАВА 8. ОБЪЕКТЫ PROMISE 223

Почему же промисы? 223
Основы промисов 224
Краткий обзор 224
Пример 226
Промисы и элементы thenable 228
Использование существующего промиса 229
Метод `then` 229
Связывание промисов в цепочки 230
Сравнение с обратными вызовами 234
Метод `catch` 235

Метод `finally` 237
Метод `throw` в обработчиках `then`, `catch` и `finally` 241
Метод `then` с двумя аргументами 243
Добавление обработчиков к уже выполненным промисам 245
Создание промисов 246
Конструктор Promise 247
Метод `Promise.resolve` 250
Метод `Promise.reject` 251
Другие служебные методы промисов 252
Метод `Promise.all` 252
Метод `Promise.race` 254
Метод `Promise.allSettled` 254
Метод `Promise.any` 255
Шаблоны промисов 255
Обрабатывать ошибки или возвращать промис 255
Серии промисов 256
Параллельные промисы 258
Антишаблоны промисов 259
Излишнее выражение new Promise(/*…*/) 259
Отсутствие обработки ошибок (или неправильная обработка) 259
Оставление ошибок незамеченными при преобразовании API обратного вызова 260
Неявное преобразование отклонения в успешное выполнение 261
Попытка использовать результаты вне цепочки 262
Использование обработчиков бездействия 262
Неправильное разветвление цепочки 263
Подклассы промисов 264
От старых привычек к новым 265
Используйте промисы вместо успешных/неудачных обратных вызовов 265

ГЛАВА 9. АСИНХРОННЫЕ ФУНКЦИИ, ИТЕРАТОРЫ И ГЕНЕРАТОРЫ 267

Асинхронные функции 267
Создание промисов асинхронными функциями 270
Оператор await использует промисы 271
Стандартная логика становится асинхронной при использовании await 272
Отклонения — это исключения, исключения — это отклонения; выполнение — это результаты, возвращаемые значения — это разрешения 273
Параллельные операции в асинхронных функциях 276
Нет необходимости возвращать await 277
Ловушка Pitfall: Использование асинхронной функции в неожиданном месте 278
Асинхронные итераторы, итерируемые и генераторы 279
Асинхронные итераторы 279

Асинхронные генераторы 283
Выражение `for-await-of` 285
От старых привычек к новым 286
Используйте асинхронные функции и await вместо явных промисов и then/catch 286

ГЛАВА 10. ШАБЛОНЫ, ПОМЕЧЕННЫЕ ФУНКЦИИ И НОВЫЕ ВОЗМОЖНОСТИ СТРОК 287

Шаблонные литералы 287
Базовая функциональность (Непомеченные шаблонные литералы) 288
Помеченные шаблонные функции (Помеченные шаблонные литералы) 290
Метод `String.raw` 295
Повторное использование шаблонных литералов 297
Шаблонные литералы и автоматическая вставка точки с запятой 297
Улучшенная поддержка Юникода 297
Юникод, а что такое строка JavaScript? 298
Экранирующая последовательность кодовой точки 300
Метод `String.fromCodePoint` 300
Метод `String.prototype.codePointAt` 300
Метод `String.prototype.normalize` 301
Итерация 303
Новые строковые методы 304
Метод `String.prototype.repeat` 304
Методы `String.prototype.startsWith` и `String.prototype.endsWith` 305
Метод `String.prototype.includes` 306
Методы `String.prototype.padStart` и `String.prototype.padEnd` 306
Методы `String.prototype.trimStart` и `String.prototype.trimEnd` 307
Обновления методов `match`, `split`, `search` и `replace` 307
От старых привычек к новым 309
Используйте шаблонные литералы вместо конкатенации строк (где это уместно) 309
Используйте помеченные функции и шаблонные литералы для DSL вместо пользовательских механизмов заполнения 310
Используйте строковые итераторы 310

ГЛАВА 11. МАССИВЫ 311

Новые методы массивов 311
Метод `Array.of` 311
Метод `Array.from` 312
Метод `Array.prototype.keys` 315
Метод `Array.prototype.values` 316
Метод `Array.prototype.entries` 317

Метод `Array.prototype.copyWithin` 318
Метод `Array.prototype.find` 321
Метод `Array.prototype.findIndex` 322
Метод `Array.prototype.fill` 322
Общая ловушка Pitfall: Использование объекта в качестве значения заполнения 323
Метод `Array.prototype.includes` 324
Метод `Array.prototype.flat` 324
Метод `Array.prototype.flatMap` 326
Итерация, расширение, деструктуризация 326
Стабильная сортировка массива 326
Типизированные массивы 327
Краткий обзор 327
Основное использование 330
Подробнее о преобразовании значений 331
Объект `ArrayBuffer`: Хранилище для типизированных массивов 333
Порядковый номер (Порядок байтов) 336
Вид `DataView`: Необработанный доступ к буферу 337
Совместное использование ArrayBuffer массивами 339
Совместное использование без перекрытия 339
Совместное использование с перекрытием 340
Подклассы типизированных массивов 341
Методы типизированного массива 341
Стандартные методы массива 341
Метод `%TypedArray%.prototype.set` 343
Метод `%TypedArray%.prototype.subarray` 343
От старых привычек к новым 344
Используйте `find` и `findIndex` для поиска в массивах вместо циклов (где это уместно) 344
Используйте для заполнения массивов `Array.fill`, а не циклы 344
Используйте `readAsArrayBuffer` вместо `readAsBinaryString` 345

ГЛАВА 12. КАРТЫ И МНОЖЕСТВА 347

Коллекции Map или карты 347
Основные операции с картой 348
Равенство ключей 350
Создание карт из итерируемых 351
Итерация содержимого карты 352
Создание подклассов для карты 354
Производительность 355
Множества 355
Основные операции с множеством 356
Создание множеств из итерируемых 357

Итерация содержимого множества 357
Создание подклассов для множества 359
Производительность 359
Слабые карты (WeakMap) 360
Слабые карты не итерируемые 360
Варианты использования и примеры 360
Вариант использования: Закрытая информация 361
Вариант использования: Хранение информации для объектов, находящихся вне вашего контроля 362
Значения, ссылающиеся на ключ 364
Слабые множества (WeakSet) 369
Вариант использования: Отслеживание 370
Вариант использования: Маркировка 371
От старых привычек к новым 372
Используйте карты вместо объектов для карт общего назначения 372
Используйте множества вместо объектов для множеств 372
Используйте слабые карты для хранения личных данных вместо публичных свойств 373

ГЛАВА 13. МОДУЛИ 375

Введение в модули 375
Основы модулей 376
Спецификатор модуля 378
Базовый именованный экспорт 379
Экспорт по умолчанию 381
Использование модулей в браузерах 383
Скрипты модуля не задерживают синтаксический анализ 384
Атрибут `nomodule` 384
Спецификаторы модулей в Интернете 385
Использование модулей в Node.js 386
Спецификаторы модулей в Node.js 388
Node.js добавляет дополнительные возможности модулей 389
Переименование экспорта 389
Повторный экспорт экспорта из другого модуля 390
Переименование импорта 391
Импорт объекта пространства имен модуля 392
Экспорт объекта пространства имен другого модуля 393
Импорт модуля только из-за побочных эффектов 394
Импорт и экспорт записей 394
Импорт записей 394
Экспорт записей 395
Импорт в режиме реального времени и доступности только для чтения 397
Экземпляры модуля зависят от базы realm 400

Как загружаются модули 400
Получение и синтаксический анализ 402
Создание экземпляра 405
Выполнение 406
Обзор временной мертвой зоны (TDZ) 406
Циклические зависимости и TDZ 407
Обзор синтаксиса импорта/экспорта 408
Разновидности экспорта 408
Разновидности импорта 410
Динамический импорт 411
Динамический импорт модуля 411
Пример динамического модуля 413
Динамический импорт в немодульных скриптах 416
Встряхивание дерева 418
Бандлинг (Объединение) 420
Метаданные импорта 420
Модули воркеров 421
Загрузка веб-воркера в качестве модуля 421
Загрузка воркера Node.js в качестве модуля 422
Воркер находится в собственной базе realm 422
От старых привычек к новым 423
Используйте модули вместо псевдопространств имен 423
Используйте модули вместо обертывания кода в функции области видимости 424
Используйте модули, чтобы избежать создания мегалитических файлов кода 424
Конвертируйте CJS, AMD и другие модули в ESM 424
Не изобретайте велосипед, используйте хорошо обслуживаемый бандлер 424

ГЛАВА 14. РЕФЛЕКСИЯ — ОБЪЕКТЫ REFLECT И PROXY 425

Объект Reflect 425
Метод `Reflect.apply` 427
Метод `Reflect.construct` 427
Метод `Reflect.ownKeys` 429
Методы `Reflect.get` и `Reflect.set` 429
Другие функции `Reflect` 431
Объект `Proxy` 431
Пример: Регистрирующий прокси 435
Ловушки прокси 442
Общие возможности 442
Ловушка `apply` 443
Ловушка `construct` 443
Ловушка `defineProperty` 443
Ловушка deleteProperty 445
Ловушка `get` 446

Ловушка `getOwnPropertyDescriptor` 447
Ловушка `getPrototypeOf` 448
Ловушка `has` 449
Ловушка `isExtensible` 449
Ловушка `ownKeys` 449
Ловушка `preventExtensions` 450
Ловушка `set` 451
Ловушка `setPrototypeOf` 451
Пример: Скрытие свойств 452
Отключаемые прокси 456
От старых привычек к новым 456
Используйте прокси, а не полагайтесь на потребляющий код, чтобы не изменять объекты API 457
Используйте прокси для отделения кода реализации от инструментального кода 457

ГЛАВА 15. ОБНОВЛЕНИЯ РЕГУЛЯРНЫХ ВЫРАЖЕНИЙ 459

Свойство `flags` 459
Новые флаги 460
Липкий флаг (`y`) 460
Флаг Юникода (`u`) 461
Флаг «все точки» (`s`) 461
Именованные группы захвата 462
Основная функциональность 462
Обратные ссылки 466
Заменяющие токены 467
Утверждения ретроспективной проверки 467
Позитивная ретроспективная проверка 468
Негативная ретроспективная проверка 469
Жадность проявляется в принципе справа налево в ретроспективных проверках 469
Ссылки и нумерация групп захвата 470
Функциональные возможности Юникода 471
Экранирование кодовой точки 471
Экранирование свойства Юникода 472
От старых привычек к новым 476
Используйте липкий флаг (y) вместо создания подстрок и использования «^» при синтаксическом анализе 476
Используйте флаг «все точки» (s) вместо использования обходных путей для сопоставления всех символов (включая разрывы строк). 477
Используйте именованные группы захвата вместо анонимных 477
Используйте ретроспективные проверки вместо различных обходных путей 478
Используйте экранирование кодовой точки вместо суррогатных пар в регулярных выражениях 478
Используйте шаблоны Юникода вместо обходных путей 478

ГЛАВА 16. СОВМЕСТНО ИСПОЛЬЗУЕМАЯ ПАМЯТЬ 479

Введение 479
Здесь водятся драконы! 480
Поддержка браузера 481
Основы совместно используемой памяти 482
Критические секции, блокировки и условные переменные 483
Создание совместно используемой памяти 484
Совместно используется память, но не объекты 489
Условия гонки, вышедшие из строя хранилища, устаревшие значения, значения с разрывами, тиринг и многое другое 490
Объект Atomics 492
Низкоуровневые возможности объекта Atomics 495
Использование Atomics для приостановки и возобновления потоков 497
Пример совместно используемой памяти 498
Здесь водятся драконы! (Снова) 519
От старых привычек к новым 525
Используйте совместно используемые блоки вместо многократного обмена большими блоками данных 525
Используйте Atomics.wait и Atomics.notify вместо разделения заданий воркеров для поддержки цикла событий (при необходимости) 525

ГЛАВА 17. РАЗЛИЧНЫЕ АСПЕКТЫ 527

Тип данных BigInt 527
Создание значения типа BigInt 529
Явное и неявное преобразование 530
Производительность 531
Массивы BigInt64Array и BigUint64Array 531
Служебные функции 531
Новые целочисленные литералы 532
Двоичные целочисленные литералы 532
Восьмеричные целочисленные литералы, попытка № 2 533
Новые математические методы 534
Общие математические функции 534
Поддержка низкоуровневых математических функций 535
Оператор возведения в степень (**) 535
Изменения в `Date.prototype.toString` 537
Изменения в `Function.prototype.toString` 538
Дополнения конструктора Number 538
«Безопасные» целые числа 538
Константы `Number.MAX_SAFE_INTEGER` и `Number.MIN_SAFE_INTEGER` 539
Метод `Number.isSafeInteger` 540

Метод `Number.isInteger` 540
Методы `Number.isFinite` и `Number.isNaN` 540
Методы `Number.parseInt` и `Number.parseFloat` 541
Свойство `Number.EPSILON` 541
Символ `Symbol.isConcatSpreadable` 541
Различные хитрости синтаксиса 542
Оператор нулевого слияния 543
Опциональная цепочка 543
Необязательные привязки `catch` 546
Разрывы строк Юникода в JSON 546
Правильно сформированный JSON из метода JSON.stringify 546
Различные стандартные библиотеки/глобальные дополнения 547
Метод Symbol.hasInstance 547
Свойство `Symbol.unscopables` 547
Объект `globalThis` 549
Свойство `description` символа 549
Метод `String.prototype.matchAll` 550
Приложение Б: Возможности, доступные только для браузера 550
HTML-подобные комментарии 551
Хитрости регулярного выражения 552
Расширение управляющего символа экранирования (\cX) 552
Допуск недопустимых последовательностей 553
Метод `RegExp.prototype.compile` 553
Дополнительные встроенные свойства 553
Дополнительные свойства объекта 554
Дополнительные строковые методы 555
Различные фрагменты свободного или неясного синтаксиса 555
Когда же `document.all` есть… или нет? 557
Оптимизация хвостового вызова 558
От старых привычек к новым 561
Используйте двоичные литералы 561
Используйте новые математические функции вместо различных математических обходных путей 562
Используйте оператор нулевого слияния для значений по умолчанию 562
Используйте опциональную цепочку вместо проверок && 562
Уберите привязку ошибки (e) из “catch (e)”, если она не используется 562
Используйте оператор возведения в степень (**) вместо метода Math.pow 563

ГЛАВА 18. ГРЯДУЩИЕ ФУНКЦИОНАЛЬНЫЕ ВОЗМОЖНОСТИ КЛАССА 565

Публичные и приватные поля класса, методы и акцессоры 565
Определения публичного поля (свойства) 566

Приватные поля 572
Приватные методы и акцессоры экземпляра 579
Приватные методы 579
Приватные акцессоры 584
Публичные статические поля, приватные статические поля и приватные статические методы 585
Публичные статические поля 585
Приватные статические поля 586
Приватные статические методы 586
От старых привычек к новым 587
Используйте определения свойств вместо создания свойств в конструкторе (где это уместно) 587
Используйте приватные поля вместо префиксов (где это уместно) 588
Используйте приватные методы вместо функций вне класса для приватных операций 588

ГЛАВА 19. ВЗГЛЯД В БУДУЩЕЕ... 591

Оператор `await` верхнего уровня 592
Обзор и примеры использования 592
Пример 594
Обработка ошибок 599
Слабые ссылки и обратные вызовы очистки 600
Слабые ссылки 601
Обратные вызовы очистки 604
Индексы соответствия RegExp 609
Метод `String.prototype.replaceall` 611
Выражение `Atomics.asyncWait` 612
Различные хитрости синтаксиса 613
Числовые разделители 613
Поддержка Hashbang 614
Осуждаемые устаревшие возможности RegExp 614
Спасибо, что прочитали! 615

ПРИЛОЖЕНИЕ. ФАНТАСТИЧЕСКИЕ ВОЗМОЖНОСТИ И ГДЕ ОНИ ОБИТАЮТ 617

Функциональные возможности в алфавитном порядке 617
Новые положения 623
Новый синтаксис, ключевые слова, операторы, циклы и тому подобное 624
Новые литеральные формы 626
Дополнения и изменения стандартной библиотеки 626
Прочее 629

АЛФАВИТНЫЙ УКАЗАТЕЛЬ 631

ОБ АВТОРЕ

Ти Джей Краудер — инженер-программист с 30-летним стажем, и примерно половину этого времени он посвятил JavaScript. Он руководит британской компанией Farsight Software, которая занимается консалтингом и разработкой программного обеспечения. Будучи одним из 10 лучших авторов книги «Stack Overflow» и *лучшим* автором по тегу JavaScript, он любит использовать свои знания и опыт, чтобы помочь другим разработчикам с решением встречающихся технических проблем. Акцент делается не только на передачу знаний, но и на помощь в процессе решения проблем.

Ти Джей начал программировать в подростковом возрасте в Калифорнии, играя с Apple II и Sinclair ZX-80 и -81, используя BASIC и язык ассемблера. Свою первую настоящую работу с компьютерами он получил много лет спустя на должности технической поддержки в компании, выпускавшей продукт для судебных репортеров на ПК (DOS). Работая в службе поддержки, он самостоятельно выучил язык программирования C из руководств TurboC в офисе, переработал недокументированный сжатый двоичный формат файлов компании и использовал эти знания для создания в свободное время столь востребованной функции для продукта (которую компания вскоре начала поставлять). Вскоре его перевели в отдел разработки программного обеспечения. В течение следующих лет Ти Джей получил возможность и ресурсы вырасти в программиста и в итоге в ведущего инженера, создавая различные продукты, включая их первый продукт для Windows.

В своей следующей компании он взял на себя роль специалиста по предоставлению услуг и обучению разработчиков, в рамках которой он настраивал пользовательскую часть корпоративного продукта компании (используя VB6, JavaScript и HTML) и проводил обучающие занятия для фронтенд-разработчиков; в конечном счете он начал создавать обучающие курсы. Переезд из США в Лондон вернул его на должность разработчика, где он смог улучшить свои навыки разработки программного обеспечения и расширить познания в языках SQL, Java и JavaScript, прежде чем перейти на независимые контракты.

С тех пор в силу обстоятельств Ти Джей занимался в основном удаленной разработкой с закрытым исходным кодом для различных компаний и организаций (агентство НАТО, орган местного самоуправления Великобритании и различные частные фирмы), работая преимущественно на JavaScript, SQL, C# и (в последнее время) TypeScript. Стремление находиться в сообществе программистов привело его сначала в список рассылки PrototypeJS, затем в Stack Overflow, а теперь и на другие платформы.

Англичанин и американец по рождению, американец по воспитанию, Ти Джей живет в деревне в центральной Англии со своей женой и сыном.

О ТЕХНИЧЕСКОМ РЕДАКТОРЕ

Хаим Краузе — опытный программист, и его более чем тридцатилетний опыт подтверждает этот факт. Он работал ведущим инженером технической поддержки интернет-провайдеров еще в 1995 году, старшим инженером поддержки разработчиков в Borland для Delphi и более десяти лет работал в Силиконовой долине на различных должностях, включая инженера технической поддержки и инженера поддержки разработчиков. В настоящее время он является специалистом по военным симуляторам в Колледже командования и Генерального штаба армии США, работая над такими проектами, как разработка серьезных игр для использования в тренировочных целях. Он также стал автором нескольких обучающих видеокурсов по темам Linux и был техническим рецензентом более двух десятков книг.

О ТЕХНИЧЕСКОМ КОРРЕКТОРЕ

Марсия К. Уилбур — технический консультант в области полупроводников, специализирующийся на промышленном интернете вещей — IoT (IIoT) и искусственном интеллекте — AI. Марсия получила ученую степень в области компьютерных наук, технических коммуникаций и информационных технологий. Как президент Copper Linux User Group, она активно участвует в сообществе разработчиков, ведущих West Side Linux + Pi, и East Valley, ведущих регулярные проекты Raspberry Pi, Beaglebone, Banana Pi/Pro и ESP8266, включая домашнюю автоматизацию, игровые консоли, видеонаблюдение, сеть, мультимедиа и другие «Pi приколы».

БЛАГОДАРНОСТИ

«Я написал книгу» почти никогда не может быть точным утверждением.

Конечно, все слова в книге без кавычек принадлежат мне. Но ни одно из них не было бы здесь, если бы не другие. Другие предлагали поддержку, перспективу, воодушевление; другие просматривали черновики, чтобы помочь отсечь плевелы и выявить скрытый внутри самородок полезности; другие задавали вопросы на различных форумах на протяжении многих лет, давая мне возможность практиковать искусство и мастерство объяснения тех скудных знаний, которые я смог получить; другие отвечают на мои вопросы напрямую и указывают на ресурсы; другие помогают отточить мой язык и улучшить мое словесное мастерство.

Вся эта болтовня означает: я в долгу перед многими людьми. Это моя первая настоящая книга, и я допустил почти все возможные ошибки. Люди, перечисленные дальше, нашли и исправили столько ошибок, сколько смогли. Все, что осталось, принадлежит мне.

Прежде всего, спасибо моей жене Венди за ее удивительную поддержку, за то, что она была бесконечным и добровольным источником силы и источником вдохновения, за то, что вернула меня, когда (как выразился Джеймс Тейлор) я ловлю себя на том, что нахожусь в тех местах, куда я не должен был себя отпускать.

Спасибо моему сыну Джеймсу за то, что он так долго терпел, когда папу запирали в офисе.

Спасибо моей матери Вирджинии и отцу Норману за то, что они привили мне любовь к языку, обучению, чтению и письму. Это подарки на всю жизнь.

Спасибо моему лучшему другу Джоку за то, что он всегда был терпеливым собеседником и источником поддержки — и, если уж на то пошло, за то, что он сыграл важную роль в том, чтобы я придал высший приоритет программированию.

Спасибо всем моим редакторам и рецензентам в Wiley & Sons: Джиму Минателу и Питу Гохану за поддержку и сохранение проекта даже перед лицом разочарования автора; Дэвиду Кларку и Хаиму Краузе за редакторскую и техническую рецензию; Ким Кофер за ее отличную редактуру и помощь с грамматикой, синтаксисом и ясностью; Нэнси Белл за ее корректуру; художникам и композиторам в отделе продакшена; и всем остальным, кого я не назвал по имени, за поддержку всех аспектов проекта.

Спасибо Андреасу Бергмайеру за то, что он помог мне расставить все точки над «i» и перечеркнуть все «t». Острый взгляд и глубокое понимание Андреаса оказали огромную помощь на протяжении всей книги.

Спасибо члену Технического комитета № 39 (TC39) Даниэлю Эренбергу из Igalia за то, что он помог мне лучше понять, как работает TC39, а также за любезный обзор и помощь в доработке Главы 1. Его мягкие исправления, вклад и понимание значительно улучшили эту главу.

Спасибо члену Технического комитета № 39 Ларсу Т. Хансену из Mozilla (соавтору предложения JavaScript о совместно используемой памяти и атомарности) за его любезную и неоценимую помощь в правильном изложении деталей и области применения в главе 16. Его глубокие знания и перспектива сыграли здесь решающую роль.

И, наконец, спасибо вам, дорогой читатель, за то, что уделили свое время и внимание моим усилиям. Я надеюсь, что эта книга сослужит вам хорошую службу.

ВВЕДЕНИЕ

Эта книга предназначена всем программистам JavaScript (или TypeScript), которые хотят быть в курсе функций, добавленных в JavaScript за последние несколько лет, и оставаться в курсе роста и развития языка. Вы можете найти почти всю информацию, содержащуюся в этой книге, *где-нибудь* в Интернете, если будете достаточно внимательно искать и осторожно относиться к сайтам, которым доверяете. В этой книге собраны все технические подробности, а также показано, как отслеживать изменения по мере их внесения. Можно оставаться в курсе последних событий с помощью веб-сайта книги по адресу: `https://thenewtoys.dev`. Здесь я освещаю новые функции по мере их появления.

О ЧЕМ ЭТА КНИГА?

Вот краткий обзор того, что содержится в каждой главе.

Глава 1 «Новые возможности в ES2015–ES2020 и далее» — книга начинается с представления различных игроков в мире JavaScript и некоторых важных терминов. Затем в ней описывается определение «Новых возможностей» для целей книги. Описывается способ добавления новых функций в JavaScript, как и кем этот процесс управляется, и объясняется, как следить за этим процессом и участвовать в нем. Он заканчивается введением некоторых инструментов для использования новых возможностей в старых средах (или использования самых новых возможностей в текущих средах).

Глава 2 «Объявления блочной области видимости для `let` и `const`» охватывает новое объявление ключевых слов `let` и `const` и новые поддерживаемые ими области видимости, в том числе охват области видимости в глубину в циклах, в частности, новая обработка данных в циклах `for`.

Глава 3 «Функции» охватывает множество новых возможностей, связанных с функциями: стрелочные функции, значения параметров по умолчанию, остаточные параметры rest, свойство `name` и различные улучшения синтаксиса.

Глава 4 «Классы» охватывает новые возможности классов: основы, что такое `class`, а что нет, подклассы, ключевое слово `super`, встроенные подклассы `Array` и `Error`, а также псевдосвойство `new.target`. Глава 4 не описывает приватные поля и другие функции, проходящие через процесс подачи предложений. Они представлены в главе 18.

Глава 5 «Объекты» охватывает вычисляемые имена свойств, сокращенные свойства, получение и настройку прототипа объекта, новый тип символа (Symbol) и его связь с объектами, синтаксис метода, порядок свойств, синтаксис расширения свойств и множество новых функций объекта.

Глава 6 «Возможности итерации: итерируемые объекты, итераторы, циклы `for-of`, итеративные расширения, генераторы» охватывает итерацию, новый мощный инструмент для коллекций и списков, и генераторы, новый мощный способ взаимодействия с функциями.

Глава 7 «Деструктуризация» описывает этот важный новый синтаксис и то, как его использовать для извлечения данных из объектов, массивов и других итеративных объектов, с использованием значений по умолчанию, глубокого выбора и многого другого.

Глава 8 «Объекты Promise» углубляется в этот важный новый инструмент для управления асинхронными процессами.

Глава 9 «Асинхронные функции, итераторы и генераторы» подробно описывает новый синтаксис `async/await`, что позволяет использовать привычные логические структуры потока с асинхронным кодом, так же как асинхронные итераторы и генераторы работают с новым циклом `for-await-of`.

Глава 10 «Шаблоны, помеченные (тегированные) функции и новые возможности строк» описывает синтаксис литералов шаблонов, функции тегов и массу новых строковых функций, таких как улучшенная поддержка Юникода, обновления знакомых методов и множество новых.

Глава 11 «Массивы» охватывает широкий спектр новых методов работы с массивами, различные другие обновления массива, типизированные массивы, такие как `Int32Array`, и расширенные функции для взаимодействия с типизированными данными массива.

Глава 12 «Карты и множества» расскажет вам о новых коллекциях ключ/значение `Map` и `Set`, а также об их «слабых» (Weak) родственниках — `WeakMap` и `WeakSet`.

Глава 13 «Модули» представляет собой глубокое погружение в этот захватывающий и мощный способ организации вашего кода.

Глава 14 «Рефлексия — объекты Reflect и Proxy» рассказывает о новых мощных функциях динамического метапрограммирования объектов `Reflect` и `Proxy` и о том, как они связаны друг с другом.

Глава 15 «Обновления регулярных выражений» описывает все сделанные за последние несколько лет обновления регулярных выражений, такие как новые флаги, именованные группы захвата, утверждения ретроспективной проверки и новые возможности Юникода.

Глава 16 «Совместно используемая память» охватывает сложную и потенциально трудную область совместного использования памяти между потоками в программе JavaScript, в том числе описывает `SharedArrayBuffer` и объект `Atomics`, а также фундаментальные концепции и примечания о подводных камнях.

Глава 17 «Различные аспекты» охватывает широкий спектр вещей, которые не совсем вписывались в другие: BigInt, новые синтаксисы целочисленных литералов (двоичный, новый восьмеричный), необязательные привязки `catch`, новые математические методы, оператор возведения в степень, различные дополнения к объекту `Math`, оптимизация хвостовых вызовов, оператор нулевого слияния, необязательное связывание и «Приложение Б» (только-для-браузера) функции, определенные по соображениям совместимости.

Глава 18 «Грядущие функциональные возможности класса» переносит нас в недалекое будущее, описывая усовершенствования еще не завершенных классов, но которые находятся на стадии разработки: объявления открытых полей, частные поля и частные методы.

Глава 19 «Взгляд в будущее…» завершает книгу обзором будущего, некоторых дальнейших улучшений, находящихся в стадии разработки: оператор `await` верхнего уровня, WeakRefs (слабые ссылки) и обратные вызовы очистки, индексы соответствия регулярных

выражений, выражние `Atomics.asyncWait`, пара новых синтаксических функций, устаревшие функции регулярных выражений и различные предстоящие дополнения к стандартной библиотеке.

Приложение «Фантастические возможности и где они обитают» (приносим извинения Джоан Роулинг) содержит списки новых возможностей и рассказывает, в какой главе рассказывается о каждой из них. В этих списках: Возможности в алфавитном порядке; Новые положения; Новый синтаксис, Ключевые слова, Операторы, Циклы и тому подобное; Новые Литеральные формы; Дополнения и изменения Стандартной библиотеки; Разное.

КОМУ СТОИТ ЧИТАТЬ ЭТУ КНИГУ

Вам стоит прочесть эту книгу, если:

- у вас есть по крайней мере базовое понимание JavaScript, и
- вы хотите изучить новые возможности, появившиеся в последние годы.

Это не академическая книга только для экспертов. Это прагматичная книга для обычных программистов JavaScript.

Почти каждый, взявший в руки эту книгу, уже будет знать кое-что из ее содержимого, но почти никто из читателей не будет знать *всего* этого. Может быть, вы имеете представление об основах `let` и `const`, но еще не совсем понимаете работу функции `async`. Возможно, промисы кажутся вам старомодными, но вы видели незнакомый вам синтаксис в каком-то современном коде, с которым столкнулись. Здесь вы найдете все об этих ситуациях с ES2015 по ES2020 (и далее).

КАК ПОЛЬЗОВАТЬСЯ ЭТОЙ КНИГОЙ

Прочтите главу 1. Она определяет множество терминов, которые я использую в остальной части книги. Пропуск главы 1, скорее всего, поставит вас в тупик.

После этого у вас есть выбор: читать главы по порядку или в произвольном порядке?

Я расположил главы в таком порядке по определенным причинам. И я описываю что-то в более поздних главах на основе содержания предыдущих глав. Например, вы узнаете о промисах в главе 8, что важно для понимания функций `async` в главе 9. Естественно, я рекомендую прочитать их в предложенном порядке. Но я уверен, что вы умный человек и знаете, что делаете. Если вы будете читать главы книги в произвольном порядке, у вас все получится.

Но я предлагаю вам прочитать — или, по крайней мере, просмотреть — все главы (возможно, за исключением главы 16; подробнее об этом через минуту). Даже если вы думаете, что знаете какую-то функцию, вероятно, в этой книге есть что-то, чего вы не знаете или только думаете, что знаете. Например: возможно, вы планируете пропустить главу 2, потому что уже знаете все, что нужно знать о `let` и `const`. Может быть, вы даже знаете, почему

```
for (let i = 0; i < 10; ++i) {/*...*/
    setTimeout(() => console.log(i));
   }
```

создает десять *разных* переменных с именем `i`, и почему использование

```
let a = "ay";
```

```
var b = "bee";
```

в глобальной области видимости создает свойство `window.b`, но не создает свойство `window.a`. Даже если вы это сделаете, стоит просмотреть главу 2, просто чтобы убедиться, что нет других складок, которые вы не заметили.

Глава 16 — это немного особый случай: речь идет о совместном использовании памяти между потоками. Большинству программистов JavaScript не нужно разделять память между потоками. *Некоторым* из вас необходимо это сделать, и именно поэтому в книге есть глава 16. Но большинству это не нужно, и если вы относитесь к этой категории, просто отложите в памяти тот факт, что, *если* вам понадобится совместно используемая память в какой-то момент в будущем, вы можете вернуться и узнать о ней в главе 16. Это просто замечательно.

Помимо всего этого: приводите примеры, экспериментируйте с ними и, прежде всего, получайте удовольствие.

Весь код для самостотельной работы можно загрузить на странице:
http://addons.eksmo.ru/it/JS_Full%20Code.zip

1

Новые возможности в ES2015–ES2020 и далее

СОДЕРЖАНИЕ ГЛАВЫ

- Определения, что есть что и терминология
- Объяснение версий JavaScript (ES6? ES2020?)
- Что это за «новые возможности»?
- Процесс создания новых функций JavaScript
- Инструменты для использования JavaScript следующего поколения

Язык JavaScript сильно изменился за последние несколько лет.

Если бы вы были активным разработчиком JavaScript в 2000-х годах, какое-то время вам было бы простительно думать, что JavaScript стоит на месте. После 3-го издания спецификации в декабре 1999 года разработчики целых 10 лет ждали следующего издания. Со стороны казалось, что ничего не происходит. На самом деле была проделана огромная работа. Ее результаты просто не нашли своего места в официальной спецификации и нескольких движках JavaScript. Мы могли бы (но не будем) потратить целую главу — если не книгу — на создание различных групп, занимающих важные положения по отношению к JavaScript, и как они некоторое время не могли договориться об общем пути продвижения вперед. Ключевым моментом является то, что они в итоге *договорились* о дальнейших действиях на судьбоносной встрече в Осло в июле 2008 года после долгих предварительных переговоров. Эта общая точка соприкосновения, которую Брендан Эйх (создатель JavaScript) позже назвал Harmony, проложила путь к спецификации 5-го издания в декабре 2009 года (4-е издание так и не было завершено) и заложила основу для постоянного прогресса.

И как же стремительно все продвинулось!

В этой главе вы получите обзор новых функций, появившихся с 2009 года (они подробно рассматриваются в остальной части книги). Вы узнаете, кто отвечает за продвижение JavaScript, какой процесс сейчас используется для этого и как вы можете отслеживать, что происходит, и (при желании) участвовать. Вы узнаете об инструментах, которые

можно использовать для написания современного JavaScript сегодня, даже если вам приходится ориентироваться на среды, не идущие в ногу со временем.

ОПРЕДЕЛЕНИЯ, ЧТО ЕСТЬ ЧТО И ТЕРМИНОЛОГИЯ

Чтобы поговорить о том, что происходит с JavaScript, нам нужно определить некоторые имена и общую терминологию.

Что такое Ecma? ECMAScript? TC39?

То, что мы называем «JavaScript», стандартизировано ассоциацией Ecma International[1] как «ECMAScript». Это организация по стандартизации, ответственная за несколько стандартов вычислений. Стандарт ECMAScript — ECMA-262. Люди, ответственные за стандарт, состоят в Международном техническом комитете Ecma 39 («TC39»), которому поручено «стандартизировать универсальный, кроссплатформенный, вендорнейтральный язык программирования ECMAScript. Такая стандартизация включает в себя синтаксис языка, семантику, библиотеки и дополнительные технологии, поддерживающие язык»[2]. Они также управляют другими стандартами, такими как спецификация синтаксиса JSON (ECMA-404) и, в частности, спецификация API интернационализации ECMAScript (ECMA-402).

В этой книге и в обычном использовании JavaScript — это ECMAScript и наоборот. Иногда, особенно в течение десятилетия, когда разные группы занимались разными вещами, название «JavaScript» использовалось специально для обозначения языка, который разрабатывала Mozilla (у него было несколько функций, которые либо никогда не попадали в ECMAScript, либо заметно изменились до того, как они это сделали). Но с момента появления Harmony это использование все больше устаревает.

Что такое ES6? ES7? ES2015? ES2020?

Наличие всех этих различных сокращений может сбить с толку. Немалую роль в этом сыграло то, что у некоторых обозначений указаны номера изданий, а у других — годы релиза. В этом разделе объясняется, что это такое и почему существуют два вида обозначений.

Вплоть до 5-го издания TC39 ссылался на версии спецификации с использованием номера издания. Полное название спецификации 5-го издания таково:

- Стандарт ECMA-262.
- 5-е Издание / Декабрь 2009.
- ECMAScript Language Specification.

Поскольку «5-е Издание ECMAScript» — это слишком многословно, естественным было сказать «ES5».

---

[1] Ранее известная как Европейская ассоциация производителей компьютеров (ECMA), но теперь в названии организации с заглавной буквы пишется только буква *E* в Ecma. — *Здесь и далее прим. авт., если не указано иное.*

[2] `http://www.ecma-international.org/memento/TC39.htm`

Начиная с 6-го издания в 2015 году комитет TC39 внедрил процесс постоянного совершенствования, в котором спецификация представляет собой рабочий проект редактора[3] с ежегодными снимками. (Подробнее об этом позже в текущей главе.) Когда они это сделали, к названию языка был добавлен год:

- Стандарт ECMA-262.
- 6-е Издание / Июнь 2015.
- Спецификация языка ECMAScript® 2015.

Таким образом, стандарт 6-го издания ECMAScript («ES6») определяет ECMAScript 2015, или сокращенно «ES2015». До публикации «ES6» само по себе стало модным словом и до сих пор широко используется. (К сожалению, оно часто используется неточно, ссылаясь не только на функции из ES2015, но и на те, которые появились позже в ES2016, ES2017 и т. д.)

Вот почему существуют два стиля: стиль, использующий издание (ES6, ES7, …), и стиль, использующий год (ES2015, ES2016, …). Выбирайте, что вам подходит. Стандарт ES6 — это ES2015 (или иногда, неправильно, ES2015+), ES7 — это ES2016, ES8 — это ES2017, и так далее до (на момент выхода этой книги) ES11, представляющий ES2020. Встречаются также «ESnext» или «ES.next», которые иногда используются для обозначения предстоящих изменений.

В этой книге я использую то, что считаю формирующимся консенсусом — старый стиль для ES5 и более ранних версий и новый стиль для ES2015 и более поздних версий.

Обычно я называю конкретную версию, в которой была введена функция. Но все вышесказанное дает понять, что факт отношения `Array.prototype.includes` к ES2016, а `Object.values` — к ES2017, на самом деле не имеет большого значения. Более важно то, что на самом деле поддерживается в ваших целевых средах, и нужно ли вам либо воздерживаться от использования определенной функции, либо переносить и/или заполнять ее.

(Подробнее о транспилировании и полифиллировании (заполнении) позже в разделе «Использование сегодняшних возможностей во вчерашних условиях и завтрашних возможностей сегодня».)

«Движки» JavaScript, браузеры и др.

В этой книге я буду использовать термин «движок JavaScript» для обозначения программного компонента, запускающего код JavaScript. Движок JavaScript должен знать, как:

- проводить синтаксический анализ JavaScript;
- интерпретировать или компилировать программу в машинный код (или все вместе);
- запускать результат в среде, работающей согласно спецификации.

Движки JavaScript также иногда называют виртуальными машинами (virtual machines), или сокращенно VM.

Конечно, одно из распространенных применений движков JavaScript — это веб-браузеры:

[3] `https://tc39.es/ecma262/`

- Google Chrome использует свой движок V8 (также используется в Chromium, Opera и Microsoft Edge v79 и более поздних версий), за исключением iOS (подробнее об этом через секунду).
- Safari от Apple (для Mac OS и для iOS) использует их движок JavaScriptCore.
- Firefox от Mozilla использует свой движок SpiderMonkey, за исключением iOS.
- Internet Explorer от Microsoft использует свой движок JScript, который становится все более устаревшим, поскольку он получает только исправления безопасности.
- Microsoft Edge v44 и более ранних версий («Устаревший Edge») использует движок Microsoft Chakra engine. В январе 2020 года был выпущен Edge v79, основанный на проекте Chromium и использующий движок V8, за исключением iOS. (Номер версии подскочил с 44 до 79, чтобы соответствовать Chromium.) Движок Chakra по-прежнему применяется в различных продуктах, использующих элемент управления Microsoft WebView, таких как надстройки Microsoft Office JavaScript, хотя на каком-то этапе он может быть заменен. (WebView2, использующий Chromium Edge, находится в стадии предварительного просмотра для разработчиков с начала 2020 года.)

Chrome, Firefox, Edge и другие браузеры, работающие на операционной системе Apple iOS для iPad и iPhone, в настоящее время не могут использовать собственные движки JavaScript, поскольку для компиляции и запуска JavaScript (а не просто его интерпретации) им необходимо выделить исполняемую память, и только собственные приложения Apple для iOS, но не сторонних поставщиков, могут это осуществлять. Таким образом, Chrome и Firefox (и другие) должны использовать JavaScriptCore от Apple на iOS, хотя они используют собственный движок на ПК и Android. (По крайней мере, на данный момент это так. Команда V8 добавила режим «только для интерпретатора» в V8 в 2019 году. Это означает, что Chrome и другие, использующие V8 браузеры, могут работать в этом режиме на iOS, поскольку ему не нужно использовать исполняемую память.) В этой книге, если я говорю, что что-то «поддерживается Chrome» или «поддерживается Firefox», я имею в виду версии, отличные от iOS, использующие V8 или SpiderMonkey соответственно.

Движки JavaScript также используются в приложениях на ПК (Electron[4], React Native[5], и др.), веб-серверах и других типах серверов (часто использующих Node.js[6]), не веб-приложения, встроенные приложения — практически везде.

ЧТО ЗА «НОВЫЕ ВОЗМОЖНОСТИ»?

В этой книге «новые возможности» — это новые функциональные возможности, добавленные в JavaScript в ES2015-ES2020 (и обзор ожидаемых в ближайшем будущем). За эти шесть обновлений язык прошел долгий путь. Ниже приводится общий обзор. (В Приложении A содержится более полный список изменений.) Некоторые термины из списка могут быть вам незнакомы: не переживайте, вы узнаете их по ходу книги.

[4] https://www.electronjs.org/

[5] https://reactnative.dev/

[6] https://nodejs.org/

- *Блочная область видимости Opt-in* (`let`, `const`): Более узкая область видимости для переменных, продуманная обработка области тела цикла `for`, «переменные», значение которых не может измениться (константы — `const`).
- *«Стрелочные» функции:* Легкие, лаконичные функции, которые особенно полезны для обратных вызовов, поскольку они закрываются над ключевым словом `this`, а не получают собственного значения `this`, устанавливаемого при их вызове.
- *Улучшения параметров функций:* Значения по умолчанию; деструктуризация параметров, остаточные параметры «rest», висящие запятые (последние или замыкающие запятые).
- *Итерируемые объекты:* Четко определенная семантика для создания и использования итерируемых объектов (таких, как массивы и строки), внутриязыковые итерационные конструкции (`for-of`, `for-await-of`); функции генератора для генерации последовательностей, которые можно повторять (включая асинхронные последовательности).
- *Синтаксис оператора Spread:* Расширение (Spread) элементов массива (или других итерируемых объектов) в новые массивы, расширение свойств объекта на новые объекты и расширение итеративных элементов до дискретных аргументов функции. Оно особенно полезно для функционального программирования или везде, где используются неизменяемые структуры.
- *Синтаксис Rest:* Объединение «остаточных» (rest) свойств объекта, значений итерации или аргументов функции в объект или массив.
- *Другие улучшения синтаксиса:* Разрешение висящей запятой в списке аргументов при вызове функции; исключение неиспользуемых идентификаторов в операторах `catch`; восьмеричные литералы нового стиля; двоичные литералы; символы-разделители в числовых литералах; и многое другое.
- *Деструктуризация:* Выбор значений из массивов/объектов в сжатом виде, отражающем синтаксис литералов объектов и массивов.
- `class`: Заметно более простой декларативный синтаксис для создания функций конструктора и связанных с ними объектов-прототипов, сохраняя при этом присущую JavaScript прототипическую природу.
- *Улучшения в асинхронном программировании:* Промисы, функции `async` и `await` — все это заметно уменьшает «ад обратного вызова».
- *Улучшения объектного литерала:* Имена вычисляемых свойств, сокращенные свойства, синтаксис метода, висящие запятые после определений свойств.
- *Шаблонные литералы:* Простой декларативный способ создания строк с динамическим содержимым и выхода за рамки строк с помощью функций шаблона с тегами.
- *Типизированные массивы:* Низкоуровневые истинные массивы для использования собственных API (и многое другое).
- *Совместно используемая память:* Возможность реального совместного использования памяти между потоками JavaScript (включая примитивы координации между потоками).
- *Улучшения в строках Юникода:* Экранирующие последовательности кодовых точек Юникода; поддержка доступа к кодовым точкам вместо кодовых единиц.
- *Улучшения регулярных выражений:* Утверждения ретроспективной проверки; именованные группы захвата; индексы захвата; экранирование свойств Юникода; нечувствительность к регистру Юникода.

- *Коллекции Map:* Коллекции ключей/значений, в которых ключи не обязательно должны быть строками.
- *Объект Set:* Коллекции уникальных значений с четко определенной семантикой.
- *WeakMap, WeakSet и WeakRef*: Встроенные модули для хранения только слабых ссылок на объекты (что позволяет собирать из них мусор).
- *Стандартные дополнения к библиотеке:* Новые методы для `Object`, `Array`, `Array.prototype`, `String`, `String.prototype`, `Math` и др.
- *Поддержка динамического метапрограммирования:* `Proxy` и `Reflect`.
- *Символы:* Гарантированно уникальные значения (особенно полезно для уникальных имен свойств).
- *Тип BigInt:* Целые числа произвольной точности.
- И многие-многие другие.

Все эти новые функциональные возможности, в частности новый синтаксис, могут быть ошеломляющими. Не беспокойтесь! Нет необходимости внедрять новые функции до тех пор, пока вы не будете готовы к этому и не будете испытывать в них потребность. Один из ключевых принципов, которого придерживается TC39, — «Не ломайте сеть». Это значит, что JavaScript должен оставаться «веб-совместимым», то есть совместимым с огромным объемом уже существующего в современном мире кода[7]. Если вам не нужна или не нравится новая функция, вам не обязательно ее использовать. Ваш старый способ реализации функции всегда будет работать. Но во многих случаях вы будете искать веские причины применения новых возможностей. В частности, новый синтаксис функции предлагает то, что проще в написании, понимании и менее подвержено ошибкам. Или, как в случае с `Proxy` и `WeakMap`/`WeakSet`, совместно используемой памятью, и другими, новый функционал предоставляет возможности, которые невозможно реализовать старыми методами.

По соображениям экономии места в этой книге рассматриваются только новые возможности в самой спецификации JavaScript — ECMA262. Но в спецификации API интернационализации ECMAScript[8] также есть несколько интересных новых возможностей, ECMA-402, с которыми стоит ознакомиться. Вы можете найти описание ECMA-402 на веб-сайте этой книги по адресу: `https://thenewtoys.dev/internationalization`.

КАК СОЗДАЮТСЯ НОВЫЕ ВОЗМОЖНОСТИ?

В этом разделе вы узнаете, кто отвечает за продвижение JavaScript, какой процесс они используют для этого, а также как следить за этим процессом и участвовать в нем.

Кто здесь главный

Ранее вы узнали, что Технический комитет 39 Ecma International (TC39) отвечает за создание и выпуск обновленных спецификаций для стандарта ECMAScript. Комитет состоит из разработчиков JavaScript, авторов фреймворков, крупных авторов/сопровождающих

[7] Комитет также заботится о существенной части кода JavaScript, который напрямую не связан с Интернетом.

[8] `https://tc39.es/ecma402/`

веб-сайтов, исследователей языков программирования, представителей всех основных движков JavaScript, влиятельных разработчиков JavaScript и других заинтересованных в успехе и будущем JavaScript сторон. Они проводят регулярные встречи, исторически шесть раз в год по три дня. Чтобы участвовать в собраниях в качестве члена, ваша организация может присоединиться к Ecma. Комитет TC39 ориентируется в совокупности желаний разработчиков, сложности реализации, проблем безопасности, обратной совместимости и многих других конструкторских решений, чтобы предложить сообществу JavaScript новые и полезные функции.

Для гарантий работы комитета как части сообщества, а не отдельно от него, TC39 поддерживает репозиторий ecma262 на GitHub[9] с актуальной спецификацией (доступен для просмотра по адресу: `https://tc39.es/ecma262/`) и хранилище предложений[10] для предложений, поступающих с помощью процесса TC39, описанного в следующем разделе. Некоторые члены также активно участвуют в дискуссионной группе TC39[11]. Записи встреч TC39 и связанные с ним материалы (слайды и др.) публикуются на `https://github.com/tc39/notes`.

Вы можете узнать больше о TC39 и о том, как принять участие в работе комитета, на сайте `https://tc39.es/`. Вы также можете узнать больше о том, как работает TC39, на `https://github.com/tc39/how-we-work`.

Процесс

TC39 принял четко определенный процесс в ноябре 2013 года и впервые опубликовал его в январе 2014 года. Процесс состоит из нескольких этапов (с 0 по 4), через которые TC39 продвигает предложения. На каждом этапе есть конкретные ожидания и четкие критерии для перехода к следующему этапу. Как только предложение удовлетворяет критериям для перехода, комитет консенсусом принимает решение о его продвижении.

Стоит ознакомиться с самим документом процесса[12], но вкратце этапы таковы:

- *Этап 0: Соломенный человек.* У кого-то появилась идея, которую, по их мнению, стоит рассмотреть, поэтому они обдумали ее, немного доработали и выдвинули. (Этот этап не совсем этап, а термин в разное время применялся к разным вещам.) Если человек, выдвигающий это предложение, не является членом комитета TC39, ему необходимо зарегистрироваться в качестве участника, не являющегося членом[13] (что может сделать любой желающий). Некоторые предложения этапа 0 в итоге попадают в репозиторий предложений TC39. Как правило, это те предложения, которые получили потенциальную поддержку действующего члена комитета. Если предложение этапа 0 вызывает достаточный интерес, член TC39 может внести его в повестку дня заседания TC39 для обсуждения и рассмотрения на этапе 1.
- *Этап 1: Предложение.* Как только предложение было внесено в комитет и был достигнут консенсус в отношении его дальнейшего изучения, комитет переводит его на этап 1 с представителем, который проведет его через весь процесс. Если

[9] `https://github.com/tc39/ecma262`
[10] `https://github.com/tc39/proposals`
[11] `https://es.discourse.group/`
[12] `https://tc39.es/process-document/`
[13] `https://tc39.es/agreements/contributor/`

для него еще нет репозитория GitHub, его создает сторонник предложения из комитета, создатель или другая заинтересованная сторона. Затем члены сообщества (независимо от того, входят они в комитет или нет) обсуждают его, развивают дальше, исследуют аналогичную технологию на других языках или в других средах, уточняют область применения, определяют общую форму решения и в целом конкретизируют идею. В результате этой работы может оказаться, что выгоды не стоят затрат, или что идею нужно разделить и добавить к другим предложениям и т. д. Но если у предложения есть основания, вовлеченные люди (состав группы которых, возможно, со временем изменился) составят некоторый первоначальный проект спецификации языка, API и семантики и передадут его в TC39 для рассмотрения на этапе 2.

- *Этап 2: Проект.* Когда оно будет готово, предложение из этапа 1 может быть представлено на заседании TC39 для рассмотрения на этапе 2. Это означает поиск консенсуса в отношении того, что предложение должно быть принято, с ожиданием, что оно, скорее всего, пройдет через весь процесс. На этапе 2 сообщество совершенствует точный синтаксис, семантику, API и т. д. и подробно описывает решение, используя язык формальной спецификации. Часто на этом этапе создаются полифиллы и/или плагины Babel, позволяющие экспериментировать с реальным использованием. В зависимости от сферы охвата предложение может оставаться на этапе 2 в течение некоторого времени по мере проработки деталей.
- *Этап 3: Кандидат.* Как только команда доработает предложение до окончательной формы проекта и создаст для него формальный язык спецификации, представитель может выдвинуть его на этап 3. Это означает поиск консенсуса в отношении того, что предложение готово к реализации в движках JavaScript. На этапе 3 само предложение практически стабильно. Ожидается, что изменения на данном этапе будут ограничены обратной связью, связанной с внедрением, например, о случаях, обнаруженных в ходе внедрения, проблемах с веб-совместимостью или трудностях внедрения.
- *Этап 4: Результат.* На данный момент функция завершена и готова к добавлению в черновик редактора по адресу: `https://tc39.es/ecma262/`. Чтобы достичь этой заключительной стадии, функция должна пройти приемочные тесты в наборе тестов TC39 test262[14]; по крайней мере две различные совместимые реализации, которые проходят тесты (например, работа в версии V8 в Chrome Canary и SpiderMonkey в Firefox Nightly или SpiderMonkey в Firefox и JavaScriptCore в Safari Tech Preview и т. д.). Как только эти критерии будут выполнены, заключительный шаг для завершения этапа 4 — запрос от команды, работающей над функцией, на извлечение в репозиторий ecma262 для включения изменений спецификации в проект редактора, а группа редакторов ECMAScript приняла этот PR.

Через такой процесс проходят *предложения*, чтобы стать частью JavaScript. Но не каждое изменение является предложением. Меньшие изменения могут быть внесены путем консенсуса на совещаниях TC39 на основе запроса на изменение спецификации. Например, вывод `Date.prototype.toString` изменено между ES2017 и ES2018

[14] `https://github.com/tc39/test262`

(см. главу 17) в результате консенсуса по запросу на извлечение, а не поэтапного предложения. Часто это редакторские изменения или изменения, отражающие реальность того, что уже делают движки JavaScript, но чего нет в спецификации. Это могут быть изменения в том, что говорится в спецификации. Они были согласованы, потому что комитет TC39 считает, что они оба желательны и «веб-совместимы» (не нарушат большого количества существующего кода) — например, изменение в ES2019, делающее `Array.prototype.sort` стабильной сортировкой (см. главу 11). Если вы хотите узнать, какие изменения рассматриваются или вносятся таким образом, посмотрите записи под ярлыками «нужен консенсус» в репозитории `https://github.com/tc39/ecma262` (и аналогично для изменений ECMA-402 в репозитории `https://github.com/tc39/ecma402`). Чтобы найти завершенные, посмотрите на ярлыки «есть консенсус», «редакционная поправка» и/или «нормативное изменение». В какой-то момент может возникнуть более формальный процесс для этих изменений с ярлыком «нужен консенсус», но пока все так, как есть.

Вовлечение в процесс

Если вы видите предложение, которое вас интересует, и вы хотите принять в нем участие, когда вам следует это сделать? Как должно выглядеть это участие?

Одна ключевая вещь: принимайте участие как можно раньше. Как только предложение достигает этапа 3, обычно рассматриваются только критические изменения, основанные на опыте внедрения. Лучший период для принятия участия — на этапах 0, 1 и 2. Именно тогда вы сможете дать представление, основанное на вашем опыте, помочь определить семантику, попробовать то, что предлагается, используя такие инструменты, как Babel (о которых вы узнаете в дальнейших разделах) и т. д. Дело не в том, что вы не можете занять полезное положение в предложении на этапе 3 (иногда необходимо выполнить задачи с точки зрения уточнения текста спецификации или помощи в написании документации для разработчиков). Просто имейте в виду, что предлагать изменения на этапе 3 обычно бесполезно, если вы не один из тех людей, кто реализует предложения в движке JavaScript, и сталкиваетесь с проблемой.

Итак: вы нашли предложение, в котором хотите принять участие; что теперь? Это зависит от вас, но вот несколько предложений:

- *Проведите свое исследование.* Внимательно прочитайте пояснительную записку к предложению (`README.md` — оно связано со списком предложений TC39) и другими документами. Если они относятся к предшествующему уровню техники (например, аналогичная функция на другом языке), полезно ознакомиться с этим предшествующим уровнем техники. Если есть начальный текст спецификации, прочтите его. (Это руководство может быть полезным в такой ситуации: `https://timothygu.me/es-howto/`.) Вы должны быть уверены, что ваше предложение будет подкреплено точной информацией.
- *Попробуйте эту функцию!* Даже если вы еще не можете использовать новую функцию, вы можете написать спекулятивный код (код, который вы не можете запустить, но можете обдумать), чтобы оценить, насколько хорошо предложение решает проблему, которую ставит перед собой. Если для этого есть плагин Babel, попробуйте написать и запустить код. Посмотрите, как эта функция работает, и оставьте отзыв о ней.

- *Ищите способы, которыми вы можете помочь.* Это не только предложения и отзывы. Например, вы можете искать проблемы, по которым был достигнут консенсус относительно того, что делать, но ни у кого не было времени на реализацию (изучение уровня техники, обновление объяснения, обновление текста спецификации). Вы можете сделать эти обновления, если вам нравятся такие задачи. Вы можете координировать свои действия с авторами предложений, обсуждая вклады в тикетах GitHub.

Принимая участие, не забывайте относиться ко всем с уважением и быть дружелюбными, терпеливыми, открытыми и внимательными. Будьте осторожны при выборе слов. У вас больше шансов оказать влияние, если вы хорошо относитесь к людям и демонстрируете дух сотрудничества, а не кажетесь сердитым, пренебрежительным или мешающим работе. Обратите внимание, что встречи TC39 и онлайн-пространства, используемые для разработки предложений, регулируются Кодексом поведения (Code of Conduct)[15]. По поводу случаев ненадлежащего поведения обратитесь в комитет по кодексу поведения TC39 (см. документ по ссылке).

СЛЕДИТЕ ЗА НОВЫМИ ВОЗМОЖНОСТЯМИ

Вам не *нужно* идти в ногу с новыми возможностями. Как я уже упоминал, ваш старый способ ведения дел никуда не денется. Но, если вы читаете эту книгу, я подозреваю, что вы хотите изменений.

Поскольку выше вы читали описание процесса, вас, возможно, поразило то, что процесс гарантирует, что новые функции попадут в реальный мир разработки *до* того, как они будут добавлены в спецификацию. Напротив, при выходе ES5 в 2009 году и ES2015 в 2015 большинство описанных в них функций не существовало (в описанном виде) ни в одном из доступных на тот момент движков JavaScript. Если спецификация соответствует новым функциям, а не наоборот, как можно узнать, какие функции появятся в ближайшем будущем? Вот несколько способов:

- Следите за репозиторием предложений на GitHub (`https://github.com/tc39/proposals`). Дошедшее до этапа 3 предложение, вероятно, будет добавлено в течение года или двух. Даже функции этапа 2 с большой вероятностью будут добавлены в конечном счете, хотя любая функция может быть отклонена TC39 практически на любом этапе.
- Ознакомьтесь с записями заседаний TC39, размещенными по адресу: `https://github.com/tc39/notes`.
- Участвуйте в дискуссионной группе TC39 (`https://es.discourse.group/`)[16].
- Обратите внимание на то, что происходит с инструментами, описанными в следующем разделе.

[15] `https://tc39.es/code-of-conduct/`

[16] Экземпляр Discourse в значительной степени заменяет список рассылки для неформального обсуждения es-discussion. Список все еще существует, но многие представители TC39 не рекомендуют его использовать.

Вы также можете следить за процессом по адресу: `https://thenewtoys.dev`. Там я продолжаю рассказывать о том, что будет дальше.

ИСПОЛЬЗОВАНИЕ ТЕКУЩИХ ФУНКЦИЙ ВО ВЧЕРАШНИХ УСЛОВИЯХ И РАЗРАБАТЫВАЕМЫХ ВОЗМОЖНОСТЕЙ СЕГОДНЯ

Что касается простого изучения функций, описанных в этой книге, вам не стоит беспокоиться о работе со средами, которые их не поддерживают. Почти все описанное (кроме глав 18 и 19) поддерживается текущими версиями браузеров Chrome, Firefox, Safari и Edge для ПК и, как минимум, Node.js. Просто используйте один из них для запуска кода.

ИСПОЛЬЗОВАНИЕ NODE.JS ДЛЯ ЗАПУСКА ПРИМЕРОВ

По умолчанию, когда вы запускаете скрипт с помощью Node.js таким образом:

```
node script.js
```

программа запускает скрипт в области модуля, а не в глобальной области видимости. Несколько примеров в этой книге демонстрируют вещи, которые происходят только в глобальном масштабе. Поэтому они не будут работать, если запустить код таким образом.

Для таких примеров используйте браузер при запуске кода либо цикл считывания/выполнения/вывода (REPL) в Node.js. Чтобы использовать REPL, не нужно указывать файл скрипта для запуска в качестве аргумента команды node. Вместо этого вы перенаправляете скрипт в него с помощью оператора < (это работает как в системах Unix/Linux/Mac OS, так и в Windows):

```
node < script.js
```

Я напомню вам сделать это, когда пример необходимо будет запустить в глобальной области видимости.

Однако на каком-то этапе вы, вероятно, захотите использовать новые функции в среде, которая их не поддерживает. Например, большая часть разработки на JavaScript по-прежнему ориентирована на веб-браузеры, а движки JavaScript в разных браузерах получают новые функции в разное время (если вообще получают — Internet Explorer не поддерживает ни одну из новых функций, обсуждаемых в этой книге[17], но все еще удерживает некоторую долю на мировом рынке на момент написания этой книги, особенно в правительстве и внутренних сетях крупных компаний).

[17] Не совсем корректно — у него есть неполная версия let и const.

Это было проблемой во время выпуска ES5, поскольку в то время очень мало этих возможностей было реализовано в любом движке JavaScript. Но большая часть ES5 была новыми стандартными библиотечными функциями, а не значительными изменениями синтаксиса, поэтому их можно было «полифилировать» (добавить, включив дополнительный скрипт, который предоставлял недостающие объекты/функции), используя различные проекты, такие как es5-shim.js[18], core-js[19], es-shims[20] или аналогичные. Однако во время работы над ES2015 в период с 2010 по 2015 год было ясно, что для хорошей разработки этого синтаксиса требуется реальный опыт работы с новым синтаксисом. Но реализации JavaScript еще не поддерживали новый синтаксис — очевидная «уловка-22»[21].

Производители инструментов спешат на помощь! Они создали такие инструменты, как Traceur[22] и Babel[23] (ранее 6to5), которые берут исходный код, использующий новый синтаксис в качестве входных данных, преобразуют его для использования старого синтаксиса и выводят код в старом стиле (опционально вместе с полифилами и другими функциями поддержки среды выполнения). Аналогичным образом TypeScript[24] поддерживал основные части того, что станет ES2015, задолго до завершения спецификации. Эти инструменты позволяют вам писать код нового стиля, но преобразовывать его в код старого стиля, прежде чем отправлять его в старые среды. Этот процесс преобразования называется по-разному — «компиляцией» или «транспилированием». Изначально это было удобно для получения отзывов об улучшениях JavaScript, запланированных для ES2015, но даже когда вышел ES2015, это был полезный способ написания кода в новом стиле, если планировалось запускать его в среде без новых функций.

На момент написания этой Traceur прекратил свое существование, но Babel используют многие разработчики JavaScript по всему миру. В Babel есть преобразования почти для всех функций, которые находятся в процессе создания, — даже для тех, которые находятся на этапе 1, хотя они могут заметно измениться перед продвижением. (Так что используйте их на свой страх и риск. Функции с этапа 3 и далее довольно безопасны в использовании.) Вы выбираете желаемые плагины преобразования, пишете свой код, используя эти функции, и Babel создает код, который вы можете использовать в средах без этих функций.

Транспилирование примера с помощью Babel

В этом разделе мы кратко рассмотрим использование Babel для переноса кода с использованием функции ES2015, называемой *стрелочной функцией*, в код, совместимый с ES5, который работает со стандартом IE11. Но это всего лишь пример. Вы могли бы так же легко использовать Babel для преобразования кода с использованием функции с этапа 3, еще не присутствующей ни в одном из движков JavaScript, в код, совместимый с ES2020.

---

[18] https://github.com/es-shims/es5-shim

[19] https://github.com/zloirock/core-js

[20] https://github.com/es-shims/

[21] Уловка-22 — целенаправленно созданная, получившаяся случайно или органично присущая ситуации правовая, административная, социальная либо логическая ошибка, состоящая в том, что попытка соблюдения некоторого правила сама по себе означает его нарушение. Индивид, подпадающий под действие таких норм, не может вести себя целесообразно. — *Прим. ред.*

[22] https://github.com/google/traceur-compiler

[23] http://babeljs.io/

[24] http://typescriptlang.org/

Babel также поддерживает некоторые преобразования, которые вообще не находятся в процессе внедрения, такие как JSX[25] (используется в некоторых фреймворках JavaScript, в частности в React[26]). По-настоящему предприимчивые люди могут написать собственные плагины преобразования только для использования в своих проектах!

Чтобы установить Babel, вам понадобятся программы Node.js и npm (Node Package Manager). Если они еще не установлены в вашей системе, то выберите один из двух путей:

- перейдите на `https://nodejs.org/` и используйте соответствующий установщик/пакет для вашей системы, чтобы установить инструмент, или
- используйте Node Version Manager, предоставляющий удобный способ установки версий Node и переключения между ними `https://github.com/nvm-sh/nvm`.

Npm поставляется в комплекте с Node.js, так что вам не нужно устанавливать его отдельно.

После того как вы их установили:

1. Создайте каталог для этого примера (например, `example` в вашем домашнем каталоге).
2. Откройте окно командной строки/терминала и перейдите в только что созданный каталог.
3. Используйте `npm`, чтобы создать файла **package.json**: Введите

   ```
   npm init
   ```

   и нажмите клавишу **Enter**. `npm` задаст ряд вопросов; отвечайте на них так, как вам нравится (или просто нажмите клавишу **Enter** в ответ на все из них). После выполнения инструмент запишет файл **package.json** в ваш пример каталога.
4. Далее установите Babel. (Следующие шаги начинаются с перехода на `https://https://babeljs.io/setup`. Затем нажмите кнопку CLI; возможно, вы захотите проверить наличие обновлений.) Введите

   ```
   npm install —save-dev @babel/core @babel/cli
   ```

   и нажмите клавишу **Enter**. `npm` загрузит и установит Babel, его интерфейс командной строки и все его зависимости в ваш пример проекта. (Вы можете получить предупреждение, связанное с модулем под названием `fsevents`, и/или некоторые предупреждения об устаревании; это нормально.)
5. На этом этапе вы можете начать использовать Babel, вызвав его напрямую. Но давайте упростим задачу, добавив запись скрипта `npm` в `package.json`. Откройте файл **package.json** в своем любимом редакторе. Если у вас нет записей `scripts` верхнего уровня, создайте одну (но текущие версии `npm` будут включать в себя скрипт `test`, показывающий сообщение об ошибках). Добавьте эту настройку внутрь записи `scripts`:

   ```
   "build": "babel src -d lib"
   ```

[25] `https://facebook.github.io/jsx/`

[26] `https://reactjs.org/`

Теперь ваш файл **package.json** должен быть похож на Листинг 1–1. (В вашем файле все еще может находиться запись `test` в `scripts`; это нормально. У него также может быть другая лицензия; я всегда меняю значение по умолчанию на MIT.) Обязательно сохраните файл.

Листинг 1-1: Пример package.json — package.json

```
{
  "name": "example",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "build": "babel src -d lib"
},
  "author": "",
  "license": "MIT",
  "devDependencies": {
    "@babel/cli": "^7.2.3",
    "@babel/core": "^7.2.2"
}
}
```

6. Babel отличается высокой модульностью. Хотя он был установлен, мы еще не поставили ему задач. В этом примере мы будем использовать один из его пресетов, чтобы указать ему преобразовать код ES2015 в код ES5, установив и затем настроив пресет. Чтобы установить пресет, введите

   ```
   npm install —save-dev babel-preset-env
   ```

   и нажмите клавишу **Enter**. На следующем шаге мы его настроим.
7. Теперь нам нужно создать конфигурационный файл для Babel,.babelrc (обратите внимание на ведущую точку). Создайте файл с этим содержимым (или используйте файл из раздела загрузки для этой главы):

   ```
   {
     "presets": [
       [
         "env",
         {
           "targets": {
             "ie": "11"
           }
         }
       ]
     ]
   }
   ```

Эта конфигурация предписывает Babel использовать его пресет `env`, который документация Babel описывает как «…интеллектуальный пресет, позволяющий вам использовать последнюю версию JavaScript без необходимости микроуправления тем, какие синтаксические преобразования… необходимы вашей целевой среде(ам)». В этой

конфигурации установка целевого параметра `"ie": "11"` сообщает пресету `env`, что вы ориентируетесь на IE11, что подходит для следующего примера. Для реального использования вам следует ознакомиться с документацией пресета `env`[27] и/или других пресетов или плагинов, которые вы, возможно, захотите использовать вместо него.

Вот и все для настройки Babel в этом примере. Теперь давайте создадим фрагмент кода для переноса. Создайте подкаталог вашего каталога `example` с именем `src` и в нем файл с именем `index.js` с содержанием Листинга 1-2. (В конце процесса я покажу вам список с расположением файлов, так что не волнуйтесь, если вы немного растеряны. Просто создайте файл; его можно переместить, если он окажется в неправильном месте.)

Листинг 1-2: Пример транспилирования входных данных ES2015 — index.js

```
var obj = {
    rex: /\d/,
    checkArray: function(array) {
        return array.some(entry => this.rex.test(entry));
    }
};
console.log(obj.checkArray(["no", "digits", "in", "this", "array"])); // ложь
console.log(obj.checkArray(["this", "array", "has", "1", "digit"])); // истина
```

Код в Листинге 1-2 использует только одну функцию ES2015+ — стрелочную, `entry => this.rex.test(entry)` в вызове `some`, который я выделил в коде. (Да, это действительно функция.) Вы узнаете о стрелочных функциях в главе 3. Краткая версия заключается в том, что они предлагают краткий способ определения функции (как вы можете видеть) и закрываются над `this` точно так же, как закрываются над переменной (вместо того, чтобы получать `this` во время вызова). Когда вызывается `obj.checkArray(...)`, `this` в вызове ссылается на `obj` *даже в пределах обратного вызова* `some`, следовательно `this.rex` ссылается на свойство `rex` из `obj`. Это было бы неверно, если бы обратный вызов был привычной функцией.

На этом этапе ваш пример каталога должен содержать следующее:

```
example/
+-- node_modules/
|   +-- (various directories and files)
+-- src/
|   +-- index.js
+--.babelrc
+-- package.json
+-- package-lock.json
```

Вы готовы к переносу! Введите

```
npm run build
```

и нажмите клавишу **Enter**. Инструмент Babel сделает свое дело, создаст для вас выходной каталог `lib` и напишет версию ES5 для `index.js` к нему. Результат в `lib/index.js` будет выглядеть примерно так, как в Листинге 1-3.

[27] https://babeljs.io/docs/en/babel-preset-env#docsNav

Листинг 1-3: Вывод примера транспилирования ES2015 — index-transpiled-to-es5.js

```
"use strict";

var obj = {
 rex: /\d/,
  checkArray: function checkArray(array) {
    var _this = this;

    return array.some(function (entry) {
      return _this.rex.test(entry);
    });
  }
};
console.log(obj.checkArray(["no", "digits", "in", "this", "array"])); // ложь

console.log(obj.checkArray(["this", "array", "has", "1", "digit"])); // истина
```

Если сравнить `src/index.js` (Листинг 1-2) с `lib/index.js` (Листинг 1-3), вы увидите только пару изменений (кроме пробелов). Сначала Babel добавил директиву `use strict;` в начало переносимого файла (напомним, что строгий режим — это функция, добавленная в ES5, она изменяет поведение нескольких вещей, которые были проблематичными по разным причинам). Это функция Babel по умолчанию, но ее можно отключить, если у вас есть код, опирающийся на свободный режим (loose mode).

Однако интересно то, как он переписал стрелочную функцию. Инструмент создал переменную с именем `_this` внутри массива `checkArray`, присвоил ей значение `this` и затем использовал привычную функцию в качестве обратного вызова `some`. Внутри функции использовалось значение `_this` вместо `this`. Это соответствует моему более раннему описанию стрелочных функций: они закрываются над `this` точно так же, как закрываются над переменной. Babel просто сделал так, чтобы это произошло так, как может понять среда ES5.

Очевидно, что это очень маленький пример, но он иллюстрирует суть и дает вам представление об одном инструменте, который вы могли бы использовать в ваших проектах. Babel может быть интегрирован в вашу систему сборки независимо от того, используете ли вы инструмент Gulp[28], Grunt[29], Webpack[30], Browserify[31], Rollup[32] или какой-то другой. На странице установки `https://babeljs.io/docs/setup/#installation` содержатся инструкции для всех основных инструментов сборки.

ОБЗОР ГЛАВЫ

JavaScript сильно изменился за последние несколько лет, особенно с 2015 года. Он будет продолжать меняться и в будущем. Но не стоит переживать о перегрузке себя всеми новыми функциями. JavaScript всегда будет обратно совместим, поэтому нет необходимости внедрять новую функцию до тех пор, пока вы не будете готовы к этому.

---

[28] `https://gulpjs.com/`
[29] `https://gruntjs.com/`
[30] `https://webpack.js.org/`
[31] `http://browserify.org/`
[32] `https://rollupjs.org/`

Новые функции охватывают широкий спектр от небольших хитростей, таких как разрешение висящей запятой в списке аргументов вызова функции и новых удобных методов в стандартной библиотеке, до серьезных улучшений, таких как декларативный синтаксис класса (глава 4), асинхронные функции `async` (глава 9) и модули (глава 13).

Люди, отвечающие за продвижение JavaScript, являются членами TC39 (Технический комитет 39 Ecma International). Он состоит из разработчиков JavaScript, исследователей языков программирования, авторов библиотек и крупных веб-сайтов, представителей основных движков JavaScript и других заинтересованных сторон в успехе и будущем JavaScript. Но любой может принять в этом участие.

Процесс создания новых функций общедоступен и открыт. Вы можете следить за прогрессом и быть в курсе событий с помощью различных репозиториев GitHub, опубликованных заметок о собраниях и дискуссионной группы TC39. Кроме того, я продолжаю освещать то, на чем заканчивается эта книга, на сайте книги по адресу: `https://thenewtoys.dev`.

Большинство функций, описанных в этой книге, поддерживаются движками JavaScript в современных браузерах, таких как Chrome, Firefox, Safari и Edge, а также недавними выпусками движков в небраузерных средах, таких как Node.js, Electron, React Native, и т. д.

Старые среды, такие как Internet Explorer, могут поддерживаться с помощью компиляции JavaScript в JavaScript (она же «перенос»), преобразования кода старого стиля в код нового стиля с помощью инструментов (таких как Babel), которые могут быть интегрированы в вашу систему сборки. Некоторые функции (например, объекты Proxy, о которых вы узнаете в главе 14) невозможно поддерживать таким образом, хотя большинство все же можно.

Отлично. Сцена готова…

Переходим к новым возможностям!

2

Объявления блочной области видимости для let и const

СОДЕРЖАНИЕ ГЛАВЫ

- Введение в `let` и `const`
- Определение «блочной области видимости» с примерами
- Затенение и поднятие: Временная мертвая зона
- Использование `const` для переменных, которые не должны изменяться
- Создание глобальных переменных, не существующих в глобальном объекте
- Использование блочной области видимости в циклах

В этой главе вы узнаете, как работают новые объявления `let` и `const` и какие проблемы они решают. На протяжении всей главы вы увидите некоторые проблематичные варианты поведения `var`, и узнаете, как `let` и `const` решают эти проблемы. Вы увидите, как `let` и `const` обеспечивают истинную блочную область видимости и предотвращают путаницу, вызванную повторными объявлениями или использованием переменной перед ее инициализацией. Узнаете, как блочная область видимости позволяет использовать `let`, чтобы избежать традиционной проблемы «замыкания в цикле», и как `const` позволяет создавать *постоянные* «переменные», значения которых не могут изменяться. Изучите, как `let` и `const` помогают избежать создания еще большего количества свойств для и без того перегруженного глобального объекта. Короче говоря, вы узнаете, почему `let` и `const` — новые переменные и почему `var` больше нет места в современном программировании на JavaScript.

ВВЕДЕНИЕ В LET И CONST

Как и директива `var`, `let` объявляет переменные:

```
let x = 2;
x += 40;
console.log(x); // 42
```

Директиву `let` можно использовать там же, где и `var`. Как и в случае с `var`, вы не должны использовать инициализатор с `let`. В таком случае значение переменной по умолчанию будет неопределенным — `undefined`:

```
let a;
console.log(a); // undefined
```

Примерно на этом сходства между директивами `let` и `var` заканчиваются. Как вы узнаете из этой главы, `var` и `let` ведут себя совершенно по-разному. Подробнее об этом позже. Пока давайте рассмотрим `const`.

Директива `const` объявляет константы:

```
const value = Math.random();
console.log(value < 0.5 ? "Heads": "Tails");
```

Константы похожи на переменные, за исключением того, что их значения не могут изменяться. Из-за этого вам действительно нужно указать инициализатор: константы не получают значения по умолчанию. Помимо создания констант вместо переменных и требования инициализации, директива `const` аналогична `let`. Это также гораздо более полезно, чем вы могли подумать. По ходу этой главы все станет на свои места.

ИСТИННАЯ БЛОЧНАЯ ОБЛАСТЬ ВИДИМОСТИ

Директива `var` выходит за пределы блоков. Если вы объявляете переменную внутри блока с помощью `var`, она находится не только в области видимости внутри этого блока, но и за его пределами:

```
function jumpOut() {
    var a = [1, 2, 3];
    for (var i = 0; i < a.length; ++i) {
        var value = a[i];
        console.log(value);
    }
    console.log("Outside loop " + value); // Почему есть возможность
                                          // использовать 'value' здесь?
}
jumpOut();
```

Автор `jumpOut`, вероятно, не имел в виду, что к значению `value` можно будет получить доступ вне цикла. Но это так. (Как и `i`.) Почему же это может стать проблемой? Есть пара причин. Первая заключается в том, что область видимости переменных должна быть как можно более узкой по соображениям удобства обслуживания. Переменные

должны существовать только до тех пор, пока они вам нужны, и не дольше. Вторая — каждый раз, когда очевидное назначение кода и его фактическое действие отличаются, вы ищете ошибки и проблемы обслуживания.

Директивы `let` и `const` решают эту проблему, поддерживая истинную *блочную область видимости*: они существуют только в пределах блока, в котором они объявлены. Перед вами пример `let`:

```
function stayContained() {
    var a = [1, 2, 3];
    for (var i = 0; i < a.length; ++i) {
        let value = a[i];
        console.log(value);
    }
    console.log("Outside loop" + value); // ReferenceError: 'value' не объявлена
}
stayContained();
```

Теперь `value` относится к области видимости блока, в котором она была объявлена. Она не существует в остальной части функции. Она существует только до тех пор, пока это необходимо, и очевидное назначение кода соответствует фактическому действию.

(В функции `stayContained` я не изменял остальные переменные с `var` на `let`. Это было сделано только для того, чтобы подчеркнуть важность изменения объявления переменной `value`. Естественно, вы можете изменить и остальные.)

ПОВТОРНЫЕ ОБЪЯВЛЕНИЯ — ЭТО ОШИБКА

Директива `var` рада дать вам возможность повториться. Вы можете объявлять одну и ту же переменную с помощью `var` столько раз, сколько захотите. Например:

```
function redundantRepetition() {
    var x = "alpha";
    console.log(x);
    // ...здесь много кода...
    var x = "bravo";
    console.log(x);
    // ...здесь много кода...
    return x;
}
redundantRepetition();
```

Этот код совершенно корректен синтаксически. Тот факт, что он объявляет `x` более одного раза, полностью игнорируется движком JavaScript. Код создает единственную переменную `x`, используемую во всей функции. Однако, как и в случае с `var` в блоке ранее, очевидное назначение кода и его фактическое действие противоречат друг другу. Повторное объявление уже объявленной переменной может быть ошибкой. В этом случае вполне вероятно, что первый автор функции `redundantRepetition` не написал фрагмент в середине, и код должен вернуть `"alpha"`. Но после этого пришел кто-то еще и добавил фрагмент кода в середину, не понимая, что переменная `x` уже используется.

Хороший тон в программировании, как и во многих других сферах, подразумевает сокращение функций, и/или инструменты lint (линтер), и/или хорошую среду IDE.

Но теперь и сам JavaScript в директивах `let` и `const` делает повторяющиеся объявления в одной и той же области ошибкой:

```
function redundantRepetition() {
    let x = "alpha";
    console.log(x);
    // ...здесь много кода...
    let x = "bravo"; // SyntaxError: Идентификатор 'x' уже объявлен
    console.log(x);
    // ...здесь много кода...
    return x;
}
redundantRepetition();
```

Это лучший вид ошибки — упреждающая. Ошибка возникает при анализе кода. Она не ждет вызова функции `redundantRepetition`, прежде чем сообщить вам о проблеме.

ПОДНЯТИЕ И ВРЕМЕННАЯ МЕРТВАЯ ЗОНА

Объявления `var` отлично *поднимаются*. При использовании `var` вы можете использовать переменную до ее объявления:

```
function example() {
    console.log(answer);
    answer = 42;
    console.log(answer);
    var answer = 67;
}
example();
```

Когда вы запускаете `example`, он выводит следующее:

```
undefined
42
```

Мы с радостью использовали переменную до того, как она была объявлена, но объявление `var`, по-видимому, было перемещено в начало функции. И перемещено только объявление, а не прикрепленный к нему инициализатор (фрагмент `= 67` из объявления `var answer = 67`).

Это происходит потому, что при вводе функции `example` движок JavaScript просматривает функцию, обрабатывающую объявления `var` и создающую необходимые переменные, прежде чем начать пошаговое выполнение любого кода. Он «поднимает» объявления в начало функции. Когда это происходит, движок инициализирует объявленные переменные со значением по умолчанию `undefined`. Но, опять же, очевидное назначение кода и его фактическое выполнение не синхронизированы. Это означает высокую вероятность ошибки. Похоже, что первая строка пытается присвоить значение переменной `answer`, которая находится в содержащей области видимости (возможно, даже глобальной), но вместо этого она использует локальную. И похоже, что автор предполагал, что при создании переменной `answer` она получит значение `67`.

При использовании `let` и `const` вы не можете использовать переменную до тех пор, пока ее объявление не будет обработано в ходе пошагового выполнения кода:

```
function boringOldLinearTime() {
    answer = 42;            // ReferenceError: 'answer' не определено
    console.log(answer);
    let answer;
}
boringOldLinearTime();
```

По-видимому, объявление `let` не поднимается в верхнюю часть функции, как `var`. Но это распространенное заблуждение: `let` и `const` тоже поднимаются. Он просто поднимаются *по-другому*.

Рассмотрите ранее сделанное замечание о том, что код, возможно, пытался присвоить значение переменной `answer` в содержащей области видимости. Давайте рассмотрим этот сценарий:

```
let answer;               // Внешняя переменная 'answer'
function hoisting() {
    answer = 42;          // ReferenceError: 'answer' не определено
    console.log(answer);
    let answer;           // Внутренняя переменная 'answer'
}
hoisting();
```

Если внутренняя переменная `answer` не существует до инструкции `let answer;` в конце, следовательно, в начале функции строка `answer = 42;` должна присваиваться внешней переменной `answer`?

Да, можно было спроектировать код так. Но насколько это сбивало бы с толку? Использование идентификатора для чего-то одного в начале области, но для чего-то другого позже в области запрашивает ошибки.

Вместо этого `let` и `const` используют концепцию, называемую *Временной мертвой зоной* (TDZ, Temporal Dead Zone), период времени в рамках выполнения кода, когда идентификатор вообще нельзя использовать, даже для ссылки на что-то в содержащей области. Так же, как при работе с `var`, движок JavaScript просматривает код в поисках объявления `let` и `const` и обрабатывает их, прежде чем начать пошаговое выполнение кода. Но вместо того, чтобы разрешить доступ к переменной `answer` и присвоить ей значение `undefined`, движок ставит метку «не инициализировано» для `answer`:

```
let answer;                 // Внешняя переменная 'answer'
function notInitializedYet() {
                            // Зарезервируйте 'answer' здесь
    answer = 42;            // ReferenceError: 'answer' не определено
    console.log(answer);
    let answer;             // Внутренняя переменная 'answer'
}
notInitializedYet();
```

Период TDZ начинается, когда выполнение кода входит в область, в которой появляется объявление, и продолжается до тех пор, пока объявление не будет выполнено (вместе

с любым прикрепленным к нему инициализатором). В этом примере внутренняя переменная `answer` зарезервирована в начале функции `notInitializedYet` (там начинается период TDZ) и инициализируется там, где находится объявление (там заканчивается TDZ). Таким образом `let` и `const` тоже поднимаются, но это происходит иначе, чем поднятие директивы `var`.

Важно понимать, что период TDZ является *временным* (связанным со временем), а не *пространственным* (связанным с пространством/местоположением). Это не раздел в верхней части области видимости, где идентификатор невозможно использовать. Это *период времени*, в течение которого идентификатор невозможно использовать. Запустите код из Листинга 2-1.

Листинг 2-1: Пример временной природы TDZ — tdz-is-temporal.js

```
function temporalExample() {
    const f = () => {
        console.log(value);
    };
    let value = 42;
    f();
}
temporalExample();
```

Если бы период TDZ был *пространственным*, если бы это был блок кода в верхней части функции `temporalExample`, где `value` нельзя использовать, этот код не работал бы. Но TDZ — *временной* период, и к тому времени, когда `f` использует `value`, объявление уже выполнено, так что проблем нет. Если поменять местами две последние строки этой функции, перемещая строку `f();` выше строки `let value = 42;`, то она потерпит неудачу, поскольку функция `f` попытается использовать значение `value`, прежде чем эта переменная была инициализирована. (Попробуйте!)

TDZ применяется к блокам в той же степени, что и к функциям:

```
function blockExample(str) {
    let p = "prefix";                // Объявление внешней переменной 'p'
    if (str) {
        p = p.toUpperCase();          // ReferenceError: 'p' не определено
        str = str.toUpperCase();
        let p = str.indexOf("X");     // Объявление внутренней переменной 'p'
        if (p != -1) {
            str = str.substring(0, p);
        }
    }
    return p + str;
}
```

Переменную `p` невозможно использовать в этой первой строке внутри блока, потому что, несмотря на объявление ее в функции, внутри блока есть объявление затенения, и оно становится владельцем идентификатора `p`. Таким образом, идентификатор может ссылаться только на новую внутреннюю переменную `p` и только после выполнения объявления `let`. Это предотвращает путаницу в отношении того, какая переменная `p` используется в коде.

НОВЫЙ ВИД ГЛОБАЛЬНЫХ ПЕРЕМЕННЫХ

При использовании `var` в глобальной области видимости создается глобальная переменная. В ES5 и более ранних версиях все глобальные переменные были также и свойствами глобального объекта. А что же изменилось с появлением ES2015? Теперь в JavaScript есть обычные глобальные переменные, созданные с помощью `var` (которые также представляют собой свойства глобального объекта), и глобальные переменные нового образца (которые не будут свойствами глобального объекта). Директивы `let` и `const` в глобальной области видимости создают эти новые глобальные переменные.

ДОСТУП К ГЛОБАЛЬНОМУ ОБЪЕКТУ

Запомните, что существует один глобальный объект. Получить к нему доступ можно с помощью this в глобальной области видимости или с помощью глобальной переменной, которую среда определяет для него (если таковой имеется), — например window или self в браузерах или global в Node.js. (В некоторых средах, таких как браузеры, это не глобальный объект, а фасад глобального объекта, но это достаточно близко.)

Вот пример использования директивы `var` для создания глобального объекта, который также является свойством глобального объекта. Обратите внимание, что вы должны выполнить это в глобальной области видимости (если вы используете Node.js или jsFiddle.net, убедитесь, что вы работаете в глобальной области, а не в области модуля или функции, как описано в главе 1).

```
var answer = 42;
console.log("answer == " + answer);
console.log("this.answer == " + this.answer);
console.log("has property? " + ("answer" in this));
```

Когда вы запустите это, вы увидите:

```
answer == 42
this.answer == 42
has property? true
```

Теперь попробуйте сделать это с помощью `let` (в новом окне):

```
let answer = 42;
console.log("answer == " + answer);
console.log("this.answer == " + this.answer);
console.log("has property? " + ("answer" in this));
```

На этот раз вы получите:

```
answer == 42
this.answer == undefined
has property ? false
```

Обратите внимание, что `answer` — это больше не свойство глобального объекта.

То же самое касается директивы `const`: она создает глобальную переменную, но не является свойством глобального объекта.

Эти глобальные переменные по-прежнему доступны в любом месте, даже если не представляют собой свойства глобального объекта, так что это не значит, что вы должны перестать избегать глобальных переменных. Учитывая сказанное, можете ли вы придумать, почему было бы полезно, чтобы переменные не были свойствами глобального объекта?

Есть несколько причин:

- Глобальный объект уже сильно перегружен свойствами в наиболее распространенной среде: веб-браузере. Не только объявленные с помощью `var` глобальные переменные попадают в объект. Он также получает свойства для всех элементов с полем `id`, большинство элементов с полем `name` и многие другие «автоматически глобальные переменные». Он просто переполнен. Хорошо было бы остановить прилив.
- Это затрудняет их обнаружение из другого кода. Чтобы использовать глобальные переменные `let` или `const`, вам необходимо знать их имена: невозможно обнаружить переменную, просмотрев имена свойств в глобальном объекте. Это не так уж полезно — если вы хотите конфиденциальности, не создавайте глобальные переменные, — но это лишь немного уменьшит утечки информации.
- Это может позволить движку JavaScript оптимизировать доступ к ним (в частности, к `const`) с помощью способов, которые нельзя применить к свойствам глобального объекта.

К теме автоматически глобальных переменных. Объявленная с помощью `let` или `const` (или `class`) глобальная переменная затеняет автоматически глобальную переменную (то есть она скрывает ее; объявление `let` или `const` «выигрывает»). А глобальная переменная, объявленная с помощью `var`, не поддерживает такой возможности. Классический пример — попытка использовать глобальную переменную с названием `name` в веб-браузере. В браузере глобальный объект — это объект Window для страницы; он содержит свойство с именем `name`, которое не может быть затенено с помощью глобальной переменной, объявленной с помощью `var`. Значение этого свойства всегда является строкой:

```
// В браузере в глобальной области
var name = 42;
console.log(typeof name); // строка
```

Хотя объявленная с помощью `let` или `const` глобальная переменная успешно затенит это значение, или любую другую автоматически глобальную переменную/свойство `window`:

```
// В браузере в глобальной области
let name = 42;
console.log(typeof name); // число
```

Однако отделение `let` и `const` от глобального объекта — лишь неотъемлемая часть направления языка, поскольку он уходит от использования глобального объекта. Об этом вы узнаете больше в последующих главах, особенно в главе 13 «Модули».

CONST: КОНСТАНТЫ В JAVASCRIPT

Мы уже коснулись некоторых общих для `let` и `const` моментов. В этом разделе мы погрузимся в понятие `const` намного глубже.

Основы `const`

Как вам известно, директива `const` создает константы:

```
const answer = 42;
console.log(answer); // 42
```

Этот процесс такой же, как создание переменной с помощью `let`, со всеми теми же правилами области видимости, Временной мертвой зоной и т. д. Разница лишь в том, что вы не можете присвоить новое значение константе, а переменной — можете.

Как вы думаете, что происходит при попытке присвоить константе новое значение? Что было бы самым полезным?

Правильно — получение сообщения об ошибке:

```
const answer = 42;
console.log(answer); // 42
answer = 67;         // TypeError: недопустимое присвоение константе 'answer'
```

> **ПРИМЕЧАНИЕ**
> Текст сообщения об ошибке варьируется от реализации к реализации, но это будет ошибка типа TypeError. На момент написания книги в одной реализации забавно и оксюморонно написано «TypeError: Присвоение переменной константе».

На первый взгляд может показаться, что вам предстоит использовать `const` только для таких вещей, как избежание использования «магических чисел» в коде. Например, стандартной задержки, которую можно использовать в коде перед отображением сообщения о занятости системы, когда вы делаете что-то, инициированное пользователем:

```
const BUSY_DISPLAY_DELAY = 300; // миллисекунды
```

Но тип `const` полезен не только для этого. Новые значения переменным присваиваются так же часто, как простое использование их для хранения неизменной информации.

Взгляните на реальный пример: в Листинге 2-2 приведен простой цикл для добавления текста к разделам в зависимости от того, содержат ли они определенный класс. Кажется, что он довольно просто использует переменные.

Листинг 2-2: Версия ES5 цикла обновления div — element-loop-es5.js

```
var list, n, element, text;
list = document.querySelectorAll("div.foo");
for (n = 0; n < list.length; ++n) {
    element = list[n];
    text = element.classList.contains("bar")? " [bar]" : "[not bar]";
    element.appendChild(document.createTextNode(text));
}
```

В этом коде есть четыре переменные. Если вы присмотритесь внимательнее, то увидите, что одна из них не похожа на остальные три. Можете ли вы понять, в чем разница?

Одна из них содержит значение, которое никогда не меняется, — `list`. Остальные (`n`, `element` и `text`) действительно относятся к переменным, но `list` — это константа.

Мы к этому еще вернемся в одном из последующих разделов, охватив несколько аспектов `const`. А пока мысленно воссоздайте применение `let` и `const` к этому коду (не удаляя идентификаторы), основываясь на том, что вы уже узнали.

Объекты, на которые ссылается const, по-прежнему изменяемы

Важно помнить, что именно нельзя изменить с помощью константы — значение *константы*. Если это значение представляет собой ссылку на объект, это не означает, что объект *неизменяемый*. Это просто означает, что вы не можете сделать так, чтобы константа указывала на *другой* объект (потому что это изменило бы значение константы). Давайте рассмотрим такой вариант:

```
const obj = {
    value: "before"
};
console.log(obj.value);          // "before"
```

Пока что у вас есть ссылка на объект в константе.

В памяти хранится что-то похожее на рисунок 2-1.

Константа `obj` не содержит объект непосредственно, она содержит ссылку на объект (показано на рисунке 2-1 как «Ref55462», но, конечно, это просто концептуальное отображение: вы никогда не увидите реальное значение ссылок на объекты). Следовательно, если вы измените состояние объекта:

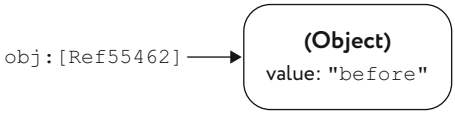


РИСУНОК 2-1

```
obj.value = "after";
console.log(obj.value);          // "after"
```

вы не меняете значение константы `obj`. В ней по-прежнему хранится ссылка на тот же объект — `obj: [Ref55462]`. Просто состояние объекта было обновлено, поэтому он сохраняет другое значение для своего свойства `value`: см. рисунок 2-2.

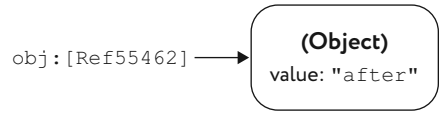


РИСУНОК 2-2

Директива `const` выполняет свою задачу, не позволяя вам изменять фактическое значение `obj`, указывать ссылку на другой объект, присваивать значение `null` или что-то совершенно иное:

```
obj = {}; // TypeError: недопустимое присвоение константе 'obj'
```

Вот практический пример — функция, добавляющая абзац с заданным HTML-кодом к родительскому элементу:

```
function addParagraph(parent, html) {
    const p = document.createElement("p");
    p.innerHTML = html;
    parent.appendChild(p);
    return p;
}
```

Так как код только изменяет состояние абзаца (путем установки свойства `innerHTML`), а не то, на что ссылается `p`, вы можете объявить `p` константой.

БЛОЧНАЯ ОБЛАСТЬ ВИДИМОСТИ В ЦИКЛАХ

Ранее в этой главе вы видели блочную область видимости. На первый взгляд это довольно просто — переменная, объявленная с помощью `let` или `const` внутри блока, доступна только внутри блока:

```
function anotherBlockExample(str) {
    if (str) {
        let index = str.indexOf("X"); // 'index' существует только внутри блока
        if (index != -1) {
            str = str.substring(0, index);
        }
    }
    // Здесь невозможно использовать 'index', он вне области
    return str;
}
anotherBlockExample();
```

Но что, если блок присоединен к циклу? Все ли итерации цикла используют одну и ту же переменную? Или для каждой итерации создаются отдельные переменные? Как ведут себя созданные внутри цикла замыкания?

Люди, разрабатывающие блочную область видимости для JavaScript, сделали умную вещь. Каждая итерация цикла получает свои собственные переменные блока — почти так же, как разные вызовы функции получают свои собственные локальные переменные.

Это означает, что вы можете использовать блочную область видимости для решения классической проблемы «замыканий в циклах».

Проблема «замыканий в циклах»

Вы наверняка знакомы с проблемой «замыкания в циклах», хотя, возможно, и не под этим названием. Запустите код в Листинге 2-3, чтобы увидеть проблему в действии.

Листинг 2-3: Проблема замыканий в циклах — closures-in-loops-problem.js

```
function closuresInLoopsProblem() {
    for (var counter = 1; counter <= 3; ++counter) {
        setTimeout(function() {
            console.log(counter);
        }, 10);
    }
}
closuresInLoopsProblem();
```

(Игнорируйте функции типа `setTimeout`, которые могут помочь нам в решении этой проблемы, — это просто плейсхолдер для любой асинхронной операции.)

Вы могли подумать, что этот код выведет 1, затем 2, затем 3, но вместо этого он выводит 4, 4, 4. Причина в том, что каждый таймер не запускает свой обратный вызов до тех пор, пока цикл не завершится. К тому моменту, когда посылаются обратные вызовы, значение счетчика `counter` равно 4, а поскольку он объявлен при помощи `var`, переменная `counter` определяется функцией `closuresInLoopsProblem`. Все три обратных вызова таймера закрываются по одной и той же переменной `counter`, поэтому все они видят значение 4.

В ES5 и ранее обычным способом решить эту проблему был ввод другой функции и передачу в нее переменной `counter` в качестве аргумента. Затем необходимо было использовать этот аргумент вместо переменной `counter` в файле **console.log**. Программисты часто делают это с помощью встроенной анонимной функции, которую они немедленно вызывают, например, в Листинге 2-4.

Листинг 2-4: Замыкания в циклах, решение в стандарте ES5 — closures-in-loops-es5.js

```
function closuresInLoopsES5() {
    for (var counter = 1; counter <= 3; ++counter) {
        (function(value) {      // Начало анонимной функции
            setTimeout(function() {
                console.log(value);
            }, 10);
        })(counter);             // Ее окончание и вызов
    }
}
closuresInLoopsES5();
```

Когда вы запустите этот код, он выводит ожидаемый результат — 1, 2, 3, потому что функции таймеры используют `value`, а не `counter`. Каждый вызов оборачивающей анонимной функции получает собственный параметр `value` для функции таймера,

чтобы она могла закрыться. Ничто не изменяет ни один из этих параметров `value`, поэтому обратные вызовы регистрируют ожидаемые значения (1, 2 и 3).

Однако благодаря директиве `let` из ES2015 вы можете решить эту задачу гораздо проще: просто замените `var` на `let`. Запустите код из Листинга 2-5.

Листинг 2-5: Замыкания в циклах, решение при помощи let — closures-in-loops-with-let.js

```
function closuresInLoopsWithLet() {
    for (let counter = 1; counter <= 3; ++counter) {
        setTimeout(function() {
            console.log(counter);
        }, 10);
    }
}
closuresInLoopsWithLet();
```

Запуск этого кода также дает вам ожидаемый результат — 1, 2, 3. Одно маленькое изменение, одно масштабное воздействие. Но как это работает? Конечно, функции все еще закрываются через `counter`. Как все они могут получать разные значения?

Точно так же, как вызовы анонимной функции создавали несколько параметров `value` для завершения функций таймера, цикл, приведенный выше в Листинге 2-5, создает несколько переменных `counter` для завершения созданных в цикле функций таймера, по одной для каждой итерации цикла. Таким образом, каждая итерация получает свою собственную переменную `counter`.

Чтобы понять, как это сделать, нам нужно более внимательно изучить, как переменные (и константы) работают в JavaScript. Это также поможет вам в некоторых последующих главах.

Привязки: Как работают переменные, константы и другие идентификаторы

Ранее в этой главе вы узнали, что константы `const` работают точно так же, как переменные `let` с точки зрения области видимости, поддерживаемых типов значений и т. д. Для этого есть веская причина. По сути, переменные и константы — это одно и то же, и спецификация называет это привязками (в данном случае *привязками идентификаторов, биндингами*): связь между идентификатором и хранилищем его значения. Когда вы создаете переменную, например, с помощью

```
let x = 42;
```

вы создаете привязку для идентификатора с именем `x` и сохраняете значение 42 в слоте хранения этой привязки. В этом случае это *изменяемая* привязка (привязка, значение которой может изменяться). Когда вы создаете константу, вы создаете *неизменяемую* привязку (привязку, значение которой не может измениться).

У привязки идентификаторов есть имена и значения. Они немного похожи на свойства объекта, правда? Как и свойства объекта, они находятся в контейнере, который

я собираюсь назвать *объектом среды*[33]. Например, объект среды для контекста, в котором выполняется этот код:

```
let a = 1;
   const b = 2;
```

будет выглядеть примерно, как показано на рисунке 2-3.

Для обработки вложенных областей объекты среды связаны друг с другом в цепочку: каждый из них содержит ссылку на тот, который находится «вне» его.

Если коду нужен идентификатор, отсутствующий в текущем объекте среды, он следует по ссылке на объект внешней среды, чтобы найти его (повторяя запрос по мере необходимости, через глобальную среду — именно так работают глобальные переменные).

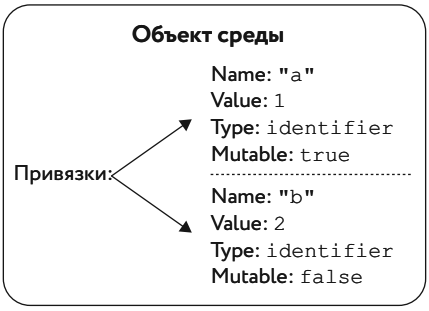


РИСУНОК 2-3

Внешняя среда задается различными способами. Например, когда выполнение кода вводит блок с идентификаторами в блочной области, объект среды для блока получает объект среды для кода, содержащего блок, в качестве его внешней среды. Когда вызывается функция, объект среды для вызова получает среду, в которой была создана функция (она сохраняется в функции; спецификация называет ее внутренним слотом среды [[Environment]] функции) в качестве ее внешней среды. Вот как работают замыкания.

Например, рассмотрим этот код (предположим, что он находится в глобальной области видимости):

```
let x = 1;
function example() {
    const y = 2;
    return function() {
        let z = 3;
        console.log(z, y, x);
    };
}
const f = example();
   f();
```

В вызове функции `f()`, когда выполнение кода попадает на строку `console.log` цепочка объектов среды выглядит примерно так, как показано на рисунке 2–4 (далее).

Давайте проследим за процессом до конца:

- Движок JavaScript создает глобальную среду (`EnvObject1`) и добавляет к ней связи с `x`, `f` и `example`.

[33] То, что я называю «объектом среды», разделено в спецификации на *Лексическую среду* и содержащуюся в ней *Запись среды*. Разделение здесь не имеет значения.

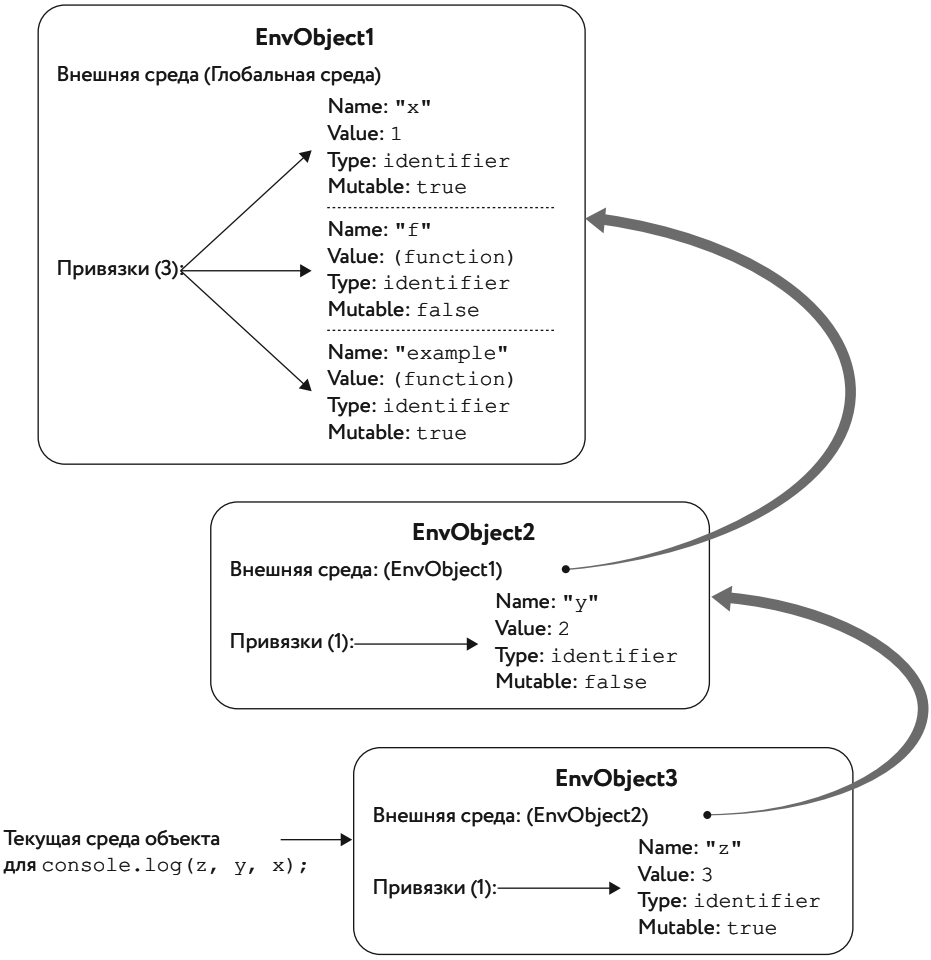


РИСУНОК 2-4

- Он создает функцию `example`, устанавливает в качестве сохраненной `example` среды текущую среду (глобальную `EnvObject1`) и устанавливает значение связей функции `example`.
- Запускается строка `let x = 1;`, присваивая `x` значение 1.
- Запускается строка `const f = example();`:
  - Она создает новый объект среды для вызова (`EnvObject2`), устанавливает свою внешнюю среду в качестве сохраненной среды `example` (`EnvObject1`).
  - Создает связь с именем `y` в объекте среды.
  - Запускается строка `const y = 2;`, присваивая `y` значение 2.
  - Создается функция, устанавливающая текущую среду (это среда `EnvObject2` для вызова функции `example`) в качестве сохраненной среды.
  - Происходит возврат функции из вызова.

    - Функция присваивается переменной `f`.
- Наконец движок запускает функцию `f();` вызовом `f`:
    - Она создает новую среду объекта для вызова (`EnvObject3`), установив в качестве внешней среды сохраненную функцией среду (`EnvObject2`, один из вызовов функции `example` ранее), и создает связанную с ней переменную `z`.
    - Переменной `z` задается значение 3.
    - Выполняется строка `console.log`.

В строке `console.log` движок обнаруживает переменную `z` в текущей среде, но ему приходится переходить во внешнюю среду (ту, что предназначена для вызова в функции `example`), чтобы найти переменную `y`. Для обнаружения `x` движок перемещается на два уровня дальше.

Как все это помогает нам понять `closuresInLoopsWithLet`? Давайте еще раз рассмотрим эту функцию; см. Листинг 2-6.

Листинг 2-6: Замыкания в циклах, решение при помощи let (снова) — closures-in-loops-with-let.js

```
function closuresInLoopsWithLet() {
    for (let counter = 1; counter <= 3; ++counter) {
        setTimeout(function() {
            console.log(counter);
        }, 10);
    }
}
closuresInLoopsWithLet();
```

При выполнении цикла `for` движок JavaScript создает новый объект для каждой итерации цикла. Каждый из них со своей собственной, отдельной переменной `counter`, поэтому каждый обратный вызов таймера закрывается на разных переменных `counter`. Вот что движок JavaScript делает с этим циклом:

1. Он создает объект среды для вызова; давайте назовем его `CallEnvObject`.
2. Он устанавливает ссылку внешней среды `CallEnvObject` на сохраненную в функции `closuresInLoopsWithLet` среду (ту, в которой она была создана, в данном случае глобальная среда).
3. Он начинает с обработки `for`, вспоминая список переменных, объявленных с `let` в части инициализации `for`. В таком случае есть только одна переменная `counter`, хотя вы могли бы получить больше.
4. Движок создает новый объект среды для части инициализации цикла, используя объект `CallEnvObject` в качестве внешней среды, и создает привязку к нему для переменной `counter` со значением 1.
5. Он создает новый объект среды (`LoopEnvObject1`) для первой итерации, используя `CallEnvObject` в качестве объекта внешней среды.
6. Ссылаясь на свой список из шага 3, он создает привязку переменной `counter` к объекту `LoopEnvObject1`, устанавливая его значение равным 1 (значение из объекта среды инициализации).

7. Движок устанавливает `LoopEnvObject1` в качестве текущего объекта среды.
8. Поскольку тест `counter <= 3` возвращает истинный результат, выполняется тело цикла `for` путем создания первой функции таймера (назовем ее `timerFunction1`), присвоив ей ссылку на `LoopEnvObject1` в качестве сохраненного объекта среды.
9. Он вызывает `setTimeout`, передавая в него ссылку на функцию `timerFunction1`.

На данный момент у вас в памяти находится что-то вроде рисунка 2-5.

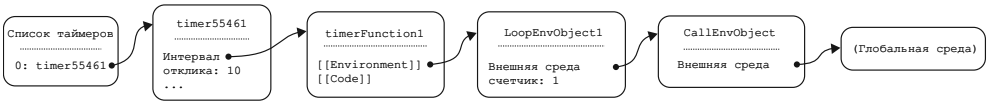


РИСУНОК 2-5

Теперь движок JavaScript готов к следующей итерации цикла:

1. Он создает новый объект среды (`LoopEnvObject2`), используя `CallEnvObject` в качестве внешней среды.
2. Используя свой список привязок из шага 3, он создает привязку для переменной `counter` к объекту `LoopEnvObject2` и устанавливает его значение на текущее значение `counter` объекта `LoopEnvObject1` (в данном случае 1).
3. Движок устанавливает `LoopEnvObject2` в качестве текущего объекта среды.
4. Движок выполняет «инкрементную» часть счетчика цикла `for`: `++counter`. Счетчик, который он увеличивает, является счетчиком в текущем объекте среды, `LoopEnvObject2`. Поскольку его значение равно 1, оно становится равным 2.
5. Движок продолжает цикл `for`: поскольку условие истинно, он выполняет тело цикла, создавая вторую функцию таймера (`timerFunction2`), предоставляя ей ссылку на `LoopEnvObject2`, чтобы она закрывала информацию в функции.
6. Он вызывает `setTimeout`, передавая ссылку на функцию `timerFunction2`.

Как видно, две функции таймера закрываются над разными объектами среды с разными копиями `counter`. В первой значение `counter = 1`, во второй — `counter = 2`. На данный момент у вас в памяти находится что-то вроде рисунка 2–6.

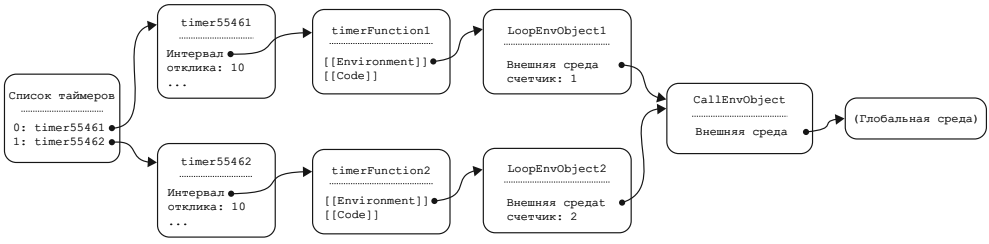


РИСУНОК 2-6

Все это происходит и на третьей и последней итерации цикла. В итоге вы получаете в памяти что-то похожее на рисунок 2-7.

Когда функции таймера вызываются таймерами (поскольку каждый из них использует отдельный объект среды со своей собственной копией переменной `counter`), вы видите 1, 2, 3 вместо 4, 4, 4, получившегося с применением `var`, где все итерации использовали один и тот же объект среды и переменную.

Вкратце, механика блочной области видимости в цикле делала именно то, что делала анонимная функция нашего решения ES5: предоставляла каждой функции таймера отдельный объект среды для закрытия с его собственной копией привязки (`counter` при решении с помощью `let`, `value` при решении по версии ES5). Но движок делал это более эффективно, не требуя отдельной функции и вызова функции.

Иногда, конечно, вам понадобится прежнее поведение. В этом случае просто объявите переменную перед циклом (что, конечно же, и сделал `var`).

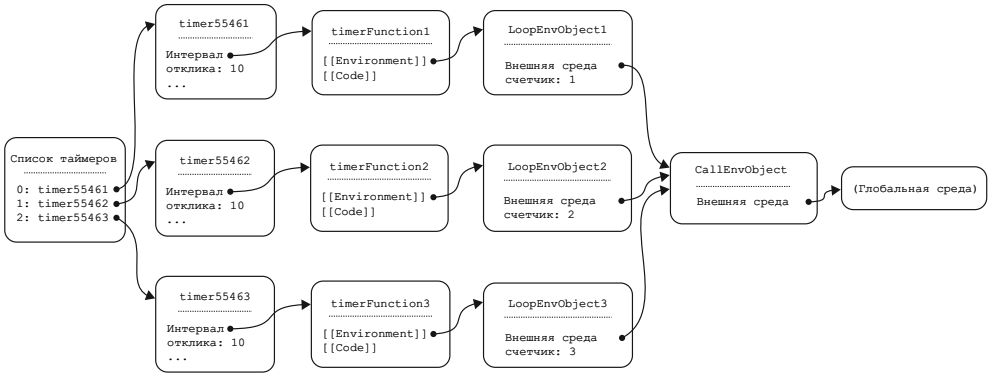


РИСУНОК 2-7

Но стиль поведения, который вы только что изучили, специфичен для циклов `for` в той части, где он отслеживает, какие переменные `let` были созданы во фрагменте инициализации `for` и копируют значение из привязок одного объекта среды к другому. Но тот факт, что блок получает свой собственный объект среды, вовсе не ограничивается циклами `for`. Среди прочего это означает, что для применения циклов `while` и `do-while` также выгодно получение блоками собственных объектов среды. Давайте посмотрим, как это происходит.

Циклы `while` и `do-while`

При применении циклов `while` и `do-while` также выгодно получение блоками собственных объектов среды. Поскольку у них нет выражения инициализации `for`, они не копируют далее значения объявленных там привязок, но блок, связанный с каждой итерацией цикла, все равно получает свою собственную среду. Давайте посмотрим на это в действии. Запустите Листинг 2-7.

Листинг 2-7: Замыкания в циклах `while` — closures-in-while-loops.js

```
function closuresInWhileLoops() {
    let outside = 1;
    while (outside <= 3) {
        let inside = outside;
        setTimeout(function() {
            console.log("inside = " + inside + ", outside = " + outside);
        }, 10);
        ++outside;
    }
}
closuresInWhileLoops();
```

При запуске `closuresInWhileLoops` результат вывода на экран будет следующим:

```
inside = 1, outside = 4
inside = 2, outside = 4
inside = 3, outside = 4
```

Все функции таймера закрываются над одной переменной `outside` (поскольку она была объявлена вне цикла), но каждая из них закрывается над своей собственной переменной `inside` (рисунок 2-8).

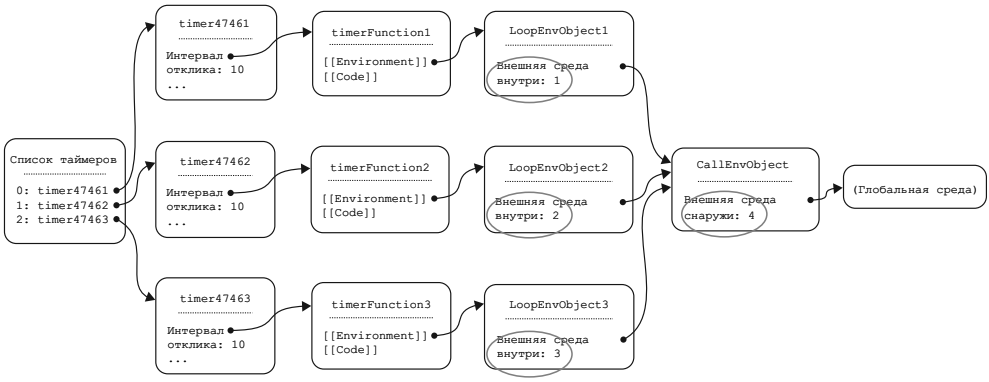


РИСУНОК 2-8

Последствия для производительности

Размышляя о том, как работает блочная область видимости в циклах, вы можете подумать: «Подождите, если я использую блочные переменные в цикле, и он должен создать новый объект среды для их хранения и настроить цепочку выполнения, и (для циклов `for`) скопировать значение привязки итерации от одного к следующему, разве это не замедлит выполнение моих циклов?»

На этот вопрос существуют два ответа:

1. Наверное, вам все равно. Помните, что преждевременная оптимизация — это просто преждевременная оптимизация. Беспокойтесь об этом, если (и когда) у вас возникнет реальная проблема с производительностью, требующая решения.

2. И да и нет. Такой цикл определенно повлечет больше накладных расходов, если движок JavaScript не оптимизировал разницу, и бывают случаи (включая наш пример замыканий в циклах), когда он не может оптимизировать разницу. В других случаях, если мы не создаем замыкания или движок может определить, что замыкания не используют переменные для каждой итерации цикла, он вполне может оптимизировать разницу. Современные движки проводят внушительную оптимизацию. Когда директива `let` из ES2015 была относительно новой, движок V8 Chrome запускал циклы `for` заметно медленнее при использовании директивы `let` для переменных, чем при объявлении с помощью `var`. Эта разница в скорости исчезла (для случаев, когда замыкания не создавались и т. п.), поскольку инженеры V8 нашли способы ее оптимизации.

Если вы столкнулись с реальной проблемой с циклом, и не важно, что у вас есть отдельные копии переменных, просто переместите их во включающую область:

```
let n;
for (n = 0; n < aReallyReallyBigNumber; ++n) {
    // ...
}
```

или, если вы не хотите, чтобы они были в этой области, оберните весь цикл в анонимный блок и объявите переменную в этом блоке:

```
function wrappingInAnonymousBlock() {
    // ...какой-то несвязанный код...

    // Теперь мы выявили проблему с производительностью/памятью для
    // каждой итерации инициализация 'n', поэтому мы используем
    // анонимный блок, чтобы получить только одну 'n'
    {
        let n;
        for (n = 0; n < aReallyReallyBigNumber; ++n) {
            // ...
        }
    }

    // ...еще какой-то несвязанный код...
}
wrappingInAnonymousBlock();
```

Константа в блоках цикла

В предыдущем разделе о `const` был показан простой цикл обновления `div` (повторяется в Листинге 2-8), где одна из переменных (`list`) остается неизменной. И я попросил вас подумать о том, как можно применить `let` и `const` к коду (без удаления каких-либо идентификаторов).

Листинг 2-8: Версия ES5 цикла обновления div — element-loop-es5.js

```
var list, n, element, text;
list = document.querySelectorAll("div.foo");
```

```
for (n = 0; n < list.length; ++n) {
    element = list[n];
    text = element.classList.contains("bar") ? " [bar]": "[not bar]";
    element.appendChild(document.createTextNode(text));
}
```

Вы придумали что-то вроде Листинга 2-9?

Листинг 2-9: Версия ES2015 цикла обновления div — element-loop-es2015.js

```
const list = document.querySelectorAll("div.foo");
for (let n = 0; n < list.length; ++n) {
    const element = list[n];
    const text = element.classList.contains("bar") ? " [bar]": "[not bar]";
    element.appendChild(document.createTextNode(text));
}
```

В то время вам, возможно, и в голову не приходило перемещать `element` и `text` в блок цикла `for`. Даже если бы вы это сделали, то, возможно, оставили бы объявление `let`, а не `const`. Но их значения внутри блока никогда не меняются, и, конечно, каждый блок получает свою собственную их копию, поэтому они являются константами в пределах его области видимости. Можно объявить их как `const` и оставить значения без изменений.

Но не стоит объявлять что-то с помощью `const` только потому, что вы не меняете это значение. Что бы вы ни делали, это вопрос стиля. Такие вещи необходимо обсуждать со своей командой и приходить к соглашению — использовать `const`, не использовать или оставить это на усмотрение разработчика. Есть прагматические преимущества в использовании констант во время разработки (на раннем этапе вы получаете ошибку при попытке изменить что-то, что товарищ по команде объявил как `const`; затем вы либо целенаправленно меняете объявление, либо не меняете значение этой константы), но ваша команда может использовать полные наборы тестов, которые также найдут ошибку (хотя и несколько позже).

Константа в циклах `for-in`

В дополнение к обычным циклам `for` из четырех частей (инициализация, тестирование, инкремент, тело цикла) JavaScript предлагает другие типы циклов `for`: `for-in`, `for-of` и `for-await-of`. (Цикл `for-of` вы изучите в главе 6, а цикл `for-await-of` — в главе 9.) Давайте просмотрим типичный цикл `for-in`:

```
var obj = {a: 1, b: 2};
for (var key in obj) {
    console.log(key + ": " + obj[key]);
}
```

Если вы собирались переписать это в ES2015+, вы бы использовали директиву `let` или `const` для переменной `key`?

Можно использовать любую из этих директив. Циклы `for-in` с лексическими объявлениями получают отдельный объект среды для каждой итерации цикла точно

так же, как это делает цикл `while`. Поскольку код в теле цикла не изменяет значение переменной `key`, ее можно при желании объявить с помощью `const`:

```
const obj = {a : 1, b : 2};
for (const key in obj) {
    console.log(key + ": " + obj[key]);
}
```

ОТ СТАРЫХ ПРИВЫЧЕК К НОВЫМ

Перед вами несколько старых привычек, которые, возможно, вам захочется заменить на новые.

Используйте `const` или `let` вместо `var`

Старая привычка: Использовать `var`.

Новая привычка: Используйте `const` или `let`. Единственными оставшимися вариантами использования для `var` станут устаревшие коды, например, содержащие верхний уровень

```
var MyApp = MyApp || {};
```

в нескольких скриптах, которые могут быть загружены на страницу. Все эти скрипты записываются в разные части объекта `MyApp`, и выполнение этого кода заменяется модулями (см. главу 13).

Если использовать `const` для «переменных», которые вы не собираетесь изменять, вы в скором времени обнаружите, что используете директиву `const` гораздо чаще, чем изначально можно было подумать. Тем более при работе со ссылками на объекты это не означает, что исчезает возможность изменить состояние объекта; это просто означает, что вы не можете изменить объект, на который ссылается константа. Если вы пишете объектно-ориентированный код (что почти неизбежно при использовании JavaScript), удивительно, как мало реальных *переменных* у вас получается.

Сохраняйте узкую область видимости переменных

Старая привычка: Перечислять переменные типа `var` в верхней части функции, поскольку именно туда в любом случае будет поднято объявление `var`.

Новая привычка: Используйте `let` и `const` в максимально узкой подходящей области, где это разумно. Это повышает удобство обслуживания кода.

Используйте блочную область видимости вместо встроенных анонимных функций

Старая привычка: Использовать встроенные анонимные функций для решения проблемы замыкания с помощью циклов:

```
for (var n = 0; n < 3; ++n) {
    (function(value) {
```

```
        setTimeout(function() {
            console.log(value);
        }, 10);
    })(n);
}
```

Новая привычка: Используйте блочную область видимости:

```
for (let n = 0; n < 3; ++n) {
    setTimeout(function() {
        console.log(n);
    }, 10);
}
```

Гораздо чище и легче читается.

Тем не менее не стоит заменять отлично именованные, повторно используемые функции блочной областью видимости без веской причины:

```
// Если это уже реализовано так, с многоразовой именованной функцией, нет
//необходимости перемещать код функции в цикл
function delayedLog(msg, delay) {
    setTimeout(function() {
        console.log(msg);
    }, delay);
}
// ...далее...
for (let n = 0; n < 3; ++n) {
    delayedLog(n, 10);
}
```

Блочная область видимости не заменяет функции правильного размера, но это полезный инструмент для устранения запутанных встроенных анонимных функций.

Если вы используете функции ограничения области видимости в глобальной области, чтобы избежать создания глобальных переменных, вы даже можете заменить их блоком (или модулем, как будет показано в главе 13). Блоки не обязательно должны быть присоединены к оператору управления потоком, такому как `if` или `for` — они могут быть автономными. Таким образом, ваша функция ограничения области может быть просто блоком определения области:

```
{                          // Блок определения области видимости
    let answer = 42;       // Переменная 'answer' является локальной для
                           // блока, а не глобальной
    console.log(answer);
}
```

3

Функции

СОДЕРЖАНИЕ ГЛАВЫ

- Стрелочные функции и лексические `this`, `super` и т. д.
- Значения параметров по умолчанию
- Остаточные параметры
- Висящие запятые в списках параметров и вызовах функций
- Свойство имени функции
- Объявления функций внутри блоков

В этой главе вы узнаете о многих интересных новых возможностях функций в современном JavaScript, таких как стрелочные функции — облегченная, лаконичная новая форма функций, которая решает целый класс проблем с обратными вызовами. Значения параметров по умолчанию для упрощения кода функции и улучшения поддержки инструментов. Параметры rest, обеспечивающие истинные массивы «остаточных» аргументов функции после именованных. Новое официальное свойство name для функций и множество различных способов его установки. А также как работают объявления функций в блоках управления потоком.

Есть три связанных с функциями вопроса, которые мы отложим до следующих глав:

- *Деструктурированные параметры описаны в главе 7.*
- *Функции объекта генератора описаны в главе 6.*
- *Асинхронные функции рассматриваются в главе 9.*

ПАРАМЕТР В СРАВНЕНИИ С АРГУМЕНТОМ

В программировании у нас есть два тесно связанных термина, которые часто используются взаимозаменяемо, — parameter и argument. Функции объявляют и используют **параметры**. Вы вызываете функцию с определенными значениями для этих параметров, они называются **аргументами**. Давайте приведем краткий пример:

```
function foo(bar) {
console.log(bar * 6);
}
foo(7);
```

Функция foo объявляет и использует параметр под названием bar. Код вызывает функцию foo с аргументом 7.

СТРЕЛОЧНЫЕ ФУНКЦИИ И ЛЕКСИЧЕСКИЕ `THIS`, `SUPER` И Т. Д.

Версия ES2015 добавила *стрелочные функции* в JavaScript. Стрелочные функции решают целый ряд проблем, особенно связанных с обратными вызовами: подтверждают, что `this` внутри функции совпадает с `this` вне ее. Они также более легкие и лаконичные, чем традиционные функции, созданные с использованием ключевого слова `function` (иногда гораздо более лаконичные).

Давайте начнем с синтаксиса, а затем более внимательно рассмотрим `this` и другие вещи, которые стрелочные функции обрабатывают иначе, чем традиционные функции.

Синтаксис стрелочной функции

Стрелочные функции бывают двух форм: с *кратким телом* (часто называемые краткими стрелочными функциями) и со стандартным *телом функции* (я называю их *подробными* стрелочными функциями, хотя они все же не такие подробные, как традиционные функции). Сначала мы рассмотрим краткую форму.

Предположим, вы хотите отфильтровать массив, сохранив только те записи, значение которых меньше 30. Вероятно, вы бы сделали это, используя метод `Array.prototype.filter` и передав функцию обратного вызова. В ES5 это выглядело бы так:

```
var array = [42, 67, 3, 23, 14];
var filtered = array.filter(function(entry) {
    return entry < 30;
});
console.log(filtered); // [3, 23, 14]
```

Это обычный вариант использования для обратных вызовов: обратный вызов, который должен сделать что-то простое и вернуть значение. Стрелочные функции обеспечивают очень лаконичный способ реализовать это:

```
const array = [42, 67, 3, 23, 14];
const filtered = array.filter(entry => entry < 30);
console.log(filtered); // [3, 23, 14]
```

Когда вы сравниваете его с традиционной привычной функцией, это почти ничем не похоже на функцию! Не волнуйтесь, вы скоро привыкнете к этому.

Как видно, краткая форма стрелочной функции — это буквально просто имя принимаемых параметров, затем стрелка (=>), указывающая анализатору, что это стрелочная функция, а затем выражение, определяющее возвращаемое значение. Вот и все. Нет ключевого слова `function`, нет фигурных скобок для определения тела функции, нет ключевого слова `return`; только параметры, стрелка и основное выражение.

Если вы хотите, чтобы стрелочная функция принимала несколько параметров, вам необходимо заключить список ее параметров в круглые скобки. Так будет ясно, что все они являются параметрами для стрелочной функции. Например, если вы хотите отсортировать массив, а не фильтровать его, вы бы сделали так:

```
const array = [42, 67, 3, 23, 14];
array.sort((a, b) => a - b);
console.log(array); // [3, 14, 23, 42, 67]
```

Обратите внимание на круглые скобки вокруг параметров. Как вы думаете, почему они требовались синтаксису?

Подумайте, как бы это выглядело, если бы их там не было:

```
array.sort(a, b => a - b);
```

Видите проблему? Верно! Выглядит так, будто мы передаем два аргумента методу `sort`: переменную под названием `a` и стрелочную функцию `b => a - b`. Следовательно, круглые скобки необходимы, если есть несколько параметров. (Если вы хотите быть последовательным, то всегда можете применять скобки, даже если есть только один параметр.)

Круглые скобки используются даже когда вам не нужно принимать какие-либо параметры, например, при обратном вызове таймера — просто оставьте их пустыми:

```
setTimeout(() => console.log("timer fired"), 200);
```

Хотя краткие стрелочные функции часто пишутся без разрывов строк, это вовсе не является требованием краткого синтаксиса. Если вам нужно было выполнить сложное вычисление в обратном вызове `array.sort`, вы могли бы поместить его в отдельную строку:

```
const array = [42, 67, 3, 23, 14];
array.sort((a, b) =>
    a % 2 ? b % 2 ? a - b : -1 : b % 2 ? 1 : a - b
);
console.log(array); // [3, 23, 67, 14, 42]
```

Этот код сортирует массив по двум группам: сначала нечетные числа, а затем четные. Тело функции по-прежнему представляет собой одно (сложное) выражение, что важно для краткой формы стрелочной функции.

Но что если ваша стрелочная функция должна содержать несколько операторов, а не однострочное выражение? Может быть, вы думаете, что вызов метода `sort` был бы понятнее, если бы вы разбили это сложное выражение (я, конечно, думаю, что так и было бы). Для этого вы можете использовать форму стандартного *тела функции* (или, как я ее называю, *подробную* форму), указав тело функции с помощью фигурных скобок после стрелки:

```
const array = [42, 67, 3, 23, 14];
array.sort((a, b) => {
    const aIsOdd = a % 2;
    const bIsOdd = b % 2;
    if (aIsOdd) {
        return bIsOdd ? a - b: -1;
}
    return bIsOdd ? 1 : a - b;
});
console.log(array); // [3, 23, 67, 14, 42]
```

Этот обратный вызов выполняет то же самое, что и предыдущий, используя несколько простых операторов, а не одно сложное выражение.

Две вещи, на которые следует обратить внимание в отношении подробной формы:

- Сразу после стрелки есть открывающая фигурная скобка. Она сообщает анализатору, что вы используете подробную форму, где тело функции находится внутри фигурных скобок, как и в случае с традиционными функциями.
- Используйте оператор `return` для возврата значения, как и в случае с традиционной функцией. Если бы вы не использовали `return`, то стрелочная функция явно не предоставляла бы возвращаемое значение, и ее вызов привел бы к значению `undefined`.

Точно так же как краткая стрелочная функция не обязательно должна быть в одной строке, подробная функция не обязательно должна быть в нескольких строках. Расстановка разрывов строк зависит только от вас.

ПРЕДУПРЕЖДЕНИЕ: ОПЕРАТОР «ЗАПЯТАЯ» И СТРЕЛОЧНЫЕ ФУНКЦИИ

Поскольку краткая форма стрелочной функции принимает выражение после =>, некоторые программисты привыкли использовать оператор «запятая», чтобы избежать использования подробной формы, если им нужно выполнить всего две или три задачи в функции. Напомним, что оператор «запятая» — один из самых странных в JavaScript: он вычисляет свой левый операнд, отбрасывает результирующее значение, а затем вычисляет правый операнд и принимает это значение в качестве результата. Пример:

```
function returnSecond(a, b) {
    return a, b;
}
console.log(returnSecond(1, 2)); // 2
```

Результат выражения return a, b; — это значение b.

Как это связано со стрелочными функциями? Это позволяет вам выполнять несколько действий в выражении, используемом краткой формой, применяя выражение в левом операнде, которое имеет побочный эффект. Допустим, у вас есть массив «обработчиков», обернутый вокруг «элементов». Вы хотите преобразовать каждую запись, передав обработчик в функцию unregister, а затем передать его элемент в функцию `register` и запомнить результат. С помощью традиционной функции вы бы сделали так:

```
handlers = handlers.map(function(handler) {
    unregister(handler);
    return register(handler.item);
});
```

Вы можете сделать этот код короче, используя подробную стрелочную функцию:

```
handlers = handlers.map(handler => {
    unregister(handler);
    return register(handler.item);
});
```

Но некоторые программисты вместо этого будут применять *краткую* стрелочную функцию, употребляя (или, возможно, злоупотребляя) оператор «запятая»:

```
handlers = handlers.map(handler =>
    (unregister(handler), register(handler.item))
);
```

(Но обычно все в одной строке.) Вы должны заключить выражение с запятой в круглые скобки: иначе анализатор подумает, что вы передаете два аргумента в метод `handlers.map` вместо одного. Поэтому единственная причина, по которой оно короче подробной формы, заключается в том, что оно позволяет избежать написания ключевого слова return.

Считаете ли вы это удобным использованием оператора запятой или злоупотреблением им — просто имейте в виду, что вы, скорее всего, увидите, как он используется таким образом в свободном мире программирования.

Есть одна загвоздка в краткой форме стрелочных функций, о которой следует знать. Краткая стрелочная функция возвращает объект, созданный с помощью объектного литерала.

Предположим, у вас есть массив строк, и вы хотите преобразовать его в массив объектов, используя строку в качестве их свойства `name`. Это звучит как задача для метода `map` массива и краткой стрелочной функции:

```
const a = ["Joe", "Mohammed", "María"];
const b = a.map(name => {name: name});          // Это не работает
console.log(b);
```

Но когда вы пытаетесь запустить это, вы получаете массив с кучей значений `undefined` внутри вместо объектов. (Или, в зависимости от создаваемого объекта, синтаксическую ошибку.) Так что же произошло?

Проблема в том, что открывающая фигурная скобка сообщает анализатору JavaScript, что вы используете подробную, а не краткую форму. Поэтому он использует эту фигурную скобку для начала тела, а затем использует содержимое (`name: name`) в качестве тела стрелочной функции, подобной этой традиционной функции:

```
const a = ["Joe", "Mohammed", "María"];
const b = a.map(function(name) {            // Это не работает
    name: name
});
console.log(b);
```

Поскольку функция ничего не возвращает (потому что анализатор думает, что вы используете подробную форму), результатом ее вызова становится значение `undefined`. В итоге вы получаете массив, заполненный значениями `undefined`. Тело функции не является синтаксической ошибкой (в этом примере), потому что оно выглядит как метка (их можно использовать с вложенными циклами, чтобы прервать внешний цикл), за которой следует отдельная ссылка на переменную. Это допустимый синтаксис, хотя он ничего не делает.

Ответ заключается в том, чтобы заключить литерал объекта в круглые скобки. Тогда анализатор не увидит фигурную скобку сразу после стрелки и поймет, что вы используете краткую форму:

```
const a = ["Joe", "Mohammed", "María"];
const b = a.map(name => ({name: name}));     // Это работает
console.log(b);
```

Или, конечно, можно использовать подробную форму и выражение `return`. Если это кажется вам более понятным при написании кода.

Краткая ремарка: литерал объекта в этом вызове метода `map` может быть еще короче благодаря *сокращенным свойствам*, о которых вы узнаете в главе 5.

Стрелочные функции и лексические this

До сих пор вы видели, что стрелочные функции могут сделать код более лаконичным. Это полезно само по себе, но это не их главный трюк. Он заключается в том, что, в отличие от традиционных функций, у них нет собственной версии `this`. Вместо этого они закрываются над `this` контекста, в котором созданы, точно так же как закрываются над переменной.

Почему это полезно? Давайте рассмотрим знакомую проблему. Вы пишете код в объектном методе и хотите использовать обратный вызов, но вам нужно, чтобы `this` в обратном вызове ссылался на ваш объект. С традиционной функцией этого не произошло бы, потому что у традиционных функций есть собственное `this`, определяемое тем, как они вызываются. Из-за этого обычно используется переменная, которую обратный вызов может закрыть в качестве обходного пути, как в этом коде ES5:

```
Thingy.prototype.delayedPrepAndShow = function() {
    var self = this;
    this.showTimer = setTimeout(function() {
        self.prep();
        self.show();
}, this.showDelay);
};
```

Сейчас используется переменная `self` вместо `this` внутри функции обратного вызова. Значит, не важно, что у обратного вызова есть своя версия `this` с другим значением.

Поскольку у стрелочных функций нет собственных `this`, они закрываются над `this`, как в ES5 пример закрывается над переменной `self`. Таким образом, вы можете переписать этот обратный вызов как стрелочную функцию без какой-либо необходимости в переменной `self`:

```
Thingy.prototype.delayedPrepAndShow = function() {
    this.showTimer = setTimeout(() => {
        this.prep();
        this.show();
}, this.showDelay);
};
```

Гораздо проще. (В главе 4 вы узнаете о новом способе создания этого метода вместо присвоения свойству `Thingy.prototype`.)

Ключевое слово `this` — не единственное, над чем закрываются стрелочные функции. Они также закрываются над аргументами (автоматический псевдомассив всех аргументов, полученный функцией) и двумя другими вещами, о них вы узнаете в главе 4 (`super` и `new.target`).

Стрелочные функции не могут быть конструкторами

Поскольку у них нет собственного `this`, стрелочные функции не могут быть функциями-конструкторами. То есть вы не можете использовать их с ключевым словом `new`:

```
const Doomed = () => {};
const d = new Doomed(); // TypeError: Doomed не является конструктором
```

В конце концов, основная цель функции-конструктора — заполнить вновь созданный объект, передаваемый функции как `this`. Если у функции нет собственного `this`, она не может устанавливать свойства для нового объекта, и ей нет смысла быть конструктором.

Явное запрещение этого позволило стрелочным функциям быть более легкими, чем традиционные функции: им не обязательно содержать свойство `prototype` с привязанным к нему объектом. Запомните, что когда вы используете функцию в качестве конструктора, прототип создаваемого нового объекта присваивается из прототипа `prototype` функции:

```
function Thingy() {
}
var t = new Thingy();
console.log(Object.getPrototypeOf(t) === Thingy.prototype); // истина
```

Поскольку движок JavaScript не может заранее знать, собираетесь ли вы использовать традиционную функцию в качестве конструктора, он должен поместить свойство `prototype` и объект, на который оно будет ссылаться, в каждую созданную вами традиционную функцию. (При условии оптимизации, конечно.)

Но поскольку стрелочные функции не могут быть конструкторами, они не получают свойство `prototype`:

```
function traditional() {
}
const arrow = () => {
};
console.log("prototype" in traditional); // истина
console.log("prototype" in arrow);       // ложь
```

Однако стрелочные функции — не единственные новые функции в современном JavaScript.

ЗНАЧЕНИЯ ПАРАМЕТРОВ ПО УМОЛЧАНИЮ

Начиная с ES2015 вы можете указать значения параметров по умолчанию. В ES5 и более ранних версиях вам пришлось бы делать это с помощью кода, подобного этому:

```
function animate(type, duration) {
    if (duration === undefined) {// (Или любая из нескольких подобных проверок)
        duration = 300;
    }
    // ...выполнение задач...
}
```

Теперь вы можете сделать это декларативно:

```
function animate(type, duration = 300) {
    // ...выполнение работы...
}
```

Это более лаконично и проще для поддержки инструментами.

Значение по умолчанию срабатывает, если при вызове функции значение параметра равно `undefined`. Если вы полностью оставите аргумент при вызове функции, значение параметра будет равно `undefined`. Если вы укажете `undefined` в качестве значения, его значение также будет `undefined` (конечно). В любом случае вместо этого функция использует значение по умолчанию.

Давайте посмотрим на это в действии: запустите код, приведенный в Листинге 3-1.

Листинг 3-1: Основные параметры по умолчанию — basic-default-parameters.js

```
function animate(type, duration = 300) {
    console.log(type + ", " + duration);
}
animate("fadeout");             // "fadeout, 300"
animate("fadeout", undefined); // "fadeout, 300" (снова)
animate("fadeout", 200);        // "fadeout, 200"
```

Код для обработки значений по умолчанию фактически вставляется в начало функции движком JavaScript. Однако позже вы увидите, что этот код находится в своей области видимости.

В этом примере вы могли бы при желании указать значения по умолчанию для `type` и `duration`:

```
function animate(type = "fadeout", duration = 300) {
    // ...выполнение задач...
}
```

И, в отличие от некоторых других языков, вы также можете указать значение по умолчанию для `type`, но не указывать значение для `duration`:

```
function animate(type = "fadeout", duration) {
    // ...выполнение задач...
}
```

Это может показаться немного странным (пожалуй, это и *есть* немного странно). Но такое решение продиктовано тем фактом, что значение по умолчанию применяется, если значение равно `undefined`. Часто это происходит из-за того, что аргумент не был задан, но может быть связано и с тем, что значение `undefined` было задано явно. Запустите код из Листинга 3-2.

Листинг 3-2: Значение по умолчанию для первого параметра — default-first-parameter.js

```
function animate(type = "fadeout", duration) {
    console.log(type + ", " + duration);
}
animate("fadeout", 300);    // "fadeout, 300"
animate(undefined, 300);    // "fadeout, 300" (снова)
animate("fadein", 300);     // "fadein, 300"
animate();                  // "fadeout, undefined"
```

Значения по умолчанию — это выражения

Параметр по умолчанию — это выражение. Оно не обязательно должно быть просто буквальным значением. Значение по умолчанию может даже вызывать функцию. Выражение по умолчанию вычисляется при вызове функции, а не при ее определении, и *только* в том случае, если значение по умолчанию необходимо для этого конкретного вызова функции.

Предположим, что вы хотите использовать разную продолжительность для каждого типа анимации и у вас есть функция, дающая длительность по умолчанию для данного типа. Вы могли бы написать функцию `animate` следующим образом:

```
function animate(type, duration = getDefaultDuration(type)) {
    // ...выполнение задач...
}
```

Функция `getDefaultDuration` не вызывается без надобности. Этот код трактуется почти так же, как следующий:

```
function animate(type, duration) {
    if (duration === undefined) {
        duration = getDefaultDuration(type);
    }
    // ...выполнение задач...
}
```

Основное различие связано с областью видимости, к которой мы сейчас вернемся.

В том примере значение по умолчанию для `duration` использует параметр `type`. Это отлично, поскольку `type` предшествует `duration`. Если `type` следует за `duration`, будет выброшена ошибка `ReferenceError`; при условии, что значение по умолчанию для `duration` было необходимо для конкретного вызова:

```
function animate(duration = getDefaultDuration(type), type) {
    // ...выполнение задач...
}
animate(undefined, "dissolve"); // ReferenceError: type не определен
```

Это похоже на то, как если бы вы попытались получить доступ к значению переменной, объявленной с помощью директивы `let` перед объявлением `let`, как показано в главе 2. С точки зрения движка JavaScript эта функция выглядит очень похоже на такой сценарий (опять же, кроме области видимости):

```
function animate() {
    let duration = /*...получаем значение аргумента 0...*/;
    if (duration === undefined) {
        duration = getDefaultDuration(type);
    }
    let type = /*...получаем значение аргумента 1...*/;
    // ...выполнение задач...
}
```

Если вы вызываете ее без переменной `duration`, код, заполняющий значение по умолчанию, пытается использовать `type`, когда он находится во временной мертвой зоне. Поэтому вы получаете сообщение об ошибке.

Правило простое: значение по умолчанию может использовать параметры, перечисленные перед ним, но не после него.

Значения по умолчанию вычисляются в их собственной области видимости

Как вы уже поняли, значения по умолчанию могут ссылаться на другие параметры, если они находятся перед значением по умолчанию в списке, и они могут ссылаться на вещи, которые являются частью внешней области видимости (например, функция `getDefaultDuration` в предыдущем примере). Но они не могут ссылаться ни на что определенное *в* теле функции, даже на то, что прошло поднятие. Запустите Листинг 3-3.

Листинг 3-3: Значения по умолчанию не могут ссылаться на что-то в теле функции — default-access-body.js

```
function example(value = x) {
    var x = 42;
    console.log(value);
}
example(); // ReferenceError: значение x не определено
```

Здесь я использовал директиву `var`, потому что у нее нет временной мертвой зоны: она поднимается в верхнюю часть области видимости, где была объявлена. Но выражение значения по умолчанию по-прежнему не могло его использовать. Причина в том, что значения по умолчанию вычисляются в их собственной области, существующей между областью, содержащей функцию, и областью внутри функции (рисунок 3-1). Это, как если бы функция была обернута в другую функцию, которая обрабатывала бы выполнение значений по умолчанию. Примерно так:

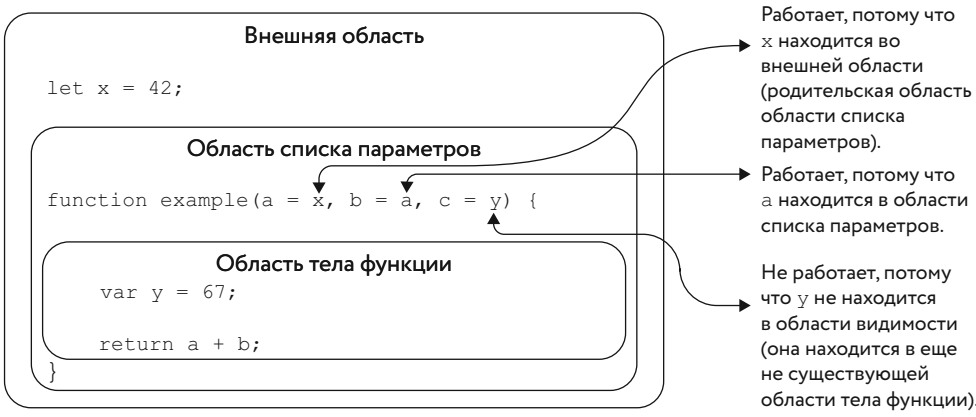


РИСУНОК 3-1

```
function example(value) {
    if (value === undefined) {
        value = x;
    }
    const exampleImplementation = () => {
        var x = 42;
        console.log(value);
    };
    return exampleImplementation();
}
example(); // ReferenceError: значение x не определено
```

Конечно, это делается не буквально, но концептуально это очень близко.

К счастью, фрагменты визуально различимы (список параметров отделен от тела функции). Поэтому легко понять, что у списка параметров есть своя область видимости.

НЕПРОСТЫЕ СПИСКИ ПАРАМЕТРОВ И ИЗМЕНЕНИЕ СТРОГОГО РЕЖИМА

Список параметров, содержащий просто голый список параметров без значений по умолчанию или других возможностей параметров из ES2015+, о которых вы узнаете позже, называется «простым». Если в списке параметров используется какая-то из новых возможностей, он называется «непростым» списком параметров (креативно, да?).

Удивительно, что функция с непростым списком параметров не может содержать директиву `"use strict"`:

```
function example(answer = 42) {
    "use strict";    // SyntaxError: Невозможно применить
                     // директиву 'use strict' в функции
                     // с непростым списком параметров
    console.log(answer);
}
```

Почему это ошибка? Потому что, если эта функция определена в коде свободного режима, директива включит строгий режим внутри функции. Проблема с этим заключается в том, что непростой список параметров эффективно включает автоматическую обработку (например, применение значений по умолчанию), которая происходит внутри списка параметров. Если определение функции отображается в свободном режиме, но функция начинается с `"use strict"`, должен ли код в списке параметров быть в строгом режиме или свободном режиме? Рассмотрим этот пример:

```
function example(callback = o => {with (o) {return answer;}}) {
    "use strict";
    console.log(callback({answer: 42}));
}
```

Должен ли этот код стать причиной синтаксической ошибки (поскольку `with` не допускается в строгом режиме) или нет? Является ли список параметров строгим или свободным?

Чтобы избежать путаницы, функции не разрешено это делать. Более того, чтобы избежать сложности синтаксического анализа, даже если определение функции появляется в уже строгом контексте, оно все равно не может содержать директиву `"use strict"`, если у нее есть непростой список параметров. Функция может наследовать только строгость контекста, в котором она определена.

Значения по умолчанию не увеличивают арность функции

Арность функции обычно определяется как количество формально объявленных параметров, содержащихся в функции. В JavaScript их можно получить из свойства `length` функции:

```
function none() {
}
console.log(none.length); // 0
function one(a) {
   }
console.log(one.length);  // 1
function two(a, b) {
}
console.log(two.length);  // 2
```

В JavaScript параметры со значениями по умолчанию не учитываются при вычислении арности, и они также препятствуют подсчету любых последующих параметров:

```
function stillOne(a, b = 42) {
}
console.log(stillOne.length);    // 1
function oneYetAgain(a, b = 42, c) {
}
console.log(oneYetAgain.length); // 1
```

Результат для функции `stillOne` достаточно прост: у нее есть один параметр без значения по умолчанию и один с таковым, поэтому ее арность равна 1. Результат для функции `oneYetAgain` более интересен тем, что ее параметр `c` не получает явного значения по умолчанию. Но поскольку параметр перед ним содержит значение по умолчанию, третий параметр не засчитывается в арности.

ОСТАТОЧНЫЕ ПАРАМЕТРЫ

Обычно при написании функции вы знаете, какие параметры ей понадобятся. При каждом вызове количество параметров останется одинаковым. Но некоторые функции должны принимать изменяющееся количество параметров. Например, до того, как метод

`Object.assign` был добавлен в ES2015, многие инструменты программистов JavaScript включали функцию `extend`. Она принимает целевой объект и один или несколько исходных объектов — функция копирует свойства из исходных объектов в целевой. В ES5 и более ранних версиях вы бы использовали псевдомассив `arguments` для доступа к этим аргументам, например:

```
function extend(target) {
    var n, source;
    for (n = 1; n < arguments.length; ++n) {
        source = arguments[n];
        Object.keys(source).forEach(function(key) {
            target[key] = source[key];
        });
    }
    return target;
}
var obj = extend({}, {a: 1}, {b: 2});
console.log(obj); // {a: 1, b: 2}
```

Но у псевдомассива `arguments` есть ряд проблем:

- В свободном режиме у него есть проблемы с производительностью, вызванные тем, как он связан с формальными параметрами (но это можно решить с помощью строгого режима).
- Это *все* аргументы, то есть вы должны индексировать за пределами тех, которые представляют ваши формальные параметры (вот почему в примере `extend` цикл `for` начинается с `n = 1` вместо `n = 0` — мы хотели пропустить аргумент параметра `target`).
- Псевдомассив похож на массив, но не является им, поэтому у него нет возможностей массива, таких как методы `forEach` или `map`.
- У стрелочных функций нет собственного объекта `arguments`.
- Субъективно говоря, у него неуклюжее название, которое не отражает специфичного для функции значения этих аргументов.

В ES2015 было добавлено решение этих проблем: *остаточный параметр* — «rest». Если вы объявляете последний параметр с многоточием (`...`) перед ним, это говорит движку JavaScript собрать все фактические аргументы с этого момента («остаточные» аргументы) и поместить их в этот параметр как подлинный массив.

При обновлении более ранней функции `extend` для использования остаточного параметра, поскольку значение остаточного параметра — фактический массив, вы можете использовать метод `forEach` (или `for-of`, о котором вы узнаете в главе 6) вместо `for`:

```
function extend(target, ...sources) {
    sources.forEach(source => {
        Object.keys(source).forEach(key => {
            target[key] = source[key];
        });
    });
    return target;
}
```

```
const obj = extend({}, {a: 1}, {b: 2});
console.log(obj); // {a: 1, b: 2}
```

Разовьем мысль. Функция `extend` сама по себе может быть стрелочной, поскольку ей больше не нужен собственный псевдомассив `arguments`:

```
const extend = (target, ...sources) => {
    sources.forEach(source => {
        Object.keys(source).forEach(key => {
            target[key] = source[key];
        });
    });
    return target;
};
const obj = extend({}, {a: 1}, {b: 2});
console.log(obj); // {a: 1, b: 2}
```

Но что произойдет, если для остаточного параметра вообще не будет указано никаких аргументов? То есть что, если мы вызовем функцию `extend` только с одним аргументом, целевым объектом и вообще без исходных объектов? (Как бы странно это ни было.) Как вы думаете, каково тогда значение `source`?

Вероятно, вы сделали заключение из реализации функции `extends`: вы получаете пустой массив, поэтому этому коду не нужно было проверять аргументы `source`, чтобы убедиться, что в них есть массив. (Разработчики этой функции могли бы пойти на то, чтобы в этом случае `source` содержали значение `undefined`, но они выбрали пустой массив для обеспечения согласованности.)

Обратите внимание, что «`...`» — не оператор, хотя выражение «оператор rest» можно услышать от множества людей. Это просто «синтаксис» или «обозначение» (в данном случае синтаксис/обозначение остаточного параметра). Вы узнаете больше о синтаксисе «`...`» и о том, почему он не оператор (и не может им быть), в главе 5.

ВИСЯЩИЕ ЗАПЯТЫЕ В СПИСКАХ ПАРАМЕТРОВ И ВЫЗОВАХ ФУНКЦИЙ

В некоторых стилях кодирования перечисляют параметры функции в отдельных строках, возможно, для размещения комментариев к строкам, описывающих их, или чего-то подобного:

```
function example(
    question,// (строка) Вопрос должен заканчиваться вопросительным знаком
    answer   // (строка) Ответ должен заканчиваться соответствующей пунктуацией
) {
    // ...
}
```

Аналогично при вызове функций иногда бывает полезно поместить каждый аргумент в отдельную строку, особенно если аргумент — длинное выражение:

```
example(
    "Do you like building software?",
    "Big time!"
);
```

В обоих случаях по мере развития кодовой базы и необходимости добавления третьего параметра в функцию вы должны добавить запятую к предыдущей строке при добавлении новой строки с новым параметром/аргументом:

```
function example(
    question,   // (строка) Вопрос должен заканчиваться вопросительным знаком
    answer,    // (строка) Ответ должен заканчиваться соответствующей
               // пунктуацией
    required   // (логическое значение) Приведенное, если необходимо
               // (поэтому значение по умолчанию равно false)
) {
    // ...
}
example(
    "Do you like building software?",
    "Big time!",
    true
);
```

Распространенная ошибка — забыть поставить запятую. Необходимость сделать это приводит к тому, что изменение кода, не связанное с этими строками, кажется связанным с ними (например, в различиях фиксации) просто потому, что вы добавили запятую.

ES2017 позволяет ставить *висящую запятую* в конце списков параметров и списков аргументов. Начиная с ES2017 вы можете написать исходную двухпараметрическую версию функции следующим образом:

```
function example(
    question, // (строка) Вопрос должен заканчиваться вопросительным знаком
    answer,   // (строка) Ответ должен заканчиваться соответствующей
пунктуацией
) {
    // ...
}
```

и вызвать ее таким образом:

```
example(
    "Do you like building software?",
    "Big time!",
);
```

Это чисто синтаксическое изменение, больше оно ничего не меняет. Функции только с аргументами `question` и `answer` до сих пор получают только два параметра. Их арность, что отражено в их свойстве `length`, до сих пор равна `2`.

Позже, при добавлении третьего параметра, вы просто добавляете новые строки — вам не нужно изменять существующие:

```
function example(
    question,  // (строка) Вопрос должен заканчиваться вопросительным знаком
    answer,    // (строка) Ответ должен заканчиваться соответствующей
               // пунктуацией
    required,  // (логическое значение) Приведенное, если необходимо
```

```
                // (поэтому значение по умолчанию равно false)
) {
    // ...
}
example(
    "Do you like building software?",
    "Big time!",
    true,
);
```

СВОЙСТВО ИМЕНИ ФУНКЦИИ

ES2015 наконец-то стандартизировал свойство `name` (имя) функции (это было нестандартное расширение в некоторых движках JavaScript в течение многих лет) и сделал это очень интересным и мощным способом, который, помимо прочего, делает многие ранее анонимные функции не анонимными.

Очевидный способ получения имени функцией — это объявление функции или выражение именованной функции:

```
// Объявление
function foo() {
}
console.log(foo.name); // "foo"
// Выражение именованной функции
const f = function bar() {
};
console.log(f.name);   // "bar"
```

Имена удобны для целей отчетности и особенно полезны в стеках вызовов при ошибках.

Интересно, что вы можете дать имя функции, даже если используете *анонимное* выражение функции. Например, если вы присваиваете результат выражения константе или переменной:

```
let foo = function() {
};
console.log(foo.name); // "foo"
```

Это не волшебные догадки движка JavaScript. В спецификации четко и тщательно определено, когда и где именно устанавливаются имена.

Не имеет значения, объявляете ли вы переменную при назначении функции или назначаете функцию позже. Важно то, когда вы создаете функцию:

```
let foo;
foo = function() {
};
console.log(foo.name); // "foo"
```

Выражения стрелочных функций работают так же, как выражения анонимных функций:

```
let foo = () => {
};
console.log(foo.name); // "foo"
```

Однако это не просто присвоение переменных/констант: есть еще множество мест, где имя становится производным от контекста.

Например, в объектном литерале, если вы присваиваете свойству результат выражения анонимной функции, функция получает имя свойства:

```
const obj = {
    foo: function() {
    }
};
console.log(obj.foo.name); // "foo"
```

(Это также работает, если вы используете синтаксис *метода*; о нем вы узнаете в главе 5.) Это работает даже при использовании функции в качестве значения параметра по умолчанию:

```
(function(callback = function() {}) {
    console.log(callback.name); // "callback"
})();
```

Однако есть одна операция, которая, на удивление, *не* задает имя функции, а присваивает объектному свойству существующего объекта. Например:

```
const obj = {};
obj.foo = function() {
};
console.log(obj.foo.name); // "" - здесь нет имени
```

Почему? Комитет TC39 счел это конкретное использование слишком большой утечкой информации. Предположим, что в приложении есть кэш обработчиков, связанных с некоторой секретной информацией, связанной с пользователем, и должно передать обработчик некоторому внешнему коду. Если обработчик создан следующим образом:

```
cache[getUserSecret(user)] = function() {};
```

и имя функции было установлено этой операцией, выдача функции-обработчика стороннему коду выдаст значение из `getUserSecret(user)`. Поэтому комитет намеренно исключил эту конкретную операцию из задания имени функции.

ОБЪЯВЛЕНИЯ ФУНКЦИЙ ВНУТРИ БЛОКОВ

В течение многих лет размещение объявления функций внутри блока не подпадало под спецификацию, но и не было запрещено. Движки JavaScript могли обрабатывать их как «допустимое расширение». Начиная с ES2015 объявления функций в блоках стали частью спецификации. Для них существуют стандартные правила, а также «устаревшая веб-семантика», которая применяется только в свободном режиме в веб-браузерах.

Во-первых, давайте посмотрим, что такое объявление функции в блоке:

```
function simple() {
    if (someCondition) {
        function doSomething() {
        }
        setInterval(doSomething, 1000);
    }
}
simple();
```

Этот код содержит объявление функции (а не ее выражение) внутри блока. Но объявления функций поднимаются, обрабатываются перед любым пошаговым кодом. Так что же может означать объявление функции внутри блока?

Поскольку синтаксис не был указан, но и не был запрещен, создатели движков JavaScript были вольны определять свое собственное значение для этого синтаксиса в своих движках. Естественно, это привело к проблемам, так как разные движки делали разные вещи. Функция `simple` ранее не доставляла особых хлопот, но обдумайте это:

```
function branching(num) {
    console.log(num);
    if (num < 0.5) {
        function doSomething() {
            console.log("true");
        }
    } else {
        function doSomething() {
            console.log("false");
        }
    }
    doSomething();
}
branching(Math.random());
```

В мире ES5 есть по крайней мере три способа справиться с такой ситуацией.

Первый и наиболее очевидный вариант — признать это синтаксической ошибкой.

Второй вариант — рассматривать код так, как если бы объявления действительно были функциональными выражениями. Например, так:

```
function branching(num) {
    console.log(num);
    var doSomething;
    if (num < 0.5) {
        doSomething = function doSomething() {
            console.log("true");
        };
    } else {
        doSomething = function doSomething() {
            console.log("false");
        };
    }
    doSomething();
}
branching(Math.random());
```

Это приводит к регистрации значения «true» или «false» в зависимости от случайного числа, что, вероятно, и было задумано автором.

Третий вариант — рассматривать код как несколько поднятых объявлений в одной и той же области:

```
function branching(num) {
    function doSomething() {
        console.log("true");
    }
    function doSomething() {
        console.log("false");
    }
    console.log(num);
    if (Math.random() < 0.5) {
    } else {
    }
    doSomething();
}
branching(Math.random());
```

Такое решение всегда регистрирует значение «false» (потому что, когда у вас есть повторяющиеся объявления в одной и той же области — что разрешено делать — выигрывает последнее). Вероятно, это не то, что предполагал автор (в данном конкретном случае), но больше соответствует тому факту, что объявления функций поднимаются, а не обрабатываются как часть пошагового кода.

Так какое же решение приняли создатели движков JavaScript?

Все три.

Некоторые движки (в основном второстепенные) работали с вариантом 1, другие — с вариантом 2, третьи — с вариантом 3. Это был полный бардак.

Когда TC39 приступил к определению семантики для объявлений функций в блоках, перед ними стояла очень сложная задача указать что-то разумное и согласованное, не делая недействительным значительный объем кода по всему миру. И они сделали две вещи:

- Они определили стандартную семантику, которая соответствует остальной части ES2015.
- Они определили «устаревшую веб-семантику» только для кода в свободном режиме в веб-браузерах.

Объявления функций внутри блоков: Стандартная семантика

Самая простая обработка — это стандартная семантика, которая всегда действует в строгом режиме (даже в веб-браузерах). Я предлагаю вам использовать строгий режим, чтобы избежать случайного написания кода, полагающегося на устаревшую семантику. Со стандартной семантикой объявления функций эффективно преобразуются в выражения функций, назначенные переменным `let` (поэтому они ограничены областью действия блока, в котором они находятся), и поднимаются в верхнюю часть блока. Давайте возьмем более раннюю функцию ветвления `branching` и добавим немного больше функций логгирования, чтобы увидеть процесс поднятия:

```
"use strict";
function branching(num) {
    console.log(num);
    if (num < 0.5) {
        console.log("true branch, typeof doSomething = " + typeof doSomething);
        function doSomething() {
            console.log("true");
        }
     } else {
        console.log("false branch, typeof doSomething = " + typeof doSomething);
        function doSomething() {
            console.log("false");
        }
    }
    doSomething();
}
branching(Math.random());
```

В строгом режиме движок JavaScript обрабатывает этот код так, как если бы он был таким:

```
"use strict";
function branching(num) {
    console.log(num);
    if (num < 0.5) {
        let doSomething = function doSomething() {
            console.log("true");
        };
        console.log("true branch, typeof doSomething = " + typeof doSomething);
    } else {
        let doSomething = function doSomething() {
            console.log("false");
        };
        console.log("false branch, typeof doSomething = " + typeof doSomething);
    }
    doSomething();
}
branching(Math.random());
```

Обратите внимание, как каждое объявление было эффективно поднято внутри своего блока, над вызовом `console.log`.

Естественно, если вы запустите это, произойдет сбой. `doSomething` в самом конце не является объявленным идентификатором в области верхнего уровня функции, поскольку `doSomething` в каждом блоке имеет блочную область действия. Итак, давайте изменим наш пример на что-то, что будет выполняться, и запустим его, см. Листинг 3-4.

Листинг 3-4: Объявление функции в блоке: строгий режим — func-decl-block-strict.js

```
"use strict";
function branching(num) {
    let f;
    console.log(num);
    if (num < 0.5) {
```

```
        console.log("true branch, typeof doSomething = " + typeof doSomething);
        f = doSomething;
        function doSomething() {
            console.log("true");
        }
    } else {
        console.log("false branch, typeof doSomething = " + typeof doSomething);
        f = doSomething;
        function doSomething() {
            console.log("false");
        }
    }
    f();
}
branching(Math.random());
```

Теперь, когда вы запускаете его, `f` в конечном счете ссылается на ту или иную функцию и регистрирует значение «true» или «false». Из-за поднятия `doSomething` ссылается на функцию, когда она назначается `f`, даже если это назначение находится выше объявления.

Объявления функций внутри блоков: Устаревшая веб-семантика

В свободном режиме в веб-браузерах применяется устаревшая веб-семантика, определенная Приложением Б спецификации. (Некоторые движки применяют ее даже вне веб-браузеров.) В спецификации отмечается, что, когда объявления функций в блоках не рассматривались как синтаксические ошибки, различные способы их обработки движками означали, что можно было полагаться только на сценарии, которые обрабатывались одинаково пересечением этих движков JavaScript. Эти три сценария:

1. Объявление и вызовы функции существуют только в пределах одного блока.
2. Функция объявляется и, возможно, используется в пределах одного блока, но также вызывается определением внутренней функции, не содержащимся в том же блоке.
3. Функция объявляется и, возможно, используется в пределах одного блока, но также вызывается в последующих блоках.

Пример функции ветвления `branching`, который мы использовали в этой главе, не подходит ни для одного из этих трех сценариев. В нем есть два объявления функций, использующих одно и то же имя в двух разных блоках, а затем ссылки на это имя в коде, следующем за этими блоками. Но если у вас есть устаревший код свободного режима, соответствующий одному из этих трех сценариев, вы можете рассчитывать, что он будет работать в кроссбраузерном режиме. Это не значит, что вы должны писать новый код, опирающийся на устаревшую веб-семантику. Вместо этого напишите код, полагающийся только на стандартную семантику (возможно, используя строгий режим для обеспечения процесса).

Унаследованная семантика во многом совпадает со стандартной семантикой, но в дополнение к переменной `let`, объявленной и находящейся внутри блока для функции, для нее также существует переменная `var` в области видимости содержащей функции (или

глобальной области видимости, если все это не находится внутри функции). В отличие от `let` в блоке, присвоение `var` не поднимается в верхнюю часть блока, это делается при достижении объявления функции в коде. (Это кажется странным, но это необходимо для поддержки случая № 2 пересечения поведения между основными движками до стандартизации.)

Опять же, пример с функцией ветвления `branching` не вписывается в список распространенных методов обработки, для решения которых предназначена устаревшая семантика. Но эта семантика все же говорит, как это должно обрабатываться сейчас (в то время как в старых движках это может не обрабатываться так): поскольку есть переменная `doSomething` типа `var` в области действия функции, вызов `doSomething` в конце работает. Давайте еще раз посмотрим на пример без «`use strict`»; и с протоколированием как до, так и после объявления функции. Загрузите и запустите Листинг 3-5.

Листинг 3-5: Объявление функции в блоке: веб-совместимость — func-decl-block-web-compat.js

```
function branching(num) {
    console.log("num = " + num + ", typeof doSomething =
        " + typeof doSomething);
    if (num < 0.5) {
        console.log("true branch, typeof doSomething = " + typeof doSomething);
        function doSomething() {
            console.log("true");
        }
        console.log("end of true block");
    } else {
        console.log("false branch, typeof doSomething = " + typeof doSomething);
        function doSomething() {
            console.log("false");
        }
        console.log("end of false block");
    }
    doSomething();
}
branching(Math.random());
```

Как вы можете видеть, на этот раз функция `doSomething` в конце не выходит за рамки. В свободном режиме движок JavaScript эффективно лечит это, например так (за исключением, конечно, `varDoSomething` и `letDoSomething`, которые становятся просто `doSomething`):

```
function branching(num) {
    var varDoSomething;
    console.log("num = " + num + ", typeof doSomething =
        " + typeof varDoSomething);
    if (num < 0.5) {
        let letDoSomething = function doSomething() {
            console.log("true");
        };
        console.log("true branch, typeof doSomething =
            " + typeof letDoSomething);
```

```
        varDoSomething = letDoSomething; // где было объявление
        console.log("end of true block");
    } else {
        let letDoSomething = function doSomething() {
            console.log("false");
        };
        console.log("false branch, typeof doSomething =
            " + typeof letDoSomething);
        varDoSomething = letDoSomething; // где было объявление
        console.log("end of false block");
    }
    varDoSomething();
}
branching(Math.random());
```

Поскольку функции присваиваются переменной `var` из области функции, они доступны за пределами блоков. Но объявления по-прежнему хранятся в своих блоках.

Лучше не писать новый код, полагаясь на устаревшую семантику. Вместо этого стоит придерживаться строгого режима.

ОТ СТАРЫХ ПРИВЫЧЕК К НОВЫМ

Со всеми этими новыми функциональными возможностями мы можем обновить множество старых привычек.

Используйте стрелочные функции вместо различных обходных путей для этого значения

Старая привычка: Применять различные обходные пути для получения доступа к вызывающему контексту `this` в обратном вызове.

- Использование переменной, например `var self = this;`
- Использование метода `Function.prototype.bind`
- Использование поддерживающего функцию параметра `thisArg`

Примеры:

```
// Использование переменной
var self = this;
this.entries.forEach(function(entry) {
    if (entry.matches(str)) {
        self.appendEntry(entry);
    }
});

// Использование метода Function.prototype.bind
this.entries.forEach(function(entry) {
    if (entry.matches(str)) {
        this.appendEntry(entry);
    }
}.bind(this));
```

```
// Использование параметра 'thisArg'
this.entries.forEach(function(entry) {
    if (entry.matches(str)) {
        this.appendEntry(entry);
    }
}, this);
```

Новая привычка: Используйте стрелочную функцию:

```
this.entries.forEach(entry => {
    if (entry.matches(str)) {
        this.appendEntry(entry);
    }
});
```

Используйте стрелочные функции для обратных вызовов, если не используете аргументы или this

Старая привычка: Применение традиционных функций для обратных вызовов, которые не используют `this` или аргументы (по причине отсутствия иного выбора):

```
someArray.sort(function(a, b) {
    return a - b;
});
```

Новая привычка: Если обратный вызов не использует `this` или аргументы, используйте стрелочную функцию. Стрелочные функции более легкие и лаконичные:

```
someArray.sort((a, b) => a - b);
```

Большинство обратных вызовов массивов (например, `sort`, `forEach`, `map`, `reduce`, …), обратные вызовы для вызовов строк `replace`, создание промисов и решение обратных вызовов (вы узнаете о промисах в главе 8), а также многое другое может стать стрелочной функцией.

Рассмотрите применение стрелочных функций и в других местах

Старая привычка: Использовать традиционные функции для всего, потому что (опять же) у вас не было выбора.

Новая привычка: Подумайте, имеет ли смысл использовать стрелочную функцию, даже если это не обратный вызов. Это будет в первую очередь вопрос стиля. Например, посмотрите на эту функцию, которая создает строку с начальными заглавными буквами из входной строки:

```
function initialCap(str) {
    return str.charAt(0).toUpperCase() + str.substring(1);
   }
```

По сравнению с версией со стрелочной функцией:

```
const initialCap = str =>
    str.charAt(0).toUpperCase() + str.substring(1);
```

Теоретически у стрелочной функции нет общих с традиционной функцией недостатков. Кроме того, вы можете использовать директиву `const`, как в этом примере, чтобы избежать переназначения идентификатора (или используйте `let`, если такое ограничение не требуется).

Конечно, традиционные функции никуда не денутся. Поднятие иногда полезно; некоторые предпочитают, чтобы ключевое слово `function` помечало функции. И если вам нравится краткость, стоит отметить, что даже

```
let x = () => {/*...*/};
```

все же чуть-чуть длиннее, чем

```
function x() {/*...*/}
```

хотя предоставленный

```
let x=()=>{/*...*/};
```

совсем чуть-чуть короче.

Это в значительной степени зависит от ваших (и вашей команды) предпочтений в стиле.

Не используйте стрелочные функции, когда вызывающей стороне необходимо контролировать значение this

Старая привычка: Использование традиционной функции в качестве обратного вызова, где важно, чтобы вызывающий объект контролировал, что представляет собой `this`.

Новая привычка: Эм… продолжайте так делать.

Иногда важно, чтобы вызывающий абонент установил, что такое `this`. Например, в коде браузера, использующем jQuery, часто требуется, чтобы jQuery контролировал, что такое `this` в обратном вызове. Или при ответе на события DOM, к которым вы уже подключились с помощью функции `addEventListener`, потому что это укажет `this` сослаться на элемент при вызове вашего обратного вызова (хотя вместо этого вы можете использовать свойство `currentTarget` объекта события). Или при определении общих методов объектов (например, поскольку они находятся в прототипе), потому что важно разрешить задавать значение `this` при их вызове.

Таким образом, хотя в некоторых случаях переключение на стрелочные функции полезно, иногда необходимо использовать старые традиционные функции (или синтаксис метода, о котором вы узнаете в главах 4 и 5).

Используйте значения параметров по умолчанию, а не код, предоставляющий значения по умолчанию

Старая привычка: Использование кода внутри функции для предоставления значения по умолчанию для параметра:

```
function doSomething(delay) {
    if (delay === undefined) {
        delay = 300;
    }
    // ...
}
```

Новая привычка: Используйте значения параметров по умолчанию везде, где это возможно:

```
function doSomething(delay = 300) {
    // ...
}
```

Используйте остаточный параметр вместо ключевого слова arguments

Старая привычка: Использование псевдомассива `arguments` в функциях, принимающих различное количество аргументов.

Новая привычка: Используйте остаточные параметры.

Рассмотрите возможность использования висящих запятых, если это оправдано

Старая привычка: Не включать висящую запятую в списки параметров и вызовы функций (поскольку они не были разрешены):

```
function example(
    question,  // (строка) Вопрос должен заканчиваться вопросительным знаком
    answer     // (строка) Ответ должен заканчиваться соответствующей
               // пунктуацией
) {
    // ...
}
// ...
example(
    "Do you like building software?",
    "Big time!"
);
```

Новая привычка: В зависимости от стиля вашего/вашей команды вы можете рассмотреть возможность использования висящих запятых, чтобы добавление дополнительных параметров/аргументов по мере развития кода не требовало изменения строк, определяющих то, что раньше было последним параметром или аргументом:

```
function example(
    question,  // (строка) Вопрос должен заканчиваться вопросительным знаком
    answer,    // (строка) Ответ должен заканчиваться соответствующей
               // пунктуацией
) {
    // ...
}
// ...
example(
    "Do you like building software?",
    "Big time!",
);
```

Однако это вопрос стиля.

4

Классы

СОДЕРЖАНИЕ ГЛАВЫ

- Что такое класс?
- Представление нового синтаксиса класса
- Сравнение с устаревшим синтаксисом
- Создание подклассов
- Наследование методов и свойств (прототипов и статических)
- Доступ к унаследованным методам и свойствам
- Создание подклассов для встроенных компонентов
- Вычисляемые имена методов
- Отказ от `Object.prototype`
- Что такое `new.target` и почему его стоит применять
- Подробнее об этом в главе 18

В этой главе вы узнаете, как работает синтаксис `class` для создания конструкторов JavaScript и связанных с ними объектов-прототипов. Мы сравним и сопоставим новый синтаксис со старым, подчеркнем, что это все то же прототипическое наследование, которым славится JavaScript, и исследуем, насколько мощен и прост новый синтаксис. Вы увидите, как создавать подклассы, включая подклассы встроенных модулей (даже те, которые не могут быть подклассами в ES5 и более ранних версий, например `Array` and `Error`), как работает ключевое слово `super`, для чего нужен метод `new.target` и как его применять.

В этой главе не рассматриваются функции, которые появятся в ES2020 или ES2021, такие как поля открытого класса, приватные поля класса и приватные методы. Описание этих вопросов см. в главе 18.

ЧТО ТАКОЕ КЛАСС?

Прежде чем рассмотреть новый синтаксис, давайте начнем с очевидного. В JavaScript на самом деле нет классов, не так ли? Он просто имитирует их с помощью прототипов, верно?

Это популярная точка зрения, потому что люди путают тип класса, предоставляемый языками на основе классов, такими как Java или C#, с общей концепцией класса в терминах информатики. Но классы — это нечто большее, чем в основном статические конструкции, предоставляемые такого рода языками. Для того чтобы язык содержал классы, он должен обеспечивать две вещи: инкапсуляцию (объединение данных и методов вместе[34]) и наследование. Наличие классов в языке — это не то же самое, что основанный на классах язык, просто язык поддерживает инкапсуляцию и наследование. Прототипические языки могут (и содержат) классы и получили их еще до изобретения JavaScript. Механизм, который они используют для обеспечения второго требования (наследования), — это объекты-прототипы.

JavaScript всегда был одним из самых объектно-ориентированных среди распространенных языков. Он, безусловно, был способен делать что-то в стиле классов, по крайней мере, с ECMAScript 1. Кто-то может педантично утверждать, что в нем не было классов в смысле информатики, пока с появлением ES5 не был добавлен метод `Object.create` для прямой поддержки наследования (хотя можно было эмулировать `Object.create` с помощью вспомогательной функции). Другие могут возразить, что даже ES5 не подходит, потому что в нем отсутствуют декларативные конструкции и простой способ ссылки на методы суперкласса.

Но педанты могут перестать спорить. Начиная с ES2015 даже эти возражения устранены: в JavaScript есть классы.

Давайте посмотрим, как они работают в современном JavaScript. Попутно мы сравним новый синтаксис со старым синтаксисом ES5 и более ранними версиями.

ПРЕДСТАВЛЕНИЕ НОВОГО СИНТАКСИСА КЛАССА

В Листинге 4-1 показан базовый пример `class`: класса, экземпляры которого представляют собой цвета, выраженные в системе RGB. (В списке опущен некоторый код тела метода, поскольку цель здесь — просто показать общий синтаксис. Код метода находится в файле в разделе загрузки, его можно запустить, и он будет показан позже.)

Листинг 4-1: Базовый класс с синтаксисом класса — basic-class.js

```
class Color {
    constructor(r = 0, g = 0, b = 0) {
        this.r = r;
        this.g = g;
        this.b = b;
    }
```

[34] Термин *инкапсуляция* может также относиться к сокрытию данных (приватные свойства и т. п.), но это не обязательно для того, чтобы у языка были «классы». (Однако JavaScript действительно обеспечивает скрытие данных с помощью замыканий и WeakMap, о которых вы узнаете в Главе 12, а вскоре и с помощью закрытых полей, о которых вы узнаете в Главе 18.)

```
    get rgb() {
        return "rgb(" + this.r + ", " + this.g + ", " + this.b + ")";
    }

    set rgb(value) {
        // ...показанный позже код...
    }

    toString() {
        return this.rgb;
    }

    static fromCSS(css) {
        // ...показанный позже код...
    }
}

let c = new Color(30, 144, 255);
console.log(String(c));             // "rgb(30, 144, 255)"
c = Color.fromCSS("00A");
console.log(String(c));             // "rgb(0, 0, 170)"
```

Это определяет класс с помощью:

- *Конструктора.*
- Трех *свойств данных* (`r`, `g`, and `b`).
- *Свойства-акцессора* (`rgb`).
- *Метода прототипа* (`toString`) (их также иногда называют *методами экземпляров*, поскольку вы обычно получаете к ним доступ через экземпляры, но *методы прототипов* более точные — фактический *метод экземпляра* будет существовать только в экземпляре, а не наследоваться от прототипа).
- *Статического метода* (`fromCSS`) (их также иногда называют *методами класса*).

Давайте разберем все по частям.

Первое, что нужно написать, — это само определение класса, которое, как вы можете видеть, представляет собой новую структуру. Как и в случае с функциями, вы определяете классы либо с помощью объявлений, либо с помощью выражений:

```
// Объявление класса
class Color {
}

// Анонимное выражение класса
let Color = class {
};

// Именованное выражение класса
let C = class Color {
};
```

Пока мы будем придерживаться объявлений, а к выражениям вернемся позже.

Объявления классов не поднимаются так, как объявления функций. Вместо этого они поднимаются (или наполовину поднимаются) так же, как объявления `let` и `const`.

Согласно описаниям Временной мертвой зоны, приведенным в главе 2: поднимается только идентификатор, а не его инициализация. Также, как и в случае с `let` и `const`, если вы объявляете класс в глобальной области видимости, идентификатор класса становится глобальным, но не будет свойством глобального объекта.

Добавление конструктора

Даже с тем, что у нас есть на данный момент, у класса есть конструктор по умолчанию, поскольку мы не предоставили явный конструктор, подробнее об этом чуть позже. А пока давайте добавим в класс конструктор, который действительно что-то делает. Код для конструктора указывается в определении `constructor`, например:

```
class Color {
    constructor(r = 0, g = 0, b = 0) {
        this.r = r;
        this.g = g;
        this.b = b;
    }
}
```

Это определяет функцию, которая будет связана с идентификатором `Color` класса. Обратите внимание на синтаксис: ключевое слово `function` используется не везде — только имя `constructor`, открывающая скобка, список параметров, если они есть (в этом случае `r`, `g` и `b`), закрывающая скобка, фигурные скобки, чтобы определить тело конструктора, а также код для конструктора внутри этих скобок. (В реальном коде вы можете проверить значения `r`, `g` и `b`. Я убрал это из примера, чтобы он остался коротким.)

> **ПРИМЕЧАНИЕ**
> После закрывающей фигурной скобки определения constructor нет точки с запятой. Определения конструктора и метода в теле класса подобны объявлениям, не заканчивающимся точкой с запятой. (Однако если точка с запятой присутствует, это допустимо. Грамматика специально учитывает их, чтобы избежать превращения этой простой ошибки в синтаксическую.)

В ES5 и более ранних версиях вы, вероятно, определили бы этот конструктор следующим образом:

```
// Более старый (~ES5), почти эквивалентный
function Color(r, g, b) {
    // ...
}
```

но поскольку объявления `class` не поднимаются, как объявления `function`, они не совсем эквивалентны. Это больше похоже на выражение функции, назначенное переменной:

```
// Более старый (~ES5), почти эквивалентный
var Color = function Color(r, g, b) {
    // ...
};
```

Здесь следует отметить одну вещь. С синтаксисом `class` вы пишете структуру для класса в целом и пишете определение конструктора отдельно внутри него, тогда как со старым синтаксисом вы просто определяли функцию (и это зависело от того, как вы использовали эту функцию, была ли это просто функция или конструктор для класса). Новый синтаксис подобен этому, поэтому вы можете определять другие аспекты класса *декларативно*, в том же контейнере: в конструкции `class`.

Как отмечалось ранее, указание конструктора явно становится необязательным. Класс `Color` прекрасно обходится без этого (кроме того, что он не задает свои свойства `x`, `y` и `z`). Если вы не указываете конструктор ключевым словом, движок JavaScript создает конструктор, который ничего не делает, точно так же как если бы он был у вас в классе:

```
constructor() {
}
```

(В подклассах он кое-что делает; вы узнаете об этом позже.)

Конструктор — это функция, но вы можете вызывать его как часть процесса создания объекта: выполнение функции конструктора должно быть результатом использования оператора `new` (либо напрямую, либо косвенно с применением `new` к подклассу) или вызова метода `Reflect.construct` (о котором вы узнаете в главе 14). Если вы попытаетесь вызвать его, не создавая объект, движок выдаст ошибку:

```
Color();     // TypeError: Конструктор класса Color не может быть
             // вызван без 'new'
```

Это устраняет целый ряд ошибок, которых не было в старом синтаксисе, когда функция, предназначенная для конструктора, также могла быть вызвана без оператора `new`, что приводило к чрезвычайно запутанным результатам (или раздутому коду, чтобы предотвратить это).

Давайте добавим этот «раздутый код» в пример `Color` ES5, чтобы он (в некоторой степени) запрещал вызовы, кроме как через оператор `new`:

```
// Более старый (~ES5), почти эквивалентный
var Color = function Color(r, g, b) {
    if (!(this instanceof Color)) {
        throw new TypeError(
            "Class constructor Color cannot be invoked without 'new'"
        );
    }
    // ...
};
```

Это на самом деле не заставляет его вызываться как часть процесса построения объекта (вы могли бы просто вызвать его через метод `Color.call` или `Color.apply` с существующим объектом), но, по крайней мере, пытается.

Вы уже можете видеть, что даже в минимальном классе вы получаете реальные преимущества с точки зрения надежности от использования нового синтаксиса, сокращая при этом шаблонный код.

Код внутри класса всегда находится в строгом режиме, даже если окружающий код нет. Поэтому, чтобы быть действительно внимательными к примеру с ES5, нам пришлось бы обернуть все это в функцию ограничения области видимости и использовать в ней строгий режим. Но давайте просто предположим, что наш код уже находится в строгом режиме, не усложняя пример с ES5.

Добавление свойств экземпляра

На данный момент стандартный способ настройки свойств для новых экземпляров класса — назначить их в конструкторе точно так же, как это делалось в ES5:

```
class Color {
    constructor(r = 0, g = 0, b = 0) {
        this.r = r;
        this.g = g;
        this.b = b;
    }
}
```

Поскольку эти свойства создаются с помощью базового присвоения, они являются настраиваемыми, доступными для записи и перечисляемыми.

В дополнение к свойствам (или вместо них), получаемым из параметров, вы можете настроить свойство, которое не берется из аргумента конструктора, просто используя литерал или значение, получаемое откуда-то еще. Если требуется, чтобы все экземпляры `Color` начинались с черного, вы, например, можете отбросить параметры `r`, `g` и `b`, и сделать так, чтобы свойства `r`, `g` и `b` начинались с нуля:

```
class Color {
    constructor() {
        this.r = 0;
        this.g = 0;
        this.b = 0;
    }
}
```

> **ПРИМЕЧАНИЕ**
> В главе 18 рассказывается о функции, которая, вероятно, будет добавлена в ES2020 или ES2021, но которая уже широко используется в транспиляции, — объявления полей (свойств) публичного класса.

Добавление метода прототипа

Теперь давайте добавим метод, который будет помещен в объект класса `prototype`, чтобы все экземпляры получили к нему доступ:

```
class Color {
    constructor(r = 0, g = 0, b = 0) {
        this.r = r;
        this.g = g;
        this.b = b;
    }
    toString() {
        return "rgb(" + this.r + ", " + this.g + ", " + this.b + ")";
    }
}
```

Обратите внимание, что между определением конструктора и определением функции `toString` нет запятой, как если бы они были в литерале объекта. Определение класса не похоже на литерал объекта — разделители в виде запятой не ставятся. (Если так сделаете, то получите синтаксическую ошибку.)

Показанный ранее синтаксис метода помещает метод в объект `Color.prototype`. Поэтому экземпляры класса наследуют этот метод от своего прототипа:

```
const c = new Color(30, 144, 255);
console.log(c.toString());          // "rgb(30, 144, 255)"
```

Сравните новый синтаксис с тем, как функции-прототипы обычно добавляются в ES5:

```
// Более старый (~ES5), почти эквивалентный
Color.prototype.toString = function toString() {
    return "rgb(" + this.r + ", " + this.g + ", " + this.b + ")";
};
```

Новый синтаксис, *синтаксис метода* ES2015, более декларативный и краткий. Он также особо помечает функцию как метод, что дает ей доступ к возможностям, которых у нее не было бы в качестве простой функции (например, `super` — вы узнаете об этом позже). Новый синтаксис также делает метод неисчислимым. Это разумное значение по умолчанию для методов в прототипах, чего нет в показанной ранее версии ES5. (Чтобы сделать его неисчислимым в коде ES5, вам нужно было бы определить его с помощью метода `Object.defineProperty` вместо использования присваивания.)

Метод также по определению не относится к функциям конструктора, и поэтому движок JavaScript не помещает в него свойство `prototype` и связанный с ним объект:

```
class Color {
    // ...
    toString() {
        // ...
    }
}
const c = new Color(30, 144, 255);
console.log(typeof c.toString.prototype); // "undefined"
```

Поскольку методы не относятся к конструкторам, попытка вызвать их при помощи оператора `new` приводит к ошибке.

Напротив, в ES5 все функции могут использоваться в качестве конструкторов, насколько известно движку. Поэтому он должен предоставить им свойство `prototype` с прикрепленным объектом:

```
// Более старый (~ES5), почти эквивалентный
var Color = function Color(r, g, b) {
    // ...
};
Color.prototype.toString = function toString() {
    // ...
};
var c = new Color(30, 144, 255);
console.log(typeof c.toString.prototype); // "object"
```

Таким образом, теоретически синтаксис метода более эффективен с точки зрения использования памяти, чем более старый синтаксис функции. На практике было бы неудивительно, если бы движки JavaScript уже могли оптимизировать ненужные свойства прототипа даже для методов ES5.

Добавление статического метода

При создании примера класса `Color` до сих пор вы видели конструктор, пару свойств экземпляра и метод прототипа. Давайте добавим *статический метод* — объект, привязанный к самому классу `Color`, а не к прототипу:

```
class Color {
    // ...

    static fromCSS(css) {
        const match = /^#?([0-9a-f]{3}|[0-9a-f]{6});?$/i.exec(css);
        if (!match) {
            throw new Error("Invalid CSS code: " + css);
        }
        let vals = match[1];
        if (vals.length === 3) {
            vals = vals[0] + vals[0] + vals[1] + vals[1] + vals[2] + vals[2];
        }
        return new this(
            parseInt(vals.substring(0, 2), 16),
            parseInt(vals.substring(2, 4), 16),
            parseInt(vals.substring(4, 6), 16)
        );
    }
}
```

> **ПРИМЕЧАНИЕ**
> Этот конкретный статический метод, fromCSS, создает и возвращает новый экземпляр класса. Не все статические методы так могут. В примере это делается при помощи вызова оператора new this(/*…*/), который работает, поскольку при вызове метода Color.fromCSS(/*…*/), как обычно, this в вызове устанавливается на объект, к свойству которого был получен доступ (в данном случае Color). Следовательно, new this(/*…*/) — это то же самое, что и new Color(/*…*/). Позже вы узнаете о доступной альтернативе в разделе о подклассах.

Ключевое слово `static` указывает движку JavaScript установить метод для объекта `Color`, а не для `Color.prototype`. Вы можете вызвать метод для `Color` напрямую:

```
const c = Color.fromCSS("#1E90FF");
console.log(c.toString());              // "rgb(30, 144, 255)"
```

Ранее в ES5 вы бы реализовали это присвоением свойства функции `Color`:

```
Color.fromCSS = function fromCSS(css) {
    // ...
   };
```

Как и в случае с методами-прототипами, использование синтаксиса метода означает, что `fromCSS` не содержит свойства `prototype` с назначенным ему объектом, как это было бы в версии ES5, и не может быть вызвано как конструктор.

Добавление свойства-акцессора

Давайте добавим *свойство-акцессор* (*свойство доступа*) к классу `Color`. Свойство-акцессор — это свойство с методом геттера, методом сеттера или и тем и другим. Давайте предоставим объекту `Color` свойство `rgb`, получающее цвет в качестве стандартной строки `rgb(r, g, b)`. Также необходимо обновить метод `toString`, чтобы он использовал это свойство вместо создания строки самостоятельно:

```
class Color {
    // ...

    get rgb() {
        return "rgb(" + this.r + ", " + this.g + ", " + this.b + ")";
    }

    toString() {
        return this.rgb;
    }
}
let c = new Color(30, 144, 255);
console.log(c.rgb);           // "rgb(30, 144, 255)"
```

Как видите, это похоже на определение акцессора в литерале объекта в ES5. В результате есть одно небольшое отличие: свойства-акцессора в конструкциях `class` не поддаются перечислению, что имеет смысл для чего-то определенного в прототипе. А свойства доступа, определенные в литералах объектов, являются перечисляемыми.

Пока что акцессор `rgb` объекта `Color` доступен только для чтения (геттер без сеттера). При необходимости добавить сеттер потребуется определить метод `set`:

```
class Color {
    // ...

    get rgb() {
        return "rgb(" + this.r + ", " + this.g + ", " + this.b + ")";
    }
```

```
    set rgb(value) {
        let s = String(value);
        let match = /^rgb\((\d{1,3}),(\d{1,3}),(\d{1,3})\)$/i.exec(
            s.replace(/\s/g, "")
        );
        if (!match) {
            throw new Error("Invalid rgb color string ' " + s + " ' ");
        }
        this.r = parseInt(match[1], 10);
        this.g = parseInt(match[2], 10);
        this.b = parseInt(match[3], 10);
    }

    // ...
}
```

Теперь можно установить значение свойства `rgb` и получить его значение:

```
let c = new Color();
console.log(c.rgb);           // "rgb(0, 0, 0)"
c.rgb = "rgb(30, 144, 255)";
console.log(c.r);             // 30
console.log(c.g);             // 144
console.log(c.b);             // 255
console.log(c.rgb);           // "rgb(30, 144, 255)"
```

Определение акцессора в ES5 немного болезненно, если вы хотите добавить его к существующему объекту в свойстве `prototype` вместо того, чтобы заменять этот объект новым:

```
// Более старый (~ES5), почти эквивалентный
Object.defineProperty(Color.prototype, "rgb", {
    get: function() {
        return "rgb(" + this.r + ", " + this.g + ", " + this.b + ")";
    },
    set: function(value) {
        // ...
    },
    configurable: true
});
```

Вы также можете определить статические свойства акцессора (хотя это и редкий случай). Просто определите акцессор с помощью ключевого слова `static` перед ним:

```
class StaticAccessorExample {
    static get cappedClassName() {
        return this.name.toUpperCase();
    }
}
console.log(StaticAccessorExample.cappedClassName); // Пример статического
                                                    // акцессора
```

Вычисляемые имена методов

Иногда требуется создавать методы с именами, определяемыми во время выполнения, а не буквально указываемым в коде. Это особенно важно при использовании типа *Symbol*, о которых вы узнаете в главе 5. В ES5 это было достаточно легко сделать с помощью синтаксиса скобок свойства-акцессора:

```
// Более старый (~ES5), почти эквивалентный
var name = "foo" + Math.floor(Math.random() * 1000);
SomeClass.prototype[name] = function() {
    // ...
};
```

В ES2015 вы можете сделать почти то же самое с синтаксисом метода:

```
let name = "foo" + Math.floor(Math.random() * 1000);
class SomeClass {
    [name]() {
        // ...
    }
}
```

Обратите внимание на квадратные скобки вокруг имени метода. Они работают точно так же, как и свойства-акцессоры:

- В них можно поместить любое выражение.
- Выражение вычисляется при вычислении определения класса.
- Если результат не относится к строковому типу или к типу Symbol (глава 5), он преобразуется в строку.
- Результат используется в качестве имени метода.

Статические методы и методы свойств-акцессоров также могут получать вычисляемые имена. Вот пример статического метода, получающего свое имя из результата выражения умножения:

```
class Guide {
    static [6 * 7]() {
        console.log("Life, the Universe, and Everything");
    }
}
Guide["42"](); // "Life, the Universe, and Everything"
```

СРАВНЕНИЕ С УСТАРЕВШИМ СИНТАКСИСОМ

Хотя в этой главе вы уже видели сравнения нового синтаксиса со старым, давайте сравним полное определение класса `Color` в синтаксисе ES2015 с его почти эквивалентной версией ES5. Сравните Листинг 4-2 с Листингом 4-3.

Листинг 4-2: Полный базовый класс с синтаксисом класса — full-basic-class.js

```
class Color {
    constructor(r = 0, g = 0, b = 0) {
        this.r = r;
        this.g = g;
        this.b = b;
    }

    get rgb() {
        return "rgb(" + this.r + ", " + this.g + ", " + this.b + ")";
    }

    set rgb(value) {
        let s = String(value);
        let match = /^rgb\((\d{1,3}),(\d{1,3}),(\d{1,3})\)$/i.exec(
            s.replace(/\s/g, "")
    );
        if (!match) {
            throw new Error("Invalid rgb color string '" + s + "'");
        }
        this.r = parseInt(match[1], 10);
        this.g = parseInt(match[2], 10);
        this.b = parseInt(match[3], 10);
}

    toString() {
        return this.rgb;
}

    static fromCSS(css) {
        const match = /^#?([0-9a-f]{3}|[0-9a-f]{6});?$/i.exec(css);
        if (!match) {
            throw new Error("Invalid CSS code: " + css);
        }
        let vals = match[1];
        if (vals.length === 3) {
            vals = vals[0] + vals[0] + vals[1] + vals[1] + vals[2] + vals[2];
        }
        return new this(
            parseInt(vals.substring(0, 2), 16),
            parseInt(vals.substring(2, 4), 16),
            parseInt(vals.substring(4, 6), 16)
        );
    }
}

// Использование класса
let c = new Color(30, 144, 255);
console.log(String(c));          // "rgb(30, 144, 255)"
c = Color.fromCSS("00A");
console.log(String(c));          // "rgb(0, 0, 170)"
```

Листинг 4–3: Базовый класс с синтаксисом старого стиля — full-basic-class-old-style.js

```
"use strict";
var Color = function Color(r, g, b) {
    if (!(this instanceof Color)) {
        throw new TypeError(
            "Class constructor Color cannot be invoked without 'new'"
        );
    }
    this.r = r || 0;
    this.g = g || 0;
    this.b = b || 0;
};

Object.defineProperty(Color.prototype, "rgb", {
    get: function() {
        return "rgb(" + this.r + ", " + this.g + ", " + this.b + ")";
    },
    set: function(value) {
        var s = String(value);
        var match = /^rgb\((\d{1,3}),(\d{1,3}),(\d{1,3})\)$/i.exec(
            s.replace(/\s/g, "")
    );
        if (!match) {
            throw new Error("Invalid rgb color string '" + s + "'");
        }
        this.r = parseInt(match[1], 10);
        this.g = parseInt(match[2], 10);
        this.b = parseInt(match[3], 10);
    },
    configurable: true
});

Color.prototype.toString = function() {
    return this.rgb;
};
Color.fromCSS = function(css) {
    var match = /^#?([0-9a-f]{3}|[0-9a-f]{6});?$/i.exec(css);
    if (!match) {
        throw new Error("Invalid CSS code: " + css);
    }
    var vals = match[1];
    if (vals.length === 3) {
        vals = vals[0] + vals[0] + vals[1] + vals[1] + vals[2] + vals[2];
    }
    return new this(
        parseInt(vals.substring(0, 2), 16),
        parseInt(vals.substring(2, 4), 16),
        parseInt(vals.substring(4, 6), 16)
    );
};

// Использование класса
var c = new Color(30, 144, 255);
console.log(String(c));          // "rgb(30, 144, 255)"
c = Color.fromCSS("00A");
console.log(String(c));          // "rgb(0, 0, 170)"
```

СОЗДАНИЕ ПОДКЛАССОВ

Новый синтаксис полезен даже для базовых классов, но по-настоящему он раскрывается при работе с подклассами. Даже просто настройка наследования конструкторов в ES5 довольно сложна и подвержена ошибкам. Использование «супер» версии метода в подклассе еще сложнее. С синтаксисом `class` все эти трудности устраняются.

Давайте создадим подкласс с названием `ColorWithAlpha`[35] для класса `Color` со свойством непрозрачности:

```
class ColorWithAlpha extends Color {
}
```

Да, это действительно все, что вам нужно сделать для создания подкласса. Кое-что еще *можно* сделать, что-то, скорее всего, вы *сделаете*, но здесь описано, что *необходимо* сделать. Этот код выполняет следующие операции:

- Создает конструктор подкласса `ColorWithAlpha`.
- Делает `Color` (функцию конструктора суперкласса) прототипом `ColorWithAlpha`, так что любые статические свойства/методы класса `Color` доступны подклассу `ColorWithAlpha`. (Мы вернемся к этому. Идея функции, имеющей прототип, отличный от `Function.prototype`, нова и может показаться немного удивительной.)
- Создает объект прототипа подкласса `ColorWithAlpha.prototype`.
- Делает `Color.prototype` прототипом этого объекта, так что объекты, созданные с помощью выражения `new ColorWithAlpha`, наследуют свойства и методы суперкласса.

На рисунке 4-1 приведена картина взаимосвязи `Color`/`ColorWithAlpha`. Обратите внимание на две параллельные линии наследования: одна линия наследования для функций конструктора (`ColorWithAlpha` от `Color` от `Function.prototype` от `Object.prototype`) и параллельная для объектов, созданных с помощью этих конструкторов (`ColorWithAlpha.prototype` от `Color.prototype` от `Object.prototype`).

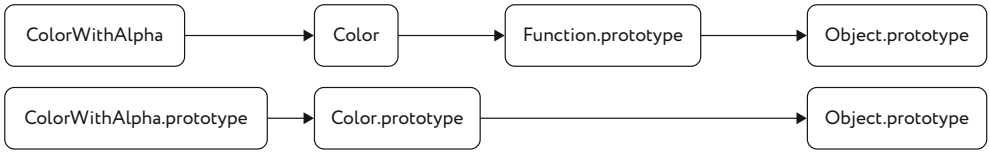


РИСУНОК 4-1

Запустите Листинг 4-4, чтобы увидеть, что основы подкласса настраиваются только с помощью этого простого объявления.

[35] Почему «alpha»? Подкласс `ColorWithAlpha` будет использовать стандарт RGBA. В цветовой модели RGBA буква «A» означает «альфа» (alpha), что относится к «альфа-каналу», в котором хранится/передается информация о прозрачности.

Листинг 4-4: Класс с базовым подклассом — class-and-basic-subclass.js

```
class Color {
    constructor(r = 0, g = 0, b = 0) {
        this.r = r;
        this.g = g;
        this.b = b;
    }

    get rgb() {
        return "rgb(" + this.r + ", " + this.g + ", " + this.b + ")";
    }

    set rgb(value) {
        let s = String(value);
        let match = /^rgb\((\d{1,3}),(\d{1,3}),(\d{1,3})\)$/i.exec(
            s.replace(/\s/g, "")
    );
        if (!match) {
            throw new Error("Invalid rgb color string '" + s + "'");
        }
        this.r = parseInt(match[1], 10);
        this.g = parseInt(match[2], 10);
        this.b = parseInt(match[3], 10);
    }

    toString() {
        return this.rgb;
    }

    static fromCSS(css) {
        const match = /^#?([0-9a-f]{3}|[0-9a-f]{6});?$/i.exec(css);
        if (!match) {
            throw new Error("Invalid CSS code: " + css);
        }
        let vals = match[1];
        if (vals.length === 3) {
            vals = vals[0] + vals[0] + vals[1] + vals[1] + vals[2] + vals[2];
        }
        return new this(
            parseInt(vals.substring(0, 2), 16),
            parseInt(vals.substring(2, 4), 16),
            parseInt(vals.substring(4, 6), 16)
        );
    }
}

class ColorWithAlpha extends Color {
}

// Использование класса
const c = new ColorWithAlpha(30, 144, 255);
console.log(String(c)); // "rgb(30, 144, 255)"
```

Обратите внимание, что, хотя в подклассе `ColorWithAlpha` нет явно определенного конструктора, его применение для создания цвета сработало просто отлично. Это потому,

что движок JavaScript предоставил конструктор по умолчанию. Вместо «ничего не делать» по умолчанию, которое движок предоставляет для базовых классов, для подклассов по умолчанию вызывается конструктор суперкласса, и ему передаются все получаемые аргументы. (Программисты иногда говорят, что подкласс *наследует* конструктор, но это не так. Движок создает отдельную функцию, вызывающую конструктор суперкласса.) Это очень мощное значение по умолчанию, означающее, что во многих случаях не нужно указывать явный конструктор в подклассах.

ПРИМЕЧАНИЕ

Существует большая разница между конструктором JavaScript по умолчанию и тем, который вам предоставит компилятор Java или C#. Если у вас есть опыт работы с Java, C# или подобными языками, вы поймете разницу. В Java и C# (и многих родственных языках) конструктор по умолчанию не принимает параметров и вызывает конструктор суперкласса без аргументов:

```
// Java
Subclass() {
    super();
}
// C#
Subclass(): base() {
}
```

Но в JavaScript конструктор по умолчанию для подкласса принимает любое количество аргументов и передает их все в конструктор суперкласса:

```
// JavaScript
constructor(/*...здесь любое количество параметров...*/) {
    super(/*...здесь все передаются в super...*/);
}
```

(Вы изучите super в следующем разделе.)

Это естественное следствие того факта, что JavaScript не использует перегрузку функций и сигнатуры функций, как это делают Java и C#. В Java или C# у вас может быть несколько «перегруженных» конструкторов, каждый из которых принимает разное количество или типы параметров. JavaScript предоставляет только один конструктор, и любые изменения в списках параметров обрабатываются внутри конструктора. Просто имеет больше смысла вызывать суперкласс по умолчанию со всеми полученными аргументами, а не с одним. Поэтому подкласс, которому не нужно изменять то, что ожидает конструктор, может просто полностью отключить конструктор.

Чтобы показать работу нового синтаксиса, давайте возьмем определение подкласса:

```
class ColorWithAlpha extends Color {
}
```

и сравним его с почти эквивалентным синтаксисом в ES5:

```
// Более старый (~ES5), почти эквивалентный
var ColorWithAlpha = function ColorWithAlpha() {
    Color.apply(this, arguments);
};
ColorWithAlpha.prototype = Object.create(Color.prototype);
ColorWithAlpha.prototype.constructor = ColorWithAlpha;
```

Это довольно небольшой фрагмент шаблонного кода с большой вероятностью возникновения ошибок, просто для настройки подкласса, и он не делает статические свойства и методы класса `Color` доступными в подклассе `ColorWithAlpha`. Это не страшно, но решение неочевидное, и допустить ошибку очень легко. Новый синтаксис более понятен, более декларативен, предлагает больше функциональности и прост в использовании.

Однако нет особого смысла создавать подкласс, если мы не собираемся заставлять его создавать что-то отличное от суперкласса. Давайте добавим некоторые функции в подкласс `ColorWithAlpha`, для которых вам нужно знать о выражении `super`.

Ключевое слово `super`

Чтобы добавить функции в подкласс `ColorWithAlpha`, во многих случаях вам потребуется знать о новом ключевом слове — `super`. Ключевое слово `super` используется в конструкторах и методах для ссылки на аспекты суперкласса. Есть два способа его применения:

- `super()`: В конструкторе подкласса вызывается `super`, как если бы это была функция для создания объекта, и суперкласс выполняет инициализацию объекта.
- `super.property` и `super.method()`: Вы ссылаетесь на свойства и методы прототипа суперкласса, обращаясь к ним через `super`, а не через `this`. (Конечно, можно использовать либо точечную нотацию, `super.property`, либо скобочную нотацию, `super["property"]`.)

В следующих двух разделах подробно рассказывается об использовании ключевого слова `super`.

> **ПРИМЕЧАНИЕ**
> Выражение super предназначено не только для использования в классах. В главе 5 вы узнаете, как super работает в методах для объектов, созданных с помощью литералов объектов вместо new.

Написание конструкторов подклассов

Подклассу `ColorWithAlpha` понадобится свойство, которого нет у класса `Color`, — прозрачность. Давайте создадим свойство с именем `a` («alpha») для его хранения. Свойство `a` будет получать значения в диапазоне от 0 до 1 (включительно). Например, значение 0,7 означает, что цвет на 70% непрозрачный (30% прозрачный).

Поскольку подкласс `ColorWithAlpha` должен принимать четвертый параметр конструктора, он больше не может просто использовать конструктор по умолчанию. Ему нужен свой собственный:

```
class ColorWithAlpha extends Color {
    constructor(r = 0, g = 0, b = 0, a = 1) {
        super(r, g, b);
        this.a = a;
    }
}
```

Сначала подкласс `ColorWithAlpha` вызывает выражение `super` и передает ему значения параметров `r`, `g` и `b`. Это создает объект и позволяет классу `Color` выполнить инициализацию этого объекта. В ES5 конструктор может выглядеть так:

```
// Более старый (~ES5), почти эквивалентный
var ColorWithAlpha = function ColorWithAlpha(r, g, b, a) {
    Color.call(this, r, g, b);
    this.a = a;
};
```

Выражение `Color.call(this, r, g, b)` почти эквивалентно выражению `super(r, g, b)`. Но есть существенная разница: в ES5 объект был создан до запуска первой строки подкласса `ColorWithAlpha`. При желании в нем можно поменять местами строки (хотя это плохой вариант):

```
var ColorWithAlpha = function ColorWithAlpha(r, g, b, a) {
    this.a = a;        // Работает, даже если это происходит до вызова Color
    Color.call(this, r, g, b);
};
```

Это не работает с выражением `class`. В самом начале кода подкласса `ColorWithAlpha` объект еще не создан. И попытка использовать `this` вызовет ошибку. Вы можете использовать `this` только после создания объекта, и он не создается до тех пор, пока вы не вызовете `super`. Попробуйте выполнить код, приведенный в Листинге 4-5.

Листинг 4-5: Получение доступа к this перед super — accessing-this-before-super.js

```
class Color {
    constructor(r = 0, g = 0, b = 0) {
        this.r = r;
        this.g = g;
        this.b = b;
    }
}
```

```
class ColorWithAlpha extends Color {
    constructor(r = 0, g = 0, b = 0, a = 1) {
        this.a = a;                  // Здесь выбрасывается ошибка
        super(r, g, b);
    }
}

// Применение:
const c = new ColorWithAlpha(30, 144, 255, 0.5);
```

Конкретная выбрасываемая ошибка будет варьироваться от одного движка JavaScript к другому. Вот несколько примеров:

- `ReferenceError`: Необходимо вызвать суперконструктор в производном классе, прежде чем обращаться к ‹this› или возвращаться из производного конструктора.
- `ReferenceError`: необходимо вызвать суперконструктор перед использованием |this| в конструкторе класса ColorWithAlpha.
- `ReferenceError`: это не определено.

Это требование существует для того, чтобы гарантировать, что инициализация создаваемого объекта выполняется на основании, описанном выше. У вас может быть код в конструкторе до вызова `super`, но он не может использовать `this` или что-либо связанное с экземпляром, до тех пор, пока объект не будет создан и у суперкласса не будет возможности инициализировать экземпляр.

Но не только применение ключевого слова `this` недоступно до вызова `super`, вы *должны* вызвать `super` в определенной точке конструктора подкласса. Если вы этого не сделаете, при возврате конструктора возникнет ошибка. Наконец, неудивительно, что попытка повторно вызвать `super` приводит к ошибке: вы не можете создать объект, когда он уже существует!

Наследование свойств и методов прототипа суперкласса и доступ к ним

Иногда подкласс переопределяет определение метода, предоставляя свой собственный вместо использования унаследованного от суперкласса метода. Например, у подкласса `ColorWithAlpha` должен быть свой метод `toString`, использующий нотацию `rgba` и включающий в себя свойство `a`:

```
class ColorWithAlpha extends Color {
    // ...

    toString() {
        return "rgba(" + this.r + ", " +
                         this.g + ", " +
                         this.b + ", " +
                         this.a + ")";
    }
}
```

Имейте в виду, что при вызове метода `toString` для экземпляра подкласса `ColorWithAlpha` будет использоваться это определение метода, а не определение из класса `Color`.

Иногда, однако, метод подкласса должен вызывать метод суперкласса как часть его реализации. Очевидно, что метод `this.methodName()` не будет работать, потому что он просто будет вызывать сам себя. Вам нужно подняться на уровень выше. Чтобы сделать это, метод суперкласса необходимо вызвать с помощью `super.methodName()`.

Давайте рассмотрим это, добавив метод `brightness`, вычисляющий яркость (свечение) цвета, к классу `Color`:

```
class Color {
    // ...

    brightness() {
        return Math.sqrt(
            (this.r * this.r * 0.299) +
            (this.g * this.g * 0.587) +
            (this.b * this.b * 0.114)
        );
    }
}
```

Не вдаваясь в математику (и это всего лишь одно определение яркости; есть и другие), эти константы регулируют цвета, чтобы учесть тот факт, что человеческие глаза воспринимают яркость красного, зеленого и синего по-разному.

Определение метода `brightness` работает только в классе `Color`, но не может работать в подклассе `ColorWithAlpha`, поскольку здесь требуется брать в расчет прозрачность: как минимум требуется приглушить яркость, основываясь на прозрачности цвета. В идеале нужно знать цвет заднего фона, чтобы учитывать яркость его цвета, так как цвет фона будет частично проступать. Следовательно, подклассу `ColorWithAlpha` требуется своя версия метода `brightness`, например:

```
class ColorWithAlpha extends Color {
        // ...
    brightness(bgColor) {
        let result = super.brightness() * this.a;
        if (bgColor && this.a !== 1) {
            result = (result + (bgColor.brightness() * (1 - this.a))) / 2;
        }
        return result;
    }
}
```

Он использует метод `brightness` класса `Color` для получения основных значений яркости цвета (`super.brightness()`), а затем применяет прозрачность. Если методу был передан цвет фона, а текущий цвет частично прозрачный, он учитывает яркость цвета фона. В Листинге 4–6 содержится полный на данном этапе код для класса `Color` и подкласса `ColorWithAlpha`, а также примеры использования метода `brightness`. Запустите его. Можно пройти этапы выполнения кода в отладчике, чтобы посмотреть выполнение в действии.

Листинг 4-6: Использование метода суперкласса с помощью super — using-superclass-method.js

```
class Color {
    constructor(r = 0, g = 0, b = 0) {
        this.r = r;
        this.g = g;
        this.b = b;
    }

    get rgb() {
        return "rgb(" + this.r + ", " + this.g + ", " + this.b + ")";
    }

    set rgb(value) {
        let s = String(value);
        let match = /^rgb\((\d{1,3}),(\d{1,3}),(\d{1,3})\)$/i.exec(
            s.replace(/\s/g, "")
    );
        if (!match) {
            throw new Error("Invalid rgb color string '" + s + "'");
        }
        this.r = parseInt(match[1], 10);
        this.g = parseInt(match[2], 10);
        this.b = parseInt(match[3], 10);
    }

    toString() {
        return this.rgb;
    }

    brightness() {
        return Math.sqrt(
            (this.r * this.r * 0.299) +
            (this.g * this.g * 0.587) +
            (this.b * this.b * 0.114)
        );
    }

    static fromCSS(css) {
        const match = /^#?([0-9a-f]{3}|[0-9a-f]{6});?$/i.exec(css);
        if (!match) {
            throw new Error("Invalid CSS code: " + css);
        }
        let vals = match[1];
        if (vals.length === 3) {
            vals = vals[0] + vals[0] + vals[1] + vals[1] + vals[2] + vals[2];
        }
        return new this(
            parseInt(vals.substring(0, 2), 16),
            parseInt(vals.substring(2, 4), 16),
            parseInt(vals.substring(4, 6), 16)
        );
    }
}
```

```
class ColorWithAlpha extends Color {
    constructor(r = 0, g = 0, b = 0, a = 1) {
        super(r, g, b);
        this.a = a;
    }

    brightness(bgColor) {
        let result = super.brightness() * this.a;
        if (bgColor && this.a !== 1) {
            result = (result + (bgColor.brightness() * (1 - this.a))) / 2;
        }
        return result;
    }

    toString() {
        return "rgba(" + this.r + ", " +
                         this.g + ", " +
                         this.b + ", " +
                         this.a + ")";
    }
}

// Начнаем с темно-серого цвета, полностью непрозрачного
const ca = new ColorWithAlpha(169, 169, 169);
console.log(String(ca));                        // "rgba(169, 169, 169, 1)"
console.log(ca.brightness());                   // 169
// Сделаем его полупрозрачным
ca.a = 0.5;
console.log(String(ca));                        // "rgba(169, 169, 169, 0.5)"
console.log(ca.brightness());                   // 84.5
// Яркость на голубом фоне
const blue = new ColorWithAlpha(0, 0, 255);
console.log(ca.brightness(blue));               // 63.774477345571015
```

Давайте рассмотрим, как можно определить метод `brightness` в ES5:

```
// Более старый (~ES5), почти эквивалентный
ColorWithAlpha.prototype.brightness = function brightness(bgColor) {
    var result = Color.prototype.brightness.call(this) * this.a;
    if (bgColor && this.a !== 1) {
        result = (result + (bgColor.brightness() * (1 - this.a))) / 2;
    }
    return result;
};
```

Выделенная строка с вызовом метода суперкласса `brightness` может быть записана несколькими способами. Этот способ явно ссылается на `Color` (суперкласс), что не идеально, поскольку вы можете изменить родительский класс при рефакторинге. Другой вариант — применить метод `Object.getPrototypeOf` к `ColorWithAlpha.prototype`:

```
    var superproto = Object.getPrototypeOf(ColorWithAlpha.prototype);
    var result = superproto.brightness.call(this) * this.a;
```

Или использовать метод `Object.getPrototypeOf` дважды:

```
var superproto = Object.getPrototypeOf(Object.getPrototypeOf(this));
var result = superproto.brightness.call(this) * this.a;
```

Все они неуклюжи, и управлять `this` приходится с помощью метода `call`. В ES2015+ выражение `super` обрабатывает это все за вас.

Этот синтаксис предназначен не только для методов, вы также можете использовать его для доступа к свойствам прототипа, не относящимся к методам. Но редко бывает, чтобы у объектов прототипа класса были свойства, отличные от метода. Еще реже возникает потребность получить к ним доступ напрямую, а не через экземпляр.

Наследование статических методов

Ранее вы узнали, как создать статический метод в классе. В JavaScript статические методы наследуются подклассами. Запустите Листинг 4-7, показывающий применение метода `fromCSS` (определенного в классе `Color`) к подклассу `ColorWithAlpha`.

Листинг 4-7: Доступ к статическим методам через подкласс — accessing-static-method-throughsubclass.js

```
class Color {
    // ...тот же код, что и в листинге 4-6...
}

class ColorWithAlpha extends Color {
    // ...тот же код, что и в листинге 4-6...
}

const ca = ColorWithAlpha.fromCSS("#1E90FF");
console.log(String(ca));                      // "rgba(30, 144, 255, 1)"
console.log(ca.constructor.name);             // "ColorWithAlpha"
console.log(ca instanceof ColorWithAlpha);    // истина
```

Обратите внимание на то, как можно вызвать метод `fromCSS` для подкласса `ColorWithAlpha`, и заметьте, что результат будет экземпляром подкласса `ColorWithAlpha`, а не экземпляром класса `Color`. Давайте посмотрим, почему это решение работает.

В начале этого раздела, посвященного подклассам, вы узнали, что использование оператора `extends` создает две цепочки наследования: одну для самого конструктора и одну для объекта `prototype` конструктора. См. рисунок 4-2, который представляет собой копию рисунка 4-1.

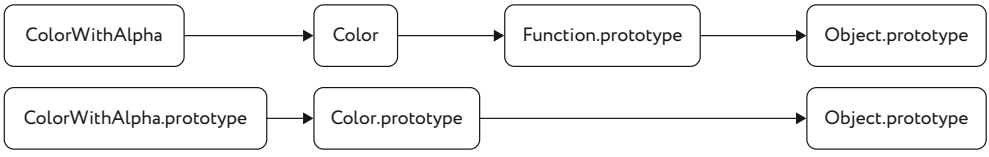


РИСУНОК 4-2

Цепочка наследования конструктора — это новый аспект в ES2015. Вплоть до ES5 не было стандартного способа создать настоящую функцию JavaScript, прототипом которой было бы что угодно, кроме `Function.prototype`. Но для конструктора подкласса имеет смысл использовать конструктор суперкласса в качестве своего прототипа, а значит, наследовать его свойства и методы.

Но почему же метод `fromCSS` создает объект подкласса `ColorWithAlpha`, а не объект класса `Color`, хотя определение метода относится к `Color`? Мы очень кратко коснулись этого в главе. Давайте снова посмотрим, как класс `Color` определяет метод `fromCSS`:

```
class Color {
    // ...

    static fromCSS(css) {
        const match = /^#?([0-9a-f]{3}|[0-9a-f]{6});?$/i.exec(css);
        if (!match) {
            throw new Error("Invalid CSS code: " + css);
        }
        let vals = match[1];
        if (vals.length === 3) {
            vals = vals[0] + vals[0] + vals[1] + vals[1] + vals[2] + vals[2];
        }
        return new this(
            parseInt(vals.substring(0, 2), 16),
            parseInt(vals.substring(2, 4), 16),
            parseInt(vals.substring(4, 6), 16)
        );
    }
}
```

Ключевым моментом здесь будет то, что класс создает объект с помощью метода `new this`, а не `new Color`. При вызове метода `fromCSS` для класса `Color`, значение `this` в вызове относится к классу `Color`. Но если вызвать этот метод для подкласса `ColorWithAlpha`, значение `this` внутри вызова будет относиться к подклассу `ColorWithAlpha`, таким образом, это конструктор, вызываемый с помощью метода `new this`. Попробуйте скопировать файл **accessing-static-method-through-subclass.js** из загружаемых и заменить метод на `new Color`.

Ключевое слово `super` в статических методах

Поскольку конструктор подкласса действительно наследуется от конструктора суперкласса, вы можете использовать выражение `super` в статическом методе подкласса для ссылки на версию суперкласса. Предположим, вы хотите разрешить использование второго аргумента в методе `fromCSS` для непрозрачности в подклассе `ColorWithAlpha` (Листинг 4-8). Реализация просто вызывает метод `super.fromCSS`, а затем добавляет непрозрачность к результирующему объекту.

Листинг 4-8: Использование ключевого слова super в статическом методе — super-in-static-method.js

```
class Color {
    // ...тот же код, что и в листинге 4-6...
}

class ColorWithAlpha extends Color {
    constructor(r = 0, g = 0, b = 0, a = 1) {
        super(r, g, b);
        this.a = a;
    }

    brightness(bgColor) {
        let result = super.brightness() * this.a;
        if (bgColor && this.a !== 1) {
            result = (result + (bgColor.brightness() * (1 - this.a))) / 2;
        }
        return result;
    }

    toString() {
        return "rgba(" + this.r + ", " +
                         this.g + ", " +
                         this.b + ", " +
                         this.a + ")";
    }

    static fromCSS(css, a = 1) {
        const result = super.fromCSS(css);
        result.a = a;
        return result;
    }
}

const ca = ColorWithAlpha.fromCSS("#1E90FF", 0.5);
console.log(String(ca));                    // "rgba(30, 144, 255, 0.5)"
```

Методы, возвращающие новые экземпляры

В этом разделе вы узнаете о шаблонах для создания новых экземпляров класса из его методов: статических, таких как `Color.fromCSS`, и методов экземпляра, таких как `slice` и `map` для массивов.

Вы уже видели статический метод, использующий один шаблон для создания экземпляра класса.

Метод `Color.fromCSS`, использующий выражение `new this(/*...*)` для создания нового экземпляра. Такая конструкция работает хорошо.

При вызове метода `Color.fromCSS` вы получаете экземпляр `Color`. При вызове `ColorWithAlpha.fromCSS` вы получаете экземпляр `ColorWithAlpha`. Это ожидаемое в большинстве случаев поведение. В методе экземпляра необходимо использовать `this.constructor`, а не `this`., поскольку `this.constructor` обычно ссылается на конструктор объекта. Например, вот метод класса `Color`, возвращающий цвет, яркость которого в два раза меньше оригинала:

```
halfBright() {
    const ctor = this.constructor || Color;
    return new ctor(
        Math.round(this.r / 2),
        Math.round(this.g / 2),
        Math.round(this.b / 2)
    );
}
```

Но, предположим, вы хотели создать подкласс `Color`, методы `fromCSS` и `halfBright` которого возвращают экземпляр `Color`. (Немного странно выглядит выполнение метода `fromCSS`, но давайте пока проигнорируем это.) Вам придется переопределить текущие реализации методов `fromCSS` и `halfBright` и использовать класс `Color` на протяжении всех этих строк:

```
class ColorSubclass extends Color {
    static fromCSS(css) {
        return Color.fromCSS(css);
    }
    halfBright() {
        return new Color(
            Math.round(this.r / 2),
            Math.round(this.g / 2),
            Math.round(this.b / 2)
        );
    }
}
```

Это нормально, если дело касается одного или двух методов. Но что делать, если класс `Color` содержал в себе несколько другие операции, создающие новые экземпляры, и вы хотели, чтобы они создали экземпляр класса `Color`, а не подкласса `ColorSubclass`? Вам придется переопределить их все. Это может привести к беспорядочности кода. (Подумайте обо всех методах массива, таких как `slice` и `map`, создающих новые массивы, и о том, как сложно было бы переопределить их все в подклассе массива, если вам потребуется поведение не по умолчанию.)

Если вы хотите, чтобы большинство методов, создающих новые экземпляры, использовали один и тот же конструктор таким образом, чтобы подклассы могли легко переопределять его, есть отличная альтернатива — `Symbol.species`.

Символы (Symbol) — это то, что мы еще не рассмотрели. Вы узнаете о них в главе 5. На данный момент запомните: они — новый вид примитивов, их можно использовать в качестве ключа свойства (как строки, но они не являются строками), и что существуют «хорошо известные», доступные в качестве свойств в функции примитивы `Symbol`, включая тот, который мы будем использовать здесь, — `Symbol.species`.

Он является частью шаблона, разработанного специально для того, чтобы позволить подклассам контролировать происходящее в методах, которым необходимо создавать новые экземпляры класса. В базовом классе (`Color` в этом примере) вместо использования конструктора у метода есть `this` от `this.constructor`, как мы и делали до сих пор. Методы, использующие шаблон вида, определяют конструктор для использования, просматривая свойство `Symbol.species` для `this` / `this.constructor`. Если результатом станет значение `null` или `undefined`, класс определяет выражение

по умолчанию, которое он использует вместо этого. В методе `fromCSS` это будет выглядеть так:

```
static fromCSS(css) {
    // ...
    let ctor = this[Symbol.species];
    if (ctor === null || ctor === undefined) {
        ctor = Color;
    }
    return new ctor(/*...*/);
}
```

Поскольку значения null и undefined лжеподобные, а функция конструктор по определению не может быть ложной, этот код можно немного упростить:

```
static fromCSS(css) {
    // ...
    const ctor = this && this[Symbol.species] || Color;
    return new ctor(/*...*/);
}
```

(Позже вы узнаете, почему мы используем там запасной вариант `Color`, а не `this`.)

Обратите внимание, что код проверяет, чтобы значение `this` было истинным (поскольку код класса в строгом режиме, `this` может быть присвоено любое значение, в том числе `undefined`).

Хотя опять же это может быть немного странно для статического метода (в зависимости от вашего варианта использования). Давайте посмотрим, как это сделать в методе прототипа `halfBright`:

```
halfBright() {
    const ctor = this && this.constructor &&
                 this.constructor[Symbol.species] || Color;
    return new Color(
        Math.round(this.r / 2),
        Math.round(this.g / 2),
        Math.round(this.b / 2)
    );
}
```

(В главе 19 вы узнаете о дополнительной возможности связывания, которая упростит первое утверждение в функции `halfBright`: `const ctor = this?.constructor?.[Symbol.species] || Color;`)

При использовании шаблона `Symbol.species` базовый класс определяет это свойство умным способом, как акцессор, возвращающий `this`:

```
class Color {
    static get [Symbol.species]() {
        return this;
    }
    // ...
}
```

Таким образом, если подкласс не переопределяет свойство `Symbol.species`, это будет просто, как если бы класс использовал `new this(/*...*/)`: будет использоваться конструктор, для которого был вызван метод (`Color` или `ColorWithAlpha`, в зависимости от контекста). Но если подкласс переопределяет свойство `Symbol.species`, вместо него используется конструктор из этого переопределения.

Складывая все эти кусочки воедино:

```
class Color {
    static get [Symbol.species]() {
        return this;
    }

    static fromCSS(css) {
        const ctor = this && this[Symbol.species] || Color;
        return new ctor(/*...*/);
    }

    halfBright() {
        const ctor = this && this.constructor &&
                     this.constructor[Symbol.species] || Color;
        return new ctor(/*...*/);
    }

    // ...
}
```

Объекты стандартной библиотеки используют этот шаблон только для методов прототипа и экземпляра, а не для статических методов. Это потому, что нет никакой веской причины для его использования в статических методах, и вы в итоге остаетесь со странным или ложным кодом, например, метод `ColorWithAlpha.fromCSS`, возвращающий экземпляр `Color` вместо `ColorWithAlpha`. Посмотрите пример в Листинге 4-9, более подробно описывающий, как стандартная библиотека использует шаблон вида.

Листинг 4-9: Использование символа Symbol.species — using-Symbol-species.js

```
class Base {
    constructor(data) {
        this.data = data;
    }

    static get [Symbol.species]() {
        return this;
    }

    static create(data) {
        // Не использует `Symbol.species`
        const ctor = this || Base;
        return new ctor(data);
    }

    clone() {
        // Использует `Symbol.species`
        const ctor = this && this.constructor &&
                     this.constructor[Symbol.species] || Base;
```

```
        return new ctor(this.data);
    }
}
// Sub1 использует поведение по умолчанию, которое обычно соответствует
вашим ожиданиям
class Sub1 extends Base {
}
// Sub2 заставляет любой метод, поддерживающий шаблон, использовать Base
вместо Sub2
class Sub2 extends Base {
    static get [Symbol.species]() {
        return Base;
    }
}

const a = Base.create(1);
console.log(a.constructor.name);          // "Base"
const aclone = a.clone();
console.log(aclone.constructor.name);     // "Base"

const b = Sub1.create(2);
console.log(b.constructor.name);          // "Sub1"
const bclone = b.clone();
console.log(bclone.constructor.name);     // "Sub1"

const c = Sub2.create(3);
console.log(c.constructor.name);          // "Sub2"
const d = new Sub2(4);
console.log(d.constructor.name);          // "Sub2"
console.log(d.data);                      // 4
const dclone = d.clone();
console.log(dclone.constructor.name);     // "Base"
console.log(dclone.data);                 // 4
```

Обратите внимание, что вызов метода `create` для `Sub1` создает экземпляр `Sub1`, и вызов метода `clone` для этого экземпляра тоже создает экземпляр `Sub1`. Вызов метода `create` для `Sub2` создает экземпляр `Sub2`, а вызов метода `clone` для `Sub2` создает экземпляр `Base`.

Ранее я обещал рассказать вам, почему мы используем явное значение по умолчанию в этом примере (`Base` и `Color` в предыдущих примерах), если значение свойства символа `Symbol.species` принимает значение `null` или `undefined`. Это то, что делают встроенные классы, такие как `Array`. Вместо этого они могли бы использовать `this` (в статических методах) или `this.constructor` (в методах прототипов) в качестве значения по умолчанию. Но, используя явное значение по умолчанию, они дают подклассам подклассов возможность запрашивать *исходное* значение по умолчанию, а не текущий конструктор подкласса. Так и с классом `Base` из предыдущего примера. В этом коде:

```
class Sub extends Base {
}

class SubSub1 extends Sub {
}
```

```
class SubSub2 extends Sub {
    static get [Symbol.species]() {
        return null;
    }
}

const x = new SubSub1(1).clone();
console.log(x.constructor.name); // "SubSub1"

const y = new SubSub2(2).clone();
console.log(y.constructor.name); // "Base", а не "SubSub2" или "Sub"
```

значения по умолчанию из класса `Base` применяется в методе `clone` подкласса `SubSub2`, а не данные из подкласса.

(Попробуйте сделать это с помощью `explicit-constructor-default.js` из перечня загрузок.)

Создание подклассов для встроенных компонентов

В ES5 некоторые встроенные конструкторы, такие как `Error` и `Array`, были заведомо недоступны для использования с правильным подклассом. Это было исправлено в ES2015. Создание подкласса встроенного компонента с помощью директивы `class` — это то же самое, что создание подкласса чего-либо еще. (Можно создавать подклассы и без синтаксиса `class` с помощью метода `Reflect.construct`; подробнее об этом в главе 14.) Давайте рассмотрим пример с массивом. В Листинге 4-10 показан очень простой класс `Elements`, расширяющий `Array` при помощи метода `style`, который вводит информацию по стилю для элементов DOM в массиве. Попробуйте запустить его на странице, содержащей по крайней мере три элемента `div` (в разделе загрузки есть файл **subclassingarray.html**, который вы можете использовать).

Листинг 4-10: Создание подкласса для элемента Array — subclassing-array.js

```
class Elements extends Array {
    select(source) {
        if (source) {
            if (typeof source === "string") {
                const list = document.querySelectorAll(source);
                list.forEach(element => this.push(element));
            } else {
                this.push(source);
            }
        }
        return this;
    }

    style(props) {
        this.forEach(element => {
            for (const name in props) {
                element.style[name] = props[name];
            }
        });
        return this;
    }
}
```

```
// Применение
new Elements()
.select("div")
.style({color: "green"})
.slice(1)
.style({border: "1px solid red"});
```

Код «Применение» в конце:

- Создает экземпляр класса `Elements`, являющегося подклассом `Array`.
- Использует свой метод `select`, чтобы добавить все элементы `div` на странице к экземпляру (используя метод массива `push`, чтобы добавить их).
- Стилизует эти элементы зеленым текстом.
- Создает новый экземпляр класса `Elements` с помощью метода массива `slice`, чтобы получить только второй и третий элементы `div`.
- Добавляет красную рамку вокруг этих элементов `div` с помощью метода `style`.

(Обратите внимание, как применение `slice` создает новый экземпляр класса `Elements`, а не просто новый экземпляр класса `Array`. Это происходит потому, что класс `Array` использует шаблон `Symbol.species`, изученный вами в предыдущем разделе, и класс `Elements` не переопределяет данные по умолчанию.)

Конечно, класс `Elements` — это всего лишь пример. В реальной реализации вы, вероятно, передали бы селектор в конструктор `Elements`, а не использовали отдельный вызов `select` для последующего заполнения экземпляра. Чтобы сделать это надежно, требуется использовать функцию, о которой вы еще не знаете (итеративный синтаксис `spread` в главе 6). Если вы хотите посмотреть, как это будет выглядеть (также с использованием `for-of` [глава 6], `Object.entries` [глава 5] и деструктуризация [глава 7]), посмотрите файл **subclassing-array-usinglater-chapter-features.js** в разделе загрузки.

Где доступен `super`

До появления ES2015 мы свободно использовали термин «*метод*» в JavaScript для обозначения любой функции, назначенной свойству объекта (или только тем, которые используют `this`). Начиная с ES2015, хотя это все еще распространено неофициально, существует различие между настоящими методами и функциями, назначенными свойствам. Код внутри настоящего метода имеет доступ к ключевому слову `super`; код в традиционной функции, назначенной свойству, не может получить такой доступ. Давайте докажем это самим себе с помощью Листинга 4-11: прочитайте его и попробуйте запустить.

Листинг 4-11: Метод в сравнении с функцией — method-vs-function.js

```
class SuperClass {
    test() {
        return "SuperClass's test";
    }
}
class SubClass extends SuperClass {
    test1() {
        return "SubClass's test1: " + super.test();
```

```
    }
}
SubClass.prototype.test2 = function() {
    return "SubClass's test2: " + super.test(); // Здесь выбрасывается ошибка
};

const obj = new SubClass();
obj.test1();
obj.test2();
```

Движок JavaScript откажется проводить парсинг этого кода, жалуясь, что применение `super` не ожидалось в строке, отмеченной в листинге. Но почему нет?

Потому что у методов есть ссылка на объект, для которого они созданы, — но у традиционных функций, назначенных свойствам, ее нет. При использовании в операции поиска свойств, такой как `super.foo`, выражение `super` полагается на внутреннее поле содержащей функции с именем [[HomeObject]]. Механизм JavaScript получает объект из поля [[HomeObject]] метода, получает его прототип, а затем ищет свойство `method` для этого объекта, например, этот псевдокод:

```
// Псевдокод
let method = (the running method);
let homeObject = method.[[HomeObject]];
let proto = Object.getPrototypeOf(homeObject);
let value = proto.foo;
```

«Но подождите, — скажете вы. — Почему выражение `super` должно волновать, где был определен метод? Оно просто относится к прототипу `this`, верно? Или прототипу прототипа `this`?»

Нет, так не может быть. Чтобы понять, почему, рассмотрим трехуровневую иерархию в Листинге 4-12.

Листинг 4-12: Трехуровневая иерархия — three-layer-hierarchy.js

```
class Base {
    test() {
        return "Base test";
    }
}
class Sub extends Base {
    test() {
        return "Sub test > " + super.test();
    }
}
class SubSub extends Sub {
    test() {
        return "SubSub test > " + super.test();
    }
}

// Применение:
const obj = new SubSub();
console.log(obj.test()); // SubSub test > Sub test > Base test
```

Когда вы создаете этот экземпляр `obj`, его цепочка прототипов выглядит так, как показано на рисунке 4-3.

РИСУНОК 4-3

Предположим, выражение `super` работало, отталкиваясь от `this`, зная, что нужно, чтобы прототип прототипа `this` использовался в методе прототипа. Когда вы вызываете `obj.test`, `this` соответствует `obj`. Когда используется метод `super.test()`, принцип такой: прототип `obj` — это `SubSub.prototype`, а прототип `SubSub.prototype` — это `Sub.prototype`, следовательно, используется `test` из `Sub.prototype`. Пока все хорошо. Потом код вызывает `test` из `Sub.prototype` с `this`, заданным `obj`. Как часть его кода `Sub.prototype.test` *тоже* вызывает метод `super.test()`. И теперь движок JavaScript застрял. Он не может сделать то же самое, что сделал, чтобы перейти от `SubSub` к `Sub`, поскольку `this` все еще равно `obj`. Следовательно, `Sub.prototype` все еще прототип прототипа `this`, и выполнение просто закончится в том же самом месте. Просмотрите код в Листинге 4-13 и запустите его. В конечном счете он вызывает ошибку переполнения стека, потому что метод `Sub.prototype.test` продолжает вызывать сам себя.

Листинг 4-13: Трехуровневая иерархия — three-layer-hierarchy.js

```
function getFakeSuper(o) {
    return Object.getPrototypeOf(Object.getPrototypeOf(o));
}
class Base {
    test() {
        console.log("Base's test");
        return "Base test";
    }
}
class Sub extends Base {
    test() {
        console.log("Sub's test");
        return "Sub test > " + getFakeSuper(this).test.call(this);
    }
}
class SubSub extends Sub {
    test() {
        console.log("SubSub's test");
        return "SubSub test > " + getFakeSuper(this).test.call(this);
    }
}

// Применение:
const obj = new SubSub();
console.log(obj.test()); // "SubSub's test", затем "Sub's test" повторно
                         // до тех пор, пока не возникнет ошибка
                         // переполнения стека
```

Вот почему методы должны содержать поле, указывающее, на каком объекте они определены (их [[Home Object]]), чтобы движок JavaScript мог получить этот объект и его прототип для доступа к `super` версии `test`. См. рисунок 4-4, чтобы проследить следующее объяснение: движок JavaScript ищет [[HomeObject]] в `obj.test`, представляющий `SubSub.prototype`, для выполнения метода `super.test()`, затем получает прототип этого объекта (`Sub.prototype`) и вызывает метод `test` для него. Движок получает [[HomeObject]] из `Sub.prototype.test` для обработки метода `super.test()` в `Sub.prototype.test` и получает *его* прототип `Base.prototype`, а затем вызывает метод `test` для него.

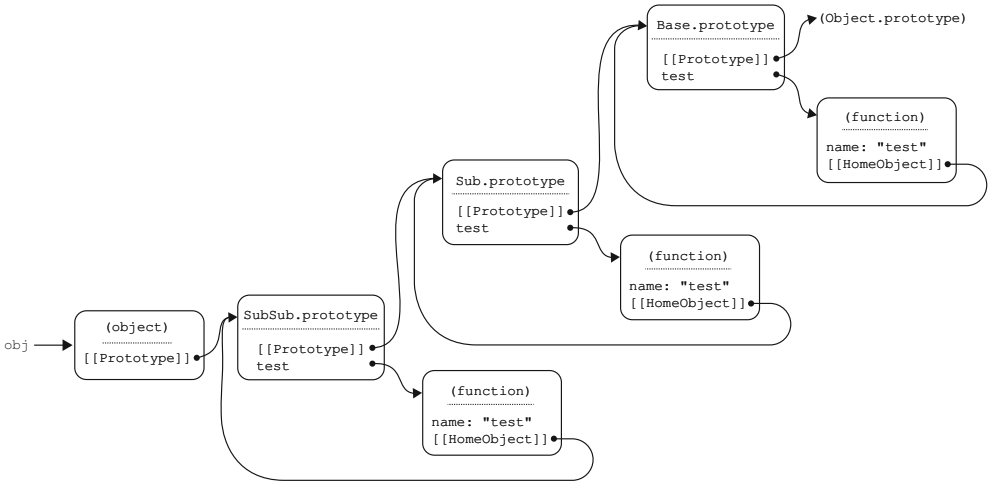


РИСУНОК 4-4

Из этого вытекает важное следствие: копирование методов от объекта к объекту (знакомый шаблон с «миксинами» (примесями) — объектами со служебными функциями, которые вы копируете в несколько прототипов) не изменяет свойство [[HomeObject]] метода. Если вы используете ключевое слово `super` в методе с миксинами, объект продолжает использовать прототип своего *исходного* домашнего объекта, а не прототип объекта, в который он был скопирован. Это может создать тонкие перекрестные помехи между исходным домашним объектом метода и объектом, на который вы его скопировали. Если вы делаете это специально (используя функцию исходного родительского элемента миксина) — это нормально. Но если вы используете ключевое слово `super` в методе с миксином, ожидая, что он будет работать в иерархии своего нового домашнего объекта, это станет ошибкой.

На этапе разработки ES2015 существовала функция, которую можно было использовать для изменения свойства [[HomeObject]] метода (в частности, для включения миксинов), но ее удалили до завершения спецификации и не добавили обратно (пока). Так что, по крайней мере, на данном этапе определите миксины с традиционными функциями или не используйте в них выражение с `super`. Если вы все-таки решите реализовать такой подход, используйте его для продолжения работы объекта в его первоначальной иерархии, а не в новой.

ОТКАЗ ОТ OBJECT.PROTOTYPE

По умолчанию даже базовый класс фактически является подклассом `Object`: его экземпляры наследуются от `Object.prototype`. Вот где они получают свои методы по умолчанию: `toString`, `hasOwnProperty` и другие подобные методы. То есть следующие два класса фактически одинаковы:

```
class A {
    constructor() {
    }
}
class B extends Object {
    constructor() {
        super();
    }
}
```

(Единственное небольшое отличие состоит в том, что прототип функции `A` — это `Function.prototype`, но прототип функции `B` — это `Object`. Хотя `A.prototype` и `B.prototype` — прототипы `Object.prototype`.)

Но, предположим, вам не нужно, чтобы экземпляры вашего класса наследовались от `Object.prototype`? То есть вы не хотите, чтобы у них по умолчанию были методы `toString`, `hasOwnProperty` и подобные. Такая необходимость возникает очень редко, но вы должны уметь реализовать это с помощью выражения `extends null`:

```
class X extends null {
}
const o = new X();
console.log(o.toString); // undefined
console.log(Object.getPrototypeOf(X.prototype) === null); // истина
```

Выражение `extends null` указывает движку JavaScript использовать `null` в качестве прототипа объекта `X.prototype` вместо `Object.prototype`, как это обычно бывает.

Однако, хотя цель этой функции всегда была ясна, существовали незначительные проблемы с точными техническими деталями в отношении выражения `extends null` в ES2015 и последующих спецификациях, что вызывало проблемы при реализации. В результате на момент написания книги выражение `extends null` еще не работает в большинстве основных движков JavaScript: выбрасывается ошибка при попытке выполнения команды `new X`. Если у вас возникает вариант, требующий применения `extends null`, обязательно протестируйте его в целевых средах. (Но вы все равно всегда так делаете, правда?)

СИНТАКСИС NEW.TARGET

Функции и конструкторы могут быть вызваны двумя способами:

- Напрямую (хотя конструкторы, созданные с помощью синтаксиса `class`, запрещают это).

- В рамках создания объекта (при помощи `new`, `super` или `Reflect.construct` [рассматриваются в главе 14]).

Иногда важно знать, как была вызвана ваша функция. Возможно, вы хотите сделать класс абстрактным (разрешать создавать экземпляры только как часть подкласса) или окончательным (не разрешать подклассы). Возможно, вам требуется функция, изменяющая результат выполнения, в зависимости от того, была ли она вызвана с помощью `new` или напрямую.

В этих случаях вы можете использовать синтаксис `new.target`, чтобы узнать, как была вызвана функция. Если функция вызывается напрямую (не через `new`), `new.target` получит значение `undefined`:

```
function example() {
    console.log(new.target);
}
example(); // undefined
```

Если текущая функция была прямой целью оператора `new`, `new.target` относится к текущей функции:

```
class Base {
    constructor() {
        console.log(new.target.name);
    }
}
new Base(); // "Base"
```

Если текущая функция была вызвана `super`, `new.target` ссылается на конструктор подкласса, который был целью `new`:

```
class Base {
    constructor() {
        console.log(new.target.name);
    }
}

class Sub extends Base {
    // (Это выражение было бы конструктором по умолчанию, но я включил
    // его для ясности, чтобы явно показать вызов `super()`.)
    constructor() {
        super();

    }

new Sub();  // "Sub"
```

Другие способы использования `Reflect.construct` описаны в главе 14.

Имея это в виду, давайте использовать его для всех трех сценариев, упомянутых в разделе введение: абстрактный класс, окончательный класс и функция, которая выполняет разные действия в зависимости от того, была ли она вызвана как конструктор или как функция.

Листинг 4-14 определяет абстрактный класс под названием `Shape` и два подкласса — `Triangle` и `Rectangle`. Абстрактный класс может быть создан напрямую, только через подкласс. Класс `Shape` делает абстрактный класс, выбрасывая ошибку, если выражение в конструкторе `new.target === Shape` истинно: это будет означать, что он создавался с помощью `new Shape` (или эквивалентного выражения). При создании подкласса `Triangle` (например) `new.target` относится к подклассу `Triangle`, а не классу `Shape`, так что конструктор не выдаст сообщение об ошибке.

Листинг 4-14: Абстрактный класс — abstract-class.js

```
class Shape {
    constructor(color) {
        if (new.target === Shape) {
            throw new Error("Shape can't be directly instantiated");
        }
        this.color = color;
    }
    toString() {
        return "[" + this.constructor.name + ", sides = " +
            this.sides + ", color = " + this.color + "]";
    }
}
class Triangle extends Shape {
    get sides() {
        return 3;
    }
}
class Rectangle extends Shape {
    get sides() {
        return 4;
    }
}
const t = new Triangle("orange");
console.log(String(t));     // "[Triangle, sides = 3, color = orange]"

const r = new Rectangle("blue");
console.log(String(r));     // "[Rectangle, sides = 4, color = blue]"

const s = new Shape("red"); // Ошибка: "Форма не может быть создана напрямую"
```

(Даже с проверкой можно создать объект, который непосредственно использует `Shape.prototype` в качестве своего прототипа, так как нет необходимости вызывать `Shape` для этого [вы можете использовать `Object.create`]. Но это предполагает, что кто-то активно работает над тем, как определяется использование класса `Shape`.)

Окончательный класс является обратным этому: он *запрещает* экземпляры подклассов. В Листинге 4-15 это делается противоположно проверке `Shape`: она выдает ошибку, если выражение `new.target` не равно собственному конструктору класса. При попытке создать экземпляр `InvalidThingy` вы получаете сообщение об ошибке. (К сожалению, в настоящее время невозможно сделать эту ошибку более упреждающей, выбрасывая ее при вычислении определения класса `InvalidThingy`, а не позже, когда используется `new InvalidThingy`.)

Листинг 4-15: Окончательный класс — final-class.js

```
class Thingy {
    constructor() {
        if (new.target !== Thingy) {
            throw new Error("Thingy subclasses aren't supported.");
        }
    }
}
class InvalidThingy extends Thingy {
}

console.log("Creating Thingy...");
const t = new Thingy();          // Это работает
console.log("Creating InvalidThingy...");
const i = new InvalidThingy();  // Ошибка: "Подклассы Thingy не поддерживаются".
```

(Как в примере класса `Shape`, с помощью `Object.create` все еще возможно создавать объекты с прототипом, наследующим `Thingy.prototype` — и, следовательно, в некотором роде объекты подкласса. Но, опять же, это предполагает активную работу с `Thingy`.)

Наконец, в Листинге 4-16 определяется функция, выполняющая разные действия в зависимости от того, была ли она вызвана как функция или через `new`. Обратите внимание: вы можете сделать это только с помощью традиционной функции, поскольку конструкторы, определенные при помощи `class`, могут быть вызваны только как часть построения объекта. Этот пример обнаруживает, что функция не была вызвана через `new` (или эквивалентное выражение), и преобразует код в вызов с помощью `new`.

Листинг 4-16: Функция, изменяющаяся в зависимости от того, как она вызывается — function-or-constructor.js

```
const TwoWays = function TwoWays() {
    if (!new.target) {
        console.log("Called directly; using 'new' instead");
        return new TwoWays();
}
    console.log("Called via 'new'");
};
console.log("With new:");
let t1 = new TwoWays();
// "Вызвана с помощью 'new'"

console.log("Without new:");
let t2 = TwoWays();
// "Вызывается напрямую; вместо этого используется 'new'"
// "Вызвана с помощью <new>"
```

ОБЪЯВЛЕНИЯ КЛАССОВ В СРАВНЕНИИ С ВЫРАЖЕНИЯМИ КЛАССОВ

Синтаксис `class`, как и `function`, может использоваться как объявление или как выражение. И, как и `function`, он зависит от контекста: если синтаксис появляется там, где допустимо либо утверждение, либо выражение, это объявление; если синтаксис появляется там, где допустимо только выражение, это выражение.

```
// Объявление
class Class1 {
}

// Анонимное выражение класса
let Color = class {
};

// Именованое выражение класса
let C = class Color {
};
```

Объявления классов

Объявления при помощи синтаксиса `class` работают аналогично объявлением с синтаксисом `function`, хотя и с важным отличием.

Общее у объявлений `function` и `class`:

- Добавление имени класса в текущую область.
- Не требуют точки с запятой после закрывающей фигурной скобки (Можно и написать точку с запятой, потому что JavaScript игнорирует ненужные точки с запятой, но в объявлении это не требуется.).

В *отличие* от объявлений функций, объявления `class`:

- Не поднимаются, они только наполовину проходят поднятие: идентификатор зарезервирован по всей области видимости, но не инициализируется до тех пор, пока объявление не будет достигнуто в пошаговом выполнении кода.
- Участвуют во Временной мертвой зоне.
- Не создают свойство глобального объекта для имени класса, если оно используется в глобальной области видимости. Вместо этого они создают глобальные переменные, которые не являются свойствами глобального объекта.

Эти последние фрагменты, вероятно, кажутся вам знакомыми. Это потому, что они совпадают с правилами, которые вы изучили в главе 2 для директив `let` и `const`. Эти правила применимы и к объявлениям с помощью `class`.

В Листинге 4-17 демонстрируются различные аспекты объявления через `class`. (Код должен выполняться в глобальной области видимости, чтобы продемонстрировать, что происходит с объявлениями через `class` в глобальной области видимости. Помните, что вы не можете запустить код в Node.js обычным способом. Необходимо использовать браузер, либо запустить его в REPL-среде Node.js. См. главу 1.)

Листинг 4-17: Объявления классов — declarations.js

```
// Попытка использовать здесь `TheClass` приведет к ошибке ReferenceError
// из-за Временной мертвой зоны

let name = "foo" + Math.floor(Math.random() * 1000);

// Объявление
class TheClass {
    // Поскольку объявление обрабатывается как часть
    // пошагового кода, здесь можно использовать `name` и быть уверенными,
    // что у класса будет присвоенное выше значение
    [name]() {
        console.log("This is the method " + name);
    }
} // <== Здесь не ожидается точки с запятой

// Был создан глобальный класс
console.log(typeof TheClass);      // "функция"

// Но нет свойства для глобального объекта
console.log(typeof this.TheClass); // "undefined"
```

Выражения классов

Выражения `class` работают аналогично выражениям `function`:

- У них есть как именованные, так и анонимные формы.
- Они не добавляют имя класса в область, в которой отображаются, но делают имя класса доступным в самом определении класса (если у него есть имя).
- Они приводят к значению (конструктору класса), которое может быть присвоено переменной или константе, передано в функцию или проигнорировано.
- Движок JavaScript выведет значение свойства `name` для класса, созданного с помощью анонимного выражения класса, из контекста, используя те же правила, которые вы изучили в главе 3 для выражений анонимных функций.
- При использовании в качестве правой части присваивания они не завершают выражение присваивания, если выражение класса стоит последним в выражении присваивания, за ним должна следовать точка с запятой. (Во многих случаях автоматическая вставка точки с запятой [ASI, Automatic Semicolon Insertion] может исправить ошибку пропуска точки с запятой, но не всегда.)

В Листинге 4-18 демонстрируются аспекты выражения `class`.

Листинг 4-18: Выражения классов — class-expressions.js

```
let name = "foo" + Math.floor(Math.random() * 1000);

// Выражение
const C = class TheClass {
    [name]() {
        console.log("This is the method " + name +
                    " in the class " + TheClass.name);
```

```
        // Имя класса в области -^
        // в рамках определения
    }
}; // <== Здесь необходима точка с запятой (хотя ASI предоставит ее
   //     при возможности)

// Имя класса не было добавлено в эту область
console.log(typeof TheClass);       // "undefined"

// Значением выражения является класс
console.log(typeof C);              // "функция"
```

ЕЩЕ НЕ ВСЕ

В ближайшее время в определениях `class` появится еще несколько функций (вероятно, в ES2021): объявления полей (свойств) открытого класса, частные поля, частные методы, публичные и частные статические поля и частные статические методы. Многие из них вы можете использовать сегодня с помощью транспиляции. Все это описано в главе 18.

ОТ СТАРЫХ ПРИВЫЧЕК К НОВЫМ

В этой главе всего один вариант «от старой привычки к новой».

Использование класса при создании функций конструктора

Старая привычка: Использовать традиционный синтаксис функций для создания функций-конструкторов (потому что у вас не было выбора!):

```
var Color = function Color(r, g, b) {
    this.r = r;
    this.g = g;
    this.b = b;
};
Color.prototype.toString = function toString() {
    return "rgb(" + this.r + ", " + this.g + ", " + this.b + ")";
};
console.log(String(new Color(30, 144, 255)));
```

Новая привычка: Используйте синтаксис `class`:

```
class Color {
    constructor(r, g, b) {
        this.r = r;
        this.g = g;
        this.b = b;
}
    toString() {
        return "rgb(" + this.r + ", " + this.g + ", " + this.b + ")";
    }
};
console.log(String(new Color(30, 144, 255)));
```

Это не означает, что необходимо начать использовать `class`, если вы *не* используете функции конструктора. Функции конструктора и их свойство `prototype` (как в ES5, так и в ES2015 +) — это лишь один из способов использования прототипического наследования JavaScript. Но если вы используете функции конструктора, то, учитывая все преимущества лаконичности, выразительности и функциональности, использование синтаксиса `class` явно выгодно.

5

Объекты

СОДЕРЖАНИЕ ГЛАВЫ

- Вычисляемые имена свойств
- Стенография свойств
- Получение и настройка прототипа объекта
- Свойство `__proto__` в браузерах
- Синтаксис метода и `super` вне классов
- Тип данных Symbol
- Новые полезные функции объектов
- Метод `Symbol.toPrimitive`
- Порядок свойств
- Синтаксис расширения свойств

В этой главе вы узнаете о новых функциях объектов в ES2015+, от функций, помогающих писать менее повторяющийся или более лаконичный код, до функций, позволяющих делать что-то новое.

ВЫЧИСЛЯЕМЫЕ ИМЕНА СВОЙСТВ

В ES5, если вы хотели создать объект со свойством, имя которого взято из переменной, вам нужно было сначала создать объект, а затем добавить к нему свойство как отдельную операцию. Например:

```
var name = "answer";
var obj = {};
obj[name] = 42;
console.log(obj); // {answer: 42}
```

Это немного неудобно, поэтому в ES2015 добавили вычисляемые имена свойств, использующие квадратные скобки (`[]`). Такие же квадратные скобки использовались в предыдущем коде, в самом определении свойства:

```
var name = "answer";
var obj = {
    [name]: 42
};
console.log(obj); // {answer: 42}
```

(Здесь я использовал `var`, чтобы подчеркнуть, что это не связано с директивой `let`, но далее я вернусь к использованию директив `let` и `const`.)

Скобки в определении свойства работают точно так же, как скобки, которые вы всегда использовали при получении/настройке значений свойств. Можно использовать любое выражение в квадратных скобках, и его результат будет использоваться в качестве имени:

```
let prefix = "ans";
let obj = {
    [prefix + "wer"]: 42
};
console.log(obj); // {answer: 42}
```

Выражение вычисляется немедленно, как часть вычисления литерала объекта, при вычислении каждого определения свойства (в порядке исходного кода). Чтобы проиллюстрировать это, следующий пример делает в точности то же, что и предыдущий, за исключением наличия дополнительных временных переменных `temp` и `name`:

```
let prefix = "ans";
let temp = {};
let name = prefix + "wer";
temp[name] = 42;
let obj = temp;
console.log(obj); // {answer: 42}
```

Вы можете увидеть порядок в действии здесь:

```
let counter = 0;
let obj = {
    [counter++]: counter++,
    [counter++]: counter++
};
console.log(obj);        // {"0": 1, "2": 3}
```

Обратите внимание на порядок, в котором переменная `counter` использовалась и увеличивалась: сначала для получения имени первого свойства (0, которое преобразуется в «0» при использовании в качестве имени свойства), после для получения значения этого свойства (1), а затем для получения имени для второго свойства (2, которое преобразуется в «2»), и, наконец, для получения значения этого свойства (3).

СТЕНОГРАФИЯ СВОЙСТВ

Довольно часто возникает желание создать объект со свойствами, значения которых берутся из переменных (или других идентификаторов области видимости, например, параметров) с тем же именем. Например, предположим, у вас есть функция `getMinMax`,

которая находит минимальное и максимальное число в массиве и возвращает объект со свойствами `min` и `max`. Версия ES5 обычно выглядит так:

```
function getMinMax(nums) {
    var min, max;
    // (опущено - нахождение `min` и `max`, перебором массива `nums`)
    return {min: min, max: max};
}
```

Обратите внимание на повторение при создании объекта в конце, когда приходится дважды указывать `min` и `max` (один раз для имени свойства и повторно — для его значения).

Начиная с ES2015 вы можете использовать *стенографию (сокращение) свойств*. Вы просто указываете идентификатор один раз, и он используется как для присвоения имени свойству, так и для определения того, откуда можно получить значение свойства:

```
function getMinMax(nums) {
    let min, max;
    // (опущено - нахождение `min` и `max`, перебором массива `nums`)
    return {min, max};
}
```

Естественно, сделать это можно только тогда, когда значение поступает из простого идентификатора в области видимости (переменная, параметр и т. д.). Если значение поступает из результата любого другого выражения, вам все равно нужно использовать форму `name: expression`.

ПОЛУЧЕНИЕ И НАСТРОЙКА ПРОТОТИПА ОБЪЕКТА

Всегда было возможно создать объект с наследованием от определенного объекта-прототипа с помощью функции конструктора. В ES5 была добавлена возможность делать это напрямую при помощи метода `Object.create` и получать прототип объекта через метод `Object.getPrototypeOf`. В ES2015 появилась возможность установки прототипа существующего объекта. Это очень необычное желание, но теперь это возможно.

Метод `Object.setPrototypeOf`

Основной способ сделать это — использовать метод `Object.setPrototypeOf`, который принимает объект для изменения и прототип для его предоставления. Взгляните на пример:

```
const p1 = {
    greet: function() {
        console.log("p1 greet, name = " + this.name);
    }
};
const p2 = {
    greet: function() {
        console.log("p2 greet, name = " + this.name);
    }
};
```

```
const obj = Object.create(p1);
obj.name = "Joe";
obj.greet(); // p1 greet, name = Joe
Object.setPrototypeOf(obj, p2);
obj.greet(); // p2 greet, name = Joe
```

Выполнение `obj` в этом коде начинается с использования `p1` в качестве прототипа объекта. Следовательно, первый вызов метода `obj.greet()` использует `greet` из `p1` и отображает `"p1 greet, name = Joe"`. Затем код изменяет прототип `obj` на `p2`, и следовательно второй вызов `greet` использует `greet` из `p2` и отображает `"p2 greet, name = Joe"`.

Опять же, изменение прототипа объекта после его создания необычно и вполне может привести к деоптимизации объекта, что значительно замедлит поиск свойств в нем. Но, если вам абсолютно необходимо это сделать, теперь это можно сделать с помощью стандартного механизма.

Свойство `__proto__` в браузерах

В среде браузера вы можете использовать свойство-акцессор с именем `__proto__` для получения и установки прототипа объекта, но не делайте этого в новом коде. Официально он не определен для отсутствующих в браузере движков JavaScript (хотя движок может предоставить его в любом случае).

> **ПРИМЕЧАНИЕ**
> Свойство __proto__ — это устаревшая функция, она была стандартизирована только для официального описания поведения, которое у нее использовалось как нестандартное расширение. В новом коде вместо него используйте getPrototypeOf и setPrototypeOf.

Вот пример использования свойства `__proto_`. Это тот же код, что и пример из предыдущего раздела с использованием метода `Object.setPrototypeOf`, единственное изменение — предпоследняя строка:

```
const p1 = {
    greet: function() {
        console.log("p1 greet, name = " + this.name);
    }
};
const p2 = {
    greet: function() {
        console.log("p2 greet, name = " + this.name);
    }
};
const obj = Object.create(p1);
obj.name = "Joe";
obj.greet(); // p1 greet, name = Joe
obj.__proto__ = p2;
obj.greet(); // p2 greet, name = Joe
```

Свойство-акцессор `__proto__` определяется выражением `Object.prototype`. Таким образом, объект, для которого вы его используете, должен наследовать от `Object.prototype` (прямо или косвенно), чтобы вы могли его использовать. Например, объект, созданный с помощью `Object.create(null)`, не содержит свойства `__proto__`, как и любой объект, использующий этот объект в качестве своего прототипа.

Буквальное указание имени свойства `__proto__` в браузерах

Свойство `__proto__` может также использоваться в литерале объекта для установки прототипа результирующего объекта:

```
const p = {
    hi() {
        console.log("hi");
    }
};
const obj = {
    __proto__: p
};
obj.hi(); // "hi"
```

Это явный синтаксис, а *не* побочный продукт свойства-акцессора `__proto__`, и он работает только тогда, когда указан буквально. Например, вы не можете использовать для этого вычисляемое имя свойства:

```
const p = {
    hi() {
        console.log("hi");
    }
};
const name = "__proto__";
const obj = {
    [name]: p
};
obj.hi(); // TypeError: obj.hi не является функцией
```

И, опять же, не используйте свойство `__proto__` в новом коде. В первую очередь применяйте `Object.create` для создания объекта с правильным прототипом. Еще можно использовать `Object.getPrototypeOf` и `Object.setPrototypeOf`, если необходимо получить или установить прототип позже. Таким образом, вам не нужно беспокоиться о том, выполняется ли код в браузере или нет, и (в случае свойства-акцессора) наследует ли объект от `Object.prototype` или нет.

СИНТАКСИС МЕТОДА И ПРИМЕНЕНИЕ `SUPER` ВНЕ КЛАССОВ

В главе 4 вы узнали об определениях методов в конструкциях `class`. ES2015 также добавляет синтаксис метода к литералам объектов. Там, где раньше вы написали бы что-то многословное с ключевым словом `function`, например:

```
var obj1 = {
    name: "Joe",
```

```
    say: function() {
        console.log(this.name);
    }
};
obj1.say(); // "Joe"
```

новый синтаксис метода позволяет вам написать так:

```
const obj2 = {
    name: "Joe",
    say() {
        console.log(this.name);
    }
};
obj2.say(); // "Joe"
```

То есть вы полностью исключаете двоеточие и ключевое слово `function`.

Синтаксис метода — это не просто сокращенный синтаксис для определения функций в литералах объектов, хотя в большинстве случаев алгоритм их действия схож. Как и в классах, синтаксис метода выполняет как больше, так и меньше, чем инициализируемое с помощью традиционной функции свойство:

- У метода нет свойства `prototype` с объектом на нем, и метод нельзя использовать в качестве конструктора.
- Метод получает ссылку на объект, для которого он определен, — его *домашний объект*. В примере с использованием традиционной функции (`obj1`) была создана только ссылка от объекта к функции через свойство объекта `say`; см. рисунок 5-1. Метод синтаксиса в примере создает связь в обе стороны: от объекта (`obj2`) к функции (через свойство `say`), и от функции к ее домашнему объекту (`obj2`) через поле [[HomeObject]], о которым вы узнали в главе 4; см. рисунок 5-2.

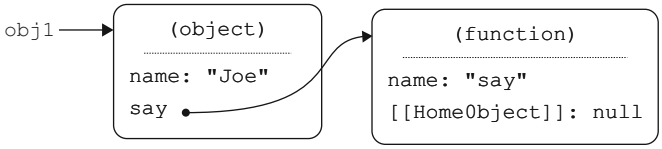


РИСУНОК 5-1

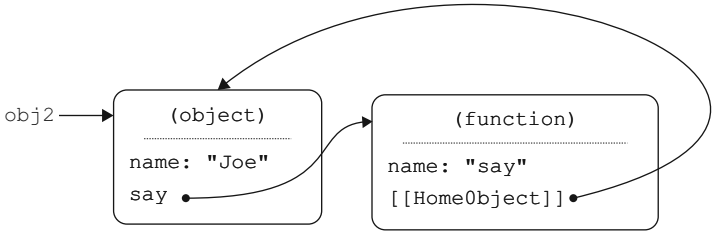


РИСУНОК 5-2

Цель ссылки из метода обратно на объект — поддержка использования `super` внутри метода. Предположим, вы хотели создать объект, функция `toString` которого использовала `toString` своего прототипа, но потом сделали все это заглавными буквами. В ES5 вам пришлось бы явно вызывать функцию прототипа, например:

```
var obj = {
    toString: function() {
        return Object.prototype.toString.call(this).toUpperCase();
    }
};
console.log(obj.toString()); // "[OBJECT OBJECT]"
```

В ES2015+ вместо этого можно использовать синтаксис метода и ключевое слово `super`:

```
const obj = {
    toString() {
        return super.toString().toUpperCase();
    }
};
console.log(obj.toString()); // "[OBJECT OBJECT]"
```

Имя метода не обязательно должно быть буквальным идентификатором. Как и в случае с ключами свойств, оно может быть строкой (с использованием двойных или одинарных кавычек) или вычисляемым именем:

```
const s = "ple";
const obj = {
    "biz-baz" () {      // Буквальное строковое имя метода
        console.log("Ran biz-baz");
    },
    ["exam" + s]() {   // Вычисляемое имя метода
        console.log("Ran example");
    }
};
obj["biz-baz"](); // "Запущен biz-baz"
obj.example();    // "Запущен example"
```

На первый взгляд ключевое слово `super` в методах литералов объектов кажется довольно ограниченным: прототипом объекта будет `Object.prototype`, так как он создается объектным литералом. Но, как описано ранее, в ES2015+ можно изменить прототип объекта после его создания, а в браузерах вы можете использовать свойство `__proto__` в литерале, чтобы установить для прототипа начальное значение, отличное от `Object.prototype` (хотя лучше не делать этого в новом коде).

Способ реализации `super` гарантирует, что, если вы это сделаете, он будет работать как ожидалось:

```
const obj = {
    toString() {
        return super.toString().toUpperCase();
    }
};
```

```
Object.setPrototypeOf(obj, {
    toString() {
        return "a different string";
    }
});
console.log(obj.toString()); // "A DIFFERENT STRING"
```

Вот тут-то и появляется [[HomeObject]]. Способ, которым движок JavaScript обрабатывает ссылку на свойство в выражении `super` (например, `super.toString`), вкратце заключается в следующем:

1. Он получает значение внутреннего слота текущей функции [[HomeObject]]. В примере это объект, на который ссылается `obj`.
2. Он получает текущий прототип этого объекта, начиная с момента запуска кода `super.toString`, а не с момента создания функции.
3. Он ищет свойство этого объекта-прототипа и использует его.

Ко времени, когда в примере вызывается `obj.toString()`, код уже изменил прототип объекта, на который ссылается `obj`. Таким образом новый метод `toString` прототипа используется вместо исходного.

Теоретически ссылка [[HomeObject]] существует независимо от того, использует ли метод синтаксис `super` или нет. На практике движки JavaScript, скорее всего, оптимизируют ссылку, если ключевое слово `super` не использовалось (а также не использовались такие функции, как `eval`, которые ограничивают способность движков оптимизировать код в методе).

ТИП ДАННЫХ `SYMBOL`

Вплоть до ES5 имена свойств («ключи») всегда были строками (даже те, которые часто записывались в виде чисел, например индексы массивов). С ES2015 это изменилось: ключи свойств теперь могут быть строкового типа или типа Symbol.

Symbol (*Символ*) — это уникальное примитивное значение. Уникальность — его главная цель и особенность. Хотя использование символа выходит за рамки свойств объектов, я включил их в эту главу, потому что именно там вы обычно будете их использовать.

«ИМЯ» ИЛИ «КЛЮЧ»?

Все-таки «имя» свойства или «ключ» свойства? Ответ — да. В спецификации сказано следующее:

«Свойства идентифицируются с помощью ключевых значений. Значение ключа свойства — это либо строковое значение ECMAScript, либо символ. Все строковые и символьные значение, включая пустую строку, допустимы в качестве ключей свойств. *Имя свойства* — это ключ свойства, представляющий собой строковое значение».

Таким образом, теоретически, если значение может быть строкой или символом — это «ключ» свойства, если только строкой — это «имя» свойства. Но по-прежнему распространено (в том числе в самом тексте спецификации) ссылаться на «свойство с названием P», где P может быть строкой или символом. И, как вы узнаете позже, функция ES5 `Object.keys` включает строки, но не символы (если бы она была добавлена в ES2015, возможно, она называлась бы `Object.names` — или включала бы в себя символы). Так что не придавайте слишком большого значения «ключу» и «имени». Программисты часто говорят «имя», когда должны говорить «ключ» (по крайней мере, по определению спецификации).

Почему же символы?

Сейчас мы рассмотрим, как создавать и использовать символы. Но сначала давайте зададим вопрос: почему так важно, чтобы значения, используемые в качестве ключей свойств, были гарантированно уникальными?

Гарантированно уникальные значения просто в целом полезны (как показывает популярность GUID). Но одним из ключевых факторов, стимулирующих использование символов в JavaScript, было то, что они позволяли добавлять в ES2015 несколько вещей, которые нельзя было бы добавить иначе. Некоторые из новых функций требуют поиска новых свойств объектов и их использования, если они существуют. Это не было бы обратно совместимо, если бы ключи были строковыми значениями.

Например, функция `toString` в `Object.prototype` создает строку в форме `"[object XYZ]"`. Для объектов, созданных встроенными конструкторами, такими как `Date`, значением `XYZ` будет имя конструктора (`"[object Date]"`, `"[object Array]"`, и т. д.). Но до ES2015 для экземпляров, созданных вашими собственными конструкторами или обычными объектами, строковое значение было бы просто `"[object Object]"`.

В ES2015 комитет TC39 хотел получить возможность, позволяющую программисту решать, каким должно быть значение `XYZ` с помощью метода `toString` по умолчанию, указав свойство объекта с именем, которое должно использоваться. Проблема в том, что нет безопасного строкового имени, которое они могли бы выбрать для этого свойства: любое выбранное ими строковое имя могло использоваться в мире программирования для другой цели, поэтому его использование может привести к нарушению существующего кода. Но они также не могли просто начать использовать имя конструктора: это также нарушило бы существующий код, который полагался на просмотр поля `"[object Object]"` для этих объектов. На новую функцию нужно было подписаться.

Символы спешат на помощь! Выражение `Object.toString` ищет свойство, идентифицируемое символом, а не строкой. Это означит, что оно не будет конфликтовать с любым другим свойством, которое может существовать для объектов, определенных до ES2015 (поскольку символы еще не существовали). Вот это свойство в действии:

```
class Example1 {
}
class Example2 {
    get [Symbol.toStringTag]() {
        return "Example2";
    }
}
console.log(new Example1().toString()); // "[object Object]"
console.log(new Example2().toString()); // "[object Example2]"
```

Выражение `Object.prototype.toString` ищет значение `Symbol.toStringTag`, предопределенное *хорошо известным символом*[36]: в ES5 вернулось бы значение `"[object Object]"`, а в ES2015+ происходит поиск свойства `Symbol.toStringTag`. И если значение выражения строкового типа, метод `toString` использует это значение для присвоения части `XYZ` в результирующей строке `"[object XYZ]"`. В противном случае используется `Object`, как и было ранее. В примере выражение `new Example1().toString()` возвращает `"[object Object]"`, поскольку у объекта не было свойства с соответствующим именем символа. Но вызов `new Example2().toString()` возвращает `"[object Example2]"`, поскольку там было свойство (унаследованное от `Example.prototype`) и значение `"Example2"`.

Обычный вариант использования выражения `toStringTag` — классы, как и в предыдущем примере. Но оно так же хорошо работает для простых объектов, что полезно, если вы предпочитаете другие типы фабрик объектов, а не функции конструктора:

```
// Прямое использование
const obj1 = {
    [Symbol.toStringTag]: "Nifty"
};
console.log(obj1.toString()); // "[object Nifty]"
// С помощью прототипа
const p = {
    [Symbol.toStringTag]: "Spiffy"
};
const obj2 = Object.create(p);
console.log(obj2.toString()); // "[object Spiffy]"
```

Хватит вопросов «*почему*» к символам, пора рассмотреть вопросы «*как*».

Создание и использование символов

Вы получаете новый, уникальный символ, вызывая функцию `Symbol`. Это не конструктор: не забывайте, что символы являются примитивами, поэтому ключевое слово `new` не используется. Как только у вас есть свой символ, вы можете добавить его к объекту, используя обозначение имени вычисляемого свойства во время создания или обозначение в скобках после создания:

```
const mySymbol = Symbol();
const obj = {
    [mySymbol]: 6                    // Вычисляемое имя свойства
};
```

[36] Подробнее о хорошо известных символах вы узнаете позже в этой главе.

```
const anotherSymbol = Symbol();
obj[anotherSymbol] = 7;              // Обозначения в скобках
console.log(obj[mySymbol]);          // 6
console.log(obj[anotherSymbol]);     // 7

В качестве вспомогательного средства для отладки вы можете дать символу
описание при его создании, передав строку в Symbol:

const mySymbol = Symbol("my symbol");
console.log(mySymbol); // Symbol(my symbol)
```

В среде, где `console.log` отображает представление символа, выполнение `console.log` для `mySymbol` в этом коде может отобразить значение `"Symbol(my symbol)"`. Оно является строкой, возвращаемой методом `toString` для символов.

Начиная с ES2019 описание символа доступно в качестве свойства `description` для символа:

```
const mySymbol = Symbol("my symbol");
console.log(mySymbol.description); // "my symbol"
```

Описание — это просто описание, оно не представляет собой никакого другого значения. (Позже вы узнаете о `Symbol.for`, в котором описание используется также и для другой цели.) Оно не представляет собой значение символа, и два разных символа могут получить одно и то же описание, но при этом быть разными:

```
const a = Symbol("my symbol");
console.log(a);       // Symbol(my symbol)
const b = Symbol("my symbol");
console.log(b);       // Symbol(my symbol)
console.log(a === b); // ложь
const obj = {
    [a]: 6,
    [b]: 7
};
console.log(obj[a]);  // 6
console.log(obj[b]);  // 7
```

Свойства, привязанные к символам, а не к строкам, не включены в цикл `for-in` или массив, возвращаемый методом `Object.keys`, даже если эти свойства перечислимые (и собственные[37] свойства, в случае `Object.keys`).

Символы не для конфиденциальности

Одно из распространенных заблуждений относительно символов заключается в том, что они предназначены для обеспечения конфиденциальности. Например, рассмотрим этот код:

[37] Помните, что объекты могут наследовать свойства, и у них могут быть свои *собственные* свойства. «Собственное» свойство — это свойство, существующее непосредственно в объекте, а не наследуемое.

```
const everUpward = (() => {
    const count = Symbol("count");
    return {
        [count]: 0,
        increment() {
            return ++this[count];
        },
        get() {
            return this[count];
        }
    };
})();
console.log(everUpward.get());             // 0
everUpward.increment();
console.log(everUpward.get());             // 1
console.log(everUpward["count"]);          // undefined
console.log(everUpward[Symbol("count")]); // undefined
```

Если у вас нет доступа к символу, хранящемуся в `count`, вы не сможете получить это свойство, верно? В конце концов, это не свойство `"count"`, а константа `count` является приватной для функции, в которой она объявлена.

Нет, символ, хранящийся в `count`, не приватный (хотя константа, ссылающаяся на него, является таковой), потому что символы *обнаруживаемые*. Например, вы можете получить символы, используемые объектом, с помощью метко названного метода `Object.getOwnPropertySymbols`. Поскольку их можно обнаружить, символы не обеспечивают никакой конфиденциальности для свойств. Возможно, они внесут немного неясности, особенно если вы не даете им описания. Но никакой конфиденциальности. Действительно приватные свойства будут введены в ES2020 или позже с помощью совершенно другого механизма (см. главу 18). Или вы можете использовать WeakMap (глава 12) для хранения действительно приватной информации объекта.

Так почему же возникает это заблуждение? Частично из-за пути, по которому символы попали в JavaScript: они начали свое существование как «объекты с частными именами», и изначально предполагалось, что они действительно будут использоваться в качестве механизма для создания частных свойств объектов. Со временем их название изменилось, они стали примитивами, и использовать их для приватных свойств не получилось. Но наличие гарантированного уникального идентификатора оказалось весьма полезным, поэтому символы были сохранены.

Глобальные символы

Когда вы вызываете ключевое слово `Symbol` для создания своего символа, только ваш код знает о нем, если вы каким-то образом не сделаете его доступным для другого кода (например, сделав его ключом свойства объекта).

Обычно вам это и требуется. Но могут возникнуть сложности, особенно для авторов библиотек, особенно когда вам нужно использовать символы в разных базах данных *realm*.

Realm — это общий контейнер, в котором находится фрагмент кода, состоящий из глобальной среды, внутренних компонентов для этой среды (массив, объект, дата и т. д.), всего кода, загруженного в эту среду, и других битов состояния и т. п. Если подумать о браузере, то база realm, в которой живет ваш код, — это окно страницы,

на которую вы его включили, и все связанные с этим окном материалы (простите за технический жаргон).

Однако у вашего кода есть доступ не только к своей собственной базе данных realm: он также может получить доступ к другим базам realm. У дочернего окна база realm отличается от базы у родительского окна, у веб-воркера (web worker) своя собственная база realm, и она отличается от создавшей его, и т. д. Но во многих случаях код в этих базах может передавать объекты между ними. Это не проблема, но если коду в двух разных базах данных realm необходимо совместно использовать доступ к свойству, идентифицируемому символом, тогда символ также должен быть общим. Вот тут-то и пригодятся *глобальные символы*.

Предположим, что вы пишете класс с именем `BarkCounter`, и `BarkCounter` хранит информацию об объектах с помощью символа. Посмотрите довольно забавный пример в Листинге 5-1.

Листинг 5-1: Забавный класс BarkCounter, версия 1 — barkcounter-version-1.js

```
const BarkCounter = (() => {
    const barks = Symbol("nifty-barks");

    return class BarkCounter {
        constructor(other = null) {
            this[barks] = other && barks in other ? other[barks]: 0;
}

        bark() {
            return ++this[barks];
}

        showBarks(label) {
            console.log(label + ": Barks = " + this[barks]);
}
};
})();
```

Экземпляры класса `BarkCounter` подсчитывают количество раз, когда вы вызываете для них функцию `bark` («гав» — я же говорил вам, что это глупо). Они отвечают на вызов `bark`, увеличивая количество и возвращая новое значение. Они хранят количество вызовов функции `bark` в свойстве, связанном с символом, хранящимся в константе `barks`. Если вы передаете объект в конструктор `BarkCounter`, у которого есть свойство, связанное с этим символом, он копирует количество `barks` из этого экземпляра; в противном случае он начинается с 0. Вы можете попросить его показать количество `barks` на экране, но не можете попросить его выдать количество `barks`.

Вот `BarkCounter` в действии:

```
const b1 = new BarkCounter();
b1.bark();
b1.bark();
b1.showBarks("b1");           // b1: Barks = 2
const b2 = new BarkCounter(b1);
b2.showBarks("b2");           // b2: Barks = 2
```

Обратите внимание, что этот код успешно скопировал подсчет `barks` из `b1` в `b2`. Это потому, что код создания `b2` отправляет `b1` в тот же конструктор `BarkCounter`, который его создал. Таким образом, тот же символ используется при поиске количества `barks` в `b1`.

Давайте рассмотрим пример с пересечением баз realm, используя `BarkCounter` в главном окне и в элементе `iframe`, передавая экземпляр `BarkCounter` из главного окна в элемент `iframe`. См. Листинги 5-2 и 5-3.

Листинг 5-2: Использование BarkCounter при пересечении БД realm: Main, версия 1 — barkcounter-main-1.html

```
<!doctype html>
<html>
<head>
<meta charset="UTF-8">
<title>BarkCounter Across Realms - Main - Version 1</title>
</head>
<body>
<script src="barkcounter-version-1.js"></script>
<script>
var barkCounterFrame = document.createElement("iframe");
barkCounterFrame.addEventListener("load", function() {
    const b1 = new BarkCounter();
    b1.bark();
    b1.bark();
    b1.showBarks("main");           // main: Barks = 2
    barkCounterFrame.contentWindow.useBarkCounter(b1);
});
barkCounterFrame.src = "barkcounter-frame-1.html";
document.body.appendChild(barkCounterFrame);
</script>
</body>
</html>
```

Листинг 5-3: Использование BarkCounter при пересечении БД realm: Frame, версия 1 — barkcounter-main-1.html

```
<!doctype html>
<html>
<head>
<meta charset="UTF-8">
<title>BarkCounter Across Realms - Frame - Version 1</title>
</head>
<body>
<script src="barkcounter-version-1.js"></script>
<script>
function useBarkCounter(b1) {
    b1.showBarks("frame-b1");
    const b2 = new BarkCounter(b1);
    b2.showBarks("frame-b2");
}
</script>
</body>
</html>
```

Файл **barkcounter-main-1.html** загружается в элемент iframe, и сразу после загрузки он создает экземпляр `BarkCounter` (`b1`) и передает его глобальной функции под названием `useBarkCounter` в элементе iframe. Главное окно и iframe — это разные БД realm.

Функция `useBarkCounter` из файла **barkcounter-frame-1.html** получает экземпляр `BarkCounter`, как и в показанном выше коде, передает его в конструктор `BarkCounter`, который пытается скопировать количество barks из `b1` в новый экземпляр.

Скопируйте эти файлы вместе с **barkcounter-version-1.js** на свой локальный веб-сервер. Потом откройте в своем браузере консоль инструментов разработчика и загрузите файл **barkcounter-main-1.html**. Так будет выглядеть результат:

```
main: Barks = 2
frame-b1: Barks = 2
frame-b2: Barks = 0
```

Свойство `barks` не было скопировано. (Если бы это было так, то в последней строке было бы показано 2, а не 0.) Почему?

Проблема в том, что копия `BarkCounter` загружена в главное окно отдельно от загруженной в элемент `iframe` копии `BarkCounter`, и они оба создают свои собственные символы `barks`. Эти символы по определению не равны, поэтому, когда конструктор `BarkCounter` попытался получить количество barks из `b1`, он не увидел ни одного и использовал 0 в качестве отправной точки.

В идеале все загруженные копии `BarkCounter` должны использовать один и тот же символ для `barks`. Вы *могли бы* сделать это, передав символ из главного окна в элемент `iframe` вместе с `b1`, но такое решение быстро все усложняет. В этом конкретном случае вы также можете просмотреть свойства `b1` и проверить, были ли какие-либо из них названы символом с соответствующим описанием. Но это работает не во всех ситуациях (например, если у символа нет описания или другой символ получил такое же описание).

Вместо этого можно опубликовать символ в *глобальном реестре символов*, связав его со строковым ключом, используя функцию `Symbol.for` вместо функции `Symbol`. Это небольшая модификация кода `BarkCounter`, изменяющая эту строку:

```
const barks = Symbol("nifty-barks");
```

на эту:

```
const barks = Symbol.for ("nifty-barks");
```

Функция `Symbol.for` проверяет, есть ли в глобальном реестре символ для предоставленного вами ключа, и возвращает его при наличии. Если такого символа нет, он создает новый (используя ключ в качестве описания), добавляет его в реестр и возвращает этот новый символ.

Это изменение есть в версиях «-2» файлов (**barkcounter-version-2.js, barkcountermain-2.html** и **barkcounter-frame-2.html**). Загрузите их на свой локальный сервер, запустите в браузере, и вы увидите:

```
main: Barks = 2
frame-b1: Barks = 2
frame-b2: Barks = 2
```

Обе копии `BarkCounter` используют один глобальный символ — вызов функции `Symbol.for` в файле **main** создает и регистрирует его, чтобы вызов функции `Symbol.for` в файле **frame** извлек зарегистрированный символ. Таким образом копирование числа `barks` из `b1` в `b2` работает, поскольку оба конструктора `BarkCounter` использовали один и тот же символ для свойства.

Как и в случае с глобальными переменными, избегайте глобальных символов, когда у вас есть альтернативы, но используйте их в подходящих ситуациях. При их использовании обязательно выбирать достаточно уникальное имя, чтобы избежать конфликтов с другими вариантами применения или с теми символами, которые могут быть определены в будущих версиях спецификации.

Вы увидите особенно удобное использование глобальных символов в главе 6 при изучении *итераторов* и *итерируемых*.

И последнее, что касается глобальных символов. Если нужно знать, находится ли символ в глобальном реестре символов, и если да, то какому ключу он присвоен, вы можете использовать функцию `Symbol.keyFor`:

```
const s = Symbol.for("my-nifty-symbol");
const key = Symbol.keyFor(s);
console.log(key); // "my-nifty-symbol"
```

Функция `Symbol.keyFor` возвращает значение `undefined`, если переданный вами ей символ отсутствует в глобальном реестре.

Хорошо известные символы

Спецификация определяет несколько *хорошо известных символов*, которые используются различными операциями в языке. Эти символы доступны через свойства функции `Symbol`. Один из них вы видели в главе 4 (`Symbol.species`), а другой — ранее в этой главе (`Symbol.toStringTag`). Вот список всех известных символов в ES2015-ES2018 (в ES2019 не добавили ни одного, и не похоже, что в ES2020 планируется добавлять) и глава, в которой они рассматриваются:

- `Symbol.asyncIterator` (глава 9);
- `Symbol.hasInstance` (глава 17);
- `Symbol.isConcatSpreadable` (глава 17);
- `Symbol.iterator` (глава 6);
- `Symbol.match` (глава 10);
- `Symbol.replace` (глава 10);
- `Symbol.search` (глава 10);
- `Symbol.species` (глава 4);
- `Symbol.split` (глава 10);
- `Symbol.toPrimitive` (глава 5);
- `Symbol.toStringTag` (глава 5);
- `Symbol.unscopables` (глава 17).

Все определенные в настоящее время хорошо известные символы относятся к глобальным (поэтому они являются общими для разных баз данных realm), хотя спецификация оставляет открытой возможность использования в будущем некоторых неглобальных.

НОВЫЕ ФУНКЦИИ ОБЪЕКТОВ

В ES2015 и ES2017 были добавлены новые функции в конструктор `Object`. Давайте пробежимся по ним.

Метод Object.assign

Функция `Object.assign` — это JavaScript-версия общей функции «extend», которая копирует свойства из одного или нескольких исходных объектов в целевой объект. Предположим, у вас есть функция, представляющая окно сообщения пользователю. Она поддерживает параметры `title`, `text`, `icon`, `className` и `buttons`, и все они (и даже сам объект options) являются необязательными[38]. Это довольно много вариантов. Вы могли бы принять их в качестве параметров функции, но при этом возникают некоторые проблемы:

- Необходимо помнить порядок при записи вызовов функции (хотя ваша среда IDE может помочь).
- Если большинство из них необязательные, код в функции, определяющий, какие параметры предоставил вызывающий, а какие пропустил, может стать довольно сложным.
- Если несколько параметров относятся к одному типу (большинство параметров в этом примере строковые), коду функции может быть не просто сложно, но и невозможно определить, какие из них предоставил вызывающий.

Хотя вы могли бы использовать функционал значений параметров по умолчанию, о котором говорилось в главе 3, и передать значение `undefined` выбранным параметрам, это довольно неудобно.

В этой ситуации обычно используется объект со свойствами для параметров вместо дискретных параметров. Чтобы сделать некоторые из этих свойств необязательными, обычно прибегают к копированию переданного объекта и примешиванию значений по умолчанию из объекта по умолчанию. В ES5 без каких-либо вспомогательных функций это выглядит примерно так:

```
function showDialog(opts) {
    var options = {};
    var optionName;
    var hasOwn = Object.prototype.hasOwnProperty;
    for (optionName in defaultOptions) {
        if (hasOwn.call(defaultOptions, optionName)) {
            options[optionName] = defaultOptions[optionName];
        }
    }
    if (opts) {// помните, что `opts` - это необязательный
        for (optionName in opts) {
            if (hasOwn.call(opts, optionName)) {
                options[optionName] = opts[optionName];
            }
```

[38] Хорошо, конечно, параметр `text`, вероятно, потребуется в реальном мире. Но давайте предположим, что здесь он необязателен.

```
        }
    }
    // (Используйте `options.title`, `options.text` и т. д.)
}
```

Это настолько распространено и настолько громоздко, что функция «extend» стала распространенной во многих наборах инструментов ((jQuery, Underscore/Lodash, и т. д.) и стала идиомой в JavaScript. Традиционная функция «extend» принимает целевой объект, а затем любое количество исходных объектов, копируя собственные, перечислимые свойства из исходных объектов в целевой объект, а после возвращает целевой объект в качестве возвращаемого значения.

Что и делает метод `Object.assign`. Здесь функция `showDialog` использует `Object.assign` вместо использования встроенного кода (и переходит к `const`, хотя это необязательно):

```
function showDialog(opts) {
    const options = Object.assign({}, defaultOptions, opts);
    // (Используйте `options.title`, `options.text` и т. д.)
}
```

Намного аккуратнее! Этот код создает объект и передает его в качестве первого аргумента («целевой» или «target» объект), за которым следуют исходные объекты для копирования, а затем сохраняет этот объект (который является возвращаемым значением) в `options`.

Метод `Object.assign` работает через исходные объекты (`defaultOptions` и `opts` в примере) в порядке слева направо. Таким образом, последний «выигрывает», когда несколько из них содержит значение для одного и того же свойства (поскольку его значение перезаписывает все предыдущие). Это означает, что итоговый объект `options` получит параметры от `opts` для имеющихся у `opts` параметров — или значения по умолчанию из `defaultOptions` для тех, которых нет у `opts`.

Метод `Object.assign` пропускает аргументы со значениями `null` или `undefined`, следовательно коду `showDialog` нет необходимости выполнять обработку случая, когда объекту `opts` совсем не были предоставлены значения при вызове.

Как и функции «extend», которые вдохновили его, метод `Object.assign` копирует только собственные, перечислимые свойства из исходных объектов, а не унаследованные и не помеченные как неисчислимые точно так же, как это делали циклы в громоздкой версии. Он возвращает целевой объект. Однако он копирует свойства, заданные символами, а также свойства, заданные строками, чего не было в предыдущем примере с циклом `for-in` (`for-in` перебирает только свойства объекта со строковыми именами).

В главе 7 вы узнаете о другом способе решения этой конкретной проблемы с помощью *параметров деструктуризации*, а позже в этой главе — о *свойстве расширения* ES2018, которое работает очень похоже на метод `Object.assign`, но не полностью заменяет его.

Метод `Object.is`

В ES2015 метод `Object.is` сравнивает два значения в соответствии с абстрактной операцией SameValue спецификации. Операция SameValue похожа на === (строгое равенство), за исключением:

- Значение `NaN` равно самому себе (за исключением того, что `NaN === NaN` это ложь).
- Положительный ноль (`+0`) и отрицательный ноль (`-0`) не равны друг другу (а `+0 === -0` верно).

Следовательно:

```
console.log(Object.is(+0, -0));   // ложь
console.log(Object.is(NaN, NaN)); // истина
```

а для всего остального функция работает просто как оператор «===».

Метод `Object.values`

В ES5 добавили метод `Object.keys`, возвращающий массив имен собственных перечислимых свойств объекта, ключами которых являются строки (а не символы). Метод `Object.values`, добавленный в ES2017, — его логический аналог: он предоставляет массив значений тех же свойств.

```
const obj = {
    a: 1,
    b: 2,
    c: 3
};
console.log(Object.values(obj)); // [1, 2, 3]
```

Значения унаследованных свойств, неисчислимых свойств и свойств, обозначенных символами, не включаются.

Метод `Object.entries`

Завершая список способов доступа к собственным, перечислимым свойствам со строковыми ключами, укажем добавленный в ES2017 метод `Object.entries`, который возвращает массив массивов `[name, value]`:

```
const obj = {
    a: 1,
    b: 2,
    c: 3
};
console.log(Object.entries(obj)); // [["a", 1], ["b", 2], ["c", 3]]
```

Метод `Object.entries` — довольно мощный инструмент, особенно в сочетании с циклом `for-of`, о котором вы узнаете в главе 6; деструктуризацией, о которой вы узнаете в главе 7; и методом `Object.fromEntries`, о котором вы узнаете… сейчас!

Метод Object.fromEntries

Метод `Object.fromEntries` — это служебная функция, добавленная в ES2019. Она принимает список (любого итерируемого объекта) пар ключ/значение и создает из них объект:

```
const obj = Object.fromEntries([
    ["a", 1],
    ["b", 2],
    ["c", 3]
]);
console.log(obj);
// => {a: 1, b: 2, c: 3}
```

Метод `fromEntries` противоположен `Object.entries`. Он также удобен для превращения коллекции Map (глава 12) в объект, поскольку `Map.prototype.entries` возвращает точный тип списка, который ожидает `Object.fromEntries`.

Функция Object.getOwnPropertySymbols

Метод ES2015 `Object.getOwnPropertySymbols` — это противоположность метода ES5 `Object.getOwnProperty Names`. Как следует из названия метода, он возвращает массив символов для собственных свойств объекта с символьными именами, в то время как `Object.getOwnPropertyNames` возвращает массив строк для объектов со строковыми именами.

Метод Object.getOwnPropertyDescriptors

Метод `Object.getOwnPropertyDescriptors` из ES2017 возвращает объект с дескрипторами свойств для всех собственных свойств объекта (включая неисчисляемые и те, которые обозначаются символами, а не строками).

Одно из его применений — передача возвращаемого объекта в `Object.defineProperties` для другого объекта, чтобы скопировать определения в этот объект:

```
const s = Symbol("example");
const o1 = {
    // Свойство, названное при помощи символа
    [s]: "one",
    // Свойство-акцессор
    get example() {
        return this[s];
},
    set example(value) {
        this[s] = value;
},
    // Свойство данных
    data: "value"
};
// Неперечисляемое свойство
Object.defineProperty(o1, "nonEnum", {
    value: 42,
    writable: true,
    configurable: true
```

```
});
// Копирует эти свойства в новый объект
const descriptors = Object.getOwnPropertyDescriptors(o1);
const o2 = Object.defineProperties({}, descriptors);
console.log(o2.example); // "one"
o2.example = "updated";
console.log(o2[s]);      // "updated", свойство-акцессор записано в свойство [s]
console.log(o2.nonEnum); // 42
console.log(o2.data);    // "value"
```

Сравните это с использованием метода `Object.assign` в той же ситуации: он копировал бы не свойство-акцессор в качестве свойства-акцессора, а значение, возвращаемое свойством-акцессором, на момент вызова метода `Object.assign`. Он также не будет копировать неперечисляемое свойство.

МЕТОД SYMBOL.TOPRIMITIVE

Метод `Symbol.toPrimitive` — это новый, более мощный способ подключения к преобразованию ваших объектов в примитивные значения.

Сначала немного предыстории. Как известно, JavaScript приводит/преобразует значения в различных контекстах, включая приведение/преобразование объектов в примитивные значения. Выполняемая операция может предпочесть, чтобы результатом было число или строка (или может не содержать предпочтений). Например, в `n - obj` (где `obj` — это объект), предпочтительно конвертировать `obj` в число, чтобы можно было выполнить вычитание. Но в `n + obj` нет предпочтения между числом и строкой. В других случаях, таких как `String(obj)` или `someString.indexOf(obj)`, предпочтение отдается строке. В терминах спецификации существует «подсказка» для операции ToPrimitive, которая является числом, строкой либо отсутствует (что означает отсутствие предпочтений).

Обычно вы подключаетесь к этому процессу преобразования для собственных объектов через реализацию метода `toString` (который используется операциями, предпочитающими строку) и/или метода `valueOf` (который используется всеми остальными, как предпочитающими числа, так не имеющими предпочтений). Такой вариант работает довольно хорошо, но ваш объект не сможет выполнять некоторые задачи, когда предпочтение отдается числу или предпочтения отсутствуют; эти две задачи объединяются в `valueOf`. Метод `Symbol.toPrimitive` помогает решить эту проблему. Если у вашего объекта есть метод `Symbol.toPrimitive` с ключом (либо «собственный» метод, либо унаследованный), этот метод используется вместо `valueOf` или `toString`.

Еще лучше, если он получает выбранный тип (*намек на тип*) в качестве аргумента: `"number"`, `"string"` или `"default"`, если операция не имеет предпочтения. Предположим, у вас есть этот объект:

```
const obj = {
    [Symbol.toPrimitive](hint) {
        const result = hint === "string" ? "str": 42;
        console.log("hint = " + hint + ", returning " + JSON.
stringify(result));
        return result;
}
};
```

Если для него вы используете оператор сложения (+), поскольку предпочтений нет, ваш метод `Symbol.toPrimitive` вызывается с намеком `"default"`:

```
console.log("foo" + obj);
// намек = default, возвращается значение 42
// foo42
```

Значением будет `"default"` независимо от того, является ли другой операнд строкой, числом или чем-то еще:

```
console.log(2 + obj);
// намек = default, возвращается значение 42
// 44
```

Но с помощью оператора вычитания (-) намек — это `"number"`:

```
console.log(2 - obj);
// намек = number, возвращается значение 42
// -40
```

Когда требуется строка, неудивительно, что намек — `"string"`:

```
console.log(String(obj));
// намек = string, возвращается значение "str"
// str
console.log("this is a string".indexOf(obj));
// намек = string, возвращается значение "str"
// 10 (индекс "str" в "this is a string")
```

Ваш метод должен возвращать примитивное значение, иначе произойдет ошибка типа (TypeError). Значение *не* обязательно должно совпадать с намеком. Намек — это просто подсказка; вы все равно можете вернуть (например) число, даже если намек указывает строку, или логическое значение, даже если намек указывает на число.

Метод `Symbol.toPrimitive` извлекает из спецификации некоторую магию, предназначенную для объекта Date. До ES2015 ToPrimitive использует внутреннюю операцию, которая решает, вызывать ли `valueOf` или `toString` (под названием [[DefaultValue]]), и поддерживает особый случай для объектов Date: если не было намека на тип, значение по умолчанию должно было действовать так, как если бы намек указывал на число, *за исключением* объектов Date, для которых по умолчанию используется значение String. Вот почему объекты Date преобразуются в строку, когда вы применяете к ним +, даже если они преобразуются в числа, когда вы применяете -. Без метода `Symbol.toPrimitive` это было невозможно сделать в ваших собственных объектах.

Если у вашего объекта нет метода `Symbol.toPrimitive`, спецификация определяет, что по сути выполняется следующее:

```
[Symbol.toPrimitive](hint) {
    let methods = hint === "string"
? ["toString", "valueOf"]
: ["valueOf", "toString"];
    for (const methodName of methods) {
```

```
        const method = this[methodName];
        if (typeof method === "function") {
            const result = method.call(this);
            if (result === null || typeof result !== "object") {
                return result;
            }
        }
    }
    throw new TypeError();
}
```

ПОРЯДОК СВОЙСТВ

Раньше это была одна из тех штучек, о которых кричат со всех крыш: «У свойств объекта нет порядка!»

Но народ все ждал, чтобы у свойств объекта появился порядок, в частности свойства с такими именами, как `1`, `7` и `10`, например, в массивах (помните, что эти имена относятся к строковому типу; см. врезку «Миф о массивах»). Люди в подавляющем большинстве ожидают, что движок будет их перебирать в числовом порядке, даже при использовании механизмов итерации объектов (а не массивов), таких как `for-in`. А ранние движки JavaScript придавали свойствам очевидный порядок, который более поздние движки копировали (хотя часто с небольшими отличиями), — и люди на него полагались.

МИФ О МАССИВАХ

Стандартные массивы в JavaScript — это своего рода миф: как определено в спецификации, они не являются массивами в классическом вычислительном смысле, то есть непрерывными блоками памяти, разделенными на слоты одинакового размера. (В JavaScript такие конструкции теперь есть в виде типизированных массивов, о чем вы узнаете в главе 11.) Это просто объекты, которые наследуются от Array.prototype, имеют особое поведение, связанное с их свойством length, и содержат свойства, имена которых, хотя и являются строками, представляют собой индексы массива по определению спецификации. (Индекс массива[39] — это имя свойства, полностью состоящее из цифр, которое при преобразовании в число представляет собой целое число, со значением из выражения: 0 ≤ значение < 232–1.) В остальном массивы — это такие же объекты, как и любые другие. Конечно, движки JavaScript сильно оптимизируют массивы, где это возможно, во многих случаях превращая их в настоящие массивы, но они определяются как объекты. Это порождает последствия: при применении цикла `for-in` к массиву получаемые ключи представляют собой строки.

[39] `https://tc39.github.io/ecma262/#sec-object-type`

Поскольку люди продолжали желать или даже полагаться на порядок свойств, а движки JavaScript остановились на двух похожих (но разных) порядках, чтобы сделать поведение предсказуемым, порядок был стандартизирован в ES2015, но только для определенных операций и только для «собственных» (не наследуемых) свойств. Он таков:

- Сначала свойства, ключи которых представлены строками, являющимися *целочисленными индексами*, в числовом порядке. *Целочисленный индекс* — это строка, состоящая из одних цифр и которая при преобразовании в число находится в диапазоне 0 ≤ значение < 2^{53}–1. (Это немного отличается от *индекса массива* [см. «Миф о массивах» выше], у которого есть нижняя верхняя границы.)
- Затем другие свойства, ключи которых представлены строками, в том порядке, в котором свойство было *создано* для объекта.
- Затем свойства, ключи которых представлены символами, в том порядке, в котором свойство было создано для объекта.

В ES2015-ES2019 этот порядок не распространяется на цикл `for-in` (даже не просто порядок «собственных» свойств) или массивы, возвращенные методом `Object.keys`, `Object.values` или `Object.entries`. Таким образом, движкам JavaScript не нужно было изменять порядок этих существующих операций, если их разработчики не хотели рисковать, влияя на существующий код. Новый порядок был определен для многих других методов, таких как `Object.getOwnPropertyNames` и `JSON.stringify` из ES5 и `Object.getOwnPropertySymbols` и `Reflect.ownKeys` из ES2015. Но ES2020 обновляет даже цикл `for-in` и методы, перечисленные выше, и подобные им, при условии, что ни объект, ни какой-либо из его прототипов не являются прокси (глава 14), типизированным массивом или чем-то аналогичным (глава 11), объектом пространства имен модуля (глава 13) или «экзотическим», предоставленным хостом объектом (например, элементом DOM).

В общем случае, хотя сейчас существует определенный порядок, лучше как можно реже полагаться на этот порядок свойств в объекте. Это очень хрупкое решение. Например, предположим, что у вас есть функция, которая добавляет два свойства к объекту:

```
function example(obj) {
    obj.answer = 42;
    obj.question = "Life, the Universe, and Everything";
    return obj;
}
```

Если вы вызываете его с помощью объекта без свойства `answer` или `question`, например:

```
const obj = example({});
```

тогда порядок действительно будет таким: `answer`, затем `question`, поскольку именно в таком порядке код добавляет свойства.

Но предположим, что вы передаете ему объект с уже заданным свойством `question`:

```
const obj = example({question: "What's the meaning of life?"});
```

Теперь порядок — `question`, затем `answer`, ведь `question` был добавлен к объекту первым; метод `example` просто обновил значение свойства вместо его создания.

СИНТАКСИС РАСШИРЕНИЯ СВОЙСТВ

Иногда требуется создать объект со всеми свойствами другого объекта, возможно, обновив пару их значений. Это особенно распространено при программировании с использованием «неизменяемого» подхода, при котором объект никогда не изменяется. Вместо изменения вы создаете новый объект для замены, включающий изменения.

Перед ES2018 выполнение этой операции требовало явно указать свойства, которые нужно скопировать, или написать цикл `for-in` или аналогичный ему, или с помощью вспомогательной функции типа общей идиомы `extend`, упоминавшейся ранее в этой главе, или, возможно, даже при помощи собственного метода `Object.assign` JavaScript. В ES2018+ вместо этого вы можете использовать *синтаксис расширения свойств (property spread syntax)*.

Давайте еще раз рассмотрим пример использования `Object.assign`, приведенный ранее в этой главе:

```
function showDialog(opts) {
    const options = Object.assign({}, defaultOptions, opts);
    // (Используйте `options.title`, `options.text` и т. д.)
}
```

ES2018 предоставляет способ сделать это с помощью синтаксиса вместо вызова функции: внутри литерала объекта можно использовать многоточие (`...`) перед любым выражением, чтобы «расширить» собственные, перечислимые свойства результата выражения[40]. Вот этот пример `showDialog` с использованием распространения свойств:

```
function showDialog(opts) {
    const options = {...defaultOptions, ...opts};
    // (Используйте `options.title`, `options.text` и т. д.)
}
```

Этот код создает новый объект, заполняя его свойствами из `defaultOptions`, а затем свойствами из `opts`. Как всегда в литералах объектов, свойства обрабатываются в порядке исходного кода (сверху вниз, слева направо) так что, если у `defaultOptions` и `opts` есть свойство с тем же именем, значение `opts` «побеждает».

Распространение свойств допустимо только в пределах литерала объекта. Он работает почти так же, как объект `Object.assign`: если источником свойства расширения (результат выражения, которое следует за многоточием) является `null` or `undefined`, свойство расширения ничего не делает (это *не* ошибка). Свойство расширения использует только собственные, перечислимые свойства объекта, а не те, которые унаследованы от его прототипа или неисчислимых свойств.

---

[40] Если выражение приводит к примитиву, этот примитив приводится к объекту до начала операции расширения. Например, примитивная строка приводится к объекту строкового типа.

На первый взгляд кажется, что вы могли бы использовать свойство расширения вместо метода `Object.assign` везде. Но метод `Object.assign` по-прежнему играет определенную роль: он может присваиваться *существующему* объекту, тогда как свойство расширения может использоваться только при создании *нового* объекта.

> **ПРИМЕЧАНИЕ**
> Заманчиво думать о синтаксисе ... как об операторе, но это не так. Оператор подобен функции: он получает операнды (например, аргументы функции) и выдает единственное результирующее значение (например, возвращаемое значение функции). Но никакое «результирующее значение» не может сказать «создайте эти свойства в объекте». Таким образом, многоточие — это основной синтаксис/обозначение, а не оператор. Это как скобки вокруг условия цикла `while`: в таком случае `()` не оператор группировки, скобки просто часть синтаксиса цикла `while`. Различие очень важно, потому что оператор может использоваться везде, где ожидается выражение. А ... может использоваться только в определенных конкретных местах (для свойства расширения, только внутри литерала объекта).

ОТ СТАРЫХ ПРИВЫЧЕК К НОВЫМ

Все эти новые синтаксические функции и помощники дают нам возможность обновить несколько привычек.

Использовать вычисляемый синтаксис при создании свойств с динамическими именами

Старая привычка: Создание свойств, имена которых определяются во время выполнения в качестве второго шага после создания объекта:

```
let name = "answer";
let obj = {};
obj[name] = 42;
console.log(obj[name]); // 42
```

Новая привычка: Используйте вычисляемые имена свойств:

```
let name = "answer";
let obj = {
    [name]: 42
};
console.log(obj[name]); // 42
```

Используйте сокращенный синтаксис при инициализации свойства из переменной с тем же именем

Старая привычка: Предоставлять имя свойства, даже если значение свойства поступает из идентификатора области видимости (например, переменной) с тем же именем:

```
function getMinMax() {
    let min, max;
    // ...
    return {min: min, max: max};
}
```

Новая привычка: Используйте сокращенный синтаксис свойств:

```
function getMinMax() {
    let min, max;
    // ...
    return {min, max};
}
```

Используйте метод `Object.assign` вместо пользовательских функций «extend» или явного копирования всех свойств

Старая привычка: Использовать пользовательскую функцию `extend` (или аналогичную) или кропотливо копировать все (собственные, перечислимые) свойства из одного объекта в другой существующий объект с помощью цикла.

Новая привычка: Используйте вместо этого метод `Object.assign`.

Используйте синтаксис расширения при создании нового объекта на основе свойств существующего объекта

Старая привычка: Использовать пользовательский объект `extend` или метод `Object.assign` для создания нового объекта на основе свойств существующего объекта:

```
const s1 = {a: 1, b: 2};
const s2 = {a: "updated", c: 3};
const dest = Object.assign({}, s1, s2);
console.log(dest); // {"a": "updated", "b": 2, "c": 3}
```

Новая привычка: Используйте синтаксис расширения свойств:

```
const s1 = {a: 1, b: 2};
const s2 = {a: "updated", c: 3};
const dest = {...s1, ...s2};
console.log(dest); // {"a": "updated", "b": 2, "c": 3}
```

Используйте символ, чтобы избежать коллизии имен

Старая привычка: Использовать непонятные строковые имена для свойств в попытке избежать коллизии с именами, находящимися вне вашего контроля.

Новая привычка: Используйте вместо этого символы для имен.

Используйте методы `Object.getPrototypeOf/setPrototypeOf` вместо свойства `__proto__`

Старая привычка: Получение или настройка прототипа объекта с помощью ранее не стандартизированного расширения `__proto__`.

Новая привычка: Используйте стандартные методы `Object.getPrototypeOf` и `Object.setPrototypeOf` (хотя свойство `__proto__` теперь стандартизировано для веб-браузеров).

Используйте синтаксис метода для методов

Старая привычка: Использование инициализатора свойств для функций, которые вы намереваетесь использовать в качестве методов объекта:

```
const obj = {
    example: function() {
        // ...
    }
};
```

Новая привычка: Используйте вместо этого синтаксис метода.

```
const obj = {
    example() {
        // ...
    }
};
```

Просто помните, что если вы используете ключевое слово `super`, метод получит ссылку на свой исходный объект. Поэтому, если вы скопируете его в другой объект, он будет продолжать использовать прототип своего исходного объекта, а не новый. Если вы *не* используете выражение с `super` (и не используете `eval`), хороший движок JavaScript оптимизирует эту ссылку.

6

Возможности итерации: итерируемые объекты, итераторы, циклы for-of, итеративные расширения, генераторы

СОДЕРЖАНИЕ ГЛАВЫ

- Итераторы и итерируемые объекты
- Циклы `for-of`
- Синтаксис итеративного расширения
- Генераторы и функции-генераторы

В ES2015 JavaScript получил новую функцию — обобщенную итерацию, классическую идиому «for each», популярную во многих языках. В этой главе вы узнаете о новых возможностях итераторов и итерируемых объектов, а также о том, как их создавать и использовать. Вы также узнаете о новых мощных генераторах JavaScript, упрощающих создание итераторов и расширяющих концепцию, делая общение двусторонним.

ИТЕРИРУЕМЫЕ ОБЪЕКТЫ, ИТЕРАТОРЫ, ЦИКЛЫ FOR-OF, ИТЕРИРУЕМОЕ РАСШИРЕНИЕ

В этом разделе вы узнаете о новых итераторах и итерируемых объектах JavaScript, цикле `for-of`, упрощающим использование итераторов, и удобном синтаксисе расширения (…) для итераций.

Итераторы и итерируемые объекты

Во многих языках есть своего рода конструкция «for each» для перебора элементов массива, списка или другого объекта коллекции. В течение многих лет в JavaScript был только цикл `for-in`, который не является инструментом общего назначения (он предназначен только для перебора свойств объекта). Новые «интерфейсы» итераторов и итерируемых объектов (см. Примечание) и новые языковые конструкции, которые их используют, меняют это положение.

> **ПРИМЕЧАНИЕ**
> Начиная с ES2015 в спецификации впервые упоминаются «интерфейсы» (в смысле интерфейсов кода). Поскольку JavaScript не является строго типизированным, «интерфейс» — это чисто условное обозначение. Описание спецификации заключается в том, что интерфейс «…представляет собой набор ключей свойств, связанные значения которых соответствуют конкретной спецификации. Любой объект, предоставляющий все описанные в спецификации интерфейса свойства, соответствует этому интерфейсу. Интерфейс не представлен отдельным объектом. Может быть много отдельно реализованных объектов, которые соответствуют любому интерфейсу. Отдельный объект может соответствовать нескольким интерфейсам».

Итератор — это объект с методом `next`. При каждом вызове `next` он возвращает следующее значение (если таковое имеется) в последовательности, которую он представляет, и флаг, указывающий, выполнено ли действие.

Итерируемый — это объект со стандартным методом получения итератора для его содержимого. Все объекты в стиле списка или коллекции в стандартной библиотеке JavaScript являются итерируемыми — массивы, строки, типизированные массивы (глава 11), карты (глава 12) и множества (глава 12). Простые объекты по умолчанию *не* являются итерируемыми.

Давайте рассмотрим основы.

Цикл `for-of`: Неявное использование итератора

Вы можете получить и использовать итератор напрямую, но чаще всего его используют косвенно, например, с помощью нового цикла `for-of`. Он получает итератор из итерируемого и использует его для перебора содержимого итерируемого объекта.

> **ПРИМЕЧАНИЕ**
> Цикл `for-of` — не единственная языковая возможность, которую можно использовать для неявного использования итераторов. Есть несколько новых функций, использующих итерируемые элементы

скрытно, в том числе: синтаксис итеративного расширения (далее в этой главе); деструктуризация (глава 7); метод `Promise.all` (глава 8); метод `Array.from` (глава 11), а также карты и множества (глава 12).

Массивы являются итерируемыми, поэтому давайте рассмотрим цикл по массиву с помощью `for-of`:

```
const a = ["a", "b", "c"];
for (const value of a) {
    console.log(value);
}
// =>
// "a"
// "b"
// "c"
```

Выполнение этого кода показывает `"a"`, затем `"b"`, затем `"c"`. Инструкция `for-of` возвращает итератор из массива за кулисами, а затем использует его для перебора значений, делая каждое значение доступным `value` в теле цикла.

Обратите внимание, что цикл `for-of` отличается от `for-in`. Если применить цикл `for-in` к этому массиву, вы получите `"0"`, затем `"1"`, затем `"2"` — имена свойств для массива записей. Цикл `for-of` предоставляет значения записей, определенные итератором массива. Это также означает, что конструкция `for-of` даст вам *только* значения *элементов* массива, а не значения любых других свойств, которые вы, возможно, добавили к нему (поскольку массивы являются объектами). Сравните:

```
const a = ["a", "b", "c"];
a.extra = "extra property";
for (const value of a) {
    console.log(value);
}
// Вышеизложенное приводит к результату "a", "b" и "c"

for (const key in a) {
    console.log(key);
}
// Вышеизложенное приводит к результату "0", "1", "2", и "extra"
```

Вы до этого момента замечали `const value` в примерах? Из главы 2 вам известно, что, когда при объявлении переменной в цикле `for` с помощью директивы `let`, для каждой итерации цикла создается новая переменная. Это справедливо и по отношению к циклу `for-of` или `for-in`. Но, поскольку значение переменной никогда не изменяется по инструкции цикла[41], в конструкциях `for-of`/`for-in` вы можете при желании

[41] Тогда как в цикле `for` управляющая переменная изменяется во время итерации цикла: в частности, в начале второй итерации и далее. См. раздел «Блочная область видимости в циклах» в Главе 2, в частности подробную серию шагов в подразделе «Привязки: Как работают переменные, константы и другие идентификаторы».

объявить его с помощью директивы `const`. (`let` или `var` тоже подойдут, с обычными оговорками об области видимости `var`. Если вы хотите изменить значение в теле цикла, используйте директиву `let`.)

Цикл `for-of` очень удобен, и вы будете часто им пользоваться. Но поскольку он делает все за вас, он не очень помогает разобраться в деталях итераторов. Как он получает итератор? Как он его использует? Давайте посмотрим внимательнее…

Явное использование итератора

Предположим, вы хотите использовать итератор явно. Вы бы сделали это вот так:

```
const a = ["a", "b", "c"];
const it = a[Symbol.iterator](); // Этап 1
let result = it.next();          // Этап 2
while (!result.done) {          // Этап 3.а
    console.log(result.value);   // Этап 3.б
    result = it.next();          // Этап 3.в
}
```

Давайте рассмотрим каждый этап этого процесса.

На этапе 1 получен итератор из массива. Итерируемые предоставляют для этого метод, имя которого — *хорошо известный символ* `Symbol.iterator` (вы изучили хорошо известные символы в главе 5). Вы вызываете этот метод, чтобы получить итератор:

```
const it = a[Symbol.iterator]();
```

На этапе 2 вызывается функция `next` итератора, чтобы получить *результирующий* объект. Результирующий объект — это объект, соответствующий интерфейсу *IteratorResult*, что, по сути, означает наличие у него свойства `done`. Оно указывает, завершил ли итератор итерацию. А также свойство `value`, содержащее значение для текущей итерации:

```
let result = it.next();
```

На этапе 3 выполняется цикл, используя серию подэтапов. (а) в то время как свойство `done` для результирующего объекта равно false (или, по крайней мере, falsy[42]), он (б) использует свойство `value`, затем (в) вновь вызывает функцию `next`:

```
while (!result.done) {       // Этап 3.а
console.log(result.value);    // Этап 3.б
result = it.next();           // Этап 3.в
}
```

[42] Напомним, что *falsy* (ложноподобные) значения приравниваются к `false` при использовании в качестве логических значений. Лжеподобные значения — это `0`, `NaN`, `""`, `undefined`, `null` и конечно же, `false`; все прочие значения истинноподобные. (Ну… в браузерах также есть `document.all`; вы узнаете о его странной ложности в Главе 17.)

> **ПРИМЕЧАНИЕ**
> Символы, используемые для итерации (`Symbol.iterator` и другие, о которых вы узнаете в главе 9), являются глобальными символами (см. главу 5). Поэтому итерация работает в разных базах данных realm.

Вы, вероятно, заметили, что кодирование вызова `result = it.next()` в двух местах создает незначительную опасность для обслуживания кода. Это одна из причин использовать цикл `for-of` или другие автоматические конструкции там, где это возможно. Но вы также можете выполнить вызов в выражении условия цикла `while`:

```
const a = ["a", "b", "c"];
const it = a[Symbol.iterator]();
let result;
while (!(result = it.next()).done) {
console.log(result.value);
}
```

Теперь вызов находится только в одном месте.

Хотя результирующий объект определен как содержащий свойства `done` и `value`, свойство `done` становится необязательным, если его значение будет равно `false`. А свойство `false` становится необязательным, если его значение равно `undefined`.

Это основные шаги использования итератора.

Остановка итерации на ранней стадии

Когда вы используете итератор, у вас может быть причина остановиться на раннем этапе. Например, если вы ищете что-то в последовательности итератора, вы, вероятно, остановитесь, когда найдете это. Но если вы просто перестанете вызывать `next`, итератор не узнает, что вы закончили. Вполне возможно, что он будет удерживать ресурсы, которые может потерять, если вы прекратите выполнение. В конце концов он все равно избавится от этих ресурсов при сборке мусора (если только они не являются чем-то, с чем сборщик мусора не справляется). Но у итераторов есть способ быть более активными — необязательный метод, называемый `return`. Он сообщает итератору, что вы закончили поиск и что он должен удалить все ресурсы, которые обычно очищает при достижении конца последовательности.

Поскольку это необязательно, вам нужно проверить его, прежде чем вызывать. Взгляните на пример:

```
const a = ["a", "b", "c", "d"];
const it = a[Symbol.iterator]();
let result;
while (!(result = it.next()).done) {
    if (result.value === "c") {
        if (it.return) {
            it.return();
        }
        break;
    }
}
```

(В главе 19 вы узнаете о необязательном связывании, позволяющем написать это более кратко. Вместо `if` можно просто написать `it.return?.();`. Подробности см. в этой главе.)

Спецификация говорит, что метод `return` должен возвращать результирующий объект (как делает функция `next`), у которого, «как правило», свойство `done` равно `true`, и что, если вы передаете аргумент в метод `return`, свойство `value` результирующего объекта должно, «как правило», быть этим значением аргумента. Но спецификация также говорит, что это необязательно (а в дальнейшем вы увидите пути выхода). Следовательно, вы вряд ли захотите брать в расчет то или другое поведение. Итераторы, предоставляемые самой средой выполнения JavaScript, будут вести себя так, будто у них есть метод `return`, но вам не стоит рассчитывать на то, что итераторы из сторонних библиотек будут делать это.

Вам может быть интересно, как вызвать `return` для итератора, при использовании цикла `for-of`, поскольку у вас нет явной ссылки на итератор:

```
const a = ["a", "b", "c", "d"];
for (const value of a) {
    if (value === "c") {
        if (???.return) {
            ???.return();
        }
        break;
    }
}
```

Ответ таков: вам не нужно этого делать. Конструкция `for-of` делает это за вас, когда вы выходите из цикла, не завершив его (то есть используя оператор `break`, или возвращая, или выбрасывая ошибку). Так что вам вообще не нужно беспокоиться об этом:

```
const a = ["a", "b", "c", "d"];
for (const value of a) {
    if (value === "c") {
        break; // Это просто отлично, он вызывает метод `return` итератора
    }
}
```

Существует второй необязательный метод — `throw`. Обычно его не применяют с простыми итераторами, и цикл `for-of` никогда не вызывает его (даже если выбрасывается ошибка из тела `for-of`). Мы рассмотрим `throw` позже в этой главе, там же вы узнаете о *функциях-генераторах*.

Прототип итератора объекта

Все итераторы, предоставляемые средой выполнения JavaScript, наследуются от объекта-прототипа, предоставляющего соответствующий метод `next` для итерируемого объекта, от которого вы его получили. Например, итераторы массива наследуются от объекта прототипа итератора массива, который предоставляет соответствующий метод `next` для массивов.

Этот прототип итератора массива известен в спецификации как `%ArrayIteratorPrototype%`. Строковые итераторы наследуются от `%StringIteratorPrototype%`, итераторы карт (глава 12) наследуются от `%MapIteratorPrototype%` и т. д.

> **ПРИМЕЧАНИЕ**
> Не существует общедоступных глобальных переменных или свойств, предоставляющих прямые ссылки на прототипы в стандартной среде выполнения. Но вы можете легко получить ссылку на один из них, используя `Object.getPrototypeOf` на итераторе нужного вам типа, например, итераторе массива.

У всех этих прототипов итераторов есть базовый прототип, который спецификация творчески называет `%IteratorPrototype%`. Он определяет только одно свойство, и оно может показаться немного странным, поскольку не является частью интерфейса итератора. Он определяет метод `Symbol.iterator`, эффективно возвращающий `this;`:

```
const iteratorPrototype = { // (Это имя носит концептуальный характер,
                            // а не фактический)
    [Symbol.iterator]() {
        return this;
    }
};
```

Вы, вероятно, думаете: «Но подождите, разве это не для *итерируемых объектов*? Зачем вам вместо этого использовать его для *итератора*?!»

Ответы таковы: «Да» и «Чтобы они также были итерируемыми». Перевод итератора в итерируемый объект позволяет передавать итератор в цикл `for-of` или другие конструкции, ожидающие итерируемый объект. Например, если вы хотите перебрать все записи, кроме первой в массиве, без использования метода `slice`, чтобы получить подмножество массива:

```
const a = ["one", "two", "three", "four"];
const it = a[Symbol.iterator]();
it.next();
for (const value of it) {
    console.log(value);
   }
// =>
// "two"
// "three"
// "four"
```

Вы узнаете больше об этом в разделе «Итеративные итераторы» позже в этой главе, но я хотел объяснить кажущийся странным метод, предоставляемый прототипом `%IteratorPrototype%`.

В основном вам не нужно будет заботиться об этих объектах-прототипах. Однако есть по крайней мере две ситуации, в которых вам может потребоваться получить доступ к этим прототипам:

1. Если вы хотите добавить функциональность к итераторам.
2. Если вы реализовывали итератор вручную (см. следующий раздел), чего обычно никто не делает.

Вы увидите номер 2 в следующем разделе. Давайте рассмотрим номер 1 — добавление функциональности.

Прежде чем мы это сделаем, обратите внимание, что применяются все обычные предостережения относительно изменения прототипов встроенных объектов:

А. Убедитесь, что все добавляемые вами свойства/методы не перечисляемы.
Б. Убедитесь, что их имена не будут конфликтовать с добавляемыми в перспективе функциями. Стоит рассмотреть возможность использования символов.
В. В подавляющем большинстве случаев библиотечный код (а не код приложения или страницы) должен полностью избегать модификаций прототипа.

Учитывая все сказанное, давайте добавим метод ко всем итераторам (или, по крайней мере, к тем, которые наследуются от `%IteratorPrototype%`): поиск первой записи, соответствующей условию, например, метод `find` для массивов. Мы не будем называть его `find`, потому что это нарушило бы предостережение (Б), поэтому мы назовем это `myFind`. Точно так же, как метод `find` в массивах, он примет обратный вызов и необязательный аргумент для использования в качестве `this` при вызове обратного вызова и вернет первый результирующий объект, для которого обратный вызов возвращает истинное значение, или последний результирующий объект (у которого будет свойство `done = true`), если истинного значение обратного вызова нет.

Чтобы добавить метод в `%IteratorPrototype%`, сначала мы должны получить ссылку на него. Нет общедоступной глобальной переменной или свойства, ссылающегося непосредственно на него. Но мы знаем из спецификации, что прототипом итератора массива является `%ArrayIteratorPrototype%`, и мы знаем (также из спецификации), что *его* прототипом является `%IteratorPrototype%`. Таким образом, мы можем получить прототип `%IteratorPrototype%`, создав массив, получив для него итератор и получив прототип его прототипа:

```
const iteratorPrototype = Object.getPrototypeOf(
    Object.getPrototypeOf([][Symbol.iterator]())
);
```

В спецификации даже есть примечание, в котором говорится именно об этом.

Теперь, когда у нас есть ссылка на прототип, мы можем добавить к нему наш метод. Обычно методы в прототипах не перечисляемые, но доступны для записи и настройки, поэтому мы будем использовать метод `Object.defineProperty`. Сам метод довольно прост: он вызывает `next` до тех пор, пока обратный вызов не вернет истинное значение, или пока не закончатся значения.

```
Object.defineProperty(iteratorPrototype, "myFind", {
    value(callback, thisArg) {
        let result;
        while (!(result = this.next()).done) {
            if (callback.call(thisArg, result.value)) {
                break;
}
}
        return result;
},
    writable: true,
    configurable: true
});
```

Запустите код в Листинге 6-1, чтобы увидеть его в действии. Как только новый метод определен, в примере он используется для поиска записей в массиве строк, в которых есть буква «e».

Листинг 6-1: Добавление метода myFind к итераторам — adding-myFind.js

```
// Добавление метода
const iteratorPrototype = Object.getPrototypeOf(
    Object.getPrototypeOf([][Symbol.iterator]())
);
Object.defineProperty(iteratorPrototype, "myFind", {
    value(callback, thisArg) {
        let result;
        while (!(result = this.next()).done) {
            if (callback.call(thisArg, result.value)) {
                break;
            }
        }
        return result;
    },
    writable: true,
    configurable: true
});

// Применение метода
const a = ["one", "two", "three", "four", "five", "six"];
const it = a[Symbol.iterator]();
let result;
while (!(result = it.myFind(v => v.includes("e"))).done) {
    console.log("Found: " + result.value);
}
console.log("Done");
```

Однако подобное расширение полезно только тогда, когда итератор используется явно. Позже, при изучении *функций-генераторов*, вы узнаете более простой способ сделать это.

Сделать что-либо итерируемым объектом

Как показано выше, итератор можно получить при помощи вызова метода `Symbol.iterator` итерируемого объекта. Также вы узнали, что итератор — это объект с методом

`next`, возвращающим «следующий» результирующий объект со свойствами `value` и `done`. Чтобы сделать что-то итеративным, вы просто предоставляете метод `Symbol.iterator`.

Давайте создадим псевдомассив с простым объектом:

```
const a = {
    0: "a",
    1: "b",
    2: "c",
    length: 3
};
```

Этот объект не является итеративным, поскольку простые объекты по умолчанию не относятся к ним:

```
for (const value of a) {// TypeError: a не является итеративным
    console.log(value);
}
```

Чтобы сделать объект итеративным, вам нужно добавить функцию в качестве его свойства `Symbol.iterator`:

```
a[Symbol.iterator] = /* код функции */;
```

ОШИБКИ, КОГДА ОБЪЕКТЫ НЕ ЯВЛЯЮТСЯ ИТЕРАТИВНЫМИ

В настоящее время при попытке использовать неитеративный объект в большинстве движков JavaScript там, где ожидается итеративный объект, вы получаете четкую ошибку — «`x is not iterable`» или что-то подобное. Но на некоторых движках формулировка может быть не совсем ясной: «`undefined is not a function`». Это происходит, когда движок JavaScript не содержит процедуры специальной обработки такой ситуации, потому что он ищет `Symbol.iterator` в объекте и пытается вызвать полученное значение. Если у объекта нет свойства `Symbol.iterator`, результатом поиска будет значение `undefined`, и при попытке вызова undefined в качестве функции вы получите такую ошибку. К счастью, в большинство движков добавлен процесс специальной обработки, чтобы сделать ошибку более понятной.

Следующий шаг — написать функцию, чтобы она возвращала объект итератора. Позже в этой главе вы узнаете о *функциях-генераторах*, которые вы, вероятно, будете использовать в большинстве случаев для реализации итераторов. Но поскольку функции-генераторы выполняют большую часть работы за вас, они скрывают некоторые детали. Так что давайте сначала проведем процесс вручную.

В качестве первого подхода можно внедрить итератор для этого объекта псевдомассива примерно так:

```
// Попытка 1
a[Symbol.iterator] = function() {
    let index = 0;
    return {
        next: () => {
            if (index < this.length) {
                return {value: this[index++], done: false};
            }
            return {value: undefined, done: true};
        }
    };
};
```

(Обратите внимание, что `next` — это стрелочная функция. Важно, что она использует значение `this`, с которым был вызван метод `Symbol.iterator`, независимо от способа вызова метода. Таким образом, `this` ссылается на псевдомассив.

В качестве альтернативы вы могли бы использовать синтаксис метода и `a`, а не `this` в обоих местах его применения, поскольку `next` закрывается над `a` в данном примере. Но при реализации этого решения в классе и необходимости получения доступа к информации об экземпляре, такой как `length`, у вас не было бы ничего, кроме `this`, для закрытия функции. Поэтому применение стрелочной функции имеет смысл.)

Эта версия подходит, но она не заставляет итератор наследовать от `%IteratorPrototype%`, как это делают все итераторы, получаемые из среды выполнения JavaScript. Поэтому, если вы добавите метод к `%IteratorPrototype%`, как мы делали ранее с `myFind`, у этого итератора не будет добавления. Он также не наследует свойство прототипа, делающее итераторы итеративными. Давайте сделаем так, чтобы объект наследовал от `%IteratorPrototype%`:

```
// Попытка 2
a[Symbol.iterator] = function() {
    let index = 0;
    const itPrototype = Object.getPrototypeOf(
        Object.getPrototypeOf([][Symbol.iterator]())
    );
    const it = Object.assign(Object.create(itPrototype), {
        next: () => {
            if (index < this.length) {
                return {value: this[index++], done: false};
            }
            return {value: undefined, done: true};
        }
    });
    return it;
};
```

Если вы часто пишете такие итераторы вручную, вам, вероятно, понадобится многоразовая функция, выполняющая большую часть работы за вас, в которую вы просто будете передавать свою реализацию `next`. Но опять же, вы, вероятно, будете использовать

вместо этого функции-генераторы. (Мы скоро доберемся до этого раздела.) Попробуйте выполнить код, приведенный в Листинге 6-2.

Листинг 6-2: Базовый пример итеративного объекта — basic-iterable-example.js

```
// Пример базового итератора без использования функции-генератора
const a = {
    0: "a",
    1: "b",
    2: "c",
    length: 3,
    [Symbol.iterator]() {
        let index = 0;
        const itPrototype = Object.getPrototypeOf(
            Object.getPrototypeOf([][Symbol.iterator]())
        );
        const it = Object.assign(Object.create(itPrototype), {
            next: () => {
                if (index < this.length) {
                    return {value: this[index++], done: false};
                }
                return {value: undefined, done: true};
            }
        });
        return it;
    }
};
for (const value of a) {
    console.log(value);
}
```

Этот пример предназначен только для одного объекта, но часто требуется определить класс итеративных объектов. Типичный способ сделать это — поместить функцию `Symbol.iterator` в объект-прототип, использующий класс объектов. Давайте рассмотрим пример использования синтаксиса `class`, который вы изучили в главе 4, — очень простой итеративный класс `LinkedList`. См. Листинг 6-3.

Листинг 6-3: Итератор для класса — iterator-on-class.js

```
// Пример базового итератора для класса без функции-генератора
class LinkedList {
    constructor() {
        this.head = this.tail = null;
    }

    add(value) {
        const entry = {value, next: null};
        if (!this.tail) {
            this.head = this.tail = entry;
        } else {
            this.tail = this.tail.next = entry;
        }
    }
```

```
    [Symbol.iterator]() {
        let current = this.head;
        const itPrototype = Object.getPrototypeOf(
            Object.getPrototypeOf([][Symbol.iterator]())
        );
        const it = Object.assign(Object.create(itPrototype), {
            next() {
                if (current) {
                    const value = current.value;
                    current = current.next;
                    return {value, done: false};
                }
                return {value: undefined, done: true};
            }
        });
        return it;
    }
}

const list = new LinkedList();
list.add("one");
list.add("two");
list.add("three");

for (const value of list) {
    console.log(value);
}
```

В основе этот код такой же, как и предыдущая реализация итератора; он просто определяется в прототипе, а не непосредственно в объекте. Кроме того, поскольку метод `next` не использует значение `this` (в отличие от примера псевдомассива, для этого не требуется никакой специфичной для экземпляра информации — каждый узел ссылается на следующий), его можно определить с помощью синтаксиса метода, а не как свойство, ссылающееся на стрелочную функцию.

Итерируемые итераторы

Ранее вы узнали, что все итераторы, наследующие `%IteratorPrototype%`, также *итерируемые*. Ведь `%IteratorPrototype%` предоставляет метод `Symbol.iterator`, что делает выражение `return this` — просто возвращением итератора, для которого он был вызван. Существуют различные причины сделать итераторы итеративными.

Как описано ранее, вы можете пропустить или специально обработать запись или некоторые записи, а затем обработать остальные с помощью цикла `for-of` или другого механизма, использующего итерируемый объект (а не итератор).

Или предположим, что вы хотите предоставить итератор без какого-либо итеративного объекта, например, в качестве возвращаемого значения функции. Вызывающий может захотеть использовать для него цикл `for-of` или что-то подобное. Перевод итератора в итеративный объект позволяет это сделать.

Вот пример функции, возвращающей итератор, который также является итеративным. Итератор предоставляет родительский элемент передаваемого ему элемента DOM (а затем родительский элемент этого родителя и т. д., пока у него не закончатся родители):

```
// Пример, который не наследуется от %IteratorPrototype% (если бы он это
сделал,
// нам не нужно было бы реализовывать функцию [Symbol.iterator])
function parents(element) {
    return {
        next() {
            element = element && element.parentNode;
            if (element && element.nodeType === Node.ELEMENT_NODE) {
                return {value: element};
            }
            return {done: true};
        },
        // Итератор становится итеративным
        [Symbol.iterator]() {
            return this;
        }
    };
}
```

Конечно, на самом деле вы бы не написали функцию таким образом, потому что, во-первых, вы, вероятно, написали бы вместо этого функцию-генератор (мы доберемся до этого, я обещаю!). Во-вторых, даже если бы вы этого не сделали, у вас был бы итератор, унаследованный от `%IteratorPrototype%` вместо самостоятельной реализации `Symbol.iterator`. Но попробуйте сделать это со страницей в Листинге 6-4. Затем попробуйте изменить его, чтобы итератор наследовал от `%IteratorPrototype%`, вместо самостоятельной реализации метода `Symbol.iterator`.

Листинг 6-4: Пример итерируемого итератора — iterable-iterator-example.html

```
<!doctype html>
<html>
<head>
<meta charset="UTF-8">
<title>Iterable Iterator Example</title>
</head>
<body>
<div>
    <span>
        <em id="target">Look in the console for the output</em>
    </span>
</div>
<script>

// Не совсем так, как вы бы это написали, но показывает,
// как реализовать метод Symbol.iterator,
// позволяя использовать итератор с `for-of`
function parents(element) {
    return {
        next() {
            element = element && element.parentNode;
            if (element && element.nodeType === Node.ELEMENT_NODE) {
                return {value: element};
            }
            return {done: true};
        },
```

```
        // Делает этот итератор итеративным
        [Symbol.iterator]() {
            return this;
        }
    };
}
for (const parent of parents(document.getElementById("target"))) {
    console.log(parent.tagName);
}
</script>
</body>
</html>
```

Синтаксис итеративного расширения

Синтаксис итеративного расширения предоставляет способ использования итеративного объекта путем расширения его результирующих значений в виде дискретных значений при вызове функции или создании массива. (В главе 5 рассказывается о другом виде расширения, *синтаксисе расширения свойств*. Он содержит ряд отличий и используется только в литералах объектов.)

Синтаксис расширения для массивов и других итеративных объектов был введен в ES2015. Давайте рассмотрим пример — поиск наименьшего числа в массиве чисел. Для решения задачи можно написать цикл, но вам известно, что `Math.min` принимает переменное количество аргументов и возвращает наименьший из них. Например:

```
console.log(Math.min(27, 14, 12, 64)); // 12
```

Вы могли бы использовать это, если бы у вас были дискретные числа для передачи. Но если у вас есть *массив* чисел:

```
// ES5
var a = [27, 14, 12, 64];
```

как вы в таком случае передаете их в `Math.min` в качестве дискретных аргументов? Старый способ, существовавший до ES2015, заключался в использовании `Function.prototype.apply`. Это было многословно и странно выглядело:

```
// ES5
var a = [27, 14, 12, 64];
console.log(Math.min.apply(Math, a)); // 12
```

Однако с помощью синтаксиса расширения вы можете вместо этого распределить массив в виде отдельных аргументов:

```
const a = [27, 14, 12, 64];
console.log(Math.min(... a)); // 12
```

Как видите, расширение использует многоточие (три точки подряд как остаточный параметр, описанный в главе 3, и свойство расширения, описанное в главе 5).

Вы можете использовать синтаксис расширения в любом месте списка аргументов. Например, если у вас было два числа в отдельных переменных, а также список чисел в массиве:

```
const num1 = 16;
const num2 = 50;
const a = [27, 14, 12, 64];
```

и если необходимо получить минимальное значение для всех чисел, вы могли бы сделать любое из этих действий:

```
console.log(Math.min(num1, num2, ...a)); // Math.min(16, 50, 27, 14, 12,
                                         // 64) == 12
console.log(Math.min(num1, ...a, num2)); // Math.min(16, 27, 14, 12, 64,
                                         // 50) == 12
console.log(Math.min(...a, num1, num2)); // Math.min(27, 14, 12, 64, 16,
                                         // 50) == 12
```

Место, где вы размещаете расширение, изменяет порядок аргументов, передаваемых в метод `Math.max`. Бывает, что `Math.max` не заботится о порядке, но другие, принимающие списки переменных параметров, функции вполне могут придавать этому значение (например, метод массивов `push` помещает получаемые аргументы в массив в том порядке, в котором вы их передаете).

Другое использование синтаксиса итеративного расширения — это расширение массива (или другого итерируемого) внутри литерала массива:

```
const defaultItems = ["a", "b", "c"];
function example(items) {
    const allItems = [...items, ...defaultItems];
    console.log(allItems);
}
example([1, 2, 3]); // 1, 2, 3, "a", "b", "c"
```

Обратите внимание, что порядок имеет значение. Если вы сначала поставите `defaultItems`, а затем `items`, вы получите другой результат:

```
const defaultItems = ["a", "b", "c"];
function example(items) {
    const allItems = [...defaultItems, ...items];
    console.log(allItems);
}
example([1, 2, 3]); // "a", "b", "c", 1, 2, 3
```

Итераторы, цикл `for-of` и DOM

В основном JavaScript используется в веб-браузерах или приложениях, созданных с использованием веб-технологий. Интерфейс DOM содержит различные объекты коллекций, такие как `NodeList`, возвращаемый методом `querySelectorAll`, или более старую версию `HTMLCollection`, возвращаемую методом `getElementsByTagName`, и другие более старые методы. Возможно, вы задаетесь вопросом: «Могу ли я применить цикл `for-of` к ним? Относятся ли они к итеративным?»

Коллекция `NodeList` доступна в современных, передовых браузерах (современных Chrome, Firefox, Edge и Safari), а коллекция `HTMLCollection` — во всех, кроме версии Edge, выпущенной до Chromium. Спецификация DOM WHAT-WG помечает коллекцию `NodeList` итеративной, но коллекцию `HTMLCollection` — нет. Поэтому старая версия Edge работает в соответствии со спецификацией, в то время как другие добавляют эту возможность для `HTMLCollection`, хотя это не было описано в спецификации.

В Листинге 6-5 показан забавный пример перебора коллекции `NodeList` из метода `querySelectorAll`.

Листинг 6-5: Выполнение цикла по коллекции DOM — looping-dom-collections.html

```
<!doctype html>
<html>
<head>
<meta charset="UTF-8">
<title>Looping DOM Collections</title>
</head>
<body>
<p>
    Lorem <span class="cap">ipsum</span> dolor sit amet, consectetur
adipiscing
    elit, sed do eiusmod <span class="cap">tempor</span> incididunt ut
    <span>labore</span> et dolore <span class="cap">magna</span>
    <span class="other">aliqua</span>.
</p>
<script>
for (const span of document.querySelectorAll("span.cap")) {
    span.textContent = span.textContent.toUpperCase();
}
</script>
</body>
</html>
```

Вероятно, сейчас не требуется так часто преобразовывать коллекции DOM в массивы, как это было в прошлом. Например, раньше вам приходилось преобразовывать коллекцию DOM в массив, чтобы использовать в ней цикл `forEach`. В современных браузерах в этом больше нет необходимости, потому что теперь в коллекциях есть цикл `forEach`. Но если у вас есть конкретная потребность в массиве (например, вы хотите использовать некоторые расширенные методы массива), вы можете использовать итерируемое расширение для преобразования коллекции DOM в массив. Например, этот код преобразует коллекцию `NodeList` в массив:

```
const niftyLinkArray = [...document.querySelectorAll("div.nifty > a")];
```

Если вы выполняете транспилирование для использования в старых браузерах, вам, вероятно, понадобится полифил, чтобы заставить итерацию работать с массивами. Это применимо и для коллекций DOM. Как правило, используемый вами транспилятор будет предлагать полифилы для итерации, по крайней мере, для выражения `Array.prototype` — и, возможно, также для `NodeList.prototype`. Они могут быть необязательными, вам потребуется специально их включить. Если ваш предпочтительный полифил не обрабатывает

коллекции DOM, но не `Array.prototype`, вы можете применить `Array.prototype` к коллекциям DOM.

То же самое касается выражения `Array.prototype.forEach`. См. Листинг 6-6. (Он написан по стандарту ES5, поэтому его можно использовать напрямую даже в старых средах. Его можно включить в свой пакет после любых используемых полифилов.) Но сначала дважды проверьте, что используемый вами инструмент не справляется с этой задачей за вас. Как правило, лучше использовать подготовленные полифилы, а не сворачивать собственные.

Листинг 6-6: Применение итератора `Array.prototype` к коллекциям DOM — polyfill-dom-collections.js

```
;(function() {
    if (Object.defineProperty) {
        var iterator = typeof Symbol !== "undefined" &&
                    Symbol.iterator &&
                    Array.prototype[Symbol.iterator];
        var forEach = Array.prototype.forEach;
        var update = function(collection) {
            var proto = collection && collection.prototype;
            if (proto) {
                if (iterator && !proto[Symbol.iterator]) {
                    Object.defineProperty(proto, Symbol.iterator, {
                        value: iterator,
                        writable: true,
                        configurable: true
                    });
                }
                if (forEach && !proto.forEach) {
                    Object.defineProperty(proto, "forEach", {
                        value: forEach,
                        writable: true,
                        configurable: true
                    });
                }
            }
        };

        if (typeof NodeList !== "undefined") {
            update(NodeList);
        }
        if (typeof HTMLCollection !== "undefined") {
            update(HTMLCollection);
        }
    }
})();
```

ФУНКЦИИ-ГЕНЕРАТОРЫ

Генераторы — это функции, способные делать паузу в середине выполнения, выдавать значение, по потребности принимать новое значение, а затем продолжать работу — столько раз, сколько необходимо (бесконечно, если потребуется). Это делает их очень мощными и на первый взгляд сложными, но на самом деле они довольно просты.

На самом деле они не останавливаются на середине. Функции-генераторы скрыто создают и возвращают объекты-генераторы. Объекты-генераторы являются итераторами, но с двусторонним потоком информации. Там, где итераторы только создают значения, генераторы могут как *создавать*, так и *потреблять* значения.

Вы могли бы создать объект-генератор вручную, но функции-генераторы предоставляют синтаксис, заметно упрощающий процесс (точно так же, как цикл `for-of` упрощает процесс использования итератора).

Базовая функция-генератор, просто производящая значения

Мы начнем с очень простой функции-генератора с односторонним потоком информации — создание значений. См. Листинг 6-7.

Листинг 6-7: Базовый пример генератора — basic-generator-example.js

```
function* simple() {
    for (let n = 1; n <= 3; ++n) {
        yield n;
    }
}
for (const value of simple()) {
    console.log(value);
}
```

При запуске Листинга 6-7, на экране отображаются цифры от 1 до 3:

1. Код вызывает функцию `simple`, получая объект-генератор, а затем передает этот объект в `for-of`.
2. Цикл `for-of` запрашивает у объекта-генератора итератор. Объект-генератор возвращает *себя* в качестве итератора: объекты-генераторы косвенно наследуют `%IteratorPrototype%`, таким образом у них есть выражение «`return this`», обеспечиваемое методом `Symbol.iterator`.
3. Цикл `for-of` использует метод `next` объекта-генератора для получения значений в цикле и передачи их в функцию `console.log`.

В этом коде есть несколько новых выражений.

Во-первых, обратите внимание на «`*`» после ключевого слова `function`. Это то, что делает выражение функцией-генератором. После ключевого слова `function` и перед `*` при желании можно использовать пробелы — это исключительно вопрос стиля.

Во-вторых, обратите внимание на новое ключевое слово `yield`. Оно отмечает, где функции-генератору полагается приостановиться, а после она возобновляется. Значение, указанное после него (если таковое имеется), — это значение, которое генератор выдает в этот момент. Итак, в Листинге 6-7 `yield n;` дает 1, затем 2, затем 3, а после выполнение заканчивается. Код, использующий генератор, видит эти значения через `for-of` (поскольку генераторы также являются итераторами).

ЧУВСТВИТЕЛЬНЫЕ К КОНТЕКСТУ КЛЮЧЕВЫЕ СЛОВА

Некоторые из новых функций, добавляемых в JavaScript, содержат новые ключевые слова, такие как `yield`. Но многие из этих новых ключевых слов не всегда были зарезервированными словами в JavaScript и могли использоваться в качестве идентификаторов (имен переменных и т. п.) в существующем коде. Так как же они могут внезапно стать ключевыми словами, не нарушая существующий код?

Это реализуется несколькими различными способами, но их объединяет одно. Новое ключевое слово является ключевым только в том контексте, где оно не могло появиться раньше. Если контекст подразумевает, что оно могло появиться раньше, это не ключевое слово, и код может использовать его в качестве идентификатора.

Один из способов сделать это — определить новое ключевое слово только в том месте, где идентификатор был бы синтаксической ошибкой. По определению это означает, что нет существующего рабочего кода, который был бы нарушен изменением. (Вы сможете увидеть такой пример в главе 9: `async` — это ключевое слово, когда оно непосредственно перед словом `function`. До его добавления выражение `async function` вызывало синтаксическую ошибку, как это происходит с выражением `foo function` до сих пор.)

Другой способ реализации — определить новое ключевое слово только внутри ранее не существовавшей структуры. Поскольку она раньше не существовала, там не может быть никакого ранее написанного кода, использующего идентификатор. Так работает `yield`: это всего лишь ключевое слово в функции-генераторе, а функции-генераторы являются новыми. До их добавления выражение `function*` вызывало синтаксическую ошибку. Вне функции-генератора ключевое слово `yield` может использоваться в качестве идентификатора.

Использование функций-генераторов для создания итераторов

Поскольку функции-генераторы создают генераторы, а генераторы являются формой итератора, можно использовать синтаксис функции-генератора для написания собственного итератора. Это намного проще и лаконичнее, чем весь представленный ранее код, создававший объект итератора с правильным прототипом и реализующий метод `next`. Давайте еще раз рассмотрим более раннюю реализацию примера итератора; см. Листинг 6-8 (который повторяет ранее приведенный Листинг 6-2).

Листинг 6-8: Базовый пример итеративного объекта (снова) — basic-iterable-example.js

```
// Пример базового итератора без использования функции-генератора
const a = {
    0: "a",
    1: "b",
    2: "c",
    length: 3,
    [Symbol.iterator]() {
        let index = 0;
        const itPrototype = Object.getPrototypeOf(
            Object.getPrototypeOf([][Symbol.iterator]())
        );
        const it = Object.assign(Object.create(itPrototype), {
            next: () => {
                if (index < this.length) {
                    return {value: this[index++], done: false};
                }
                return {value: undefined, done: true};
            }
        });
        return it;
    }
};
for (const value of a) {
    console.log(value);
}
```

Сравните этот код с Листингом 6-9, делающим то же самое с применением функции генератора.

Листинг 6-9: Базовый пример итеративного объекта с применением генератора — iterable-using-generator-example.js

```
const a = {
    0: "a",
    1: "b",
    2: "c",
    length: 3,
    // Следующий пример показывает более простой способ записи
    // следующей строки
    [Symbol.iterator]: function*() {
        for (let index = 0; index < this.length; ++index) {
            yield this[index];
        }
    }
};
for (const value of a) {
    console.log(value);
}
```

Гораздо проще, гораздо понятнее (как только вы узнаете, что означает `yield`). Функция-генератор выполняет всю тяжелую работу за вас. Вы пишете *логику*, а не беспокоитесь о слесарных деталях.

В этом примере используется синтаксис определения свойства:

```
[Symbol.iterator]: function*() {
```

но он мог бы использовать и синтаксис метода, поскольку функции генератора могут быть методами. Давайте рассмотрим этот вариант.

Функции-генераторы в качестве методов

Функции-генератора могут быть методами со всеми функциями методов (например, возможностью использовать синтаксис `super`). Ранее вы видели пример очень простого итеративного класса `LinkedList` с итератором, реализованным вручную. Давайте вернемся к этому классу, реализующему итератор с помощью функции-генератора вместо использования синтаксиса метода. См. Листинг 6-10.

Листинг 6-10: Класс с методом генератора для итератора — generator-method-example.js

```
class LinkedList {
    constructor() {
        this.head = this.tail = null;
    }

    add(value) {
        const entry = {value, next: null};
        if (!this.tail) {
            this.head = this.tail = entry;
        } else {
            this.tail = this.tail.next = entry;
        }
    }

    *[Symbol.iterator]() {
        for (let current = this.head; current; current = current.next) {
            yield current.value;
        }
    }
}

const list = new LinkedList();
list.add("one");
list.add("two");
list.add("three");

for (const value of list) {
    console.log(value);
}
```

Обратите внимание, как объявляется метод генератора: звездочка ставится (`*`) перед именем метода точно так же, как она указывалась перед именем функции после ключевого слова `function`. (В этом примере имя метода использует синтаксис вычисляемого имени, о котором вы узнали в главе 4, но это могло бы быть простое имя, если бы мы не определяли метод с символьным именем.) Это отражает синтаксис объявления метода

геттера и сеттера, используя *, где должны быть ключевые слова get или set. Также обратите внимание, насколько проще, понятнее и лаконичнее версия с генератором, чем версия, закодированная вручную.

Использование генератора напрямую

До сих пор вы видели, как потреблялись значения из генераторов при помощи цикла for-of, как если бы они были просто итераторами. Вы также можете использовать генератор напрямую, как и итераторы, см. Листинг 6-11.

Листинг 6-11: Базовый генератор, используемый напрямую — basic-generator-used-directly.js

```
function* simple() {
    for (let n = 1; n <= 3; ++n) {
        yield n;
    }
}
const g = simple();
let result;
while (!(result = g.next()).done) {
    console.log(result.value);
}
```

Этот код получает объект генератора (g) из функции-генератора (simple), а затем использует его точно так же, как вы итераторы, используемые ранее (поскольку генераторы — это итераторы). Это не очень увлекательно, но становится гораздо интереснее, когда вы начинаете передавать значения генератору, а не просто извлекать из него значения. Об этом вы узнаете в следующем разделе.

Потребление значений генераторами

До сих пор все показанные в этой главе генераторы создавали только значения, предоставляя операнд для оператора yield. Это действительно полезно для написания итераторов и бесконечных последовательностей, таких как числа Фибоначчи и т. п. Но у генераторов есть еще один козырь в рукаве: они тоже могут *потреблять* значения. *Результатом* действия оператора yield становится значение, передаваемое генератору; значение, которое он может *употребить*.

Вы не можете передавать значения в генераторы через цикл for-of, так что вместо этого вы должны использовать их напрямую и передавать значение, которое вы хотите передать генератору, в метод next при его вызове.

Вот действительно простой пример: генератор, суммирующий два введенных в него числа, см. Листинг 6-12.

Листинг 6-12: Базовый двухпоточный генератор — basic-two-way-generator-example.js

```
function* add() {
    console.log("starting");
    const value1 = yield "Please provide value 1";
```

```
    console.log("value1 is " + value1);
    const value2 = yield "Please provide value 2";
    console.log("value2 is " + value2);
    return value1 + value2;
}

let result;
const gen = add();
result = gen.next();    // "starting"
console.log(result);    // {value: "Please provide value 1", done: false}
result = gen.next(35);  // "value1 is 35"
console.log(result);    // {value: "Please provide value 2", done: false}
result = gen.next(7);   // "value2 is 7"
console.log(result);    // {value: 42, done: true}
```

Давайте подробно рассмотрим Листинг 6-12:

1. Выражение `const gen = add()` вызывает функцию генератора и сохраняет возвращаемый ею объект генератора в константе `gen`. Ни одна логика внутри функции генератора еще не запущена — строка «starting» не появляется в логе.
2. Метод `gen.next()` запускает код генератора до первого ключевого слова `yield`. Строка `"starting"` выводится на экран, затем операнд первого оператора `yield` — `"Please provide value 1"` вычисляется и предоставляется генератором вызывающему абоненту.
3. Это значение принимается вызывающим кодом в первом результирующем объекте, который хранится в `result`.
4. Функция `console.log(result)` отображает первый результат: `{value: "Please provide value 1", done: false}`.
5. Метод `gen.next(35)` отправляет значение 35 в генератор.
6. Код генератора продолжается: это значение (35) становится результатом оператора `yield`. (Это генератор, *потребляющий* переданное ему значение.) Код присваивает это значение переменной `value1`, регистрирует его и выдает (yield) значение `"Please provide value 2"`.
7. Вызывающий код получает это значение для следующего результирующего объекта и регистрирует этот новый результирующий объект: `{value: "Please provide value 2", done: false}`.
8. Метод `gen.next(7)` отправляет значение 7 в генератор.
9. Код генератора продолжается: это значение (7) становится результатом оператора `yield`. Код присваивает это значение переменной `value2`, вывод — значение на экран, и возвращает сумму переменный `value1` и `value2`.
10. Вызывающий код получает результирующий объект, сохраняет его в `result` и выводит его на экран. Обратите внимание, что в этот раз `value` — это сумма (42), и свойство `done` равно `true`. Свойство получает значение `true`, поскольку генератор *вернул* это значение, а не *выдал* его.

Этот пример очень прост, но он демонстрирует, как логика в функции генератора прогнозируемо приостанавливается, ожидая следующего ввода от кода, использующего генератор.

Обратите внимание, что в этом примере первый вызов метода `gen.next()` совершается без аргумента. Если вы введете туда аргумент, генератор никогда его не увидит.

Первый вызов метода `next` продвигает генератор от начала функции к первому выражению `yield`, а затем возвращает результирующий объект с первым значением, производимым `yield`. Поскольку код функции генератора получает значения от метода `next` в качестве *результата* `yield`, нет пути получения значения, переданного в первом вызове метода `next`. Чтобы предоставить генератору начальные входные данные, передайте аргумент в списке аргументов функции-генератора, а не в первом вызове метода `next`. (В примере, чтобы дать `add` значение в самом начале, стоит передать его в `add`, а не при первом вызове `next`.)

В представленном выше примере логика генератора не зависит от передаваемых ему значений, но часто эти значения могут использоваться для ветвления внутри генератора. В Листинге 6-13 приведен такой пример. Запустите его в браузере.

Листинг 6-13: Очень простая страница угадывания — guesser.html

```
<!doctype html>
<html>
<head>
<meta charset="UTF-8">
<title>Guesser</title>
<style>
.done.hide-when-done {
    display: none;
}

.running.hide-when-running {
    display: none;
}
</style>
</head>
<body>
<p id="text"> </p>
<input class="hide-when-done" type="button" id="btn-yes" value="Yes">
<input class="hide-when-done" type="button" id="btn-no" value="No">
<input class="hide-when-running" type="button" id="btn-again"
value="Go Again">
<script>
function* guesser() {
    if (yield "Are you employed / self-employed at the moment?") {
        if (yield "Do you work full-time?") {
            return "You're in full-time employment.";
        } else {
            return "You're in part-time employment.";
    }
} else {
        if (yield "Do you spend time taking care of someone instead
                of working?") {
            if (yield "Are you a stay-at-home parent?") {
                return "You're a parent.";
            } else {
                return "You must be a caregiver.";
        }
        } else {
            if (yield "Are you at school / studying?") {
                return "You're a student.";
            } else {
```

```
                if (yield "Are you retired?") {
                    return "You're a retiree.";
                }
                else {
                    return "You're a layabout!;-)";
                }
            }
        }
    }
}

function init() {
    const text     = document.getElementById("text");
    const btnYes   = document.getElementById("btn-yes");
    const btnNo    = document.getElementById("btn-no");
    const btnAgain = document.getElementById("btn-again");

    let gen;

    function start() {
        gen = guesser();
        update(gen.next());
        showRunning(true);
    }

    function update(result) {
        text.textContent = result.value;
        if (result.done) {
            showRunning(false);
    }
}

    function showRunning(running) {
        const {classList} = document.body;
        classList.remove(running ? "done": "running");
        classList.add(running ? "running": "done");
    }

    btnYes.addEventListener("click", () => {
        update(gen.next(true));
    });
    btnNo.addEventListener("click", () => {
        update(gen.next(false));
    });
    btnAgain.addEventListener("click", start);

    start();
}

init();
</script>
</body>
</html>
```

Как вы можете видеть из листинга, этот генератор довольно сильно ветвится в зависимости от потребляемых им значений. Это конечный автомат, но он написан с использованием синтаксиса логического потока, используемого в другом коде JavaScript.

Генераторы обычно задействуют циклы, иногда бесконечные. См. Листинг 6-14: в нем показан генератор «сохранить текущую сумму последних трех входных данных».

Листинг 6-14: Генератор «Сумма последних трех значений» — sum-of-last-three-generator.js

```
function* sumThree() {
    const lastThree = [];
    let sum = 0;
    while (true) {
        const value = yield sum;
        lastThree.push(value);
        sum += value;
        if (lastThree.length > 3) {
            sum -= lastThree.shift();
        }
    }
}

const it = sumThree();
console.log(it.next().value);  // 0  (еще не было передано никаких значений)
console.log(it.next(1).value); // 1  (1)
console.log(it.next(7).value); // 8  (1 + 7)
console.log(it.next(4).value); // 12 (1 + 7 + 4)
console.log(it.next(2).value); // 13 (7 + 4 + 2; 1 "отвалилось")
console.log(it.next(3).value); // 9  (4 + 2 + 3; 7 "отвалилось")
```

Генератор в Листинге 6-14 отслеживает последние три значения, которые вы ему передаете, возвращая обновленную сумму этих последних трех значений при каждом вызове. Когда вы используете его, вам не нужно самостоятельно отслеживать значения или беспокоиться о логике поддержания текущей суммы, включая вычитание из нее значений, когда они выпадают из рабочего диапазона. Вы просто вводите в него значения и используете полученную сумму.

Использование оператора return в функции-генераторе

В кратком обзоре ранее генератор, созданный функцией-генератором, делает интересную вещь, когда вы используете оператор `return` с возвращаемым значением: он создает результирующий объект, содержащий возвращаемое значение и свойство `done = true`. Взгляните на пример:

```
function* usingReturn() {
    yield 1;
    yield 2;
    return 3;
}
console.log("Using for-of:");
for (const value of usingReturn()) {
    console.log(value);
}
// =>
// 1
// 2
```

```
console.log("Using the generator directly:");
const g = usingReturn();
let result;
while (!(result = g.next()).done) {
    console.log(result);
}
// =>
// {value: 1, done: false}
// {value: 2, done: false}
console.log(result);
// =>
// {value: 3, done: true}
```

Есть пара вещей, на которые стоит обратить внимание:

- Цикл `for-of` не видит `value` при свойстве `done = true` у результирующего объекта, поскольку, как и виденный вами ранее цикл `while`, он проверяет свойство `done` и прерывается без поиска значения `value`.
- Финальная функция `console.log(result);` отображает возвращаемое значение вместе `done = true`.

Генератор выдает возвращаемое значение только один раз. Если вы продолжите вызывать метод `next` после получения результирующего объекта со свойством `done = true`, то получите результирующий объект со свойством `done = true` и значением `value = undefined`.

Приоритет оператора `yield`

У оператора `yield` очень низкий приоритет. Это означает, что большая часть следующего за `yield` выражения группируется перед его выполнением. Важно знать об этом, потому что в противном случае разумно выглядящий код может сбить вас с толку. Например, вы потребляете значение в генераторе и хотите использовать это значение в вычислении. На первый взгляд это кажется нормальным:

```
let a = yield + 2 + 30; // Неверно
```

Он запускается, но результаты получаются странными:

```
function* example() {
    let a = yield + 2 + 30; // Неверно
    return a;
}
const gen = example();
console.log(gen.next());   // {value: 32, done: false}
console.log(gen.next(10)); // {value: 10, done: true}
```

Первый вызов просто запускает генератор (помните, что код в функциях-генератора не видит значения, переданного первому вызову метода `next`). Результатом второго вызова является только переданное значение, к нему не добавляются 2 и 30. Почему? Ключ к ответу можно найти в результате первого вызова, содержащем `value = 32`.

Помните, что выражение справа от `yield` — его операнд, который вычисляется и становится значением из метода `next`. Эта строка действительно выполняет это:

```
let a = yield (+ 2 + 30);
```

а на самом деле:

```
let a = yield 2 + 30;
```

потому что унарный оператор «+», который был перед значением `2` не делает ничего с этим значением — это уже число.

При попытке использовать там умножение, а не сложение или вычитание, вы получите синтаксическую ошибку:

```
let a = yield * 2 + 30; // Ошибка
```

Это происходит потому, что нет унарного оператора «*».

Вы могли бы подумать, что стоит просто переместить оператор `yield` в конец:

```
let a = 2 + 30 + yield; // Ошибка
```

но грамматика JavaScript этого не допускает (в основном по историческим причинам и причинам сложности синтаксического анализа). Вместо этого самый понятный способ — просто изложить это в собственном выражении:

```
function* example() {
    let x = yield;
    let a = x + 2 + 30;
    return a;
}
const gen = example();
console.log(gen.next());   // {value: undefined, done: false}
console.log(gen.next(10)); // {value: 42, done: true}
```

Однако по желанию можно вместо этого использовать круглые скобки вокруг `yield`, а затем применить его в любом имеющим смысл месте выражения:

```
function* example() {
    let a = (yield) + 2 + 30;
    return a;
}
const gen = example();
console.log(gen.next());   // {value: undefined, done: false}
console.log(gen.next(10)); // {value: 42, done: true}
```

Методы `return` и `throw`: Завершение работы генератора

Генераторы, созданные функциями-генератора, реализуют оба необязательных метода интерфейса итератора — `return` и `throw`. С их применением использующий генераторы код может, по сути, ввести оператор `return` или `throw` в логику функции-генератора, где находится текущий оператор `yield`.

Ранее в этой главе описано, что, когда генератор выдает оператор `return`, вызывающий код видит результирующий объект со свойствами `value` и `done = true`, содержащими значения:

```
function* example() {
    yield 1;
    yield 2;
    return 3;
}
const gen = example();
console.log(gen.next()); // {value: 1, done: false}
console.log(gen.next()); // {value: 2, done: false}
console.log(gen.next()); // {value: 3, done: true}
```

Используя метод `return` генератора, вызывающий код может выдать оператор `return`, которого на самом деле нет в коде функции генератора. Запустите Листинг 6-15.

Листинг 6-15: Принудительный вызов оператора `return` в функции-генераторе — generator-forced-return.js

```
function* example() {
    yield 1;
    yield 2;
    yield 3;
}
const gen = example();
console.log(gen.next());     // {value: 1, done: false}
console.log(gen.return(42)); // {value: 42, done: true}
console.log(gen.next());     // {value: undefined, done: true}
```

Первый вызов функции `gen.next()` продвигает генератор к первому оператору `yield` и производит значение `1`. Затем вызов функции `gen.return(42)` заставляет генератор вернуть значение `42` — точно так же, как если бы выражение `return 42` было в его коде, где находится первый оператор `yield` (там, где код приостанавливается при вызове `return`). И как раз, если выражение `return 42` будет в коде генератора, функция завершится, последовательно операторы `yield 2` и `yield 3` будут пропущены. Поскольку функция-генератор была завершена, все следующие вызовы метода `next` просто возвращают результирующий объект, содержащий `done = true` и `value = undefined`, как показано в последнем вызове в Листинге 6-15.

Метод `throw` работает таким же образом, вводя инструкцию `throw` вместо `return`. Запустите Листинг 6-16.

Листинг 6-16: Принудительный вызов оператора throw в функции-генераторе — generator-forced-throw.js

```
function* example() {
    yield 1;
    yield 2;
    yield 3;
}
const gen = example();
```

```
console.log(gen.next());                    // {value: 1, done: false}
console.log(gen.throw(new Error("boom"))); // Не перехваченная ошибка -
                                            // Uncaught Error: boom
console.log(gen.next());                    // (строка не будет выполнена)
```

Точно так же, как если бы у генератора было выражение `throw new Error("boom");` в том месте логики, где сейчас находится `yield`.

Так же, как и в случае с операторами `return` и `throw`, генератор может использовать оператор `try/catch/finally` для взаимодействия с введенными выражениями `return` и `throw`. Запустите Листинг 6-17.

Листинг 6-17: Улавливание принудительного вызова оператора `throw` в функции-генераторе — generator-catch-forced-throw.js

```
function* example() {
    let n = 0;
    try {
        while (true) {
            yield n++;
    }
    } catch (e) {
        while (n >= 0) {
            yield n--;
        }
    }
}
const gen = example();
console.log(gen.next());             // {value: 0, done: false}
console.log(gen.next());             // {value: 1, done: false}
console.log(gen.throw(new Error())); // {value: 2, done: false}
console.log(gen.next());             // {value: 1, done: false}
console.log(gen.next());             // {value: 0, done: false}
console.log(gen.next());             // {value: undefined, done: true}
```

Обратите внимание, как выражение `throw` обрабатывается генератором: он отвечает переходом от подсчета вверх (в блоке `try`) к отсчету в обратном порядке (в блокт `catch`). Это не особенно практично само по себе, и это не то, что стоит делать без очень веской причины (было бы лучше написать генератор, чтобы получить возможность передать ему значение через функцию `gen.next()`, и это значение укажет генератору изменить направление). Но такое решение может быть практичным, когда генератор подключается к другому генератору обычным путем с помощью `try/catch` или при вызове одной функции из другой. Подробнее о связанных друг с другом генераторах читайте в следующем разделе.

Остановка генератора или итеративного объекта: yield*

Генератор может передать с помощью выражения `yield*` управление другому генератору (или любому итерируемому), а затем снова подключиться, когда этот генератор (или итерируемый) будет выполнен. Запустите Листинг 6-18.

Листинг 6-18: Пример оператора yield* — yield-star-example.js

```
function* collect(count) {
    const data = [];
    if (count < 1 || Math.floor(count) !== count) {
        throw new Error("count must be an integer >= 1");
    }
    do {
        let msg = "values needed: " + count;
        data.push(yield msg);
    } while (--count > 0);
    return data;
}

function* outer() {
    // Собирает ли `collect` два значения:
    let data1 = yield* collect(2);
    console.log("data collected by collect(2) =", data1);
    // Собирает ли `collect` три значения:
    let data2 = yield* collect(3);
    console.log("data collected by collect(3) =", data2);
    // Возвращает массив результатов
    return [data1, data2];
}

let g = outer();
let result;
console.log("next got:", g.next());
console.log("next got:", g.next("a"));
console.log("next got:", g.next("b"));
console.log("next got:", g.next("c"));
console.log("next got:", g.next("d"));
console.log("next got:", g.next("e"));
```

Выполнение Листинга 6-18 приводит к следующему результату:

```
next got: {value: "values needed: 2", done: false}
next got: {value: "values needed: 1", done: false}
data collected by collect(2) = ["a", "b"]
next got: {value: "values needed: 3", done: false}
next got: {value: "values needed: 2", done: false}
next got: {value: "values needed: 1", done: false}
data collected by collect(3) = ["c", "d", "e"]
next got: {value: [["a", "b"], ["c", "d", "e"]], done: true}
```

Когда функция-генератор `outer` прерывается с помощью `yield*` для выполнения функции `collect`, вызовы метода `next` генератора `outer` вызывают метод `next` для генератора из функции `collect`, производя эти значения и передавая потребляемые. Например, эта строка из функции `outer`:

```
let data1 = yield* collect(2);
```

расшифровывается примерно (но только примерно) так:

```
// Показательно, но не совсем правильно
let gen1 = collect(2);
```

```
let result, input;
while (!(result = gen1.next(input)).done) {
    input = yield result.value;
}
let data1 = result.value;
```

Однако, кроме того, среда выполнения JavaScript гарантирует, что во время выполнения генератором выражения `yield*` вызовы операторов `throw` и `return` распространяются вниз по конвейеру к глубочайшему генератору/итератору и действуют оттуда в обратном направлении. (В то время как в «показательном» коде выше вызов оператора `throw` или `return` для этого генератора не будет передан: он просто вступит в силу в генераторе, для которого был вызван.)

В Листинге 6-19 показан пример использования оператора `return` для завершения работы внутреннего генератора вместе с его внешним генератором.

Листинг 6-19: Перенаправленный оператор return — forwarded-return.js

```
function* inner() {
    try {
        let n = 0;
        while (true) {
            yield "inner " + n++;
        }
    } finally {
        console.log("inner terminated");
    }
}
function* outer() {
    try {
        yield "outer before";
        yield* inner();
        yield "outer after";
    } finally {
        console.log("outer terminated");
    }
}

const gen = outer();
let result = gen.next();
console.log(result);     // {value: "outer before", done: false}
result = gen.next();
console.log(result);     // {value: "inner 0", done: false}
result = gen.next();
console.log(result);     // {value: "inner 1", done: false}
result = gen.return(42); // "inner terminated" (функция inner завершена)
                         // "outer terminated" (функция outer завершена)
console.log(result);     // {value: 42, done: true}
result = gen.next();
console.log(result);     // {value: undefined, done: true}
```

Запустив этот код, обратите внимание, что, когда генератор из функции `outer` использует выражение `yield*` для передачи управления к генератору от функции `inner`, вызов оператора `return` для генератора от `outer` передается в генератор `inner` и начинает действовать там. Это можно проследить при помощи функции `console.log`

в блоке `finally`. Но это вызвало возврат не только там, но и в генераторе `outer`. Это как если бы в генераторе `inner` было значение `return 42`, а генератор `outer` возвращал возвращаемое `inner` значение в местах остановки выполнения генератора с помощью выражения `yield`.

Внешний генератор не должен прекращать выдавать значения только потому, что вы завершили работу внутреннего генератора. В разделе загрузки файл **forwarded-return-with-yield.js** точно так же, как файл **forwarded- return.js** со строкой над `console.log("outer terminated");`:

```
        yield "outer finally";
```

Запустите сценарий **forwarded-return-with-yield.js**. Вы увидите несколько иной вывод:

```
const gen = outer();
let result = gen.next();
console.log(result);     // {value: "outer before", done: false}
result = gen.next();
console.log(result);     // {value: "inner 0", done: false}
result = gen.next();
console.log(result);     // {value: "inner 1", done: false}
result = gen.return(42); // "inner terminated" (функция inner завершена)
console.log(result);     // {value: "outer finally", done: false}
result = gen.next();     // "outer terminated" (функция outer завершена)
console.log(result);     // {value: 42, done: true}
```

Обратите внимание, что вызов `gen.return` *не* вернул переданное в него значение (`42`). Вместо этого он вернул значение `"outer finally"`, созданное в генераторе outer при помощи нового выражения `yield`, и все еще хранит свойство `done = false`. Но позже, после окончательного вызова `next`, когда генератор закончил, он возвращает результирующий объект со значениями `value = 42` и `done = true`. Может показаться странным, что `42` был сохранен, а позже возвращен, но именно так работают блоки `finally` и вне функций генератора. Например:

```
function example() {
    try {
        console.log("a");
        return 42;
    } finally {
        console.log("b");
    }
}
const result = example(); // "a"
                          // "b"
console.log(result);      // 42
```

Генератор может возвращать значение, отличное от указанного в вызове метода `return`. Это, опять же, похоже на использование оператора `return`. Что возвращает эта функция, не являющаяся генератором?

```
function foo() {
    try {
        return "a";
```

```
    } finally {
        return "b";
    }
}
console.log(foo());
```

Верно! Она возвращает `"b"`, потому что сама переопределяет первый метод `return` одним из блока `finally`. Как правило, это плохая практика, но это *возможно*. Функции-генераторы тоже могут выполнять эту операцию, включая переопределение введенного метода `retur`:

```
function* foo(n) {
    try {
        while (true) {
            n = yield n * 2;
        }
    } finally {
        return "override";  // (Как правило, плохая практика)
    }
}
const gen = foo(2);
console.log(gen.next());    // {value: 4, done: false}
console.log(gen.next(3));   // {value: 6, done: false}
console.log(gen.return(4)); // {value: "override", done: true}
```

Обратите внимание, что результирующий объект, возвращаемый из метода `return`, содержит выражение `value = "override"`, а не значение `value = 4`.

Поскольку выражение `yield*` пересылает эти вызовы самому внутреннему генератору, функция `inner` в Листинге 6-19 может делать то же самое. Вот сокращенный пример этой операции:

```
function* inner() {
    try {
        yield "something";
    } finally {
        return "override"; // (Как правило, плохая практика)
    }
}
function* outer() {
    yield* inner();
}
const gen = outer();
let result = gen.next();
console.log(gen.return(42)); // {value: "override", done: true}
```

Обратите внимание, что вызов метода `return` передается от генератора `outer` к `inner`. Это помогает `inner` эффективно передать выражение `return` в строку `yield`, но после это выражение `return` было переопределено одним из выражений блока `finally`. Затем вызов выражения `return` также заставляет функцию `outer` возвратить возвращаемое `inner` значение, даже если в `outer` нет кода для этого.

Использование метода `throw` в этой ситуации проще для понимания: он ведет себя точно так же, как если бы самый внутренний активный генератор/итератор использовал

оператор `throw`. Поэтому, естественно, оно распространяется через (эффективный) стек вызовов, пока/если он не пойман с помощью `try/catch` (или переопределен с помощью `return` в блоке `finally`, как и в предыдущем примере):

```
function* inner() {
    try {
        yield "something";
        console.log("inner - done");
    } finally {
        console.log("inner - finally");
    }
}
function* outer() {
    try {
        yield* inner();
        console.log("outer - done");
    } finally {
        console.log("outer - finally");
    }
}
const gen = outer();
let result = gen.next();
result = gen.throw(new Error("boom")); // функция inner - блок finally
                                       // функция outer - блок finally
                                       // Не перехваченная ошибка - Uncaught
Error: "boom"
```

ОТ СТАРЫХ ПРИВЫЧЕК К НОВЫМ

Перед вами некоторые достойные внимания изменения в коде. Выбирайте подходящие исходя из своего стиля программирования.

Используйте конструкции с итеративными элементами

Старая привычка: Использовать циклы `for` или `forEach`, чтобы перебрать значения массива:

```
for (let n = 0; n < array.length; ++n) {
    console.log(array[n]);
}
// или
array.forEach(entry => console.log(entry));
```

Новая привычка: Если вам не нужен индекс в теле цикла, используйте цикл `for-of`:

```
for (const entry of array) {
    console.log(entry);
}
```

Тем не менее все еще остаются варианты для применения циклов `for` и `forEach`:

- если вам нужна индексная переменная в теле цикла[43];
- если вы передаете ранее существовавшую функцию в цикл `forEach` вместо создания новой.

Используйте возможности итеративных коллекций DOM

Старая привычка: Преобразовать коллекцию DOM в массив только для того, чтобы перебирать ее циклом, или использовать выражение `Array.prototype.forEach.call` для коллекции DOM:

```
Array.prototype.slice.call(document.querySelectorAll("div")).forEach(div =>
{
    // ...
});
// или
Array.prototype.forEach.call(document.querySelectorAll("div"), div => {
    // ...
});
```

Новая привычка: Убедитесь, что коллекции DOM итеративны в вашей среде (возможно, путем полифилирования), и используйте эту итеративность напрямую:

```
for (const div of document.querySelectorAll("div")) {
    // ...
}
```

Используйте интерфейсы итераторов и итеративных объектов

Старая привычка: Определять пользовательские методы итерации для своих собственных типов коллекций.

Новая привычка: Сделать типы коллекций итеративными, реализовав функцию `Symbol.iterator` и итератор (возможно, написанный с использованием функции-генератора).

Используйте синтаксис итеративного расширения в большинстве мест, где вы применяли Function.prototype.apply

Старая привычка: Использовать выражение `Function.prototype.apply` при использовании массива для предоставления дискретных аргументов функции:

```
const array = [23, 42, 17, 27];
console.log(Math.min.apply(Math, array)); // 17
```

[43] В главе 11 вы изучите новый метод `entries` для массивов, который в сочетании с *итеративной деструктуризацией*, описанной в главе 7, также позволит вам при желании использовать цикл `for-of`, даже если вам нужен индекс.

Новая привычка: Использовать синтаксис расширения:

```
const array = [23, 42, 17, 27];
console.log(Math.min(...array)); // 17
```

Используйте генераторы

Старая привычка: Написание очень сложных конечных автоматов, которые можно лучше смоделировать с помощью синтаксиса потока кода.

Новая привычка: Определите логику в функции-генераторе и используйте результирующий объект генератора. Однако для конечных автоматов, которые *не* получится лучше смоделировать с помощью синтаксиса потока кода, функция генератора может оказаться неправильным выбором.

7

Деструктуризация

СОДЕРЖАНИЕ ГЛАВЫ

- Деструктуризация объекта
- Деструктуризация массива/итеративного элемента
- Деструктуризация значений по умолчанию
- Деструктуризация параметров

В этой главе вы узнаете о *деструктуризации*, предоставляющей мощный синтаксис для извлечения значений из объектов и массивов в переменные. Вы узнаете, как новый синтаксис помогает получать доступ к содержимому объектов и массивов более кратко и/или выразительно.

КРАТКИЙ ОБЗОР

Деструктуризация — это извлечение объектов из структуры, в которой они находятся, в данном случае с помощью синтаксиса. Вы выполняли ручную деструктуризацию постоянно во время программирования (вероятно, не называя это так!). Например, принимая значение свойства объекта и помещая его в переменную:

```
var first = obj.first; // Деструктуризация вручную
```

Код проводит *деструктуризацию* переменной `first`, копируя ее из объекта `obj` (то есть извлекая ее из структуры объекта).

Синтаксис деструктуризации, добавленный в ES2015 и расширенный в ES2018, подарил новый, более краткий (хотя и не обязательно) и мощный способ сделать это, предоставляя значения по умолчанию, переименование, вложенность и остаточный синтаксис. Он также обеспечивает четкость и выразительность, особенно при применении к параметрам функций.

БАЗОВАЯ ДЕСТРУКТУРИЗАЦИЯ ОБЪЕКТА

Новый синтаксис дает вам новый способ деструктуризации объектов, обычно с меньшим количеством повторений и меньшим количеством ненужных переменных. (Поначалу это решение может показаться более объемным, но проявите немного терпения.) Вместо того чтобы писать:

```
let obj = {first: 1, second: 2};
let a = obj.first;          // Деструктуризация вручную, старый вариант
console.log(a);              // 1
```

можно написать:

```
let obj = {first: 1, second: 2};
let {first: a} = obj;       // Новый синтаксис деструктуризации
console.log(a);              // 1
```

Эта деструктуризация сообщает движку JavaScript поместить значение свойства `first` в переменную `a`. Опять же, на данный момент это и длиннее, и неудобнее, чем форма записи вручную, но через мгновение вы увидите, как она станет более лаконичной и выразительной.

Обратите внимание, что шаблон деструктуризации `{first: a}` выглядит точно так же, как литерал объекта. Это сделано намеренно: синтаксис для них *точно такой же*, даже несмотря на то, что они служат противоположным целям. Литерал объекта собирает структуру воедино, в то время как шаблон деструктуризации объекта таким же образом разбирает эту структуру на части. В литерале объекта `{first: a}` `first` — это имя создаваемого свойства, `a` — *источник* значения свойства. В шаблоне деструктуризации объекта `{first: a}` все наоборот: `first` — это имя свойства для считывания, `a` — это *цель*, которую нужно поставить в значение. Движок JavaScript узнает, пишете ли вы литерал объекта или шаблон деструктуризации объекта из контекста: литерал объекта не может находиться в левой части оператора присвоения (например), а шаблон деструктуризации нельзя использовать там, где ожидается значение, например, в правой части присвоения.

Тот факт, что литералы объектов и шаблоны деструктуризации объектов — фактически зеркальные отражения друг друга, становится еще более очевидным, если сделать что-то подобное:

```
let {first: a} = {first: 42};
   console.log(a); // 42
```

См. Рисунок 7-1 для визуализации первой строки. Сначала литерал объекта вычисляется, создается новый объект и помещается буквальное значение 42 в свойство `first`, затем при выполнении присвоения правой стороне шаблон деструктуризации принимает значение свойства `first` и помещает его в переменную `a`.

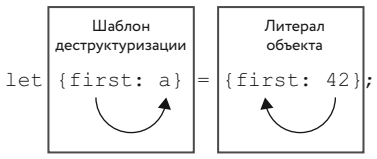


РИСУНОК 7-1

Целью в предыдущем примере была переменная `a`, объявляемая с помощью директивы `let`. Но целью шаблона деструктуризации может быть все, что можно присваивать: переменная, свойство объекта, элемент массива и т. д. — если элемент может находиться в левой части присваивания, вы можете назначить его с помощью деструктуризации. (Это также означает, что вы можете сделать целью другой шаблон деструктуризации. Вы узнаете об этом позже в разделе «Вложенная деструктуризация».)

Давайте продолжим с начальным примером в этом разделе. Так же, как и при ручной деструктуризации, если у объекта нет свойства, которое вы пытаетесь считать, переменная получит значение `undefined`:

```
let obj = {first: 1, second: 2};
let {third: c} = obj;
console.log(c);       // undefined, obj не содержит свойства с именем 'third'
```

Эта деструктуризация точно такая же, как `let c = obj.third;`, поэтому она делает то же самое.

Очень часто возникает желание использовать имя свойства также и в качестве имени переменной, поэтому вместо того, чтобы повторять его таким образом:

```
let obj = {first: 1, second: 2};
let {first: first} = obj;
console.log(first);         // 1
```

вы можете просто не писать двоеточие и имя после него:

```
let obj = {first: 1, second: 2};
let {first} = obj;
console.log(first);         // 1
```

Теперь код более лаконичен, чем эквивалент, поскольку старый эквивалент повторяет название:

```
let first = obj.first;
```

Выражение `{first}` выглядит знакомо, поскольку это противоположность изученной в главе 5 стенографии свойств: если бы это был литерал объекта, было бы создано свойство под названием `first` и получено начальное значение для свойства в области видимости идентификатора с названием `first`. Опять же, синтаксис деструктуризации объекта точно такой же, как синтаксис литерала объекта; разница между ними заключается только в том, что движок JavaScript делает с ним.

Поскольку использование одного и того же имени встречается очень часто, сейчас я собираюсь продемонстрировать это в примерах. Позже в этой главе я вернусь к использованию другого имени для переменной и к причинам, по которым вы, возможно, захотите это сделать.

> **ПРИМЕЧАНИЕ**
> Я использую директиву `let` в различных примерах, но деструктуризация — это не дополнительная возможность директивы `let` (или `const`, или `var`). Как (и где) вы объявляете переменные не имеет значения — деструктуризация не связано с объявлением переменных. Подробнее об этом чуть позже.

Деструктуризация становится действительно мощной, когда вы указываете более одного свойства:

```
const obj = {first: 1, second: 2};
let {first, second} = obj;
console.log(first, second); // 1 2
```

Этот код эквивалентен такому:

```
const obj = {first: 1, second: 2};
let first = obj.first, second = obj.second;
console.log(first, second); // 1 2
```

Вот тут-то и появляется выразительность. Поместив `{first, second}` с левой стороны, при наличии объекта справа, становится ясно (как только вы привыкнете к этому!), что вы берете от объекта. Напротив, в более длинной форме свойство `second` скрыто в середине выражения и легко упускается из виду. Это также позволяет писать/читать меньше кода в этом примере: вместо того, чтобы писать `first`, `second`, и `obj` дважды (или читать их дважды при чтении кода), требуется писать/читать эти слова только один раз.

Однако речь идет не только о краткости и выразительности, но и о том, чтобы избежать ненужных временных переменных. Предположим, вам нужно вызвать функцию, возвращающую объект, и вам просто нужны два свойства от этого объекта. Вместо этого:

```
const obj = getSomeObject();
let first = obj.first;
let second = obj.second;
```

можно написать так:

```
let {first, second} = getSomeObject();
```

Вообще нет необходимости в переменной `obj`!

Деструктуризация не ограничивается инициализаторами переменных/констант. Можно использовать ее в любой операции присвоения (и в списках параметров, как показано в следующем разделе). Есть одна маленькая «загвоздка»: если вы выполняете присваивание, в котором анализатор JavaScript ожидает оператор (а не выражение), вам нужно заключить выражение присваивания в круглые скобки, поскольку в противном случае анализатор обрабатывает начальную фигурную скобку (`{`) как начало блока.

```
let first, second;
// ...
{first, second} = getSomeObject();   // SyntaxError: Неожиданный токен =
({first, second} = getSomeObject()); // Это работает
```

Использование круглых скобок сообщает анализатору, что он работает с выражением, и поэтому начальный знак «{» не может быть началом блока.

Деструктуризация — это просто синтаксический сахар для выбора свойств с помощью эквивалентного кода старого стиля. Зная это, как вы думаете, что произойдет со следующим кодом?

```
let {first, second} = undefined;
```

Это помогает подумать о том, как будет выглядеть версия без синтаксического сахара:

```
const temp = undefined;
let first = temp.first, second = temp.second;
```

Если вы думаете, что это ошибка, вы попали в точку. Как всегда, это ошибка `TypeError` при попытке считывания свойства из значения `undefined` (или `null`).

Аналогично, каков результат здесь?

```
let {first, second} = 42;
```

Нет, это не ошибка, а вопрос с подвохом. Помните версию без синтаксического сахара:

```
const temp = 42;
let first = temp.first, second = temp.second;
```

Как и в любой другой раз, когда вы обрабатываете число как объект, примитивное число приводится к числовому объекту, и в этом примере свойства `first` и `second` считываются из этого объекта. Разумеется, они не существуют (пока кто-то не изменит метод `Number.prototype`!), следовательно `first` и `second` получают значение `undefined`. (Однако попробуйте использовать метод `toString` вместо `first` и запишите тип результата.)

БАЗОВАЯ (И ИТЕРАТИВНАЯ) ДЕСТРУКТУРИЗАЦИЯ МАССИВА

Вы также можете проводить деструктуризацию массивов и других итеративных объектов. Неудивительно, что в синтаксисе используются квадратные скобки (`[]`) вместо фигурных (`{}`), используемых при деструктуризации объектов:

```
const arr = [1, 2];
const [first, second] = arr;
console.log(first, second); // 1 2
```

Фактически как при деструктуризации объекта используется тот же синтаксис, что и при деструктуризации литерала объекта, так и при деструктуризации массива используется тот же синтаксис, что в случае с литералом массива. Получаемое каждой целью

значение зависит от того, где оно находится в шаблоне. Таким образом, при использовании выражения `[first, second]` свойство `first` получает значение из элемента `arr[0]`, поскольку он находится на 0-й позиции, а `second` получает значение из `arr[1]`, поскольку он находится на 1-й позиции. Точно так же, как если бы вы написали:

```
const arr = [1, 2];
const first = arr[0], second = arr[1];
console.log(first, second); // 1 2
```

Допускается не использовать не требующиеся вам элементы. Обратите внимание, что в следующем коде нет переменной в 0-й позиции:

```
const arr = [1, 2];
const [, second] = arr;
console.log(second); // 2
```

Вы также можете оставить пропуски посередине:

```
const arr = [1, 2, 3, 4, 5];
const [, b,,, e] = arr;
console.log(b, e); // 2 5
```

Конечно, как и в случае с литералами массива, которые они зеркально отражают, читаемость страдает, если пропустить больше пары элементов. (Альтернативный вариант см. в разделе «Использование отличных имен» далее в этой главе.)

В отличие от деструктуризации объекта, вам не нужно заключать выражение в круглые скобки, когда вы выполняете присваивание, а не инициализацию, где ожидается оператор. Начальная квадратная скобка (`[`) не является двусмысленной, как начальная фигурная скобка (`{`) — по крайней мере, не в обычной ситуации:

```
const arr = [1, 2];
let first, second;
[first, second] = arr;
console.log(first, second); // 1 2
```

Однако, если вы привыкли полагаться на автоматическую вставку точки с запятой (ASI), а не писать точки с запятой явно, остерегайтесь начинать оператор с квадратной скобки: ASI часто считает, что это часть выражения в конце предыдущей строки, если нет указывающей на обратное точки с запятой. Код в предыдущем примере был бы хорош и без точек с запятой. Но это будет не справедливо, если у вас окажется (например) вызов функции перед деструктуризацией:

```
const arr = [1, 2]
let first, second
console.log("ASI hazard")
[first, second] = arr  // TypeError: Невозможно установить свойство
                       // 'undefined' для undefined
console.log(first, second)
```

Это не удается, потому что анализатор обрабатывает выражение `[first, second]` в качестве свойства-акцессора (с выражением через запятую внутри). Такое выражение устанавливает свойство для любого возвращаемого функцией `console.log` значения, как если бы вы написали

```
console.log("ASI hazard")[first, second] = arr
```

Серия шагов заканчивается попыткой установить значение свойства результату функции `console.log`. Поскольку `console.log` возвращает значение `undefined`, код терпит неудачу — невозможно задать свойства значению `undefined`. (Подобные опасности использования ASI не всегда приводят к появлению сообщений об ошибках: иногда вы просто получаете странные ошибки. Но этот код выдал бы ошибку.)

Если вы полагаетесь на ASI, вам, вероятно, уже известно об этой опасности и выработалась привычка ставить точку с запятой в начале строки, которая в противном случае начиналась бы с открывающей квадратной скобки, что решает проблему:

```
const arr = [1, 2]
let first, second
console.log("ASI hazard")
; [first, second] = arr
console.log(first, second) // 1, 2
```

Использование деструктуризации просто означает, что у вас может быть больше строк, начинающихся с квадратных скобок.

ЗНАЧЕНИЯ ПО УМОЛЧАНИЮ

Как показано в предыдущем разделе, если у объекта нет свойства, указанного вами в вашем шаблоне деструктуризации, цель получит значение `undefined` (точно так же, как при деструктуризации вручную):

```
const obj = {first: 1, second: 2};
const {third} = obj;
console.log(third); // undefined
```

С помощью деструктуризации можно указать значение по умолчанию, применяемое только в случае отсутствия свойства или полученного значения `undefined`:

```
const obj = {first: 1, second: 2};
const {third = 3} = obj;
console.log(third); // 3
```

Это идентично параметрам функции по умолчанию, показанным в главе 3. Значение по умолчанию вычисляется и используется только в том случае, если результатом извлечения свойства становится значение `undefined` (либо потому, что объект получает свойства, либо свойство получило значение `undefined`). И, как и параметры функции по умолчанию, хотя это похоже на уловку `third = obj.third || 3`, это менее проблематично. Значение по умолчанию применяется только в том случае, если

эффективное значение равно `undefined`, а не в том случае, если это любое другое лжеподобное значение[44]:

```
const obj = {first: 1, second: 2, third: 0};
const {third = 3} = obj;
   console.log(third); // 0, не 3
```

«ЛЖЕПОДОБНЫЕ» И «ИСТИННОПОДОБНЫЕ»

Лжеподобное значение — любое, которое приводит к значению `false` при использовании в качестве логического значения, например, в условии оператора `if`. Лжеподобные значения — это `0`, `""`, `NaN`, `null`, `undefined` и, конечно же, `false`. (`document.all` из DOM также лжеподобный, вы узнаете об этом в Главе 17.) Все остальные значения *истинноподобные*.

Давайте подробнее рассмотрим значения деструктуризации по умолчанию. Запустите код в Листинге 7-1.

Листинг 7-1: Значения деструктуризации по умолчанию — default-destructuring-value.js

```
function getDefault(val) {
    console.log("defaulted to " + val);
    return val;
}
const obj = {first: 1, second: 2, third: 3, fourth: undefined};
const {
    third = getDefault("three"),
    fourth = getDefault("four"),
    fifth = getDefault("five")
} = obj;
// "defaulted to four"
// "defaulted to five"
console.log(third); // 3
console.log(fourth); // "four"
console.log(fifth); // "five"
```

Обратите внимание, что функция `getDefault` не вызывалась для `third`, поскольку у `obj` есть свойство с именем `third` и значением не-`undefined`. Но она вызывается для `fourth`, потому что хотя у `obj` есть свойство с именем `fourth`, его значение — `undefined`. Также функция `getDefault` вызывается для `fifth`, потому что у `obj`

[44] *Оператор нулевого слияния*, о котором вы узнаете в Главе 17, приближает вас почти вплотную: утверждение `const third = obj.third ?? 3;` будет использовать только операнд с правой стороны, если значение левого операнда равно `undefined` или `null`. Описание этих вопросов см. в Главе 17. Так что это решение аналогично, но как параметры, так и значения по умолчанию для деструктуризации срабатывают только для значения `undefined`.

вобще нет свойства `fifth`, следовательно, его фактическое значение — `undefined`. Также обратите внимание, что вызовы функции `getDefault` шли по порядку: сначала для `fourth` и потом для `fifth`. Деструктуризация выполняется в порядке исходного кода.

По этой причине более поздние целевые элементы могут ссылаться на значения более ранних целевых элементов в их значениях по умолчанию (например, значения по умолчанию в списках параметров). Так, например, если вы проводите деструктуризацию `a`, `b` и `c` из объекта и хотите, чтобы значением по умолчанию для `c` стало выражение `a * 3`, это можно сделать декларативно:

```
const obj = {a: 10, b: 20};
const {a, b, c = a * 3} = obj;
console.log(c); // 30
```

Вы можете ссылаться только на целевые элементы, определенные *ранее* в шаблоне. Например, такое решение не работает:

```
const obj = {a: 10, b: 20};
const {c = a * 3, b, a} = obj; // ReferenceError: a не определено
console.log(c);
```

В этом будет смысл, если рассматривать версию без синтаксического сахара. Переменная `a` еще не объявлена:

```
const obj = {a: 10, b: 20};
const c = typeof obj.a === "undefined"
? a * 3       // ReferenceError: a не определено
: obj.c;
const b = obj.b;
const a = obj.a;
console.log(c);
```

Однако если переменные уже были объявлены, их можно использовать. Как вы думаете, каков результат выполнения этого кода?

```
let a, b, c;
const obj = {a: 10, b: 20};
({c = a * 3, b, a} = obj);
   console.log(c);
```

Вы сказали, что он выдаст результат `NaN`? Верно! Потому что при вычислении значения по умолчанию для переменной `c`, значение `a` равно `undefined`. А `undefined * 3` равно `NaN`.

СИНТАКСИС REST В ШАБЛОНАХ ДЕСТРУКТУРИЗАЦИИ

Вы видели синтаксис `rest` в главе 3, когда изучали использование остаточного параметра в списке параметров функции. Вы также можете использовать синтаксис rest при деструктуризации, и он работает таким же образом:

```
const a = [1, 2, 3, 4, 5];
const [first, second, ...rest] = a;
console.log(first);      // 1
console.log(second);     // 2
console.log(rest);       // [3, 4, 5]
```

В этом коде `first` получает значение первого элемента массива, `second` — значение второго ввода, и `rest` возвращает новый массив, содержащий остаточные значения. Запись rest должна быть в конце шаблона (точно так же, как *параметр* rest должен быть в конце списка параметров функции).

Синтаксис rest для деструктуризации массива был в ES2015. Стандарт ES2018 также добавил синтаксис rest для деструктуризации объектов, что является обратным расширению свойств объекта (глава 5):

```
const obj = {a: 1, b: 2, c: 3, d: 4, e: 5};
const {a, d, ...rest} = obj;
console.log(a);     // 1
console.log(d);     // 4
console.log(rest);  // {b: 2, c: 3, e: 5}
```

Запись rest получает новый объект со свойствами исходного объекта, которые не были использованы другими записями в шаблоне деструктуризации. В этом примере объект получил свойства `b`, `c` и `e`, но не `a` и `d`, потому что `a` и `d` были поглощены более ранней частью шаблона. Так же, как и запись rest итеративной деструктуризации, она должна быть последней в шаблоне.

ИСПОЛЬЗОВАНИЕ ОТЛИЧАЮЩИХСЯ ИМЕН

За исключением начала этой главы все показанные вам до сих пор деструктуризации объектов использовали имя исходного свойства в качестве имени целевой переменной/константы. Давайте еще раз посмотрим на слегка обновленный пример, показанный раньше:

```
const obj = {first: 1, second: 2};
let {first} = obj;
console.log(first);      // 1
```

Предположим, у вас была веская причина не использовать это имя свойства в качестве имени переменной. Например, имя свойства не является допустимым идентификатором:

```
const obj = {"my-name": 1};
let {my-name} = obj;     // SyntaxError: Неожиданный токен -
let {"my-name"} = obj;   // SyntaxError: Неожиданный токен}
```

Имена свойств могут содержать практически все, что угодно, но имя идентификатора подчинено довольно строгим правилам (например, никаких тире) и, в отличие от имен свойств, для идентификаторов нет формата в кавычках.

Если бы вы делали это вручную, вы бы просто использовали другое имя:

```
const obj = {"my-name": 1};
const myName = obj["my-name"];
console.log(myName);      // 1
```

Вы также можете сделать это с помощью деструктуризации, включив явное имя переменной вместо использования сокращенного синтаксиса:

```
const obj = {"my-name": 1};
const {"my-name": myName} = obj;
console.log(myName);      // 1
```

Это эквивалент созданной ранее вручную версии. Помните, что синтаксис идентичен синтаксису инициализатора объекта. Это включает в себя тот факт, что имя свойства может быть заключено в кавычки.

Если имя свойства представляет собой допустимое имя идентификатора, но вы просто по какой-то причине не хотите его использовать, кавычки не требуются:

```
const obj = {first: 1};
const {first: myName} = obj;
console.log(myName);      // 1
```

Это также удобно, когда вы выбираете набор определенных индексов из массива. Вспомните предыдущий пример, оставляющий пропуски в шаблоне:

```
const arr = [1, 2, 3, 4, 5];
const [, b,,, e] = arr;
console.log(b, e); // 2, 5
```

Такой вариант работает, но отслеживание количества пробелов (особенно перед `e`) затрудняет его чтение. Так как массивы — это объекты, можно сделать это более понятным, используя деструктуризацию объектов, а не деструктуризацию массивов:

```
const arr = [1, 2, 3, 4, 5];
const {1: b, 4: e} = arr;
console.log(b, e); // 2 5
```

Поскольку числовые константы — это допустимые имена свойств, но не допустимые идентификаторы (вы не можете создавать переменные с именами `1` и `4`), кавычки не требуются, но необходимо их переименовать (в `b` и `e` в данном случае).

Эта уловка с индексами работает, потому что индексы массивов — это имена свойств. Следовательно, уловка не работает с итерируемыми объектами в целом, только с массивами. (Если итерируемый объект конечен, вы можете использовать метод `Array.from`, чтобы получить для него массив, а затем применить уловку с индексами, хотя это может быть излишним.)

При явном указании целевого элемента целью деструктуризации может быть все, что возможно использовать в операции присвоения. Это может быть свойство объекта, например:

```
const source = {example: 42};
const dest = {};
({example: dest.result} = source);
console.log(dest.result); // 42
```

ВЫЧИСЛЯЕМЫЕ ИМЕНА СВОЙСТВ

В главе 5 вы узнали о вычисляемых именах свойств в литералах объектов. Поскольку при деструктуризации объектов используется тот же синтаксис, что и при работе с литералами объектов, можно использовать вычисляемые имена свойств при деструктуризации:

```
let source = {a: "ayy", b: "bee"};
let name = Math.random() < 0.5 ? "a" : "b";
let {[name]: dest} = source;
console.log(dest); // "ayy" в половине случаев, "bee" в оставшейся половине
```

В этом коде половину времени переменная `name` получает значение `"a"`, а другую половину она получает значение `"b"`. Следовательно переменная `dest` половину времени получает свое значение из свойства `a`, а вторую половину — из свойства `b`.

ВЛОЖЕННАЯ ДЕСТРУКТУРИЗАЦИЯ

До сих пор вы видели, как значения элементов массива и свойств объекта берутся только с верхнего уровня массива/объекта. Но синтаксис деструктуризации может быть более глубоким, если использовать *вложенность* в шаблоне. Вы никогда не догадаетесь, но то же самое происходит при создании вложенных объектов/массивов в литералах объектов и массивов! Кто бы мог подумать. Давайте рассмотрим это.

Напомним, что в следующем примере массив `array` — целевой элемент для значения свойства `a`:

```
const obj = {a: [1, 2, 3], b: [4, 5, 6]};
let {a: array} = obj;
console.log(array); // [1, 2, 3]
```

В этом случае целевой элемент (часть справа от:) — это переменная или константа и т. д., в которую должно записываться значение деструктуризации. Это было справедливо для всех примеров, которые вы видели до сих пор. Но если просто требуются первые две записи в этом массиве в качестве дискретных переменных, можно вместо этого создать целевой шаблон деструктуризации массива с целевыми переменными внутри него:

```
const obj = {a: [1, 2, 3], b: [4, 5, 6]};
let {a: [first, second]} = obj;
console.log(first, second); // 1 2
```

Обратите внимание, как это точно отражает создание объекта со свойством, инициализированным новым массивом, построенным с использованием значений из `first` и `second`. Мы просто используем его по другую сторону знака равенства, потому что разбираем структуру на части, а не собираем ее в одно целое.

Естественно, это работает так же хорошо и для объектов:

```
const arr = {first: {a: 1, b: 2}, second: {a: 3, b: 4}};
const {first: {a}, second: {b}} = arr;
console.log(a, b); // 1 4
```

Код получил свойство `a` из объекта, ссылающегося на свойство `first`, и свойство `b` из объекта, ссылающегося на свойство `second`.

Внешняя структура также может быть массивом вместо объекта:

```
const arr = [{a: 1, b: 2}, {a: 3, b: 4}];
const [{a}, {b}] = arr;
console.log(a, b); // 1 4
```

Этот код получил свойство `a` из объекта в первой записи массива и свойство `b` из объекта во второй записи массива.

Как и в случае с литералами объектов и массивов, объем вложенности, который вы можете использовать, практически неограничен.

ДЕСТРУКТУРИЗАЦИЯ ПАРАМЕТРОВ

Деструктуризация предназначена не только для операций присвоения. Вы также можете деструктировать параметры функции:

```
function example({a, b}) {
    console.log(a, b);
}
const o = {a: "ayy", b: "bee", c: "see", d: "dee"};
example(o);              // "ayy" "bee"
example({a: 1, b: 2});   // 1 2
```

Обратите внимание на деструкцию объекта (`{a, b}`), используемую в списке параметров. Функция `example` принимает один параметр и деструктурирует его на две локальные привязки, почти как если написать:

```
function example(obj) {
    let {a, b} = obj;
    console.log(a, b);
}
```

(Это «почти» из-за временного параметра `obj` и потому, что, если список параметров содержит какие-либо значения параметров по умолчанию, это происходит в области, специфичной для списка параметров, а не в теле функции. Кроме того, параметры больше похожи на переменные, объявленные с помощью `var`, чем на объявленные с помощью `let` переменные, по историческим причинам.)

Деструктурированный параметр не обязательно должен быть первым или единственным, он может находиться в любом месте списка параметров:

```
function example(first, {a, b}, last) {
    console.log(first, a, b, last);
}
```

```
const o = {a: "ayy", b: "bee", c: "see", d: "dee"};
example("alpha", o, "omega");               // "alpha" "ayy" "bee" "omega"
example("primero", {a: 1, b: 2}, "ultimo"); // "primero" 1 2 "ultimo"
```

Концептуально вы можете представить это как один большой вложенный шаблон деструктуризации массива, где список параметров заключен в квадратные скобки, чтобы сформировать шаблон деструктуризации массива, а затем знак равенства и массив аргументов справа (рисунок 7-2).

РИСУНОК 7-2

Следовательно, эта функцию и ее вызов:

```
function example(first, {a, b}, last) {
    console.log(first, a, b, last);
}
example(1, {a: 2, b: 3}, 4); // 1 2 3 4
```

работает точно так же, как присвоение, с аргументами в виде массива:

```
let [first, {a, b}, last] = [1, {a: 2, b: 3}, 4];
console.log(first, a, b, last); // 1 2 3 4
```

Естественно, вы также можете использовать итеративную деструктуризацию:

```
function example([a, b]) {
    console.log(a, b);
}
const arr = [1, 2, 3, 4];
example(arr);                  // 1, 2
example(["ayy", "bee"]);       // "ayy", "bee"
```

И аналогично деструктуризация значений по умолчанию:

```
function example({a, b = 2}) {
    console.log(a, b);
}
const o = {a: 1};
example(o);              // 1 2
```

Переменная `b` получила значение по умолчанию, поскольку ее нет в `o`. Она также получила бы значение по умолчанию, если находилась бы в `o`, но ее значение было бы `undefined`.

В этом примере, хотя b не существовало для передаваемого объекта, объект все равно *был* передан. Но как насчет случая, когда вообще не передается ни один объект? Учитывая ту же функцию example, что, по вашему мнению, происходит со следующим вызовом?

```
example();
```

Если вы решили, что результатом станет ошибка, то попали в точку. Это словно сделать так:

```
let {a, b = 2} = undefined;
```

или так, если рассматривать весь список параметров:

```
let [{a, b = 2}] = [];
```

Невозможно считать свойства из значения undefined, попытка это сделать приводит к ошибке типов TypeError.

Вернемся к тому, что вы узнали в главе 3. Что можно использовать для случая, когда объект вообще не передается?

Да! Значение параметра по умолчанию. Держитесь за свои шляпы:

```
function example({a, b = "prop b def"} = {a: "param def a", b: "param def b"}) {
    console.log(a, b);
}
example();                      // "param def a" "param def b"
example({a: "ayy"});            // "ayy " "prop b def"
```

Код может немного сбить с толку, поэтому давайте изобразим логику на схеме (рисунок 7-3):

- Выражение {a, b = "prop b def"} — это шаблон деструктуризации параметра.
- Фрагмент = "prop b def" шаблона — значение по умолчанию для деструктурированного свойства b (если его нет в переданном объекте или значение равно undefined).
- Выражение = {a: "param def a", b: "param def b"} — значение параметра по умолчанию (если для этого параметра не передавались значения, или значение аргумента равно undefined).

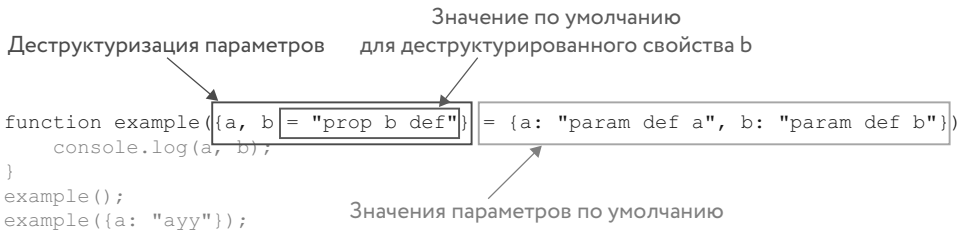


РИСУНОК 7-3

Используя эти знания, давайте рассмотрим два вызова. Первый вызов:

```
example();
```

ничего не передавал для параметра, поэтому было применено значение параметра по умолчанию `{a: "param def a", b:"param def b"}`. Выражение содержит значение для `b`, таким образом деструктуризованное свойство `b` получает значение `"param def b"`. Второй вызов:

```
example({a: "ayy"});
```

передает объект без значения `b`, следовательно, значение параметра по умолчанию не применяется (аргумент для параметра есть, и его значение не равно `undefined`), но деструктуризованное значение по умолчанию для `b` применяется, поскольку у объекта нет свойства `b`. Таким образом `b` получает значение `"b def"`.

Когда вы работали над этим примером, вам, возможно, пришла в голову мысль о параметрах функции по умолчанию. Если этого не произошло, остановитесь на мгновение и подумайте о различных изученных в этом разделе аспектах. (Подсказка: это связано со списками параметров, являющимися неявными шаблонами деструктуризации, см. рисунок 7–3…)

Поняли? Это немного сложно, не переживайте, если пока этого не видите! Вот ответ: параметры функции по умолчанию фактически являются просто значениями деструктуризации по умолчанию. Список параметров функции `example` эквивалентен списку деструктуризации при вызове `example` без параметров:

```
let [{a, b = "b def"} = {a: "param def a", b: "param def b"}] = [];
console.log(a, b); // "param def a", "param def b"
```

Последнее замечание о деструктуризации в списках параметров: использование деструктуризации в списке параметров делает его *непростым*. В главе 3 рассказывается, что функция с непростым списком параметров не может содержать директиву `"use strict"`. Если вы хотите, чтобы она работала в строгом режиме, она должна отображаться в находящемся в строгом режиме контексте.

ДЕСТРУКТУРИЗАЦИЯ В ЦИКЛАХ

В JavaScript есть присвоения, которые не похожи на присвоения. В частности, циклы `for-in` и `for-of`[45] присваивают значения переменной цикла в начале каждой итерации цикла. В них тоже можно использовать деструктуризацию. От этого мало пользы в цикле `for-in` (ключи, предоставляемые циклом `for-in`, всегда являются строками, и редко требуется провести деструктуризацию строки, хотя такая возможность есть, поскольку строки итерируемые), но это может быть очень удобно при использовании цикла `for-of`.

Предположим, есть массив объектов, и необходимо выполнить цикл по нему, используя свойства `name` и `value` объектов:

[45] И цикл `for-await-of`, о нем вы узнаете в Главе 9.

```
const arr = [
    {name: "one", value: 1},
    {name: "two", value: 2},
    {name: "forty-two", value: 42}
];
for (const {name, value} of arr) {
    console.log("Name: " + name + ", value: " + value);
}
```

Или предположим, что есть объект, и требуется перебрать его «собственные» (не унаследованные) свойства имен и значений. Можно использовать цикл `for-in`:

```
const obj = {a: 1, b: 2, c: 3};
   for (const name in obj) {
    if (obj.hasOwnProperty(name)) {
        const value = obj[name];
        console.log(name + " = " + value);
}
}
```

Или вы могли бы избавиться от этой проверки методом `hasOwnProperty` с помощью цикла `for-of` и изученной в главе 5 функции `Object.entries`:

```
const obj = {a: 1, b: 2, c: 3};
for (const entry of Object.entries(obj)) {
    console.log(entry[0] + " = " + entry[1]);
}
```

Это не очень значимые имена. С помощью итеративной деструктуризации можно сделать этот код намного понятнее:

```
const obj = {a: 1, b: 2, c: 3};
for (const [name, value] of Object.entries(obj)) {
    console.log(name + " = " + value);
}
```

ОТ СТАРЫХ ПРИВЫЧЕК К НОВЫМ

По сути, в этом разделе можно просто написать: «Используйте деструктуризацию везде, где это имеет смысл». Но вот несколько конкретных примеров, где смысл действительно есть.

Используйте деструктуризацию при получении только некоторых свойств от объекта

Старая привычка: Получать объект из функции и запоминать его в переменной только для того, чтобы затем использовать лишь небольшое количество его свойств:

```
const person = getThePerson();
console.log("The person is " + person.firstName + " " + person.lastName);
```

Новая привычка: Используйте деструктуризацию в ситуациях, когда позже сам объект не будет востребован:

```
const {firstName, lastName} = getThePerson();
console.log("The person is " + firstName + " " + lastName);
```

Используйте деструктуризацию для объектов options

Старая привычка: Использование объектов options для функций с большим количеством параметров.

Новая привычка: Рассмотрите возможность использования деструктуризации.

Перед деструктуризацией функция, принимающая пять параметров, где значение параметра `five` по умолчанию основано на значении параметра `one`, может выглядеть следующим образом:

```
function doSomethingNifty(options) {
    options = Object.assign({}, options, {
        // Параметры по умолчанию
        one: 1,
        two: 2,
        three: 3,
        four: 4
});
    if (options.five === undefined) {
        options.five = options.one * 5;
}
    console.log("The 'five' option is: " + options.five);
    // ...
}
doSomethingNifty(); // Параметр 'five': 5
```

Это скрывает значения по умолчанию внутри кода функции. Такой подход означает, что любое сложное значение по умолчанию должно быть выделено в специальный случай. Таким образом вам предстоит выполнять доступ к свойству вашего объекта `options` каждый раз, когда вы используете параметр.

Вместо этого можно определить свои параметры по умолчанию как значения по умолчанию для деструктуризации. Обработайте значение по умолчанию `five` как значение по умолчанию для деструктуризации, предоставьте общий пустой объект для значения параметра по умолчанию и используйте полученные привязки напрямую:

```
function doSomethingNifty({
        one = 1,
        two = 2,
        three = 3,
        four = 4,
        five = one * 5
} = {}) {
    console.log("The 'five' option is: " + five);
    // ...
}
doSomethingNifty(); // Параметр 'five': 5
```

8

Объекты Promise

СОДЕРЖАНИЕ ГЛАВЫ

- Создание и применение промисов
- Объекты «promise» и «thenable»
- Шаблоны промисов
- Антишаблоны промисов

В этой главе вы узнаете о *промисах* (*Promise*) ES2015, упрощающих и стандартизируютщих обработку одноразовых асинхронных операций, сокращая «ад обратного вызова» (и прокладывая путь для асинхронных функций ES2018, описанных в главе 9). Промис — это объект, представляющий асинхронный результат (например, завершение HTTP-запроса или запуск одноразового таймера). Промисы используются для наблюдения за результатом асинхронного процесса. Также можно дать промис другому коду, чтобы он мог наблюдать этот результат. Промисы — это версия шаблона JavaScript, который называется по-разному: *промис*, *обещание*, *фьючерс* или *отсрочка*. Он в значительной степени опирается на уровень техники, в частности на спецификацию Promises/A+1 и предшествующую ей работу.

Прежде чем мы начнем: возможно, вы слышали, что промисы устарели, поскольку в ES2018 добавлены функции `async` (глава 9). По большей части это неверно. Это правда, что вы будете взаимодействовать с промисами по-другому при использовании функций `async`. Но вы все равно будете взаимодействовать с ними, и вам все равно нужно получить четкое представление о том, как они работают. Кроме того, бывают случаи, когда вы будете использовать промисы напрямую, даже в функции `async`.

ПОЧЕМУ ЖЕ ПРОМИСЫ?

Промис не делает ничего сам по себе; это просто способ наблюдать за результатом чего-то асинхронного. Промисы не делают операции асинхронными. Они просто предоставляют средство *наблюдения* за завершением операций, которые *уже* являются асинхронными. Так что же в них хорошего?

До промисов для этого обычно использовались простые обратные вызовы, но у них есть некоторые недостатки:

- Объединение нескольких асинхронных операций (последовательно или параллельно) быстро приводит к глубоко вложенным обратным вызовам («ад обратного вызова»).
- У обратных вызовов нет стандартного способа указать на возникшую ошибку. Вместо этого в каждой применяемой функции или API необходимо определять, как они должны сообщать об ошибке. Существует несколько распространенных отличных друг от друга способов, что затрудняет объединение нескольких библиотек/модулей или переход от проекта к проекту.
- Отсутствие стандартного способа индикации успеха/неудачи означает, что вы не можете использовать общие инструменты для управления сложностью.
- Добавление обратного вызова к уже завершенному процессу также не стандартизировано. В некоторых случаях это означает, что обратный вызов никогда не вызывается. В других он вызывается синхронно, а иногда асинхронно. Проверка описания для каждого используемого API при каждом возникновении такой ситуации отнимает много времени и легко забывается. Ошибки, которые вы получаете, когда предполагаете или неправильно запоминаете, часто незаметны и их трудно обнаружить.
- Добавление нескольких обратных вызовов к операции либо невозможно, либо не стандартизировано.

Промисы решают все эти проблемы, предоставляя простую стандартную семантику. Вы увидите, как это происходит, когда будете изучать эту семантику на протяжении всей главы. Попутно вы увидите примеры использования промисов по сравнению с простым применением обратных вызовов, иллюстрирующим указанные выше проблемы.

ОСНОВЫ ПРОМИСОВ

В этом разделе сначала рассматриваются основы промисов. В последующих разделах вы узнаете более подробно о различных аспектах их создания и использования. Впереди довольно много терминологии.

Краткий обзор

Промис — это объект с тремя возможными состояниями:

- *Ожидание:* Промис находится в ожидании/выполняется/еще не выполнен.
- *Успешно выполнен:* Промис был выполнен со значением. Обычно это означает успешное выполнение.
- *Выполнен с ошибкой (отклонен):* Промис был выполнен с указанием причины отклонения. Обычно это означает сбой или отклонение выполнения.

Промис начинается с ожидания, а затем может быть *выполнен* (успешно выполнен или выполнен с ошибкой) только один раз. Выполненный промис не может вернуться

в состояние ожидания. Успешно выполненный промис не может измениться на выполненный с ошибкой или наоборот. После выполнения состояние промиса никогда не меняется.

Успешно выполненный промис получает *значение успешного выполнения* (обычно просто *значение*). Выполненный с ошибкой промис получает причину отклонения (иногда просто *причину*, хотя *ошибка* также встречается часто).

Когда промис *получает решение*, он либо выполняется со значением, либо ставится в зависимость от другого промиса[46]. Когда он ставится в зависимость от другого промиса, вы *разрешаете его значение* другим промисом. Это означает, что в конечном счете он может быть выполнен или отклонен в зависимости от того, что произойдет с другим промисом.

Когда выполнение промиса отклонено, должна быть указана причина этого отклонения.

ОШИБКИ И ПРИЧИНЫ ОТКЛОНЕНИЯ

В коде этой главы вы заметите, что все примеры причин отклонения являются экземплярами `Error`. Я делаю так, потому что отклонение промиса подобно использованию инструкции `throw`. В обоих случаях полезно получить информацию о стеке, которую предоставляет Error. Однако это вопрос стиля и лучших практик. Можно использовать любое желаемое значение в качестве причины отклонения — как и любое желаемое значение в `throw` и `try/catch`.

У вас нет возможности напрямую наблюдать за состоянием промиса и значением/причиной отклонения (если таковые имеются). Их можно получить, только добавив функции-обработчики, которые вызывает промис. У промиса JavaScript есть три метода для регистрации обработчиков:

- `then`: Добавляет обработчик *успешнго выполнения* для вызова если/когда промис выполнен успешно[47].
- `catch`: Добавляет обработчик для состояния *выполнено с ошибкой* для вызова если/когда промис был отклонен.
- `finally`: Добавляет *итоговый* (*finally*) *обработчик* для вызова если/когда промис был выполнен, независимо от того, с каким результатом (добавлено в ES2018).

Один из ключевых аспектов промисов заключается в том, что `then`, `catch` и `finally` *возвращают новый промис* в качестве результата. Новый промис связан с промисом, для которого вы вызвали метод: он будет выполнен или отклонен в зависимости от того, что происходит с этим исходным промисом и что делают ваши функции-обработчики. Мы вернемся к этому ключевому моменту по ходу главы.

---

[46] Или элемента thenable. Вы узнаете об элементе thenable подробнее далее в этой главе.

[47] Это то, что делает версия с одним аргументом. Вы также можете вызвать его с двумя аргументами, о чем вы узнаете в разделе «Метод `then` с двумя аргументами» далее в этой главе.

Пример

В Листинге 8-1 приведен простой пример создания и использования промиса с использованием `setTimeout` для наблюдаемой асинхронной операции. Запустите пример несколько раз. В половине случаев выдаваемый промис исполняется, в другой половине отклоняется.

Листинг 8-1: Простой пример промиса — simple-promise-example.js

```
function example() {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            try {
                const succeed = Math.random() < 0.5;
                if (succeed) {
                    console.log("resolving with 42 (will fulfill the promise)");
                    resolve(42);
                } else {
                    console.log("rejecting with new Error('failed')");
                    throw new Error("failed");
                }
            } catch (e) {
                reject(e);
            }
        }, 100);
    });
}

example()
.then(value => { // обработчик выполнения
    console.log("fulfilled with", value);
})
.catch(error => { // обработчик отклонения
    console.error("rejected with", error);
})
.finally(() => { // итоговый обработчик
    console.log("finally");
});
```

В этом коде три части:

1. Создание промиса: выполняется функцией `example`, она создает и возвращает промис.
2. Разрешение или отклонение промиса: выполняется обратным вызовом таймера функции `example`.
3. Применение промиса: выполняется с помощью кода, вызывающего `example`.

Функция `example` использует конструктор `Promise` для создания промиса, передаваемого в функцию. У функции, которой вы передаете конструктор `Promise`, причудливое имя — *функция-исполнитель* (*executor function*). Конструктор `Promise` вызывает функцию-исполнитель (синхронно), передавая ей две функции в качестве аргументов: `resolve` и `reject`. Это общепринятые названия. Но, конечно, вы можете использовать любые желаемые названия для параметров в вашей функции-исполнителе. Иногда можно

встретить аргумент с именем `fulfill` (выполнено) вместо `resolve` (разрешено), хотя это не очень правильно, в чем вы сейчас убедитесь.

Функции `resolve` и `reject` определяют, что происходит с промисом:

- Если вызвать `resolve` с другим промисом[48], он будет *разрешен* другим промисом (он будет выполнен или отклонен в зависимости от того, что произойдет с этим другим промисом). Вот почему `fulfill` не является точным именем для первой функции. Когда она вызывается с другим промисом, она не выполняет созданный вами промис, а просто разрешает его другим промисом, которой может быть отклонен.
- Если вы вызываете `resolve` с любым другим значением, промис выполняется с этим значением.
- При вызове `reject` промис отклоняется с использованием переданной вами в `reject` причины (обычно `Error`, но это вопрос стиля). В отличие от `resolve`, выполнение `reject` не отличается, если вы передаете ей промис. Он всегда использует полученное значение в качестве причины отклонения.

Промис может быть разрешен или отклонен *только* с помощью этих функций. Ничто не может разрешить или отклонить его без их применения. Они недоступны в качестве методов для самого объекта promise, поэтому вы можете передать объект promise другому коду, зная, что получающий его код может только использовать промис, а не разрешать или отклонять его.

В случае `example` вызывается метод `setTimeout` для запуска асинхронной операции. Код должен использовать `setTimeout` или что-то подобное, потому что, опять же, промисы — это просто способ *наблюдать* за завершением асинхронных операций, а не *создавать* асинхронные операции[49]. По истечении таймера обратный вызов таймера `example` вызывает функцию `resolve` или `reject`, в зависимости от случайного результата.

Переходим к тому, как используется промис. Использующий функцию `example` код применяет цепочку вызовов, в данном случае за `then` следует `catch`, за ним `finally` — для подключения обработчика выполнения, обработчика отклонения и итогового обработчика соответственно. Объединение в цепочки довольно часто встречается в методах промисов по причинам, о которых вы узнаете по мере изучения главы.

Запустите код в Листинге 8-1 достаточное количество раз, чтобы увидеть как выполнение, так и отклонение. Когда промис выполняется, цепочка вызывает обработчик выполнения со значением выполнения, затем обработчик `finally` (который не получает никакого значения). Когда промис отклоняется, цепочка вызывает обработчик отклонения с указанием причины отклонения, а затем обработчик `finally` (который, опять же, не получает никакого значения). Он также вызовет обработчик отклонения (а затем обработчик `finally`), если исходный промис был выполнен, но обработчик выполнения выдал ошибку — подробнее об этом чуть позже.

Обработчики выполнения, отклонения и `finally` служат тем же целям, что и блоки в операторах `try`/ `catch`/`finally`:

---

[48] Или, опять же, любой элемент thenable. Вы изучите элемент thenable в следующем разделе.

[49] Здесь есть небольшое предостережение, к которому мы вернемся позже.

```
try {
    // выполнение какой-то задачи
    // обработчик выполнения
} catch (error) {
    // обработчик отклонения
} finally {
    // итоговый обработчик
}
```

Есть небольшие различия, о которых вы узнаете позже, особенно в отношении итогового блока `finally`. Но это хороший способ задуматься об обработчиках выполнения, отклонения и `finally`.

Использование выражения `new Promise` встречается довольно редко по сравнению с использованием промисов из функции API или получаемых из `then`, `catch` и `finally`. Поэтому мы сосредоточимся в первую очередь на использование промисов, затем вернемся к подробностям их создания.

Промисы и элементы thenable

Промисы JavaScript следуют правилам, определенным спецификацией ECMAScript, которые в значительной степени основаны на предшествующем уровне техники, в частности на спецификации Promises/A+ и предшествующим ей работам. Промисы JavaScript полностью соответствуют спецификации Promises/A+ и содержат некоторые дополнительные функциональные возможности, намеренно не охватываемые этой спецификацией, такие как `catch` и `finally`. (Спецификация Promises/a+ намеренно минималистична и определяет только выражение `then`, потому что возможности `catch` и `finally` могут быть реализованы при помощи `then` в версии с двумя аргументами. Об этом будет рассказано в следующем разделе.)

Одним из аспектов спецификации Promises/A+ представляется концепция «thenable», отличная от «promise». Вот как эта спецификация определяет их:

- «Промис» — это объект или функция с методом `then`, поведение которого соответствует спецификации Promises/A+.
- «Thenable» — это объект или функция, которая определяет метод `then`.

Таким образом, все промисы являются thenable, но не все thenable являются промисами. Объект может определять метод с именем `then`, не работающим так, как определено для промисов. Такой объект будет относиться к thenable, но не к промисам.

Почему это так важно? Совместимость.

Иногда реализации промиса необходимо знать, является ли значение простым значением, которое реализация может использовать для выполнения промиса, или thenable, которое должно разрешить промис. Код делает это, проверяя, является ли значение объектом с методом `then`. Если это так, предполагается, что объект относится к thenable, и реализация промиса использует свой метод `then` для преобразования промиса в thenable. Это несовершенно, поскольку у объекта может быть метод `then`, означающий что-то совершенно не связанное с промисами. Но это было лучшее решение в то время, когда промисы добавлялись в JavaScript, потому что оно позволяло нескольким существующим на тот момент (и все еще существующим) библиотекам промисов, взаимодействовать без необходимости обновления каждой реализации с добавлением

какой-либо другой функции, чтобы пометить его промисы как промисы. (Например, это был бы отличный вариант использования символа [глава 5], если бы можно было обновить каждую библиотеку, чтобы добавить его.)

До этого момента в нескольких местах я использовал термин «промис» со сноской, хотя «thenable» был бы более точным. Теперь, когда вы знаете, что такое thenable, я начну использовать его там, где это уместно.

ИСПОЛЬЗОВАНИЕ СУЩЕСТВУЮЩЕГО ПРОМИСА

Как вы видели, подключение к выполнению, отклонению или завершению промисов осуществляется с использованием обработчиков, зарегистрированных с помощью `then`, `catch` и `finally`. В этом разделе вы узнаете об этом более подробно, изучите некоторые общие шаблоны использования промисов, а также узнаете, как промисы решают некоторые проблемы, упомянутые ранее в разделе «Почему же промисы?».

Метод `then`

Вы уже бегло просмотрели связанный с выполнением промиса метод `then` в действии[50]. Давайте рассмотрим `then` немного подробнее. Рассмотрим этот очень простой вызов `then`:

```
p2 = p1.then(result => doSomethingWith(result.toUpperCase()));
```

Как вы уже узнали, код регистрирует обработчик выполнения, создавая и возвращая новый промис (`p2`), который будет выполнен или отклонен в зависимости от того, что происходит с исходным промисом (`p1`) и тем, что выполняется в вашем обработчике. Если `p1` отклоняется, обработчик не вызывается, а `p2` отклоняется с указанием причины отклонения `p1`. Если `p1` выполняется, вызывается обработчик. Вот что происходит с `p2` в зависимости от того, что делает ваш обработчик:

- Если вы возвращаете элемент thenable, `p2` будет разрешен для него (он будет выполнен или отклонен в зависимости от того, что произойдет с этим thenable).
- Если вы возвращаете любое другое значение, `p2` выполняется с этим значением.
- Если вы используете метод `throw` (или вызываете функцию с `throw`), `p2` отклонятся, используя результат `throw` (обычно `Error`) в качестве причины отклонения.

Эти три пункта, вероятно, кажутся знакомыми. Это потому, что они почти идентичны показанным ранее пунктам, в которых говорится, что вы можете делать с получаемыми в функции-исполнителе промиса функциями `resolve` и `reject`. Возврат чего-либо из вашего обработчика `then` аналогичен вызову `resolve`. Использование `throw` в обработчике аналогично вызову `reject`.

Эта возможность возвращать thenable — один из ключей к промисам. Поэтому давайте рассмотрим ее более внимательно.

---

[50] Позже вы узнаете о версии с двумя аргументами, также способной привести к отклонению.

Связывание промисов в цепочки

Вы можете вернуть промис (или какой-либо элемент thenable) обработчика `then`/`catch`/`finally`, чтобы разрешить созданный ими промис. Это означает, что, если требуется реализовать серию асинхронных операций, обеспечивающих промисы/thenable, можно использовать `then` для первой операции и оставить обработчик для возврата thenable для второй операции. И так может повторяться нужное количество раз. Вот цепочка с тремя операциями, возвращающими промисы:

```
firstOperation()
   .then(firstResult => secondOperation(firstResult)) // или:.
then(secondOperation)
.then(secondResult => thirdOperation(secondResult * 2))
.then(thirdResult => {/* Use `thirdResult` */})
.catch(error => {console.error(error);});
```

Когда этот код выполняется, функция `firstOperation` запускает первую операцию и возвращает промис. Вызовы `then` и `catch` настраивают обработчики для выполнения того, что произойдет дальше (рисунок 8-1). Дальнейшее зависит от произошедшего в первой операции: если она выполняет свой промис, запускается первый обработчик выполнения и запускает вторую операцию, возвращая промис, предоставляющий функцию `secondOperation` (рисунок 8-2).

Первая операция запускается, возвращает промис; `then/catch` настраивает цепочку

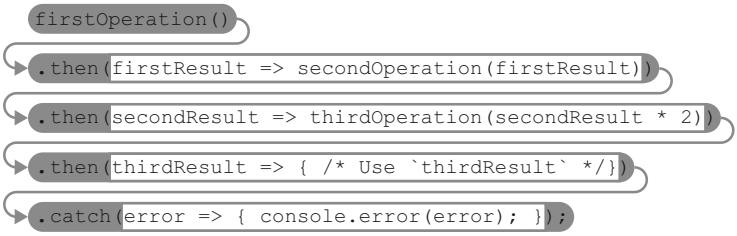


РИСУНОК 8-1

Первая операция выполнена успешно, обработчик запускает вторую

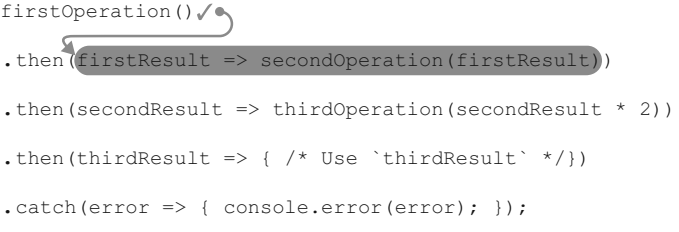


РИСУНОК 8-2

Если вторая операция выполняет свой промис, это выполняет промис, возвращенный первым `then`, и запускается следующий обработчик выполнения (рисунок 8-3): он запускает третью операцию и возвращает промис ее результата.

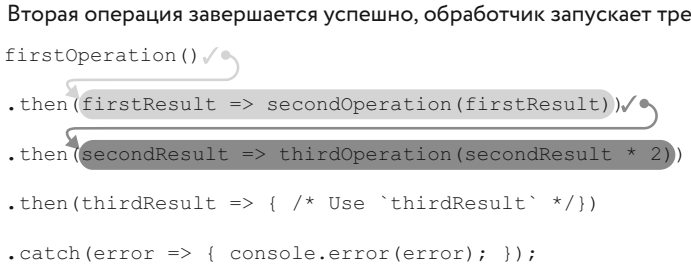


РИСУНОК 8-3

Если третья операция выполняет свой промис, это выполняет промис и второго `then`. Это вызывает код, использующий третий результат. При условии, что код не выдает или не возвращает отклоненный промис, цепочка завершается (рисунок 8-4). Обработчик отклонения в этом случае вообще не запускается, поскольку отклонений не было.

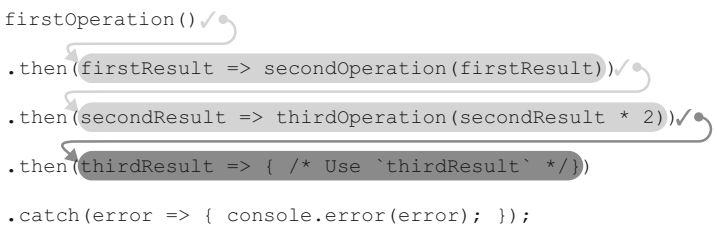


РИСУНОК 8-4

Если первая операция отклоняет свой промис, это отклоняет промис из первого `then`, что отклоняет промис второго `then`, что отклоняет промис *третьего* `then`, что в конце вызывает обработчик отклонения. Ни один из обратных вызовов `then` не вызывается, потому что они были предназначены для выполнения, а не для отклонения (рисунок 8-5).

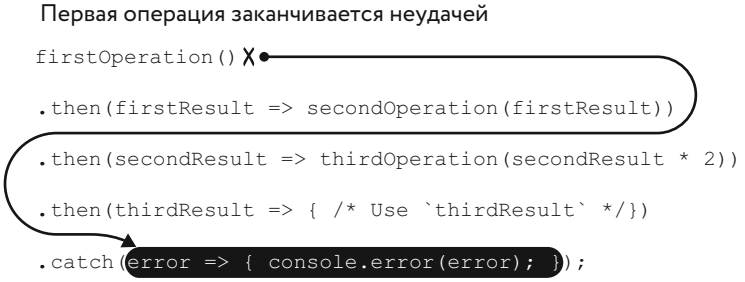


РИСУНОК 8-5

Если первая операция завершена успешно, и ее обработчик выполнения начинает вторую операцию и возвращает свой промис, но вторая операция терпит неудачу и отклоняет свой промис, то это отклоняет промис из первого `then`, что отклоняет промис из второго `then`, что отклоняет промис из третьего `then`, что вызывает обработчик отклонения (рисунок 8-6). (Аналогично, если первая операция завершается успешно, но обработчик выполнения вызывает инструкцию `throw`, происходит то же самое: остальные обработчики выполнения пропускаются, и выполняется обработчик отклонения.)

Первая операция завершается успешно, но вторая терпит неудачу

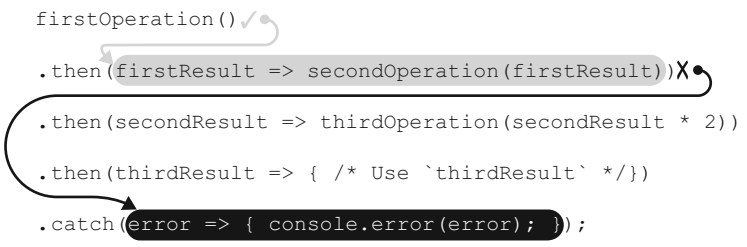


РИСУНОК 8-6

Естественно, то же самое верно, если первая и вторая операции выполняют свои промисы, но третья отклоняет свой промис (или запускающий его обработчик вызывает `throw`) (рисунок 8-7).

Первая и вторая операции завершаются успешно, но третья завершается неудачей

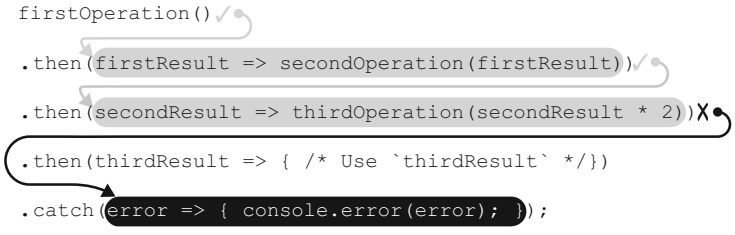


РИСУНОК 8-7

Первая, вторая и третья операции завершаются успешно, но конечный обработчик запускает `throw`

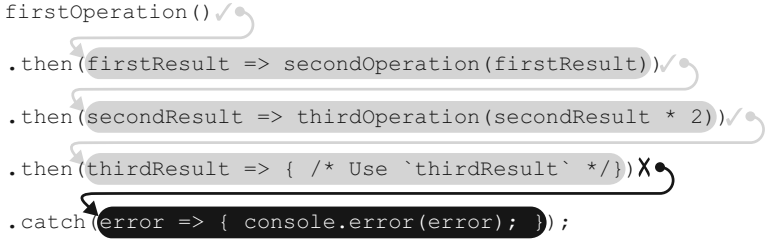


РИСУНОК 8-8

Если все три операции завершаются успешно, запускается обработчик итогового выполнения. Если этот обработчик вызывает `throw`, либо напрямую, либо путем вызова функции, которая реализует инструкцию `throw`, запускается обработчик отклонения (рисунок 8-8).

Это показывает, что цепочка промисов такого рода очень похожа на блок `try/catch` вокруг трех синхронных операций (и некоторого итогового кода), например:

```
// Та же логика с синхронными операциями
try {
    const firstResult = firstOperation();
    const secondResult = secondOperation(firstResult);
    const thirdResult = thirdOperation(secondResult * 2);
// Используйте здесь `thirdResult`
} catch (error) {
    console.error(error);
}
```

Точно так же, как разделение основной логики от логики ошибок полезно в синхронном коде `try/catch`, полезно разделение основной логики (выполнения) от логики ошибок (отклонения) в цепочках промисов.

Вы можете увидеть, как разыгрываются эти различные сценарии, многократно запустив файл **basic-promise-chain.js** из перечня загрузок. Этот файл использует метод `Math.random`, чтобы дать каждой операции (первой, второй и третьей) 20%-ную вероятность сбоя и дать обработчику итогового выполнения 20%-ную вероятность реализации инструкции `throw`.

ОБРАБОТЧИКИ — ЭТО ПРОСТО ФУНКЦИИ

Функции-обработчики — это просто функции, в них нет ничего особенного. В примере в этом разделе я использовал стрелочную функцию там, где мне на самом деле не нужно было:

```
firstOperation()
.then(firstResult => secondOperation(firstResult))
.then(secondResult => thirdOperation(secondResult * 2))
.then(thirdResult => { /* Use `thirdResult` */ })
.catch(error => { console.error(error); });
```

Хотя можно было бы написать этот код следующим образом:

```
firstOperation()
.then(secondOperation)
.then(secondResult => thirdOperation(secondResult * 2))
.then(thirdResult => { /* Use `thirdResult` */ })
.catch(error => { console.error(error); });
```

Нет необходимости оборачивать функцию `secondOperation` в стрелочную функцию, если вы просто передаете в нее значение

из первой операции. Вместо этого просто передайте ссылку на функцию в метод `then`.

Это безопасно даже в случае применения функции, принимающей необязательные вторые, третьи и т. д. аргументы, потому что функции-обработчики промисов всегда получают ровно один аргумент — значение (`then`) или причину отклонения (`catch`). Это означает, что у вас нет проблем, как были бы, скажем, если функция массива `map` и метод `parseInt`, где `a = a.map(parseInt)`, не работают правильно, потому что `map` вызывает свою функцию обратного вызова с тремя аргументами, а не с одним, а `parseInt` принимает необязательный второй аргумент.

Сравнение с обратными вызовами

Давайте сравним цепочку промисов из предыдущего раздела с простым использованием обратных вызовов. Для этого сравнения предположим, что функции, запускающие асинхронные операции, принимают обратный вызов: если операция завершается успешно, они вызывают обратный вызов с `null` в качестве первого аргумента и результирующим значением в качестве второго. Если операция завершается неудачей, они вызывают обратный вызов с ошибкой в качестве первого аргумента. (Это один из наиболее распространенных шаблонов для обратных вызовов, популяризированный Node.js среди прочих.) Вот как выглядела бы предыдущая последовательность, используя это соглашение:

```
firstOperation((error, firstResult) => {
    if (error) {
        console.error(error);
        // (Возможно, вы бы использовали здесь `return;`, чтобы избежать `else`)
    } else {
        secondOperation(firstResult, (error, secondResult) => {
            if (error) {
                console.error(error);
                // Возможно: `return;`
            } else {
                thirdOperation(secondResult * 2, (error, thirdResult) => {
                    if (error) {
                        console.error(error);
                        // Возможно: `return;`
                    } else {
                        try {
                            /* Используйте `thirdResult` */
                        } catch (error) {
                            console.error(error);
                        }
                    }
                });
            }
        });
    }
});
```

Вы можете запустить этот код с помощью `basic-callback-chain.js` из перечня загрузок.

Обратите внимание, что каждый шаг должен быть вложен в обратный вызов предыдущего шага. Это знаменитый «ад обратного вызова», и одна из причин, по которой некоторые программисты используют только двухсимвольный отступ (несмотря на то, как трудно это для чтения), чтобы их глубоко вложенные обратные вызовы не исчезали с правой стороны экрана. *Можно* написать этот код, не вкладывая обратные вызовы друг в друга, но это неудобно и многословно.

На самом деле вокруг вызова функции `second Operation` должен быть блок `try/catch` и еще один блок вокруг вызова функции `thirdOperation`. Но мне не хотелось усложнять пример, подчеркивающий идею о том, что промисы делают такой код проще.

Метод `catch`

Вы уже знаете, что метод `catch` регистрирует обработчик, вызываемый при отклонении промиса. За исключением того, что обработчик, который вы ему даете, вызывается при отклонении, а не при выполнении — `catch` *точно такой же*, как `then`. В таком случае:

```
p2 = p1.catch(error => doSomethingWith(error))
```

метод `catch` регистрирует обработчик отклонения для `p1`, создавая и возвращая новый промис (`p2`), исполняемый или отклоняемый в зависимости от того, что происходит с изначальным промисом (`p1`), и что выполняется в обработчике. Если `p1` выполняется, обработчик не вызывается, а `p2` выполняется со значением выполнения `p1`. Если `p1` отклоняется, вызывается обработчик. Вот что происходит с `p2` в зависимости от того, что делает ваш обработчик:

- Если вы возвращаете элемент thenable, `p2` будет разрешен этим элементом.
- Если вы возвращаете любое другое значение, `p2` выполняется с этим значением.
- Если вы используете метод `throw`, `p2` отклонятся, используя результат throw в качестве причины отклонения.

Звучит знакомо? Верно! Это именно то, что делает метод `then` со своим обработчиком. В свою очередь, именно это делает функция `resolve`, которую вы получаете в функции-исполнителе промисов. Согласованность полезна: вам не нужно запоминать разные правила для разных механизмов разрешения.

Одна вещь, на которую следует обратить особое внимание, заключается в том, что, если вы возвращаете не-thenable из обработчика `catch`, промис из `catch` выполняется. Это преобразует отклонение от `p1` в выполнение `p2`. Если хорошо подумать, это работает, как `catch` в блоке `try/catch`: если вы повторно выбрасываете ошибки в блоке `catch`, то блок `catch` пресекает распространение ошибки. Метод `catch` для промисов работает таким же образом.

У этого есть небольшой недостаток: вы можете непреднамеренно замаскировать ошибки, поместив обработчик `catch` в неправильное место. Рассмотрим:

```
someOperation()
.catch(error => {
```

```
        reportError(error);
})
.then(result => {
        console.log(result.someProperty);
});
```

В этом коде, если операция, запущенная с помощью функции `SomeOperation`, завершается неудачей, функция `reportError` вызывается с ошибкой (пока все хорошо). Но вы получите сообщение об ошибке на экране, оно выглядит примерно так:

```
Uncaught (in promise) TypeError: Cannot read property 'someProperty' of
undefined
```

Почему код пытается считать свойство `someProperty` из значения `undefined`? Промис отклонен, так почему же вызывается обработчик `then`?

Вернемся к нашей аналогии с `try`/`catch`/`finally`. Код выше — логический (не буквальный) эквивалент этого:

```
let result;
try {
    result = someOperation();
} catch (error) {
    reportError(error);
    result = undefined;
}
console.log(result.someProperty);
```

Оператор `catch` поймал ошибку и вывел ее, но ничего не сделал для распространения ошибки. Поэтому код после блока `try`/`catch` все еще работает и пытается использовать выражение `result.someProperty`. Оно терпит неудачу, потому переменная `result` получила значание `undefined`.

Версия с промисами делает то же самое, потому что обработчик отклонения поймал ошибку, обработал ее, а затем ничего не вернул. Это очень похоже на возврат значения `undefined`. Это выполняет промис из `catch` со значением `undefined`. Следовательно, промис вызывает свой обработчик выполнения, переданный в `undefined` в качестве значения `result`, и, когда обработчик попытался использовать выражение `result.someProperty`, это вызвало новую ошибку.

Возможно, вы задаетесь вопросом: «Но что это за “необработанное отклонение”?». Если промис отклоняется и нет кода для обработки отклонения, движки JavaScript сообщают о «необработанном отклонении» (unhandled rejection), поскольку обычно это ошибка. В этом примере код не обрабатывает ошибки из обработчика итогового выполнения (со значением `result.someProperty` в нем), поэтому, когда ошибка при попытке использования `result.someProperty` отклоняет промис из `then`, это «необработанное отклонение»: нет ничего, чтобы обработать эту ситуацию.

Иногда, конечно, может потребоваться поместить `catch` в середину специально для того, чтобы сделать именно это — обработать ошибку и позволить цепочке продолжаться с некоторым значением замены или аналогичным значением. Это абсолютно нормально, если вы делаете это целенаправленно.

ОПРЕДЕЛЕНИЕ НЕОБРАБОТАННЫХ ОТКЛОНЕНИЙ

Процесс определения того, обрабатывается ли отклонение, довольно сложен: он должен учитывать тот факт, что обработчик отклонения может быть добавлен *после* того, как промис был отклонен. Например, предположим, что вы вызываете функцию API, возвращающую промис, и он может быть отклонен синхронно (во время вызова функции API), потому что вы дали ей неверные значения аргументов. Или промис может быть отклонен асинхронно позже, потому что запускаемый им асинхронный процесс завершается сбоем. Ваш вызов может выглядеть следующим образом:

```
apiFunction(/*...arguments...*/)
.then(result => {
    // ...использование result...
})
.catch(error => {
    // ...обработка/сообщение об ошибке...
});
```

Если отклонение вызвано синхронно неправильным значением аргумента, промис *уже отклонен* до того, как код получит возможность прикрепить к нему обработчик отклонения. Но вам бы не хотелось, чтобы движок JavaScript жаловался на то, что это необработанное отклонение, потому что ваш код действительно обрабатывает его. Отклонение вызовет код в вашем обработчике отклонения, который обрабатывает/сообщает об ошибке. Так что это не так просто, как «Пришло время отказаться от этого промиса. Есть ли у него какие-либо прикрепленные обработчики отклонений?». Все гораздо сложнее.

Современные движки делают это довольно сложным способом, но все равно могут содержать странные ложноположительные или ложноотрицательные результаты.

Метод `finally`

Метод `finally`, как вы уже знаете, очень похож на блок `finally` из `try/catch/finally`. Он добавляет обработчик, вызываемый независимо от того, выполняется промис или отклоняется (как и блок `finally`). Как и `then` и `catch`, он возвращает новый промис, выполняемый или отклоняемый на основании того, что происходит с изначальным промисом и что делает обработчик `finally`. Но, *в отличие* от `then` и `catch`, его обработчик вызывается всегда. Также у его обработчика есть ограничения на операции с получаемыми элементами.

Вот простой пример:

```
function doStuff() {
    spinner.start();
    return getSomething()
```

```
        .then(result => render(result.stuff))
        .finally(() => spinner.stop());
}
```

Эта функция запускает вращение волчка, чтобы показать пользователю, что она что-то делает, для запуска асинхронной операции использует `getSomething` — функцию, возвращающую промис. Затем показывает результат, если промис выполнен, и останавливает волчок независимо от того, что происходит с промисом.

Обратите внимание, что эта функция не пытается обрабатывать ошибки; вместо этого она возвращает последний промис в цепочке. Это вполне нормально. Такой вариант позволяет вызывающему коду узнать, удалась ли операция, и позволяет обрабатывать ошибки на самом высоком уровне (часто это обработчик событий или что-то подобное, например, точка входа в код JavaScript). Используя `finally`, этот код может отложить обработку ошибок для вызывающего абонента, будучи уверенным, что он остановит волчок даже при возникновении ошибки.

В обычном случае обработчик `finally` не влияет на выполнение или отклонение, проходящее через него (опять же, как блок `finally`): любое возвращаемое им значение, отличное от элемента thenable, которое отклоняется/отклоняется, игнорируется. Но если выдается ошибка или возвращается отклоненный thenable эта ошибка/отклонение заменяет любое выполнение или отклонение, проходящее через него (точно так же, как `throw` в блоке `finally`). Таким образом, он не может изменить значение выполнения, даже если возвращает другое значение, но метод может изменить причину отклонения на другую, может изменить выполнение на отклонение и может *отложить* выполнение (возвращая значение, которое в конечном счете выполняется, вы очень скоро увидите пример).

Или, другими словами: это похоже на блок `finally`, который не может содержать оператор `return`[51]. В нем может содержаться оператор `throw` или может быть вызвана функция, реализующая выбрасывание ошибки, но она не может вернуть новое значение.

Запустите код в Листинге 8-2, чтобы посмотреть пример.

Листинг 8-2: Оператор `finally` возвращает значение — finally-returning-value.js

```
// Функция, возвращающая промис, который выполняется после заданной
// задержки и с заданным значением
function returnWithDelay(value, delay = 10) {
    return new Promise(resolve => setTimeout(resolve, delay, value));
}

// Функция выполняет задачу
function doSomething() {
    return returnWithDelay("original value")
        .finally(() => {
            return "value from finally";
        });
}
```

[51] Наличие `return` в блоке `finally` в любом случае обычно представляет собой плохую практику.

```
doSomething()
    .then(value => {
        console.log("value = " + value); // "value = original value"
    });
```

В этом примере функция `doSomething` вызывает функцию `returnWithDelay` и содержит обработчик `finally` (в рабочей программе это будет некая разновидность очистки). Функция `doSomething` возвращает созданный обработчиком `finally` промис. Код вызывает функцию `doSomething` и использует возвращаемый ей промис (тот, что возвращает `finally`). Обратите внимание, что обработчик `finally` возвращает значение, но промис из `finally` *не использует* его в качестве значения выполнения. Вместо этого он использует значение, полученное из промиса, к которому был вызван. Возвращаемое из обработчика `finally` значение не было элементом thenable, поэтому оно было полностью проигнорировано.

Причина, по которой обработчик `finally` не может изменять выполнение, двоякая:

- Основной вариант использования `finally` — это очистка, когда вы закончили выполнение какого-то фрагмента кода, но без влияния на результат этого фрагмента.
- Механизм промиса не может сообщить разницу между функцией, которая совсем не использует `return` (выполнение кода просто «убирает окончание» функции), функцией, которая использует `return` без операнда, и функцией, которая использует `return undefined`, потому что результатом вызова всех трех этих функций станет одно и то же — значение `undefined`. Например:

```
function a() {
}
function b() {
    return;
}
function c() {
    return undefined;
}
console.log(a()); // undefined
console.log(b()); // undefined
console.log(c()); // undefined
```

В отличие от этого, в структуре `try`/`catch`/`finally` движок JavaScript *может* сообщить о разнице между блоком `finally`, где выполнение кода убирает окончание (не влияет на возвращаемое значение функции, в которой оно находится) и тем, который выдает оператор `return`. Однако, поскольку механизм промисов не может определить эту разницу и поскольку основная цель обработчика `finally` — выполнение очистки, ничего не меняя, имеет смысл игнорировать возвращаемое значение обработчика `finally`.

Однако тот факт, что обработчик не может повлиять на значение выполнения цепочки, не означает, что он не может вернуть промис или элемент thenable. И если это произойдет, цепочка ожидает, пока этот промис/thenable не будет выполнен, точно так же, как когда обработчики `then` и `catch` возвращают значения, поскольку промис/thenable может быть отклонен. Пример приведен в Листинге 8-3.

Листинг 8-3: Оператор finally возвращает выполняемый промис — finally-returning-promise.js

```
// Функция, возвращающая промис, который выполняется после заданной
// задержки и с заданным значением
function returnWithDelay(value, delay = 100) {
    return new Promise(resolve => setTimeout(resolve, delay, value));
}

// Функция выполняет задачу
function doSomething() {
    return returnWithDelay("original value")
        .finally(() => {
            return returnWithDelay("unused value from finally", 1000);
        })
}

console.time("example");
doSomething()
    .then(value => {
        console.log("value = " + value); // "value = original value"
        console.timeEnd("example");      // example: 1100ms (или аналогичное
                                         // значение)
    });
```

Обратите внимание, что хотя значение не было изменено тем фактом, что обработчик `finally` вернул выполненный позже промис, цепочка дождалась выполнения этого промиса, прежде чем продолжить. Для завершения потребовалось примерно 1110 мс вместо всего лишь 100 мс — все из-за задержки в 1000 мс от функции `returnWithDelay` внутри обработчика `finally`.

Однако если возвращаемый обработчиком `finally` элемент thenable отклоняется, это заменяет выполнение (Листинг 8-4).

Листинг 8-4: Оператор finally вызывает отклонение — finally-causing-rejection.js

```
// Функция, возвращающая промис, который выполняется после заданной
// задержки и с заданным значением
function returnWithDelay(value, delay = 100) {
    return new Promise(resolve => setTimeout(resolve, delay, value));
}

// Функция, возвращающая промис, который был отклонен после заданной
// задержки и с заданным значением
function rejectWithDelay(error, delay = 100) {
    return new Promise((resolve, reject) => setTimeout(reject, delay, error));
}

console.time("example");
returnWithDelay("original value")
    .finally(() => {
        return rejectWithDelay(new Error("error from finally"), 1000);
    })
    .then(value => {
        // Не вызывается
```

```
        console.log("value = " + value);
    })
    .catch(error => {
        console.error("error = ", error);    // "error =  Error: error from
                                             // finally"
    })
    .finally(() => {
        console.timeEnd("example");          // example: 1100ms (или
                                             // аналогичное значение)
    });
```

Метод `throw` в обработчиках `then`, `catch` и `finally`

Пару раз в этой главе вы слышали, что, если выполнить метод `throw` в обработчике `then`, `catch` или `finally`, это вызовет отклонение созданного в `then`/`catch`/`finally` промиса со значением из throw. Давайте рассмотрим такой пример из реального веб-программирования.

В современных браузерах старый объект `XMLHttpRequest` заменяется новой функцией `fetch`, возвращающей промис. Она, как правило, проще в использовании. По сути, вы выполняете функцию `fetch(url)`, и она возвращает промис, выполняемый или отклоняемый в зависимости от успеха/неудачи сетевой операции. В случае успеха значение выполнения представляет собой объект `Response` с различными свойствами, а также методами (которые также возвращают промисы) для чтения тела ответа как текст, как объект `ArrayBuffer` (см. главу 11), как проанализированный по стандартам JSON текст, но затем разбора его как JSON, как объект blob и т. д.

Таким образом, наивное использование `fetch`, скажем, для извлечения некоторых данных в формате JSON, зачастую выглядит следующим образом:

```
// Неверно
fetch("/some/url")
    .then(response => response.json())
    .then(data => {
        // Делает что-нибудь с данными...
    })
    .catch(error => {
        // Делает что-нибудь с данными...
    });
```

Однако здесь упущен важный шаг. Так как нужно было проверить результат свойства `status` при использовании `XMLHttpRequest`, теперь нужно проверить состояние ответа (прямо или косвенно) с помощью `fetch` так же, ведь промис из `fetch` отклоняется только в случае возникновения ошибки сети. Все остальное — ошибка сервера 404, ошибка сервера 500 и т. д. — приравнивается к выполнению с этим кодом состояния, а не становится отклонением, потому что сетевая операция прошла успешно, даже несмотря на сбой HTTP-операции.

Однако в 99,99% случаев при использовании `fetch` вам неважно, почему вы не получили то, что просили (была ли это сетевая ошибка или ошибка HTTP) — вам важно, что вы этого не получили. Самый простой способ справиться с этим — выбросить ошибку из первого обработчика `then`:

```
fetch("/some/url")
    .then(response => {
        if (!response.ok) {
            throw new Error("HTTP error " + response.status);
        }
        return response.json();
    });
    .then(data => {
        // Делает что-нибудь с данными...
    })
    .catch(error => {
        // Делает что-нибудь с данными...
    });
```

Этот обработчик `then` преобразует выполнение с ошибкой HTTP в отклонение выполнения промиса, проверяя удобное свойство `response.ok` (его значение равно `true`, если код статуса ответа HTTP-код успешный, в противном случае значение равно `false`) и использует метод `throw`, если значение равно `false`, чтобы выполнить отклонение. Поскольку эта проверка нужна почти всегда, вы, вероятно, хотели бы получить выполняющую ее служебную функцию. И поскольку обработчику ошибок может потребоваться доступ к ответу, вы могли бы создать пользовательский подкласс `Error`, чтобы обеспечить такой доступ. Поскольку сообщение `message` в `Error` представляет собой строку, у вас могло бы получиться что-то похожее на Листинг 8-5.

Листинг 8-5: Оболочка fetch, преобразующая ошибки HTTP в отклонения — fetch-convertinghttp-errors.js

```
class FetchError extends Error {
    constructor(response, message = "HTTP error " + response.status) {
        super(message);
        this.response = response;
    }
}
const myFetch = (...args) => {
    return fetch(...args).then(response => {
        if (!response.ok) {
            throw new FetchError(response);
        }
        return response;
    });
};
```

С учетом этого вы могли бы написать предыдущий пример очевидным способом:

```
myFetch("/some/url")
    .then(response => response.json())
    .then(data => {
        // Делает что-нибудь с данными...
    })
    .catch(error => {
        // Делает что-нибудь с данными...
    });
```

Обработчик `then` в `myFetch` превращает выполнение-с-ошибкой-HTTP в отклонение, используя метод `throw` в своем обработчике.

Метод then с двумя аргументами

Все примеры `then`, которые вы видели до сих пор, передают ему только один обработчик. Но `then` может принимать *два* обработчика: один для выполнения, а другой для отклонения.

```
doSomething()
    .then(
        /*f1*/ value => {
            // Выполнить что-то с `value`
        },
        /*f2*/ error => {
            // Выполнить что-то с `error`
        }
    );
```

В этом примере, если промис от функции `doSomething` выполняется, вызывается первый обработчик; если он отклонен, вызывается второй обработчик.

Использование `p.then(f1, f2)` — *не* то же самое, что и `p.then(f1).catch(f2)`. Существует большая разница: версия `then` с двумя аргументами придает обоих обработчиков к вызываемому вами промису, но использование `catch` после `then` придает вместо этого обработчик отклонения возвращаемому `then` промису. Это означает, что при использовании `p.then(f1, f2)`, `f2` вызывается только если `p` отклонен, но не в том случае, если `f1` выбрасывает ошибку и возвращает отклоняемый промис. При использовании `p.then(f1).catch(f2)`, `f2` вызывается, если `p` отклонен *или* если `f1` выбрасывает ошибку и возвращает отклоняемый промис.

ОБА АРГУМЕНТА THEN НЕОБЯЗАТЕЛЬНЫЕ

Если вы хотите подключить обработчик отклонения и не подключать обработчик выполнения, можно сделать это, передав значение undefined в качестве первого аргумента: `.then(undefined, rejectionHandler)`.

«Подождите, — скажите вы, — разве не для этого нужен оператор `catch`?» Да, так и есть. Оператор `catch` — это буквально просто оболочка вокруг `then`, эффективно определенная, как в этом примере с использованием синтаксиса метода:

```
catch(onRejected) {
    return this.then(undefined, onRejected);
}
```

Оператор `finally` тоже представляет собой просто оболочку вокруг then, но у него есть логика в обработчиках, которые он передает `then`, вместо того чтобы просто проходить через получаемый обработчик, как это делает `catch`.

Тот факт, что операторы `catch` и `finally` строятся чисто с точки зрения вызова then (буквально, а не только концептуально), появится позже в этой главе.

Обычно требуется версия `then` только с одним аргументом и привязать любой обработчик отклонения к промису, возвращаемому через `catch`. Но если вы хотите обрабатывать ошибки только из исходного промиса, а не из обработчика выполнения, вам следует использовать версию с двумя аргументами. Например, предположим, что у вас есть `myFetch` из предыдущего раздела, и вы хотите получить некоторые данные JSON с сервера, указав значение по умолчанию, если сервер отвечает ошибкой. Но вы не хотите обрабатывать ошибку (если она есть) при чтении и парсинге JSON (вызов `response.json()`). Использование выражения `then` с двумя аргументами имело бы смысл в таком случае:

```
myFetch("/get-message")
    .then(
        response => response.json(),
        error => ({message: "default data"})
    )
    .then(data => {
        doSomethingWith(data.message);
    })
    .catch(error => {
        // Обработка общей ошибки
    });
```

Реализовать это с `then` в версии с одним аргументом не совсем удобно. Вы либо заканчиваете тем, что ваш обработчик вызывается из-за ошибок при вызове `json` при выполнении.`then(...).catch(...)`, либо вы должны обернуть свое значение по умолчанию в объект с помощью функции `json`, чтобы получить возможность первым разместить `catch`:

```
// Неудобная версия, избегающая использования `then` с двумя аргументами
// Не лучший способ
myFetch("/get-message")
    .catch(() => {
        return {json() {return {message: "default data"};};
    })
    .then(response => response.json())
    .then(data => {
        doSomethingWith(data.message);
    })
    .catch(error => {
        // Обработка общей ошибки
    });
```

На практике обычно требуется структура.`then(f1).catch(f2)`. Но для тех относительно редких случаев, когда необходимо обрабатывать ошибки от обработчика `then` *иначе*, чем отклонение исходного промиса (как в этом примере), существует версия `then` с двумя аргументами.

ДОБАВЛЕНИЕ ОБРАБОТЧИКОВ К УЖЕ ВЫПОЛНЕННЫМ ПРОМИСАМ

В разделе «Почему же промисы?» в начале этой главы были две проблемы с применением просто обратных вызовов:

- Добавление обратного вызова к уже завершенному процессу также не стандартизировано. В некоторых случаях это означает, что обратный вызов никогда не вызывается. В других он вызывается синхронно, а иногда асинхронно. Проверка описания для каждого используемого API при каждом возникновении такой ситуации отнимает много времени и легко забывается. Ошибки, которые вы получаете, когда предполагаете или неправильно запоминаете, часто незаметны и их трудно обнаружить.
- Добавление нескольких обратных вызовов к операции либо невозможно, либо не стандартизировано.

Промисы решают эти проблемы, предоставляя стандартную семантику для добавления обработчиков завершения (включая множественные) и гарантируя две вещи:

- Обработчик будет вызван (при условии, что он предназначен для соответствующего вида: выполняется, успешно выполнен или отклонен).
- Вызов будет асинхронным.

Это означает, что если у вас есть код, который откуда-то получает промис и выполняет это:

```
console.log("before");
thePromise.then(() => {
    console.log("within");
});
console.log("after");
```

спецификация гарантирует, что код выведет `"before"`, затем `"after"`, а уже потом (если промис выполнится или уже выполнился) `"within"`. Если промис будет выполнен позже, вызов обработчика выполнения запланирован позже. Если промис уже выполнен, вызов обработчика запланирован (но не выполнен) во время вызова `then`. Вызов обработчика выполнения или отклонения запланирован путем добавления задания для него в очередь заданий промиса. Дополнительные сведения об очередях заданий см. во врезке «Задания скриптов и промисов». Запустите файл **then-on-fulfilled-promise.js** из загрузок, чтобы увидеть этот процесс в действии.

Обеспечение исключительно асинхронного вызова обработчиков даже для уже выполняемых промисов — это единственный способ, которым сами промисы делают асинхронным то, что таковым не было.

ЗАДАНИЯ СКРИПТОВ И ПРОМИСОВ

Каждый поток JavaScript в движке JavaScript запускает цикл, обслуживающий очереди заданий. Задание — это единица работы, которую поток будет выполнять от начала до конца, не выполняя ничего другого. Существуют две стандартные очереди задач: задания скриптов и задания промисов. (Или, как их называет спецификация HTML, задачи и микрозадачи.) Оценка скрипта, оценка модуля (глава 13), обратные вызовы событий DOM и обратные вызовы таймера — это примеры заданий/задач скрипта (также называемых *макро*задачами). Реакции на промисы (вызов обработчиков выполнения или отклонения промисов) — это задания/*микро*задачи промисов: при планировании вызова обработчика выполнения или отклонения движок помещает задание для этого вызова в очередь заданий промиса. Все задания в очереди заданий промиса выполняются в конце задания скрипта, перед запуском следующего задания (даже если это задание скрипта было добавлено в очередь заданий скрипта задолго до того, как задание промиса было добавлено в очередь заданий промиса). Это включает в себя задания промисов, добавленные заданием промиса. Например, если первое задание промиса в очереди планирует другую реакцию промиса, эта реакция происходит после задания промиса, которое запланировало ее, но до запуска следующего задания скрипта. Концептуально это выглядит так:

```
// Только концептуально, не буквально!
while (running) {
    if (scriptJobs.isEmpty()) {
        sleepUntilScriptJobIsAdded();
    }
    assert(promiseJobs.isEmpty());
    const scriptJob = scriptJobs.pop();
    scriptJob.run(); // Может добавлять задания в promiseJobs
                     // и/или scriptJobs
    while (!promiseJobs.isEmpty()) {
        const promiseJob = promiseJobs.pop();
        promiseJob.run(); // Может добавлять задания
                          // в promiseJobs и/или scriptJobs
    }
}
```

По сути, реакции на промисы получили более высокий приоритет, чем другие задания.

(В хост-средах могут быть и другие виды заданий. Например, в Node.js есть функция таймер `setImmediate`, которая планирует работу отличным от задания скрипта или задания промиса образом.)

СОЗДАНИЕ ПРОМИСОВ

Существует несколько способов создания промисов. Основной способ создания промиса, как вы уже узнали, — при помощи применения `then`, `catch` или `finally`

к существующему промису. Чтобы создать промис, когда у вас еще нет ни одного из них, можно использовать конструктор `Promise` или одну из нескольких служебных функций.

Конструктор Promise

Конструктор `Promise` создает промис. Существует распространенное заблуждение, что `new Promise` превращает синхронную операцию в асинхронную. Это не так. Промисы не делают операции асинхронными — они просто обеспечивают последовательное средство сообщения о результате чего-то, что *уже* является асинхронным.

Редко возникает необходимость в использовании `new Promise`. Это может показаться удивительным, но в большинстве случаев промис получают каким-то другим способом:

- При вызове чего-то (функции API и т. д.), что возвращает промис.
- Промис появляется из операторов `then`, `catch` или `finally` существующего промиса.
- Промис можно получить из служебной функции, которая знает, как преобразовать определенный стиль API обратного вызова из этого стиля обратного вызова в промис (что на самом деле является лишь конкретным примером первого пункта в этом списке).
- Промис можно получить из таких статических методов `Promise`, как `Promise.resolve` или `Promise.reject`, описанных далее в этом разделе.

Если у вас уже есть промис или thenable, вам не нужно применять `new Promise`. Либо используйте `then` для привязки к нему, либо, если это элемент thenable и вам нужен промис, используйте выражение `Promise.resolve` (о нем вы узнаете в следующем разделе).

Но иногда действительно требуется создать промис с нуля. Например, глядя на третий пункт в предыдущем списке, возможно, вы тот, кто пишет служебную функцию, преобразующую API обратного вызова в API промиса. Давайте рассмотрим этот вариант.

Предположим, у вас есть код, основанный на промисах, и вам нужно работать с не предоставляющей промис функцией API. Она принимает содержащий параметры объект, и два из этих параметров — обратные вызовы для успешного и неудачного выполнения:

```
const noop = () => {}; // Функция, которая ничего не делает
/* Что-то делает.
 * @param   data        Данные для обработки
 * @param   time        Время для принятия
 * @param   onSuccess   Обратный вызов успешного выполнения, вызванный
 *                      с двумя аргументами: результатом и кодом состояния
 * @param   onError     Обратный вызов для неудачного выполнения, вызванный
 *                      с одним аргументом (сообщением об ошибке в виде
 *                      строки).
 */
function doSomething({data, time = 10, onSuccess = noop, onError = noop} = {}) {
    // ...
}
```

Поскольку ваш код основан на промисах, было бы полезно получить основанную на промисах версию функции `doSomething`. Вы могли бы написать для нее оболочку, которая выглядит как функция `promiseSomething` в Листинге 8-6. (Надеюсь, вы дадите ей название получше, чем это.)

Листинг 8-6: Оболочка для мока функции API — mock-api-function-wrapper.js

```
function promiseSomething(options = {}) {
    return new Promise((resolve, reject) => {
        doSomething({
            ...options,
            // Поскольку `doSomething` вызывает `onSuccess` с двумя
            // аргументами, нам нужно обернуть их в объект
            // чтобы передать их `resolve`
            onSuccess(result, status) {
                resolve({result, status});
    },
            // `doSomething` вызывает `onError` со строковой ошибкой,
            // оборачивает ее в экземпляр Error
            onError(message) {
                reject(new Error(message));
            }
        });
    });
}
```

Возможно, вы помните из начала главы, что функция, которую вы передаете в конструктор `Promise`, называется *функцией-исполнителем*. Она отвечает за запуск процесса, для которого `Promise` сообщит об успехе или неудаче. Конструктор `Promise` вызывает функцию-исполнителя синхронно. Это означает, что код в функции-исполнителе выполняется до того, как функция `promiseSomething` вернет промис.

Вы видели функцию-исполнитель в начале этой главы и в Листинге 8-6. Напомним, что функция-исполнитель получает два аргумента: функцию, вызываемую для разрешения промиса (обычно ей присваивается имя `resolve`, но вы можете называть ее как угодно), и функцию, вызываемую для отклонения промиса (обычно ей присваивается имя `reject`). Ничто не может выполнить промис, если у него нет доступа к одной из этих двух функций. Каждая из них принимает один параметр: `resolve` принимает элемент thenable (в этом случае оно преобразует промис в этот элемент thenable) или значение, не являющееся thenable (в этом случае промис выполняется); `reject` принимает причину отклонения (ошибку). Если вызвать любую из них без аргументов, для значения или причины отклонения используется значение `undefined`. Если вызвать любую из них с более чем одним аргументом, используется только первый аргумент, все остальные игнорируются. Наконец, после первого вызова `resolve` или `reject` повторный вызов любой из них ничего не даст. Это не вызывает ошибки, но вызов полностью игнорируется. Так происходит потому, что как только промис будет выполнен (выполнен или преобразован в thenable) или отклонен, его невозможно изменить. Решение или отказ высечены в камне.

Давайте посмотрим, что делает функция-исполнитель в Листинге 8-6:

- Она создает новый объект options и использует свойство расширения (о котором вы узнали в главе 5), чтобы придать ему свойства переданного объекта options, добавив собственные функции `onSuccess` и `onError`.
- Для создания `onSuccess` код использует функцию, принимающую два переданных ей аргумента, и создает объект со свойствами `result` и `status`, затем вызывает `resolve` с этим объектом, чтобы выполнить промис и установить объект в качестве значения исполнения.
- Для создания `onError` код использует функцию, принимающую строку сообщения об ошибке от функции `doSomething`, оборачивает функцию в экземпляр `Error`, и вызывает `reject` с этой ошибкой.

Теперь вы можете использовать функцию `promiseSomething` в своем коде, основанном на промисах:

```
promiseSomething({data: "example"})
    .then(({result, status}) => {
        console.log("Got:", result, status);
    })
    .catch(error => {
        console.error(error);
    });
```

Для получения полностью рабочего примера запустите файл **full-mock-api-function-example.js** из загрузок.

Обратите внимание, что в отличие от `resolve`, выполнение метода `reject` не отличается, если вы передаете ей объект thenable. Он *всегда* использует полученное значение в качестве причины отклонения. Это всегда приводит к отклонению промиса; это не сводит промис к тому, что вы передаете, как это делает `resolve`. Даже если вы передаете ему элемент thenable, он не ждет, пока этот thenable выполнится; он использует сам элемент thenable в качестве причины отклонения. Например, этот код:

```
new Promise((resolve, reject) => {
    const willReject = new Promise((resolve2, reject2) => {
        setTimeout(() => {
            reject2(new Error("rejected"));
        }, 100);
    });
    reject(willReject);
})
.catch(error => {
    console.error(error);
});
```

передает промис `willReject` обработчику `catch` *немедленно*, не дожидаясь его выполнения. (Затем, спустя 100 мс, это вызывает необработанную ошибку отклонения, потому что ничто не обрабатывает отклонение `willReject`.)

В этом заключается разница между `resolve` и `reject`. В то время как `resolve` разрешает промис, включая возможность его разрешения с результатом в виде другого промиса или элемента thenable (что может включать в себя ожидание разрешения этого другого промиса/ thenable), `reject` будет всегда отклонять промис немедленно.

Метод `Promise.resolve`

Метод `Promise.resolve(x)` — это эффективное сокращение для:

```
x instanceof Promise ? x: new Promise(resolve => resolve(x))
```

За исключением использования в `Promise.resolve(x)`, `x` вычисляется только один раз. Метод возвращает свой аргумент напрямую, если этот аргумент — экземпляр `Promise` (а не просто элемент thenable). В противном случае он создает новый промис и преобразует его в переданное методу значение. Это означает, что если значение представляет собой thenable, промис преобразуется в этот thenable; если значение не относится к экземплярам thenable, то промис выполняется с помощью этого значения.

Метод `Promise.resolve` довольно удобен для преобразования thenable в собственный промис, поскольку возвращаемый промис выполняется в зависимости от того, как выполняется thenable. Это также полезно, если вы принимаете что-то, назовем это `x`; это может быть промис, thenable или простое значение: вы передаете его через `Promise.resolve`:

```
x = Promise.resolve(x);
```

и в итоге это всегда промис. Например, предположим, вам нужна служебная функция, регистрирующая значение, либо само значение, либо, если это значение thenable, его значение выполнения или причину отклонения. Вы бы запустили параметр через метод `Promise.resolve`:

```
function pLog(x) {
    Promise.resolve(x)
    .then(value => {
        console.log(value);
    })
    .catch(error => {
        console.error(error);
    });
}
```

Затем вы можете вызвать его либо с помощью простого значения:

```
pLog(42);
// => 42 (после "тика", потому что она гарантированно будет асинхронной)
```

или с промисом или другим thenable:

```
pLog(new Promise(resolve => setTimeout(resolve, 1000, 42)));
// => 42 (через ~1000ms)
```

Поскольку промис/thenable может быть отклонен, промис из метода `Promise.resolve` также может быть отклонен:

```
pLog(new Promise((resolve, reject) => {
    setTimeout(reject, 1000, new Error("failed"));
));
// => error: failed (after ~1000ms)
```

Другой вариант использования — запуск серии асинхронных операций с поддержкой промисов, где каждая цепочка зависит от выполнения предыдущей. Вы узнаете больше об этом в разделе «Серии промисов» далее в этой главе.

Метод `Promise.reject`

Метод `Promise.reject(x)` — это сокращение для:

```
new Promise((resolve, reject) => reject(x))
```

То есть это создает отклоненный промис с указанием полученной от вас причины отклонения.

Метод `Promise.reject` не такой универсальный, как `Promise.resolve`, но один из распространенных вариантов его использования — в обработчике `then`, когда требуется преобразовать выполнение в отклонение. Некоторые программисты предпочитают возвращать отклоненный промис, например, так:

```
.then(value => {
    if (value == null) {
        return Promise.reject(new Error());
    }
    return value;
}
```

вместо использования `throw`, так:

```
.then(value => {
    if (value == null) {
        throw new Error();
    }
    return value;
}
```

Они оба в конечном счете делают одно и то же, но из соображений личного стиля встречаются варианты, когда для этого используется `Promise.reject` — особенно когда это означает, что вместо подробной формы стрелочной функции можно использовать краткую функцию.

Например, приведенный ранее пример можно было бы записать с помощью краткой стрелочной функции:

```
.then(value => value == null ? Promise.reject(new Error()): value);
```

Выражение `throw` нельзя использовать в краткой форме стрелочной функции (пока), потому что `throw` — это оператор, а тело краткой стрелочной функции — это однострочное выражение. (Однако выражения `throw`[52] являются активным предложением, так что это может измениться.)

[52] `https://github.com/tc39/proposal-throw-expressions`

ДРУГИЕ СЛУЖЕБНЫЕ МЕТОДЫ ПРОМИСОВ

Помимо методов `Promise.resolve` и `Promise.reject` доступны и другие служебные функции. В этом разделе мы рассмотрим еще несколько.

Метод `Promise.all`

Предположим, у вас есть три асинхронные операции, которые вам нужно выполнить. Они не зависят друг от друга, поэтому будет хорошо, если они наложатся друг на друга. Использовать результаты требуется только тогда, когда будут получены результаты от всех трех операций. Если использовать цепочку промисов, они будут выполняться последовательно. Так как же их дождаться? Можно запустить каждый из них, запомнить его промис, а затем последовательно вызывать оператор `then` для них, но это становится довольно некрасиво:

```
let p1 = firstOperation();
let p2 = secondOperation();
let p3 = thirdOperation();
p1.then(firstResult => {
    return p2.then(secondResult => {
        return p3.then(thirdResult => {
            // Здесь используются `firstResult`, `secondResult` и `thirdResult`
        });
    });
})
.catch(error) {
    // Обработка ошибки
});
```

Метод `Promise.all` спешит на помощь! Он принимает итерируемый элемент (например массив) и ожидает, пока все элементы thenable в нем будут выполнены, возвращая промис, который, либо А) выполняется, когда выполняются *все* thenable в итерируемом элементе, либо Б) отклоняется, как только *любой* из них отклоняется. Когда выражение выполняется, значение выполнения представляет собой массив, содержащий значения выполнения thenable вместе с любыми не-thenable значениями из исходного итеративного элемента, в порядке как у вызываемого итерируемого элемента. При получении отклонения метод использует причину отклонения того элемента thenable, который был отклонен (запомните: это *первый* отклоненный thenable).

Вы можете дождаться этих трех операций, которые могут выполняться параллельно с `Promise.all` — например, так:

```
Promise.all([firstOperation(), secondOperation(), thirdOperation()])
    .then(([firstResult, secondResult, thirdResult]) => {
        // Здесь используются `firstResult`, `secondResult` и `thirdResult`
    })
    .catch(error => {
        // Обработка ошибки
    });
```

Ничего страшного, если функция `secondOperation` завершится раньше функции `firstOperation`. Метод `Promise.all` гарантирует, что значения выполнения находятся в том же порядке, что и получаемые им значения thenable: результат первого

thenable всегда хранится первым в массиве, результат второго всегда хранится вторым и т. д. Аналогично любые не-thenable значения в исходном итерируемом элементе, находятся в том же месте в результирующем массиве (рисунок 8-9).

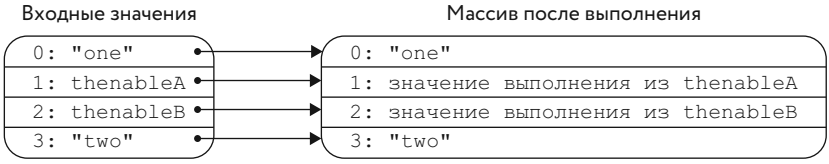


РИСУНОК 8-9

Обратите внимание на использование деструктуризации в обработчике `then` в предыдущем примере. Если вы работаете с известным количеством передаваемых в `Promise.all` записей в итерируемом элементе, в отличие от созданного с неизвестным количеством записей элемента, использование деструктуризации в обработчике `then` может повысить ясность кода. Но это необязательно.

Если вызвать метод `Promise.all` с пустым итерируемым, его промис выполнится немедленно с пустым массивом.

Если *любой* из элементов thenable отклоняется, промис из метода `Promise.all` отклоняется по этой же причине, не дожидаясь, пока другие thenable выполнятся. Это не дает вам доступа ни к каким выполнениям, которые могли произойти до отклонения, ни к каким-либо выполнениям или отклонениям, произошедшим после. Однако отклонения после возникновения отклонения бесшумно «обрабатываются», поэтому не вызывают необработанных ошибок отклонения.

Помните, что любые значения, не относящиеся к thenable, просто передаются. Это потому, что метод `Promise.all` передает все значения из итерируемого через метод `Promise.resolve` при получении, а затем использует результирующие промисы. Например, если у вас было всего две операции для выполнения и третье значение, которое вы хотели передать вместе с ними, это можно было сделать следующим образом:

```
Promise.all([firstOperation(), secondOperation(), 42])
    .then(([a, b, c]) => {
        // Здесь используются `a`, `b` и `c`
        // `c` получит значение 42 в этом примере
    })
    .catch(error => {
        // Обработка ошибки
    });
```

Значение не обязательно должно быть в конце, оно может быть где угодно:

```
Promise.all([firstOperation(), 42, secondOperation()])
    .then(([a, b, c]) => {
        // Здесь используются `a`, `b` и `c`
        // `c` получит значение 42 в этом примере
    })
    .catch(error => {
        // Обработка ошибки
    });
```

Метод `Promise.race`

Метод `Promise.race` принимает итерируемый элемент значений (обычно thenable) и наблюдает за их гонкой, предоставляя промис для результата гонки. Промис будет выполнен, как только будет выполнено первое условие, или отклонен, как только будет отклонено первое условие, используя значение выполнения или причину отклонения из этого «выигравшего» промиса. Как и метод `Promise.all`, он передает значения из итератора через метод `Promise.resolve`, поэтому не-thenable элементы также работают.

Функции-таймеры — это один из вариантов использования метода `Promise.race`. Предположим, вам нужно извлечь что-то с отсрочкой по времени, но используемый вами механизм выборки не предоставляет функции таймера. Вы можете запустить гонку между выборкой и таймером:

```
// (Предположим, что TimeoutAfter возвращает промис, отклоняемый с помощью
// ошибки таймера после заданного количества миллисекунд)
Promise.race([fetchSomething(), timeoutAfter(1000)])
    .then(data => {
        // Делает что-нибудь с данными
    })
    .catch(error => {
        // Обрабатывается ошибка или таймер
    });
```

Когда таймер выигрывает гонку, это не останавливает операцию выборки, но игнорирует более позднее завершение (в любом случае) операции выборки. Конечно, если используемый вами механизм выборки поддерживает таймер или способ его отмены, стоит использовать его вместо этого.

Метод `Promise.allSettled`

Метод `Promise.allSettled` передает все значения из переданного ему итерируемого через `Promise.resolve` и ожидает, пока все они будут выполнены, независимо от того, будут ли они выполнены успешно или отклонены, и возвращает массив объектов со свойством `status` и свойством либо `value`, либо `reason`.

Если свойство `status` получило значение `"fulfilled"` (выполнено успешно), значит, промис выполнен успешно, и свойство `value` объекта получило значение успешного выполнения. Если же свойство `status` получило значение `"rejected"` (отклонено), промис был отклонен, и свойство `reason` получило причину отклонения. (При проверке свойства `status` убедитесь, что вы правильно пишете `"fulfilled"`, легко ошибочно написать вторую «l» в «ful».)

Как и в случае с методом `Promise.all`, массив будет находиться в том же порядке, что и предоставленный итерируемый элемент, даже если элементы thenable будут расположены не по порядку.

Взгляните на пример:

```
Promise.allSettled([Promise.resolve("v"), Promise.reject(new Error("e"))])
.then(results => {
    console.log(results);
});
```

```
// Выводит что-то похожее на:
// [
//     {status: "fulfilled", value: "v"},
//     {status: "rejected", reason: Error("e")}
//]
```

Метод `Promise.any`

Метод `Promise.any`[53] на этапе 3 на момент написания был противоположностью методу `Promise.all`. Как и `Promise.all`, он принимает итерируемый элемент значений, которые передает через метод `Promise.resolve`, и возвращает промис. Но если `Promise.all` определяет успех (успешное выполнение) как «*все* последующие thenable выполнены успешно», метод `Promise.any` определяет успех как «*любой* из последующих thenable выполнен успешно». Он успешно выполняет свой промис с помощью значения успешного выполнения первого выпоенного thenable. Если все элементы thenable отклоняются, он отклоняет свой промис с помощью `Aggregate Error`, свойство `errors` которого представляет собой массив причин отклонения. Как и массив значений выполнения `Promise.all`, массив причин отклонения всегда находится в том же порядке, что и значения из полученного итерируемого.

ШАБЛОНЫ ПРОМИСОВ

В этом разделе вы узнаете о некоторых шаблонах, полезных при работе с промисами.

Обрабатывать ошибки или возвращать промис

Одно из фундаментальных правил промисов — либо обработать ошибки, либо распространить цепочку промисов (например, путем возвращения последнего результата `then`, `catch` или `finally`) на своего вызывающего абонента. Нарушение этого правила — самый большой источник некорректности программы при использовании промисов.

Предположим, у вас есть функция, извлекающая и показывающая обновленную оценку при вызове:

```
function showUpdatedScore(id) {
    myFetch("getscore?id=" + id).then(displayScore);
}
```

Что делать, если метод `myFetch` столкнется с проблемой? Ничто не справляется с отклонением этого промиса. В браузере об этом необработанном отклонении появится сообщение в консоли. В Node.js об этом появится сообщение в консоли, и это может привести к полному завершению программы[54]. Более того, что бы ни вызывало функцию `showUpdatedScore`, оно не знает, что она не сработала. Вместо этого `showUpdatedScore` должна возвращать промис от `then`:

---

[53] `https://github.com/tc39/proposal-promise-any`

[54] Во всяком случае, таков план на момент написания этой книги. На данный момент Node.js v13 по-прежнему просто показывает предупреждение при обнаружении необработанного отклонения. В предупреждении говорится, что в какой-то будущей версии необработанные отклонения завершат процесс.

```
function showUpdatedScore(id) {
    return myFetch("getscore?id=" + id).then(displayScore);
}
```

Затем, что бы ни вызывало функцию, оно может обрабатывать ошибки или передавать по цепочке по мере необходимости (возможно, в сочетании с другими асинхронными операциями, которые оно выполняет). Например:

```
function showUpdates(scoreId, nameId) {
    return Promise.all([showUpdatedScore(scoreId), showUpdatedName(nameId)]);
}
```

Каждый уровень должен либо обрабатывать отклонения, либо возвращать промис своему вызывающему коду, ожидая, что вызывающий обработает отклонения. Верхний уровень должен выполнять обработку отклонения, поскольку у него нет ничего, чему он мог бы передать цепочку:

```
button.addEventListener("click", () => {
    const {scoreId, nameId} = this.dataset;
    showUpdatedScore(scoreId, nameId).catch(reportError);
}
```

За исключением синхронных исключений, распространение встроено в механизм. С промисами вам необходимо убедиться, что распространение происходит путем возврата последнего промиса в цепочке. Но не отчаивайтесь! В главе 9 вы увидите, как это снова становится автоматическим в функции `async`.

Серии промисов

Если существует ряд операций, которые должны выполняться одна за другой, и вы хотите либо собрать все их результаты, либо просто передать каждый результат в следующую операцию, есть удобный способ сделать это — построить цепочку промисов с помощью цикла. Начать надо с массива (или любого итерируемого) функций, которые вы хотите вызвать (или, если вызывается одна и та же функция в каждом случае — массив/ итерируемый элемент аргументов, с которыми ее требуется вызвать), и перебираете циклом массив, чтобы создать цепочку промисов, связывая каждую операцию в цепочке с предыдущим успешным выполнением через `then`. Начните с вызова функции `Promise.resolve`, передающей начальное значение.

Многие делают это, используя функцию `reduce` для массивов, поэтому давайте быстро повторим концепцию `reduce`. Она принимает обратный вызов и начальное значение (как и мы его здесь используем) и вызывает обратный вызов с начальным значением и первой записью в массиве, затем снова вызывает обратный вызов с возвращаемым значением из первого вызова и следующей записью в массиве, повторяет этот процесс, пока не завершит массив, возвращая конечный результат обратного вызова. Итак, этот вызов `reduce`:

```
console.log(["a", "b", "c"].reduce((acc, entry) => acc + " " + entry,
            "String:"));
console.log(value); // "String: a b c"
```

вызывает обратный вызов со значением "String:" (начальное значение) и "a" (первая запись массива), затем вызывает его снова со значением "String: a" (результат предыдущего вызова) и "b" (следующая запись массива), затем вызывает его снова со значением "String: a b" (результат предыдущего вызова) и "c" (последняя запись массива), затем возвращает конечный результат: "String: a b c". То же самое выполняет этот код:

```
const callback = (acc, entry) => acc + " " + entry;
callback(callback(callback("String:", "a"), "b"), "c")
console.log(value); // "String: a b c"
```

но более гибко, потому что он может обрабатывать любое количество записей в массиве.

Давайте применим это к промисам. Предположим, у вас есть функция, передающая значение через ряд настраиваемых функций преобразования (например, промежуточное программное обеспечение в процессе веб-сервера или ряд фильтров данных изображений):

```
function handleTransforms(value, transforms) {
    // ...
}
```

Поскольку список преобразований варьируется от вызова к вызову, он предоставляется в виде массива функций преобразования. Каждая функция принимает значение для преобразования и выдает преобразованное значение либо синхронно, либо асинхронно. Если асинхронно, код возвращает промис. Создание цепочки промисов в цикле хорошо подходит для этой задачи: вы используете начальное значение в качестве начального значения для аккумулятора `reduce` и перебираете функции, связывая промисы вместе. Что-то вроде этого:

```
function handleTransforms(value, transforms) {
    return transforms.reduce(
        (p, transform) => p.then(v => transform(v)), // Обратный вызов
        Promise.resolve(value)                       // Начальное значение
    );
}
```

Однако код, использующий `reduce`, может быть немного сложным для чтения. Давайте рассмотрим версию, которая делает то же самое, используя цикл `for-of`:

```
function handleTransforms(value, transforms) {
    let chain = Promise.resolve(value);
    for (const transform of transforms) {
        chain = chain.then(v => transform(v));
    }
    return chain;
}
```

(Стрелочные функции в двух предыдущих примерах можно было бы удалить, а вместо `.then(v => transform(v))` просто написать `.then(transform)`.)

Давайте предположим, что мы вызываем функцию `handleTransforms` с функциями преобразования `a`, `b` и `c`. Используете ли вы код с `reduce` или код с циклом `for-of`, функция `handleTransforms` задает цепочку, а затем возвращается до того, как обратные вызовы запускаются (потому что `then` для промиса из `Promise.resolve` гарантированно запустит свой обратный вызов асинхронно). Затем происходит следующее:

- Почти сразу же запускается первый обратный вызов. Он передает начальное значение в функцию `a`. Она возвращает либо значение, либо промис. Это разрешает промис от первого вызова до `then`.
- Успешное выполнение вызывает обратный вызов ко второму `then`, передавая значение, которое функция `a` вернула в функцию `b`, а затем возвращает результат `b`. Это разрешает промис от второго вызова до `then`.
- Успешное выполнение вызывает обратный вызов к третьему `then`, передавая значение, которое функция `b` вернула в функцию `c`, а затем возвращает результат `c`. Это разрешает промис от третьего вызова до `then`.

Вызывающий код видит итоговый промис, который либо выполнен с окончательным значением от `c` или отклонен с причиной отклонения, представляемой функцией `a`, `b` или `c`.

В главе 9 вы узнаете о функциях `async`, добавленных в ES2018. С функциями `async` цикл становится еще понятнее (как только вы узнаете, что такое `await`). Вот краткое ознакомление:

```
// функции `async` рассматриваются в Главе 9.
async function handleTransforms(value, transforms) {
    let result = value;
    for (const transform of transforms) {
        result = await transform(result);
    }
    return result;
}
```

Параллельные промисы

Чтобы выполнить группу операций параллельно, запускайте их по очереди, создавая массив (или другой итеративный элемент) промисов, возвращаемых каждым из них, и используйте метод `Promise.all`, чтобы дождаться их завершения. (См. раздел выше о методе `Promise.all`.)

Например, если у вас был массив URL-адресов, все их необходимо было получить, и было бы хорошо получить их все параллельно, вы могли бы использовать `map` для запуска операций выборки и получения промисов для них, а затем метод `Promise.all` для ожидания результатов:

```
Promise.all(arrayOfURLs.map(
    url => myFetch(url).then(response => response.json())
))
.then(results => {
    // Использование массива результатов
    console.log(results);
})
.catch(error => {
```

```
    // Обработка ошибки
    console.error(error);
});
```

Учитывая примечание в окончании раздела «Серии промисов», вы, вероятно, задаетесь вопросом, делают ли функции `async` это решение намного проще. Ответ таков: отчасти, но вы все равно будете использовать метод `Promise.all`. Вот краткое ознакомление:

```
// этот код должен быть функцией `async` (Глава 9)
try {
    const results = await Promise.all(arrayOfURLs.map(
        url => myFetch(url).then(response => response.json())
    ));
    // Использование массива результатов
    console.log(results);
} catch (error) {
    // Обработка/отчет об ошибке
    console.error(error);
}
```

АНТИШАБЛОНЫ ПРОМИСОВ

В этом разделе вы узнаете о некоторых связанных с промисами распространенных антишаблонах, которых следует избегать.

Излишнее выражение new Promise(/*...*/)

Программисты-новички в промисах часто пишут такой код:

```
// Неверно
function getData(id) {
    return new Promise((resolve, reject) => {
        myFetch("/url/for/data?id=" + id)
            .then(response => response.json())
            .then(data => resolve(data))
            .catch(error => reject(error));
    });
}
```

или различные перестановки такого варианта. Как вы знаете, `then` и `catch` *уже* возвращают промисы, так что нет никакой необходимости в использовании выражения `new Promise`. Предыдущий код стоит просто написать так:

```
function getData(id) {
    return myFetch("/url/for/data?id=" + id)
        .then(response => response.json());
}
```

Отсутствие обработки ошибок (или неправильная обработка)

Отсутствие обработки ошибок или неправильная их обработка — частое явление при использовании промисов. Но, в то время как исключения автоматически распространяются

по стеку вызовов до тех пор, пока/если не будут обработаны блоком `catch`, отклонения промисов этого не делают, что приводит к появлению скрытых ошибок. (Или они *были* скрытыми ошибками, пока среды JavaScript не начали сообщать о необработанных отклонениях.)

Помните правило из предыдущего раздела о шаблонах промисов: основное правило промисов заключается в обработке ошибки, либо передаче цепочки промисов вызывающему коду, чтобы он мог их обработать.

Оставление ошибок незамеченными при преобразовании API обратного вызова

При обертывании API обратного вызова в оболочку промиса легко случайно допустить, чтобы ошибки остались необработанными. Например, рассмотрим эту гипотетическую оболочку для API, возвращающего строки из базы данных:

```
// Неверно
function getAllRows(query) {
    return new Promise((resolve, reject) => {
        query.execute((err, resultSet) => {
            if (err) {
                reject(err); // или `reject(new Error(err))` или что-то
                             // аналогичное
            } else {
                const results = [];
                while (resultSet.next()) {
                    data.push(resultSet.getRow());
                }
                resolve(results);
            }
        });
    });
}
```

На первый взгляд это выглядит разумно. Код выполняет запрос, и, если запрос возвращает ошибку, он использует ее для отклонения. Если нет, он создает массив результатов и выполняет с ними свой промис.

Но что, если метод `ResultSet.next()` выдает ошибку? Возможно, основное соединение с базой данных оборвалось. Это приведет к завершению обратного вызова, что означает, что *ни* `resolve`, *ни* `reject` не будут вызваны. Это, скорее всего, приведет к бесшумному сбою и навсегда оставит промис невыполненным.

Необходимо убедиться, что ошибки в обратном вызове перехвачены и превращены в отклонение:

```
function getAllRows(query) {
    return new Promise((resolve, reject) => {
        query.execute((err, resultSet) => {
            try {
                if (err) {
                    throw err; // или `throw new Error(err)` или что-то
                               // аналогичное
                }
```

```
                const results = [];
                while (resultSet.next()) {
                    data.push(resultSet.getRow());
                }
                resolve(results);
            } catch (error) {
                reject(error);
            }
        });
    });
}
```

Вам не нужно оборачивать вызов `query.execute` в блок `try/catch`, ведь если выбрасывается ошибка функцией-исполнителем промиса (а не обратным вызовом позже), конструктор `Promise` использует эту ошибку, чтобы автоматически отклонять промис.

Неявное преобразование отклонения в успешное выполнение

Программисты-новички в промисах, которые слышали основное правило (обрабатывать ошибки или возвращать цепочку промисов), иногда понимают его как «… *и* возвращать цепочку промисов». В итоге они пишут код, подобный этому:

```
function getData(id) {
    return myFetch("/url/for/data?id=" + id)
        .then(response => response.json())
        .catch(error => {
            reportError(error);
        });
}
```

Видите ли вы проблему в этом коде? (Это немного тонко.)

Проблема заключается в том, что при возникновении ошибки промис из функции `getData` будет *успешно выполнен* со значением `undefined`, и никогда не будет отклонен. У обработчика `catch` нет возвращаемого значения, следовательно его вызов даст результат `undefined`, а это означает, что промис, созданный в обработчике `catch`, успешно выполняется со значением `undefined` и не отклоняется. Что делает функцию `getData` довольно неудобной для использования: любой ее обработчик `then` должен проверять, где она получила значение `undefined` или запрашиваемые ей данные.

Правило гласит: «… *или* возвращает цепочку промисов», и это то, что должна делать функция `getData`:

```
function getData(id) {
    return myFetch("/url/for/data?id=" + id)
        .then(response => response.json());
}
```

Это позволяет вызывающему код использовать `then` и `catch` с результатом, зная, что обработчик `then` получие данные и обработчик `catch` получит отклонение.

Попытка использовать результаты вне цепочки

Программисты-новички в промисах часто пишут подобный код:

```
let result;
startSomething()
.then(response => {
    result = response.result;
});
doSomethingWith(result);
```

Проблема здесь заключается в том, что `result` примет значение `undefined` при запуске функции `doSomethingWith(result);`, потому что обратный вызов `then` является асинхронным. Вместо этого вызов `doSomethingWith` должен находиться внутри обратного вызова `then` (и, конечно, код должен либо обрабатывать отклонение, либо возвращать цепочку промисов к чему-то, что будет их обрабатывать):

```
startSomething()
.then(response => {
    doSomethingWith(response.result);
})
.catch(reportError);
```

Использование обработчиков бездействия

При первом использовании промисов некоторые программисты пишут обработчики бездействия, например:

```
// Неверно
startSomething()
.then(value => value)
.then(response => {
    doSomethingWith(response.data);
})
.catch(error => {throw error;});
```

Два из трех обработчиков бессмысленны (и есть еще одна проблема с показанным кодом). Какие из них?

Верно! Обработчик `then` не требуется, если вы не изменяете значение или не используете его в других операциях. Выброшенная из обработчика `catch` ошибка просто отклоняет созданный в `catch` промис. Таким образом, первый вызов `then` и вызов `catch` ничего не делают. Код может быть именно таким (с ним все еще есть проблема):

```
startSomething()
.then(response => {
    doSomethingWith(response.data);
});
```

Заметили ли вы оставшуюся проблему? Ее немного сложно увидеть без дополнительного контекста, но проблема заключается в том, что код должен обрабатывать ошибки или возвращать цепочку к чему-то, что будет их обрабатывать.

Неправильное разветвление цепочки

Еще одна распространенная проблема при начале работы с промисами — неправильное разветвление цепочки, например:

```
// Неверно
const p = startSomething();
p.then(response => {
    doSomethingWith(response.result);
});
p.catch(handleError);
```

Проблема здесь заключается в том, что ошибки из обработчика `then` не обработаны, ведь `catch` вызывается для исходного промиса, а не для промиса из `then`. Если функция `handleError` должна вызываться как для ошибок из исходного промиса, так и для ошибок из обработчика `then`, это должно быть записано просто в виде цепочки:

```
startSomething()
.then(response => {
    doSomethingWith(response.result);
})
.catch(handleError);
```

Если причина, по которой автор написал это по-другому, заключается в том, что обработка ошибок из обработчика выполнения отличается от обработки ошибок из исходного промиса, есть несколько способов реализовать это. При помощи версии `then` с двумя аргументами:

```
startSomething()
.then(
    response => {
        doSomethingWith(response.result);
},
    handleErrorFromOriginalPromise
)
.catch(handleErrorFromThenHandler);
```

Такой код предполагает, что функция `handleErrorFromOriginalPromise` никогда не выдает и не возвращает отклоняемый промис.

Или можно добавить обработчик отклонения к исходной структуре, возможно, немного изменив порядок:

```
const p = startSomething();
p.catch(handleErrorFromOriginalPromise);
p.then(response => {
    doSomethingWith(response.result);
})
.catch(handleErrorFromThenHandler);
```

ПОДКЛАССЫ ПРОМИСОВ

Создать собственный подкласс для `Promise` можно обычным путем. Например, используя синтаксис `class`, изученный в главе 4:

```
class MyPromise extends Promise {
    // ...здесь пользовательская функциональность...
}
```

При этом важно не нарушать различные фундаментальные гарантии, которые дают промисы. Например, не переопределяйте метод `then` и не заставляйте его вызывать обработчик синхронно, когда промис уже выполнен. Это нарушит гарантию того, что обработчик всегда вызывается асинхронно. Если бы вы это сделали, ваш подкласс был бы допустимым *элементом thenable*, но это не было бы промисом, что нарушает правило подклассов «является».

> **ПРИМЕЧАНИЕ**
> Вряд ли вам понадобится создавать подкласс промиса — не в последнюю очередь потому, что так легко случайно столкнуться с собственным промисом вместо вашего подкласса. Например, если вы имеете дело с предоставляющим промисы API, для использования вашего подкласса промиса потребуется обернуть промис из этого API в свой подкласс (обычно с помощью `YourPromiseSubclass.resolve(theNativePromise)`). Тем не менее при условии правильного определения подкласса Promise вы можете быть уверены, что получаемые от его собственных методов промисы будут экземплярами вашего подкласса.

Поскольку `then`, `catch`, `finally`, `Promise.resolve`, `Promise.reject`, `Promise.all`, и т. д. возвращают промисы, вы можете быть обеспокоены тем, что при создании подкласса потребуется переопределить все методы `Promise`. Хорошие новости! Реализации методов `Promise` разумны: они гарантируют, что создают новый промис, используя подкласс. На самом деле пустые оболочки `MyPromise`, показанные в начале этого раздела, полностью функциональны. И вызов `then` (или `catch`, или `finally`) для экземпляров `MyPromise` (или использование `MyPromise.resolve`, `MyPromise.reject`, и т. д.) возвращает новый экземпляр `MyPromise` с правильным поведением. Вам не нужно ничего реализовывать явно:

```
class MyPromise extends Promise {
}
const p1 = MyPromise.resolve(42);
const p2 = p1.then(() => {/*...*/});
console.log(p1 instanceof MyPromise); // истина
console.log(p2 instanceof MyPromise); // истина
```

Если вы решите выделить подкласс `Promise`, вот некоторые вещи, которые следует иметь в виду:

- Не должно быть никакой необходимости определять свой собственный конструктор, достаточно конструктора по умолчанию. Но если вы определяете конструктор:
  - Убедитесь, что вы передаете первый аргумент (функцию-исполнитель) в `super()`.
  - Убедитесь, что вы не ожидаете получения каких-либо других аргументов (потому что вы не получите их от реализаций методов `Promise`, использующих конструктор, таких как `then`).
- Если вы переопределяете методы `then`, `catch` или `finally`, убедитесь, что не нарушаете основных обеспечиваемых ими гарантий (например, не выполнять обработчик синхронно).
- Помните, что `then` — центральный метод экземпляров промисов. Операторы `catch` и `finally` вызывают `then`, чтобы выполнить свою работу (в буквальном смысле, а не только концептуально). Если вам нужно подключиться к процессу подключения обработчиков к промису, вам нужно только переопределить `then`.
- Если вы переопределяете `then`, помните, что у него есть два параметра (обычно они называются `onFulfilled` и `onRejected`), и оба необязательные.
- Если вы создаете новые методы, создающие промисы, не вызывайте свой собственный конструктор напрямую (например, `new MyPromise()`), это недружелюбно по отношению к подклассам. Вместо этого используйте шаблон `Symbol.species`, показанный в главе 4 (используйте выражение `new this.constructor[Symbol.species](/*...*/)` в методе экземпляра или `new this[Symbol.species](/*...*/)` в статическом методе) или непосредственно используйте выражение `new this.constructor(/*...*/)` в методе экземпляра и `new this(/*...*/)` в статическом методе. Исходный класс `Promise` выполняет последнее, он не использует шаблон `species`.
- Аналогично, если вы хотите использовать метод `MyPromise.resolve` или `MyPromise.reject` в коде метода `MyPromise`, не используйте их непосредственно, используйте выражение `this.constructor.resolve/reject` в методе экземпляра или `this.resolve/reject` в статическом методе.

Опять же, маловероятно, что вам действительно понадобится подкласс `Promise`.

ОТ СТАРЫХ ПРИВЫЧЕК К НОВЫМ

На самом деле здесь есть только одна «старая привычка», которую стоит изменить.

Используйте промисы вместо успешных/неудачных обратных вызовов

Старая привычка: Создание функции, запускающей одноразовый асинхронный процесс, которая принимает обратный вызов (или два, или три), чтобы сообщить об успехе, сбое и завершении.

Новая привычка: Вместо этого верните промис явно или неявно с помощью функции `async` (о которой вы узнаете в главе 9).

9

Асинхронные функции, итераторы и генераторы

СОДЕРЖАНИЕ ГЛАВЫ

- Асинхронные функции `async`
- Оператор `await`
- Итераторы и генераторы `async`
- Выражение `for-await-of`

В этой главе вы узнаете о функции `async` и операторе `await` из ES2018. Они предоставляют синтаксис для написания асинхронного кода, используя тот же знакомый поток структуры управления, который вы используете при написании синхронного кода (циклы `for`, операторы `if`, `try/catch/finally`, вызовы функций и ожидание их результатов и т. д.). Вы также узнаете об итераторах `async`, генераторах `async` и выражении `for-await-of`.

Прежде чем приходить к этой главе, вам стоит изучить главу 8, если вы этого еще не сделали. Это важно, поскольку функции `async` основаны на промисах.

АСИНХРОННЫЕ ФУНКЦИИ

В некотором смысле синтаксис `async/await` — это синтаксический сахар для создания и использования промисов. И все же он полностью преобразует то, как вы пишете свой асинхронный код, позволяя писать логический поток, а не создавать только синхронный поток и использовать обратные вызовы для асинхронных фрагментов. Синтаксис `async/await` коренным образом изменяет и упрощает написание асинхронного кода.

В не-`async` функции пишется серия операций, которые движок JavaScript будет выполнять по порядку, не позволяя при этом происходить чему-либо еще (см. врезку «Один поток на одну базу realm»). Этот код может передавать обратный вызов чему-то, что позже вызовет его асинхронно — но код, выполняющий такую операцию, просто передает функцию для последующего вызова, а сам вызов выполняется не сразу.

Например, рассмотрим этот код (предположим, что все эти функции синхронны):

```
function example() {
    let result = getSomething();
    if (result.flag) {
        doSomethingWith(result);
} else {
        reportAnError();
}
}
```

В функции `example` код вызывает функцию `getSomething`, проверяет возвращенный флаг объекта, а затем либо вызывает функцию `doSomethingWith`, передаваемую в объект, либо вызывает функцию `reportAnError`. Все это происходит одно за другим, и в этот период времени больше ничего не происходит[55].

ОДИН ПОТОК НА ОДНУ БАЗУ REALM

Ранее, когда я сказал, что движок выполняет шаги не-async функции «...не позволяя при этом происходить чему-либо еще...», существует неявный квалификатор «...в этом потоке...». JavaScript определяет семантику только для одного потока на базу данных realm (например, вкладку браузера), иногда разделяя этот один поток на несколько баз realm (например, несколько вкладок в браузере из одного источника, которые могут напрямую вызывать код друг друга). Это означает, что только один поток получает прямой доступ к переменным и остальными используемым кодом элементам. (В главе 16 объясняется, как объект `SharedArrayBuffer` позволяет обмениваться данными между потоками, но не переменными.) Хотя существуют некоторые среды JavaScript, допускающие несколько потоков в одной базе realm (виртуальная машина Java VM — один из примеров, она запускает JavaScript с помощью поддержки скриптов), они очень редки, и для них в JavaScript нет стандартной семантики. Для этой главы примите стандартный тип среды. Его можно найти в Node.js или браузере с одним потоком в базе данных realm.

До появления функций `async` (асинхронных) выполнение аналогичных действий с асинхронными операциями включало передачу обратных вызовов, возможно, обратных вызовов промисов. Например, использование `fetch` (замена `XMLHttpRequest` в современных браузерах) в не-`async` функции может привести к получению такого кода:

[55] Кроме этого, любой поток может быть подвешен (приостановлен) средой. Не может произойти только выполнение *другой* работы в той же базе realm JavaScript после паузы.

```
function getTheData(spinner) {
    spinner.start();
    return fetch("/some/resource")
        .then(response => {
            if (!response.ok) {
                throw new Error("HTTP status " + response.status);
            }
            return response.json();
        })
        .then(data => {
            useData();
            return data;
        })
        .finally(() => {
            spinner.stop();
        });
}
```

Этот код сначала осуществит вызов метода `spinner.start`, затем запрос `fetch`, затем вызов оператора `then` для возвращаемого запросом промиса, затем вызов `then` для *этого* возвращаемого промиса, затем вызов `finally` для *этого* возвращаемого промиса и возврат промиса из оператора `finally`. Все это происходит в непрерывной последовательности, и между операциями ничего нет. Позже, когда запрос завершается, эти обратные вызовы выполняются, но это происходит после возврата значения функции `getTheData`. После возврата `getTheData` и до завершения этих операций поток может выполнять другие действия.

Вот тот же код в функции `async`:

```
async function getTheData(spinner) {
    spinner.start();
    try {
        let response = await fetch("/some/resource");
        if (!response.ok) {
            throw new Error("HTTP status " + response.status);
        }
        let data = await response.json();
        useData(data);
        return data;
    } finally {
        spinner.stop();
    }
}
```

Код выглядит синхронным, не так ли? Но это не так. В нем есть места, где происходит пауза и ожидается завершение асинхронного процесса. Во время ожидания поток может выполнять другие действия. Эти места отмечены ключевым словом `await` (ожидание).

Четыре ключевые особенности функций `async`:

- Функции `async` неявно создают и возвращают промисы.
- В функции `async` оператор `await` потребляет промисы, отмечая точку, в которой код будет асинхронно ожидать выполнения промиса. Пока функция ожидает выполнения промиса, поток может запускать другой код.

- В функции `async`, код, традиционно рассматриваемый как синхронный (цикл `for`, `a + b`, `try`/`catch`/`finally` и т. д.), является асинхронным, если он содержит оператор `await`. Логика та же, но время выполнения отличается: могут быть паузы, чтобы ожидаемый промис был выполнен.
- Исключения — это отклонения, и отклонения — это исключения; инструкции `return` — это разрешения, а успешные выполнения — это результаты (если вы применяете оператор `await` для промиса, вы увидите значение успешного выполнения промиса в качестве результата `await`).

Давайте рассмотрим каждый из них более подробно.

Создание промисов асинхронными функциями

Функция `async` создает и возвращает промис скрытно, разрешая или отклоняя этот промис на основе кода внутри функции. Ниже приведено *схематичное* отображение того, как функция `async` `getTheData`, показанная ранее, концептуально обрабатывается движком JavaScript. Это не совсем верно в буквальном смысле, но достаточно близко по смыслу, чтобы получить представление о том, как движок ее обрабатывает.

```
// НЕ так, как вы бы написали это сами
function getTheData(spinner) {
    return new Promise((resolve, reject) => {
        spinner.start();
        // Внутренней функцией-исполнителем этого промиса является блок `try`
        new Promise(tryResolve => {
            tryResolve(
                Promise.resolve(fetch("/some/resource"))
                .then(response => {
                    if (!response.ok) {
                        throw new Error("HTTP status " + response.status);
                    }
                    return Promise.resolve(response.json())
                        .then(data => {
                            useData(data);
                            return data;
                        });
                })
            );
        })
        .finally(() => {
            spinner.stop();
        })
        .then(resolve, reject);
    });
}
```

Вы бы написали ее по-другому, напрямую с помощью промисов — не в последнюю очередь потому, что в главе 8 рассказывалось, что вам не нужно выражение `new Promise`, когда уже есть промис из `fetch`. Но это разумное указание на то, как «под капотом» работает функция `async`.

Заметьте, что часть функции `async` до первого оператора `await` (вызовы методов `spinner.start` и `fetch`) работает *синхронно*: в схематичной версии этот код

перемещен в функции-исполнители промиса, которые вызываются синхронно. Это важная часть выполнения функции `async`: она создает свой промис и выполняет его код синхронно до первого оператора `await` или `return` (или логика убирает окончание функции). Если этот синхронный код выдает ошибку, функция `async` отклоняет свой промис. Как только синхронный код завершится (нормально или с ошибкой), функция `async` возвращает свой промис вызывающему коду. Это делается для того, чтобы процесс мог запускаться синхронно, и завершиться асинхронно — точно так же, как это делает функция-исполнитель промиса.

После прочтения главы 8 и, в частности, ее раздела, посвященного антишаблонам, ваша первая реакция на «схематичный» код вполне может быть: «Но тут используется `new Promise` (дважды!). Код должен просто связать промис с `fetch`». Это правда, но способ выполнения с функцией `async` должен быть универсальным настолько, чтобы обеспечить возможность вашему коду не использовать `await` и, следовательно, не будет никакого промиса для связки. Хотя нет особых причин для использования функции `async`, которая никогда не использует оператор `await`. Этот пример также должен гарантировать, что обработчик `finally` выполняется, если существует вызов исключение `fetch`, чего не было бы, если бы функция просто отключила цепочку от промиса `fetch`. Вы бы написали функцию по-другому, если бы делали это вручную, но у функций `async` есть пара аспектов, которые они должны обеспечить:

- Возвращаемый промис должен быть *нативным*. Простой возврат результата `then` будет означать, что результатом может быть сторонний промис. Или, возможно, даже не промис, если объект, для которого вызывается `then`, — это *thenable*, а не промис. (Хотя это можно было бы сделать, используя `Promise.resolve`.)
- И любая, возникающая даже во время синхронной части кода, ошибка превращается в отклонение, а не в синхронное исключение.

Использование выражения `new Promise`, хотя в большинстве случаев оно — антишаблон для написанного вручную кода, когда у вас есть промис для создания цепочки, удовлетворяет обоим этим требованиям. Оба эти аспекта могли бы быть решены другими способами, но этот простой способ — предписание спецификации.

Оператор await использует промисы

Другой момент, который стоит рассмотреть в «схематичной» версии функции `getTheData`, — `await` потребляет промисы. Если операнд `await` не является нативным промисом, движок JavaScript создает собственный промис и использует значение операнда для его разрешения. Затем движок JavaScript ожидает выполнения этого промиса, прежде чем продолжить выполнение последующего кода в функции, как если бы он использовал `then` и передавался в обработчиках выполнения и в обработчиках отклонения. Помните, что разрешение промиса не обязательно означает его выполнение. Если операнд для `await` представлен изменяемым значением, собственный промис преобразуется в это изменяемое значение и будет отклонен, если изменяемое значение будет отклонено. Если значение не представлено элементом thenable, собственный промис успешно выполняется с этим значением. Оператор `await` выполняет это при помощи

эффективного применения метода `Promise.resolve` к значению операнда везде, где `await` его использует. (Не сам метод `Promise.resolve`, а базовую выполняемую им операцию, создавая промис и разрешает его со значением.)

Стандартная логика становится асинхронной при использовании await

В функции `async` код, традиционно синхронный, становится асинхронным, если задействован оператор `await`. Например, рассмотрим эту асинхронную функцию:

```
async function fetchInSeries(urls) {
    const results = [];
    for (const url of urls) {
        const response = await fetch(url);
        if (!response.ok) {
            throw new Error("HTTP error " + response.status);
        }
        results.push(await response.json());
    }
    return results;
}
```

Для запуска этого кода используйте файлы **async-fetchInSeries.html, async-fetchInSeries.js, 1.json, 2.json** и **3.json** из загрузок. Вам нужно включить их через веб-сервер для запуска кода в большинстве браузеров, поскольку выполнение ajax-запросов со страниц, загруженных с URL-адресов `file://`, часто запрещено.

Если бы это была не-`async` функция (или в ней не использовались операторы `await`), то вызовы `fetch` запускались бы параллельно (все сразу), а не серийно (друг за другом). Но поскольку это функция `async` и используется `await` в цикле `for-of`, цикл выполняется асинхронно. Версия этого кода с использованием промисов напрямую может выглядеть примерно так:

```
function fetchInSeries(urls) {
    let chain = Promise.resolve([]);
    for (const url of urls) {
        chain = chain.then(results => {
            return fetch(url)
                .then(response => {
                    if (!response.ok) {
                        throw new Error("HTTP error " + response.status);
                    }
                    return response.json();
                })
                .then(result => {
                    results.push(result);
                    return results;
                });
        });
    }
    return chain;
}
```

Для запуска этого кода используйте файлы **promise-fetchInSeries.html, promise-fetchInSeries.js, 1.json, 2.json** и **3.json** из загрузок. Опять же, вам нужно будет включить их через веб-сервер, а не просто открывать HTML локально.

Обратите внимание, что цикл `for-of` просто задает цепочку промисов (как показано в главе 8), завершаясь до выполнения любой асинхронной работы. Также обратите внимание, насколько запутанно и сложно следить за кодом.

Это основная часть возможностей функций `async`: вы пишете логику старым знакомым способом, используя оператор `await` для обработки ожидания асинхронных результатов вместо того, чтобы разбивать свой логический поток функциями обратного вызова.

Отклонения — это исключения, исключения — это отклонения; выполнение — это результаты, возвращаемые значения — это разрешения

Аналогично тому, как `for-of` и `while` и другие операторы управления ходом выполнения приспособлены для обработки асинхронной работы внутри функций `async`, операторы `try/catch/finally`, `throw` и `return` тоже адаптированы к этим процессам:

- Отклонения — это исключения. Когда применяется `await` для промиса, и промис отклоняется, результат превращается в исключение и может быть уловлен с помощью `try/catch/finally`.
- Исключения — это отклонения. Если вы применяете оператор `throw` в функции `async` (и не используете `catch`), это превращается в отклонение от промиса функции.
- Значения `return` — это разрешения. Если выполняете инструкцию `return` для функции `async`, это разрешает ее промис со значением операнда, отправленного в `return` (либо успешно выполняет промис, если предоставляется не-thenable значение, либо разрешает промис со значением thenable, если оно предоставлено).

Вы видели это раньше в примере с функцией `getTheData`. Там используются операторы `try/finally`, чтобы убедиться, что `spinner.stop` вызывается при завершении операции. Давайте посмотрим повнимательнее. Запустите код из Листинга 9-1.

Листинг 9-1: Операторы try/catch в асинхронной функции — async-try-catch.js

```
function delayedFailure() {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            reject(new Error("failed"));
        }, 800);
    });
}
async function example() {
    try {
        await delayedFailure();
```

```
        console.log("Done"); // (Выполнение не добирается до этого места)
    } catch (error) {
        console.error("Caught:", error);
    }
}
example();
```

Функция `delayedFailure` возвращает промис, который она позже отклонит, но при использовании `await` функция `async`, как в `example`, видит это как исключение и предоставляет `try/catch/finally` для его обработки.

ВЫЗОВЫ АСИНХРОННЫХ ФУНКЦИЙ НА ВЕРХНЕМ УРОВНЕ

В Листинге 9-1 вызывается функция async на верхнем уровне скрипта. При этом важно, чтобы либо вы знали, что функция async никогда не выдаст ошибку (то есть возвращаемый ей промис никогда не будет отклонен), либо вы добавили обработчик catch к возвращаемому промису. Это справедливо для функции example из Листинга 9-1, потому что все тело функции находится в блоке try, с добавленным к нему catch (по крайней мере, это справедливо, если по какой-то причине функция `console.error` выбрасывает исключение). В общем случае, когда функция `async` вполне может завершиться сбоем, убедитесь, что ошибки обрабатываются:

```
example().catch(error => {
    // Обработка/отчет об ошибке
});
```

Давайте посмотрим на это с другой стороны: `throw`. Запустите код из Листинга 9-2.

Листинг 9-2: Операторы throw в асинхронной функции — async-throw.js

```
function delay(ms, value) {
    return new Promise(resolve => setTimeout(resolve, ms, value));
}
async function delayedFailure() {
    await delay(800);
    throw new Error("failed");
}
function example() {
    delayedFailure()
        .then(() => {
            console.log("Done"); // (Выполнение не добирается до этого места)
        })
        .catch(error => {
            console.error("Caught:", error);
        });
}
example();
```

Эта версия функции `delayedFailure` относится к функциям `async`; она ожидает 800 мс и после применяет оператор `throw` для выброса исключения. Эта не-`async` версия функции `example` использует методы промиса, а не `await`, и видит это исключение как отклонение промиса, поймав его через обработчик отклонений.

Хотя в коде в Листингах 9-1 и 9-2 используется выражение `new Error`, это просто условность (и, возможно, лучшая практика). Поскольку JavaScript позволяет выбрасывать любое значение и использовать любое значение в качестве причины отклонения, не стоит использовать `Error`. См. Листинг 9-3, например, в нем вместо этого использованы строки. Но, как говорилось в главе 8, у использования экземпляров `Error` есть преимущества при отладке благодаря имеющейся у них информации о стеке вызовов.

Листинг 9-3: Примеры ошибок асинхронной функции — async-more-error-examples.js

```
// `reject` просто использует строку
function delayedFailure1() {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            reject("failed 1"); // Отклонение со значением, которое
                                // не относится к экземплярам Error
        }, 800);
    });
}
async function example1() {
    try {
        await delayedFailure1();
        console.log("Done"); // (Выполнение не добирается до этого места)
    } catch (error) {
        console.error("Caught:", error); // Caught: "failed 1"
    }
}
example1();

// `throw` просто использует строку
function delay(ms, value) {
    return new Promise(resolve => setTimeout(resolve, ms, value));
}
async function delayedFailure2() {
    await delay(800);
    throw "failed 2"; // Выбрасывается значение, которое не относится
                      // к экземплярам Error
}
function example2() {
    delayedFailure2()
        .then(() => {
            console.log("Done"); // (Выполнение не добирается до этого места)
        })
        .catch(error => {
            console.error("Caught:", error); // Caught: "failed 2"
        });
}
example2();
```

Параллельные операции в асинхронных функциях

Применение оператора `await` в функции `async` приостанавливает выполнение функции до тех пор, пока ожидаемый в `await` промис выполняется. Но что, если вы хотите выполнить серию операций, которые могут выполняться параллельно в функции `async`?

Это одна из ситуаций, когда вы замечаете, что снова используете промисы (или, по крайней мере, методы промисов) непосредственно, даже в функции `async`. Предположим, у вас есть функция `fetchJSON`, подобная этой:

```
async function fetchJSON(url) {
    const response = await fetch(url);
    if (!response.ok) {
        throw new Error("HTTP error " + response.status);
    }
    return response.json();
}
```

Теперь предположим, что у вас есть три ресурса, требующие извлечения, и естественным будет делать это параллельно. Вам *не* стоит делать так:

```
// Не делайте этого, если требуется параллельное выполнение
const data = [
    await fetchJSON("1.json"),
    await fetchJSON("2.json"),
    await fetchJSON("3.json")
];
```

Причина в том, что операторы будут выполняться последовательно (один за другим), а не параллельно. Помните, что движок JavaScript оценивает выражения, составляющие содержимое массива, перед созданием массива и присваивает их переменной `data`, поэтому функция приостанавливается на строке `await fetchJSON("1.json")` до тех пор, пока этот промис не будет выполнен, затем на строке `await fetchJSON("2.json")`, и т. д.

Вспомните: в главе 8 было описано, что существует нечто, специально разработанное для обработки параллельных операций с промисами, — метод `Promise.all`. Даже при использовании функции `async` нет никаких причин, запрещающих его использовать:

```
const data = await Promise.all([
    fetchJSON("1.json"),
    fetchJSON("2.json"),
    fetchJSON("3.json")
]);
```

Оператор `await` ожидает промис из `Promise.all`, а не отдельные промисы из вызовов `fetchJSON`. Эти промисы заполняют передаваемый в `Promise.all` массив.

Можно применить `Promise.race` и различные методы таким же образом.

Нет необходимости возвращать await

Возможно, вы заметили, что функция `fetchJSON` в этом разделе заканчивается так:

```
return response.json();
```

а не так:

```
return await response.json();
```

Здесь нет необходимости применять `await`. Функция `async` использует возвращаемое значение для разрешения промиса, созданного функцией, поэтому, если возвращаемое значение это thenable, оно уже фактически «ожидаемое». Написать `return await`, почти то же самое, что написать `await await`. Иногда попадаются выражения `return await`, и кажется, что код ведет себя точно так же, но это не совсем так. Если операнд представлен элементом thenable, а не нативным промисом, он добавляет дополнительный уровень разрешения промиса, задерживающий выполнение на один асинхронный цикл (или «тик»). То есть версия с оператором `await` выполняется немного позже, чем без `await`. Посмотрите, как это работает, выполнив следующее (файл **return-await-thenable.js** в загрузках):

```
function thenableResolve(value) {
    return {
        then(onFulfilled) {
            // Thenable может вызывать свой обратный вызов таким образом,
            // синхронно; нативный промис - нет. В этом примере используется
            // синхронный обратный вызов. Это не создаст впечатление, что
            // механизм, используемый, который должен сделать его
            // асинхронным, является причиной дополнительного "тика".
            onFulfilled(value);
        }
    };
}
async function a() {
    return await thenableResolve("a");
}
async function b() {
    return thenableResolve("b");
}
a().then(value => console.log(value));
b().then(value => console.log(value));
// b
// a
```

Обратите внимание, что обратный вызов функции `b` запускается раньше, чем обратный вызов `a`, даже когда функция `a` вызывается раньше. Обратный вызов `a` должен ожидать дополнительного асинхронного цикла из-за наличия оператора `await`.

Это также должно быть верно, если ожидается нативный промис. Но ES2020 включает в себя изменение спецификации, позволяющее оптимизировать выражение `return await nativePromise` для удаления дополнительного асинхронного «тика», и эта оптимизация уже находит свое применение в движках JavaScript. Если вы используете

Node.js v13 или более позднюю версию[56], или обновленную версию Chrome, Chromium или Brave, все они поддерживают версию V8 с оптимизацией. Если вы используете Node.js v12 или более раннюю версию (без флагов) или Chrome v72 или более раннюю версию, они используют версию V8 без оптимизации. Запустите следующий код (файл **return-await-native.js**) на чем-то современном, чтобы увидеть оптимизацию в действии:

```
async function a() {
    return await Promise.resolve("a");
}
async function b() {
    return Promise.resolve("b");
}
a().then(value => console.log(value));
b().then(value => console.log(value));
```

С движком, поддерживающим оптимизацию, вы сначала видите `a`, затем `b`. Движок без нее сначала отобразит `b`, затем `a`.

Итог: вам не нужно использовать выражение `return await`, просто используйте `return`.

Ловушка Pitfall: Использование асинхронной функции в неожиданном месте

Предположим, вы выполняете операцию `filter` над массивом:

```
filteredArray = theArray.filter((entry) => {
    // ...
});
```

и вам требуется использовать данные из асинхронной операции в обратном вызове `filter`. Возможно, вы вводите `await fetch(entry.url)` в код:

```
// Завершается неудачей
filteredArray = theArray.filter((entry) => {
    const response = await fetch(entry.url);
    const keep = response.ok ? (await response.json()).keep: false;
    return keep;
});
```

Но вы получите сообщение об ошибке с жалобой на ключевое слово `await`, из-за попытки использовать его вне функции `async`. Соблазн заключается в том, чтобы включить `async` в обратный вызов `filter`:

```
// Неверно
filteredArray = theArray.filter(async (entry) => {
    const response = await fetch(entry.url);
    const keep = response.ok ? (await response.json()).keep: false;
    return keep;
});
```

[56] Node.js v11 и v12 тоже поддерживают это, после флага `--harmony-await-optimization`.

Теперь ошибок нет… но ваш массив `filteredArray` по-прежнему содержит все значения исходного массива! И он не ждет завершения операций `fetch`. Как вы думаете, почему так происходит? (Подсказка: что возвращает функция `async` и что ожидает `filter`?)

Верно! Проблема заключается в том, что функция `async` всегда возвращает промис, а функция `filter` не ожидает промис, она ожидает флаг keep/don't keep. Промисы — это объекты, а объекты истинноподобны, поэтому `filter` рассматривает каждый возвращенный промис как флаг, указывающий сохранить запись.

Это часто возникает, когда люди только начинают использовать `async`/`await`. Это мощный инструмент, но важно помнить, что при написании функции обратного вызова вам нужно подумать о том, как будет использоваться возвращаемое значение этого обратного вызова. Используйте функцию `async` в качестве обратного вызова только в том случае, если то, что ее вызывает (`filter` в этом примере), ожидает получить от нее промис.

Есть несколько мест, где вполне допустимо использовать функцию `async` в качестве обратного вызова API, не зависящего от промиса. Одним из хороших примеров будет использование `map` в массиве для построения массива промисов, предаваемого в метод `Promise.all` или аналогичный. Но в большинстве случаев, если вы заметите, что пишете функцию `async` в качестве обратного вызова к чему-то, что явно не связано с промисами, дважды проверьте, не попадете ли вы в эту яму.

АСИНХРОННЫЕ ИТЕРАТОРЫ, ИТЕРИРУЕМЫЕ И ГЕНЕРАТОРЫ

В главе 6 рассказано об итераторах, итерируемых и генераторах. В ES2018 уже существуют асинхронные версии их всех. Если вы еще не прочитали главу 6, вам следует сделать это, прежде чем продолжить чтение этого раздела.

Вы запомните, что итератор — это объект с методом `next`, возвращающий результирующий объект со свойствами `done` и `value`. *Асинхронный* итератор — это итератор, возвращающий *промис* результирующего объекта, а не сам результирующий объект.

Вы также запомните, что *итерируемый* элемент — это объект, содержащий метод `Symbol.iterator`, который возвращает итератор. У *асинхронных* итерируемых есть аналогичный метод — `Symbol.AsyncIterator`, возвращающий асинхронный итератор.

Наконец, вы запомните, что функция-генератор обеспечивает синтаксис для создания объектов генераторов, которые являются объектами, производящими и потребляющими значения и содержащими методы `next`, `throw` и `return`. *Асинхронная* функция-генератор создает асинхронный генератор, выдающий *промисы* значений, а не сами значения.

В следующих двух разделах мы рассмотрим асинхронные итераторы и генераторы более подробно.

Асинхронные итераторы

Асинхронный итератор — это, повторим, просто итератор, чей метод `next` предоставляет промис результирующего объекта, а не сам результирующий объект. Как и в случае с итератором, можно написать его вручную (либо создав `next` в методе `async`, либо

вернув промис вручную), либо использовать асинхронную функцию-генератор (поскольку асинхронные генераторы также являются асинхронными итераторами). Или можно даже написать его полностью вручную, при необходимости используя `new Promise`.

В подавляющем большинстве случаев, когда требуется асинхронный итератор, лучше всего писать асинхронную функцию-генератор. Однако, как и в случае с итераторами, полезно понять лежащую в их основе механику. Вы узнаете об асинхронных генераторах в следующем разделе. В этом разделе давайте реализуем асинхронный итератор вручную, чтобы вы могли хорошо ознакомиться с его механикой.

Возможно, вы помните из главы 6, что все итераторы, получаемые из самого JavaScript, наследуются от объекта-прототипа, спецификация называет его `%IteratorPrototype%`. Объект `%IteratorPrototype%` предоставляет метод по умолчанию `Symbol.iterator`, возвращающий сам итератор, так что эти итераторы — итерируемые. Это удобно для выражений `for-of` и аналогичных. Он также предоставляет место для добавления функций в итераторы. Асинхронные итераторы работают одинаково: любой асинхронный итератор, получаемый из самого JavaScript (в отличие от стороннего кода), наследуется от объекта, который спецификация называет `%AsyncIteratorPrototype%`. Он предоставляет метод по умолчанию `Symbol.asyncIterator`, возвращающий сам итератор, так что асинхронные итераторы также являются асинхронными итерируемыми. Это будет полезно, когда вы узнаете о выражениях `for-await-of` позже в этой главе.

Как и в случае с `%IteratorPrototype%`, нет общедоступной глобальной переменной или свойства, ссылающихся на `%AsyncIteratorPrototype%`, и это более неудобно, чем добиться решения при помощи `%IteratorPrototype%`. Вот как это делается:

```
const asyncIteratorPrototype =
    Object.getPrototypeOf(
        Object.getPrototypeOf(
            (async function *(){}).prototype
        )
    );
```

Возможно, это и к лучшему, что вы не будете часто реализовывать асинхронные итераторы вручную. Если вам нужны кровавые подробности, стоящие за этим, см. врезку «Как получить `%AsyncIteratorPrototype%`».

Теперь, когда вы знаете, как получить прототип для вашего асинхронного итератора, пришло время его создать. Ранее в этой главе была покзана функция `fetchInSeries`: она извлекала несколько URL-адресов один за другим с помощью метода `fetch` и предоставляла массив со всеми результатами. Предположим, вы хотите, чтобы функция извлекала их по отдельности и предоставляла этот результат, прежде чем переходить к следующему. Это пример использования асинхронного итератора (Листинг 9-4).

КАК ПОЛУЧИТЬ %ASYNCITERATORPROTOTYPE%?

Поскольку нет общедоступной глобальной переменной или свойства, непосредственно ссылающихся на `%AsyncIteratorPrototype%`, при необходимости получения этого прототипа придется реализовывать опосредованно.

Краткий код для этого приведен в основном тексте. Вот версия, разделенная на отдельные этапы, чтобы упростить ее понимание:

```
let a = (async function *(){}).prototype; // Получение прототипа,
который эта асинхронная
                                              // функция генератора
будет назначать экземплярам
let b = Object.getPrototypeOf(a);     // Его прототип
                                      // %AsyncGeneratorPrototype%
let asyncIteratorPrototype =
        Object.getPrototypeOf(b);     // Его прототип
                                      // %AsyncGeneratorPrototype%
```

Вы создаете асинхронную функцию-генератор (то есть функцию, которая при вызове создает асинхронный генератор), используя синтаксис генератора async, о котором вы узнаете в следующем разделе. Затем вы получаете его свойство prototype (прототип, который он назначит создаваемым им генераторам). Затем вы получаете его прототип, представленный прототипом объектов асинхронного генератора (который спецификация называет `%AsyncGeneratorPrototype%`). Затем вы получаете его прототип, представленный прототипом для асинхронных итераторов — `%AsyncIteratorPrototype%`.

Опять же, возможно, это хорошо, что вы не будете создавать асинхронные итераторы вручную. Вместо этого просто используйте функцию асинхронного генератора.

Листинг 9-4: Использование асинхронной функции для создания асинхронного итератора — async-iterator- fetchInSeries.js

```
function fetchInSeries([...urls]) {
    const asyncIteratorPrototype =
        Object.getPrototypeOf(
            Object.getPrototypeOf(
                async function*(){}
            ).prototype
        );
    let index = 0;
    return Object.assign(
        Object.create(asyncIteratorPrototype),
        {
            async next() {
                if (index >= urls.length) {
                    return {done: true};
                }
```

```
                const url = urls[index++];
                const response = await fetch(url);
                if (!response.ok) {
                    throw new Error("Error getting URL: " + url);
                }
                return {value: await response.json(), done: false};
            }
        }
    );
}
```

Реализация функции `fetchInSeries` возвращает асинхронный итератор, реализованный в виде функции `async`. Поскольку это функция `async`, при каждом вызове она возвращает промис, выполняющийся при помощи оператора `return` или отклоняющийся посредством оператора `throw`.

Возможно, вам интересно узнать о деструктуризации в списке параметров. Для чего требуется `([...urls])`, вместо простого `(urls)`? Используя деструктуризацию, код создает защитную копию переданного массива, поскольку вызывающий код может изменить исходный массив. Это не имеет ничего общего с асинхронными итераторами как таковыми — просто довольно стандартная практика для функции, работающей с полученным массивом асинхронно.

Вот один из способов использования функции `fetchInSeries`, получив итератор вручную и вызвав метод `next`:

```
// В асинхронной функции
const it = fetchInSeries(["1.json", "2.json", "3.json"]);
let result;
while (!(result = await it.next()).done) {
    console.log(result.value);
}
```

Обратите внимание, что в коде применяется выражение `await it.next()` для получения следующего результирующего объекта, поскольку `next` возвращает промис.

Вы можете увидеть это в действии. Возьмите файлы **async-iterator-fetchInSeries.html, async-iteratorfetchInSeries.js, 1.json, 2.json**, и **3.json** из загрузок и поместите их в каталог на своем локальном веб-сервере, а затем откройте через HTTP. И снова простое открытие HTML непосредственно из файловой системы не сработает из-за ajax.

В реальном коде вы, вероятно, просто написали бы функцию-генератор `async`, чтобы создать асинхронный итератор. Если все-таки предстоит написать его вручную, имейте в виду, что, поскольку метод `next` запускает асинхронную операцию и возвращает для нее промис, вы можете получить следующий вызов `next` до завершения операции предыдущего вызова. Код в примере в этом разделе подходит для такого решения. Но это было бы не так, если бы в коде использовалась такая версия метода `next`:

```
// Неверно
async next() {
    if (index >= urls.length) {
        return {done: true};
    }
    const url = urls[index];
    const response = await fetch(url);
```

```
    ++index;
    if (!response.ok) {
        throw new Error("Error getting URL: " + url);
    }
    return {value: await response.json()};
}
```

Поскольку `await` находится между проверкой `index` на соответствие длине `urls.length` и инкрементом `++index`, два перекрывающихся вызова `next` будут получать один и тот же URL-адрес (а второй будет пропущен). Это трудная для диагностики ошибка.

Давайте посмотрим, как надо писать асинхронный генератор.

Асинхронные генераторы

Неудивительно, что асинхронная функция-генератор — это комбинация функции `async` и функции-генератора. Она создается с помощью ключевого слова `async`, или синтаксиса «*», указывающего на функцию-генератор. При вызове создается асинхронный генератор. Внутри асинхронной функции-генератора можно использовать оператор `await`, чтобы дождаться завершения асинхронной операции, и `yield`, чтобы получить значение (и использовать значение, как и с не-`async` функцией-генератором).

Листинг 9-5 для изучения версии асинхронного генератора `fetchInSeries`. (Вы можете запустить его через `async-generator-fetchInSeries.html` в разделе загрузки.)

Листинг 9-5: Использование функции асинхронного генератора — async-generator-fetchInSeries.js

```
async function* fetchInSeries([...urls]) {
    for (const url of urls) {
        const response = await fetch(url);
        if (!response.ok) {
            throw new Error("HTTP error " + response.status);
        }
        yield response.json();
    }
}
```

Это намного проще, чем писать его вручную! Основная логика остается той же, но вместо того, чтобы использовать закрытие над переменной `index`, вы можете просто использовать цикл `for-of` с оператором `await` внутри, чтобы перебрать URL-адреса, и получить с помощью `yield` каждый результат. Также генератор автоматически получает соответствующий прототип — `fetchInSeries.prototype`, наследуемый от `%AsyncGeneratorPrototype%`, наследуемый от `%AsyncIteratorPrototype%`.

Код для его использования (пока вручную: вы узнаете о более простом способе использования выражения `for-await-of` в следующем разделе) такой же, как и раньше:

```
// В асинхронной функции
const g = fetchInSeries(["1.json", "2.json", "3.json"]);
```

```
let result;
while (!(result = await g.next()).done) {
    console.log(result.value);
}
```

Посмотрите еще раз на эту строку в коде предыдущей асинхронной функции-генераторе:

```
yield response.json();
```

Это выражение непосредственно получает промис. Иногда эту строку пишут следующим образом:

```
yield await response.json(); // Не лучший сопособ
```

и они обе, *кажется*, делают одно и то же. Почему оператор `await` здесь необязателен? Потому что в асинхронной функции-генераторе `yield` автоматически применяет `await` к любому заданному вами операнду, так что в этом нет необходимости. Это похоже на `return await` в функции `async`: просто добавляется еще один уровень разрешения промисов, потенциально задерживая его еще на один «тик» (подробности см. в разделе «Нет необходимости возвращать await», рассмотренном ранее в этой главе).

До сих пор мы использовали асинхронный генератор только для *создания* значений, но генераторы также могут *использовать* значения: вы можете передавать значения в оператор `next`, которые генератор увидит в качестве результата оператора `yield`. Так как не-`async` генератор создает и использует значения и генератор `async` автоматически оборачивает значения, полученные через `yield` в промисах, вас может удивить, что он будет автоматически ожидать промис с помощью `await`, если вы его передадите в `next`. Нет, это не так. Если вы передадите промис в метод `next`, код генератора `async` увидит этот промис в качестве результата оператора `yield` и будет должен явно ожидать его при помощи `await` (или использовать `then` и т. д.). Хотя это несимметрично тому, что происходит, когда выдается промис, это означает, что у вас есть контроль над операциями с промисом, если он предоставлен вашему коду асинхронным генератором. Возможно, вы хотите собрать несколько из них в массив, а затем применить к ним выражение `await Promise.all`. Отсутствие автоматического ожидания значения, полученного от `yield`, дает вам такую гибкость.

Давайте изменим функцию `fetchInSeries`, чтобы получить возможность передать в метод `next` флаг, указывающий, что необходимо пропустить следующую запись (Листинг 9-6).

Листинг 9-6: Асинхронный генератор, использующий значения — async-generator-fetchInSeries-with-skip.js

```
async function* fetchInSeries([...urls]) {
    let skipNext = false;
    for (const url of urls) {
        if (skipNext) {
            skipNext = false;
        } else {
            const response = await fetch(url);
```

```
            if (!response.ok) {
                throw new Error("HTTP error " + response.status);
            }
            skipNext = yield response.json();
        }
    }
}
```

Чтобы увидеть этот код в действии, скопируйте файлы **async-generator-fetchInSeries-with-skip.html, asyncgenerator-fetchInSeries-with-skip.js, 1.json, 2.json** и **3.json** на свой локальный веб-сервер и откройте HTML-файл оттуда (через HTTP).

Последнее замечание об асинхронных генераторах: как только генератор приостанавливается и возвращает промис из `next`, выполнение кода генератора не продолжается до тех пор, пока этот промис не будет выполнен — даже при повторном вызове `next`. Повторный вызов `next` возвращает, как положено, второй промис, но не продвигает выполнение генератора вперед. Когда первый промис выполняется, генератор продвигается вперед и выполняет второй промис. Это относится и к вызовам методов `return` и `throw` генератора.

Выражение `for-await-of`

До сих пор вы видели только примеры явного использования асинхронного итератора, получения объекта итератора и вызова его метода `next`:

```
// В асинхронной функции
const it = fetchInSeries(["1.json", "2.json", "3.json"]);
let result;
while (!(result = await it.next()).done) {
    console.log(result.value);
}
```

Но точно так же, как существует цикл `for-of` для более удобного использования синхронного итератора (глава 6), есть и цикл `for-await-of` для более удобного использования асинхронного итератора:

```
for await (const value of fetchInSeries(["1.json", "2.json", "3.json"])) {
    console.log(value);
}
```

Цикл `for-await-of` получает итератор от того, что ему предается при помощи вызова его метода `Symbol.asyncIterator`[57], а затем автоматически срабатывает `await`, ожидая результата вызова метода `next`.

Чтобы увидеть этот код в действии, скопируйте файлы **for-await-of.html, async-generator-fetchInSeries-withskip.js, 1.json, 2.json** и **3.json** на свой локальный веб-сервер и откройте HTML-файл оттуда (через HTTP).

[57] Помните, что если вы передаете ему асинхронный *итератор*, а не асинхронный *итерируемый* элемент, это сработает при условии, что итератор был создан правильно (то есть содержит метод `Symbol.asyncIterator`, возвращающий `this`).

ОТ СТАРЫХ ПРИВЫЧЕК К НОВЫМ

Перед вами некоторые старые привычки, которые стоит пересмотреть, чтобы использовать новые возможности, описанные в этой главе.

Используйте асинхронные функции и await вместо явных промисов и then/catch

Старая привычка: Явное использование синтаксиса промиса:

```
function fetchJSON(url) {
    return fetch(url)
        .then(response => {
            if (!response.ok) {
                throw new Error("HTTP error " + response.status);
            }
            return response.json();
        });
}
```

Новая привычка: Использовать `async`/`await`. Так вы можете написать логику своего кода без необходимости использовать обратные вызовы для асинхронных частей:

```
async function fetchJSON(url) {
    const response = await fetch(url);
    if (!response.ok) {
        throw new Error("HTTP error " + response.status);
    }
    return response.json();
}
```

10

Шаблоны, помеченные функции и новые возможности строк

СОДЕРЖАНИЕ ГЛАВЫ

- Шаблонные литералы
- Помеченные шаблонные функции
- Улучшенная поддержка Юникода в строках
- Итерация строк
- Новые строковые методы
- Обновления методов `match`, `split`, `search` и `replace`

В этой главе вы узнаете о новых шаблонных литералах ES2015 и помеченных шаблонных функциях, а также о новых возможностях строк, таких как улучшенная поддержка Юникода, итерации и добавленных методах.

ШАБЛОННЫЕ ЛИТЕРАЛЫ

Шаблонные литералы ES2015 предоставляют способ создания строк (и других элементов) с использованием литерального синтаксиса, объединяющего текст и встроенные замены. Вы знакомы с другими типами литералов, такими как строковые литералы, отделенные кавычками (`"hi"`), и литералы регулярных выражений, разделенные при помощи слешей (`/\s/`). Шаблонные литералы разделяются обратными апострофами (`` ` ``), также называемыми обратными кавычками. Этот символ находится в разных местах на разных языковых раскладках клавиатуры: на англоязычных клавиатурах он обычно находится слева

сверху около клавиши Esc. Шаблонные литералы бывают двух видов: непомеченные и помеченные[58]. Сначала мы рассмотрим непомеченные (автономные) шаблонные литералы, а затем помеченные.

Базовая функциональность (Непомеченные шаблонные литералы)

Непомеченный шаблонный литерал создает строку. Вот простой пример:

```
console.log(`This is a template literal`);
```

Пока что кажется, что это не дает ничего такого, чего не может дать строковый литерал, но шаблонные литералы обладают несколькими удобными функциями. В шаблонном литерале вы можете использовать замены для заполнения содержимого из любого выражения. Подстановка начинается со знака доллара (`$`), за которым сразу следует открывающая фигурная скобка (`{`), и заканчивается закрывающей фигурной скобкой (`}`). Все, что находится между фигурными скобками — это тело подстановки: выражение JavaScript. Выражение вычисляется при вычислении шаблонного литерала. Его результат используется вместо подстановки. Взгляните на пример:

```
const name = "Fred";
console.log(`My name is ${name}`);                  // My name is Fred
console.log(`Say it loud! ${name.toUpperCase()}!`); // Say it loud! FRED!
```

Если результат выражения не является строкой в непомеченном шаблонном литерале, он преобразуется в единицу.

Если вам нужно, чтобы в тексте фактически был знак доллара, за которым следует фигурная скобка, экранируйте знак доллара:

```
console.log(`Not a substitution: \${foo}`); // Не подстановка: ${foo}
```

Избегать знаков доллара не стоит, если за ними не следует открывающая фигурная скобка.

Еще одна удобная функция заключается в том, что, в отличие от строковых литералов, шаблонные литералы могут содержать неэкранированные новые строки, которые сохраняются в шаблоне. Этот код:

```
console.log(`Line 1
Line 2`);
```

выводит

```
Line 1
Line 2
```

[58] В русскоязычной литературе также встречаются определения «нетеговые» и «теговые» литералы. — *Прим. науч. ред.*

Обратите внимание, что любой начальный пробел в строке, следующей за новой строкой, *включается* в шаблон. Так что это:

```
for (const n of [1, 2, 3]) {
    console.log(`Line ${n}-1
    Line ${n}-2`);
}
```

выводит

```
  Line 1-1
      Line 1-2
  Line 2-1
      Line 2-2
  Line 3-1
      Line 3-2
```

Поскольку телом подстановки выступает любое выражение JavaScript, вы можете использовать новые строки и отступы, если тело достаточно сложное. Это просто пробел в выражении, поэтому он не включается в строку:

```
const a = ["one", "two", "three"];
console.log(`Complex: ${
    a.reverse()
     .join()
     .toUpperCase()
}`);                        // "Complex: THREE, TWO, ONE"
```

Выражение в теле подстановки никоим образом не ограничено, это полное выражение. Среди прочего это означает, что вы можете поместить шаблонный литерал в шаблонный же литерал, хотя его может быстро стать трудно читать и поддерживать:

```
const a = ["text", "from", "users"];
const lbl = "Label from user";
show(`<div>${escapeHTML(`${lbl}: ${a.join()}`)}</div>`);
```

Это работает просто отлично. Но с точки зрения стиля, вероятно, лучше переместить этот внутренний шаблонный литерал вовне для простоты:

```
const a = ["text", "from", "users"];
const lbl = "Label from user";
const userContent = `${lbl}: ${a.join()}`;
show(`<div>${escapeHTML(userContent)}</div>`);
```

Внутри шаблонного литерала все стандартные экранирующие (escape) последовательности работают так же, как и в строковых литералах: `\n` создает новую строку, `\u2122` — символ «™» и т. д. Это означает, что для того, чтобы фактически поместить обратный слеш в шаблон, вам нужно экранировать его точно так же, как в строковом литерале: `\\`.

Помеченные шаблонные функции (Помеченные шаблонные литералы)

В дополнение к использованию в непомеченном варианте для создания строк шаблонные литералы в сочетании с *помеченными функциями* полезны для других целей.

Помеченная функция — это функция, предназначенная для вызова с использованием синтаксиса вызова помеченной функции, в котором не используются круглые скобки « ( ) », как это бывает в обычных вызовах. Вместо них указывается имя функции, за которым следует шаблонный литерал (возможно с пробелом между ними):

```
example`This is the template to pass to the function`;
// или
example `This is the template to pass to the function`;
```

Этот код вызывает функцию `example`. Это новый стиль вызова функций для JavaScript (начиная с ES2015). При вызове таким образом функция `example` получает шаблон из шаблонного литерала (массив жестко закодированных текстовых сегментов литерала) в качестве своего первого аргумента, за которым следуют дискретные аргументы для значений, полученных в результате вычисления выражений подстановки. Для примера запустите код из Листинга 10-1.

Листинг 10-1: Пример базовой помеченной функции — tag-function-example.js

```
function example(template, value0, value1, value2) {
    console.log(template);
    console.log(value0, value1, value2);
}
const a = 1, b = 2, c = 3;
example`Testing ${a} ${b} ${c}.`;
```

Этот код выводит следующее:

```
["testing ", " ", " ", "."]
1 2 3
```

Обратите внимание, что массив шаблонов содержит начальное слово `"testing "`, включая висящий пробел, пробел между тремя заменами и текст после последней замены в конце шаблона (точка). Следующие за этим аргументы обладают значениями подстановки (`value0` содержит результат `${a}`, `value1` содержит результат `${b}`, и т. д.).

Если ваша функция не ожидает фиксированного количества подстановок, обычно для значений подстановок используется остаточный параметр, например:

```
function example(template, ...values) {
    console.log(template);
    console.log(values);
}
const a = 1, b = 2, c = 3;
example`Testing ${a} ${b} ${c}.`;
```

Этот код выводит почти то же самое, что и раньше, за исключением того, что теперь значения находятся в массиве (`values`):

```
["testing ", " ", " ", "."]
[1, 2, 3]
```

Вычисленные значения подстановки не преобразуются в строки, ваша помеченная функция получает *фактическое* значение. Это значение может быть примитивом, подобным числам в массиве `values` в предыдущем примере, или ссылкой на объект, или ссылкой на функцию — вообще любым значением. В Листинге 10-2 приведен пример, просто чтобы подчеркнуть этот момент; в большинстве случаев остальной части этого раздела мы будем использовать подстановки, приводящие к строкам.

Листинг 10-2: Помеченная функция получает не строковое значение — non-string-value-example.js

```
const logJSON = (template, ...values) => {
    let result = template[0];
    for (let index = 1; index < template.length; ++index) {
        result += JSON.stringify(values[index - 1]) + template[index];
    }
    console.log(result);
};

const a = [1, 2, 3];
const o = {"answer": 42};
const s = "foo";
logJSON`Logging: a = ${a} and o = ${o} and s = ${s}`;
```

Запуск этого кода отображает:

```
Logging: a = [1,2,3] and o = {"answer":42} and s = "foo"
```

Как вы можете видеть, что `logJSON` получает массив и объект, а не их версии, преобразованные в строку.

Метод `Array.prototype.reduce` удобен при чередовании записей из массивов `template` и `values`, как и `logJSON`. Массив `template` гарантированно получает по крайней мере одну запись[59] и будет на одну запись длиннее, чем массив `values`; следовательно, метод `reduce` без начального значения хорошо подходит для скрепления их вместе. Листинг 10-3 для изучения версии `logJSON`, использующей метод `reduce` вместо цикла.

[59] Вы, наверное, помните, что вызов метода `reduce` для пустого массива выдает ошибку, если ему не было предоставлено начальное значение. Но поскольку массив `template` гарантированно никогда не будет пустым, у нас нет такой проблемы с `logJSON` (при условии, что это вызывается как помеченная функция).

Листинг 10-3: Помеченная функция получает не строковое значение (с reduce) — non-string-value- example-reduce.js

```
const logJSON = (template, ...values) => {
    const result = template.reduce((acc, str, index) =>
        acc + JSON.stringify(values[index - 1]) + str
    );
    console.log(result);
};

const a = [1, 2, 3];
const o = {"answer": 42};
const s = "foo";
logJSON`Logging: a = ${a} and o = ${o} and s = ${s}`;
```

Этот подход хорошо работает для помеченных функций. Например, если требуется эмулировать поведение создания строк для помеченных шаблонных литералов, вы можете сделать это следующим образом:

```
function emulateUntagged(template, ...values) {
    return template.reduce((acc, str, index) => acc + values[index - 1] + str);
}
const a = 1, b = 2, c = 3;
   console.log(emulateUntagged`Testing ${a} ${b} ${c}.`);
```

(Однако вы, вероятно, не стали бы использовать метод `reduce`, чтобы делать именно то, что код там делает. Позже вы узнаете о более простом варианте для этого конкретного примера. Но метод `reduce` полезен, если вы выполняете какую-то операцию со значениями в процессе построения результата.)

Создание строк из шаблонов — очень эффективный вариант использования, но существует также множество вариантов, не связанных со строками. Функции тегов и шаблонные литералы позволяют создавать практически любой доменный язык (DSL), который вам может понадобиться.

Регулярные выражения — хороший пример.

Неудобно использовать конструктор `RegExp`, потому что он принимает строку, следовательно, любые обратные слеши, предназначенные для регулярного выражения, должны быть экранированы (и любые обратные слеши, предназначенные для использования буквально, должны быть экранированы дважды: один раз для строкового литерала и один раз для регулярного выражения; всего четыре обратных слеша). Это одна из причин, по которой существуют литералы регулярных выражений в JavaScript.

Но если требуется использовать значение переменной в регулярном выражении, вам придется использовать конструктор регулярных выражений и бороться с обратными слешами и отсутствием ясности — использовать литерал в таком случае не получится. Помеченные функции спешат на помощь!

«Но подождите, — спросите вы, — разве строки в параметре `template`… не просто строки? Разве экранирующие последовательности, созданные обратными слешами, если таковые имеются, уже не были обработаны?»

Хорошая мысль! Были обработаны, да; именно поэтому массив `template` содержит дополнительное свойство, называемое `raw`. (При этом используется тот факт, что

массивы — это объекты, поэтому у них могут быть свойства, не являющиеся элементами массива.) Свойство `raw` содержит массив *необработанного текста* из текстовых сегментов шаблона. Запустите код из Листинга 10-4.

Листинг 10-4: Помеченная функция, показывающая необработанные сегменты строки — tag-function-raw-strings.js

```
function example(template) {
    const first = template.raw[0];
    console.log(first);
    console.log(first.length);
    console.log(first[0]);
}
example`\u000A\x0a\n`;
```

Этот код берет необработанную версию первого переданного вами текстового сегмента (единственного в примере) и выводит его, его длину и только первый его символ. Результат будет следующим:

```
\u000A\x0a\n
12
\
```

Обратите внимание, что обратные слеши — это на самом деле обратные слеши; экранирующие последовательности не были интерпретированы. Также обратите внимание, что они не были преобразованы в какую-то каноническую форму; они таковы, *как написано в шаблонном литерале*. Вы знаете это, потому что `\u000A`, `\x0a` и `\n` кодируют один и тот же символ (U+000A, новая строка). Но необработанная версия именно такая. Это необработанное содержимое этого текстового сегмента в шаблонном литерале.

Используя этот массив `raw`, вы можете создать помеченную функцию для создания регулярного выражения, в котором используется текст шаблонного литерала, э-э, литерально:

```
const createRegex = (template, ...values) => {
    // Создает исходный код из необработанных текстовых сегментов и значений
    // (в следующем разделе вы увидите кое-что, что может заменить
    // этот вызов reduce)
    const source = template.raw.reduce(
        (acc, str, index) => acc + values[index - 1] + str
);
    // Проверяет, что он находится в форме /expr/flags
    const match = /^\/(.+)\/([a-z]*)$/.exec(source);
    if (!match) {
        throw new Error("Invalid regular expression");
}
    // Получение выражения и флагов, создание
    const [, expr, flags = ""] = match;
    return new RegExp(expr, flags);
};
```

НЕДОПУСТИМЫЕ ЭКРАНИРУЮЩИЕ ПОСЛЕДОВАТЕЛЬНОСТИ В ШАБЛОННЫХ ЛИТЕРАЛАХ

В ES2015-ES2017 экранирующие последовательности в шаблонных литералах были ограничены допустимыми экранирующими последовательностями JavaScript. Так, например, `\ufoo` вызовет синтаксическую ошибку, потому что экранирующая последовательность Юникода должна содержать цифры после `\u`, а не `foo`. Однако это было ограничением для DSL, поэтому в ES2018 его сняли. Если фрагмент текста содержит недопустимую экранирующую последовательность, ее ввод в `template` получит значение `undefined` и необработанный текст в `template.raw`:

```
const show = (template) => {
    console.log("template:");
    console.log(template);
    console.log("template.raw:");
    console.log(template.raw);
};
show`Has invalid escape: \ufoo${","}Has only valid escapes: \n`;
```

Этот код выведет

```
template:
[undefined, "Has only valid escapes: \n"]
template.raw:
["Has invalid escape: \\ufoo", "Has only valid escapes: \\n"]
```

С помощью этой помеченной функции можно писать регулярные выражения со встроенными переменными без необходимости двойного экранирования:

```
const alternatives = ["this", "that", "the other"];
const rex = createRegex`/\b(?:${alternatives.map(escapeRegExp).join("|")})\b/i`;
```

Этот код допускает применение функции `escapeRegExp`, не относящейся к стандартной библиотеке JavaScript, но входящей в инструментарий многих программистов. Запустите Листинг 10-5 для изучения полного примера (включая функцию `escapeRegExp`).

Листинг 10-5: Полный пример с createRegex — createRegex-example.js

```
const createRegex = (template, ...values) => {
    // Создает исходный код из необработанных текстовых сегментов и значений
    // (в следующем разделе вы увидите кое-что, что может заменить
    // этот вызов reduce)
    const source = template.raw.reduce(
        (acc, str, index) => acc + values[index - 1] + str
    );
```

```
    // Проверяет, что он находится в форме /expr/flags
    const match = /^\/(.+)\/([a-z]*)$/.exec(source);
    if (!match) {
        throw new Error("Invalid regular expression");
    }
    // Получение выражения и флагов, создание
    const [, expr, flags = ""] = match;
    return new RegExp(expr, flags);
};
// Из предложения комитета TC39: https://github.com/benjamingr/RegExp.escape
const escapeRegExp = s => String(s).replace(/[\\^$*+?.()|[\]{}]/g, "\\$&");

const alternatives = ["this", "that", "the other"];
const rex = createRegex`/\b(?:${alternatives.map(escapeRegExp).join("|")})\b/i`;

const test = (str, expect) => {
    const result = rex.test(str);
    console.log(str + ":", result, "=>", !result == !expect ? "Good": "ERROR");
};
test("doesn't have either", false);
test("has_this_but_not_delimited", false);
test("has this ", true);
test("has the other ", true);
```

Однако регулярные выражения — это лишь один из примеров DSL. Вы можете создать помеченную функцию для использования логических выражений, похожих на человеческие, для запроса дерева объектов JavaScript, используя подстановки как для значений, так и для дерева (или деревьев) для поиска:

```
// Гипотетический пример
const data = [
    {type: "widget", price: 40.0},
    {type: "gadget", price: 30.0},
    {type: "thingy", price: 10.0},
    // ...
];
// ...вызывается в ответ на ввод данных пользователем...
function searchClick(event) {
    const types = getSelectedTypes();     // Возможно `["widget", "gadget"]`
    const priceLimit = getSelectedPriceLimit(); // Возможно `35`
    const results = search`${data} for type in ${types} and price < ${priceLimit}`;
    for (const result of results) {
        // ...показать результат...
    }
}
```

Метод `String.raw`

При использовании помеченной функции метод `String.raw` возвращает строку с необработанными текстовыми сегментами из шаблона в сочетании с любыми вычисленными значениями подстановки. Например:

```
const answer = 42;
console.log(String.raw`Answer:\t${answer}`); // Answer:\t42
```

Обратите внимание, что экранированная последовательность `\t` не была интерпретирована: результирующая строка буквально содержит обратный слеш, за которым следует буква `t`. Для чего это может быть полезно?

Такое решение может потребоваться в тот момент, когда вы хотите создать строку без интерпретации экранированной последовательности в строке. Например:

- Указание жестко заданного пути в служебном скрипте на компьютере с Windows:

```
fs.open(String.raw`C:\nifty\stuff.json`)
```

- Создание регулярного выражения, содержащего обратный слеш и переменную часть (альтернатива функции `createRegex`, показанной ранее):

```
new RegExp(String.raw`^\d+${separator}\d+$`)
```

- Вывод последовательностей LaTeX или PDF (они также могут содержать обратный слеш).

В принципе, метод `String.raw` полезен в любой ситуации, когда требуется введенная вами необработанная строка (возможно, с подстановками), а не интерпретируемая строка.

Он также очень полезен, когда используется другими помеченными функциями. Например, в помеченной функции `createRegex`, чтобы создать необработанный источник нашего DSL (в данном случае регулярное выражение), нам нужно было снова собрать массив `raw` текстовых сегментов и значений, переданных в помеченную функцию с помощью подстановок, например, так:

```
const source = template.raw.reduce(
    (acc, str, index) => acc + values[index - 1] + str
);
```

Для определенного класса помеченных функций (тех, которые собирают шаблон и значения, возможно, предварительно обработанные, в строку), это общее требование — и это именно то, что делает метод `String.raw`. Таким образом, мы можем вызвать его, чтобы он выполнил эту часть задачи за нас, заменив вызов `reduce`. Поскольку он вызывается без шаблонного литерала, следует использовать для вызова обычную нотацию `()`, а не нотацию тега:

```
const source = String.raw(template, ...values);
```

Это упрощает функцию `createRegex`:

```
const createRegex = (template, ...values) => {
    // Создает исходный код из необработанных текстовых сегментов и значений
    const source = String.raw(template, ...values);
    // Проверяет, что он находится в форме /expr/flags
    const match = /^\/(.+)\/([a-z]*)$/.exec(source);
    if (!match) {
        throw new Error("Invalid regular expression");
    }
```

```
    // Получение выражения и флагов, создание
    const [, expr, flags = ""] = match;
    return new RegExp(expr, flags);
};
```

Запустите файл **simpler-createRegex.js** из перечня загрузок, чтобы увидеть это в действии.

Повторное использование шаблонных литералов

Один из часто задаваемых вопросов о шаблонных литералах таков: как использовать их повторно? В конце концов, результат непомеченного шаблонного литерала — это не объект шаблона, а строка. Что делать, если вам нужен шаблон для повторного использования? Например, предположим, что часто требуется вывести имя, фамилию, а затем псевдоним или «дескриптор» в круглых скобках. Вы можете использовать шаблонный литерал типа `` `${firstName} ${lastName} (${handle})` ``, но он немедленно вычисляется и превращается в строку. Как использовать его повторно?

Это классический случай чрезмерного обдумывания чего-то, но он возникает снова и снова.

Как это часто бывает, когда возникает вопрос: «Как мне повторно использовать это?», ответ — обернуть это в функцию:

```
const formatUser = (firstName, lastName, handle) =>
    `${firstName} ${lastName} (${handle})`;
console.log(formatUser("Joe", "Bloggs", "@joebloggs"));
```

Шаблонные литералы и автоматическая вставка точки с запятой

Если вы предпочитаете писать свой код без точек с запятой (полагаясь на автоматическую вставку точки с запятой), вы, вероятно, привыкли избегать начала строки с открывающей круглой или квадратной скобки (`(` или `[`), потому что это может быть объединено с концом предыдущей строки. Это может привести к нежелательному поведению, такому как ошибочный вызов функции или выражение свойства-акцессора, которое должно было быть началом массива, и т. п.

Шаблонные литералы добавляют новую «ASI hazard:» если вы начинаете строку с обратной метки, чтобы начать шаблонный литерал, это можно рассматривать как помеченный вызов функции, на которую есть ссылка в конце предыдущей строки (точно так же, как открывающая скобка).

Поэтому, если вы полагаетесь на ASI, добавьте обратную метку в свой список символов, перед которыми вы ставите точку с запятой в начале строки, точно так же, как `(` или `[`.

УЛУЧШЕННАЯ ПОДДЕРЖКА ЮНИКОДА

ES2015 заметно улучшил поддержку Юникода в JavaScript, добавив несколько функций к строкам и регулярным выражениям, чтобы упростить работу с полным набором символов Юникода. В этом разделе рассматриваются улучшения строк. Усовершенствования регулярных выражений описаны в главе 15.

Для начала немного терминологии и краткий обзор, прежде чем перейдем к новым функциям.

Юникод, а что такое строка JavaScript?

Если у вас уже есть четкое представление о стандарте Юникода и форматах преобразования Юникода (UTFs), кодовых точках, кодовых единицах и т. д., тогда: строка JavaScript — это серия кодовых единиц UTF-16, которая допускает недопустимые суррогатные пары. Если вы один из немногих, хорошо понявших это предложение людей, можете перейти к следующему разделу. Если вы относитесь к подавляющему большинству тех, кто этого понял не все (потому что давайте посмотрим правде в глаза, это довольно загадочные термины!), читайте дальше.

Человеческий язык сложен; системы письма на человеческом языке сложны вдвойне. Английский язык — один из самых простых: если отбросить некоторые детали, каждая *графема* английского языка («минимально отличительная единица письма в контексте конкретной системы письма»[60]) представляет собой одну из 26 букв или 10 цифр. Но многие языки так не работают. У некоторых, например, французского, есть базовый алфавит и небольшое количество диакритических знаков, используемых для изменения некоторых из этих букв (например, серьезное ударение на «a» в «voilà»); такое «à» воспринимается носителями языка как одна графема, даже если она составлена из частей. В деванагари, системе письма, используемой в Индии и Непале, есть базовый алфавит для слогов с заданной согласной и гласным звуком по умолчанию («a» в «about»). Например, «na» — это «буква» न. Для слогов с другими гласными звуками (или без гласного звука) деванагари использует диакритические знаки. Чтобы написать «ni» вместо «na», базовая буква для «na» (न) изменяется диакритическим знаком для получения звука «i» (ि), создавая «ni» (नि). «नि» воспринимается носителями языка как единая графема, хотя (опять же) она состоит из частей. В китайском языке используется несколько тысяч различных графем, и (опять же, не вдаваясь в подробности) слова обычно состоят из одной-трех. Так что это было сложно еще до того, как мы добавили в эту смесь компьютеры.

Чтобы справиться со всей этой сложностью, Юникод определяет кодовые точки — значения в диапазоне от 0 × 000000 (0) до 0 × 10FFFF (1 114111) с определенными значениями и свойствами, обычно записываемые с помощью «U +», за которыми следуют от четырех до шести шестнадцатеричных цифр. Кодовые точки — это не «символы», хотя это распространенное недопонимание. Кодовая точка *может* быть «символом» сама по себе (например, английская буква «a»), или это может быть «базовый символ» (например, «न», слог «na» в Деванагари), или «комбинирующий символ» (например, диакритический «ि», который превращает «na» в «ni») или несколькими другими вещами. В Юникоде также есть несколько кодовых точек для элементов, которые вообще не являются графемами (таких, как пробел нулевой ширины), а также для вещей, которые вообще не являются частями слов (например, смайликов).

Первоначально в Юникоде использовался диапазон от 0 × 0000 до 0 × FFFF, умещавшийся в 16 бит (это называлось «UCS2» — 2-байтовый универсальный набор символов). Кодовая точка может содержаться в одном 16-битном значении. Системы тогда использовали 16-битные «символы» для хранения строк. Когда Юникод пришлось расширить за пределы 16 бит (для 0 × 000000 до 0 × 10FFFF требуется 21 бит), это означало, что не все

[60] https://unicode.org/faq/char_combmark.html

кодовые дальше будут вписываться в 16-битные значения. Для поддержки этих 16-битных систем была создана концепция суррогатных пар: значение в диапазоне от 0 × D800 до 0 × DBFF является «ведущим» (или «высоким») суррогатом, и ожидается, что за ним последует значение в диапазоне от 0 × DC00 до 0 × DFFF, «завершающий» (или «низкий») суррогат. Полученная пара может быть преобразована в одну кодовую точку с помощью довольно простого вычисления. 16-битные значения называются *кодовыми единицами*, чтобы отличать их от кодовых точек. Это «преобразование» значений 21-битной кодовой точки в 16-битные значения кодовых единиц называется UTF-16. В правильно сформированном UTF-16 никогда не будет ведущего суррогата, за которым не следует конечный суррогат, или наоборот.

JavaScript — одна из современных на тот момент систем. «Строка» в JavaScript — это серия кодовых единиц UTF-16, за исключением того, что строки JavaScript допускают недопустимые суррогаты (начальный суррогат без завершающего или наоборот) и определяют для них семантику: если встречается только одна половина суррогатной пары, она должна быть обработана, как если бы это была кодовая точка, а не кодовая единица. Так, например, если суррогат 0 × D820 был найден изолированно, он будет рассматриваться как кодовая точка U+D820, а не как ведущий суррогат. (U+D820 зарезервирован и, следовательно, не имеет назначенного значения, поэтому если строка будет выведена, отобразится глиф «неизвестный символ».)

Все это означает, что одна графема с человеческой точки зрения может быть одной или несколькими *кодовыми точками* и что одна *кодовая точка* может быть представлена одной или двумя *кодовыми единицами* UTF-16 (строка JavaScript «символы»). Смотрите примеры в таблице 10-1.

Таблица 10-1. Примеры кодовых точек и кодовых единиц

«Символ»	Кодовая точка(и)	Кодовая единица(ы) UTF-16
Английская «a»	U+0061	0061
Деванагари «नि»	U+0928 U+093F	0928 093F
Смайлик эмодзи (☺)	U+1F60A	D83D DE0A

Однако английская графема «a» представляет собой единую кодовую точку и единую кодовую единицу UTF-16, графема «नी» деванагари «ni» требует двух кодовых точек, каждая из которых (это случается) вписывается в одну кодовую единицу UTF16. Смайлик эмодзи представляет собой одну кодовую точку, но требует двух кодовых единиц UTF-16 и, следовательно, двух «символов» JavaScript:

```
console.log("a".length); // 1
console.log("नि".length); // 2
console.log("☺".length); // 2
```

UTF-16 подходит JavaScript, но существуют также UTF-8 и UTF32. UTF-8 кодирует одну-четыре 8-битные кодовые единицы в кодовые точки (с возможностью расширения до пяти или шести кодовых единиц, если Юникод должен расширяться дальше).

UTF-32 — это просто однозначное сопоставление кодовых точек с кодовыми единицами с использованием (как вы уже догадались!) 32-битных кодовых единиц.

Фух! Какой внушительный контекст. Теперь, когда у нас есть эти знания, давайте посмотрим, что нового в этой сфере.

Экранирующая последовательность кодовой точки

В строковом литерале JavaScript раньше было так, что если необходимо было использовать экранированную последовательность для записи кодовой точки, требующей две кодовые единицы UTF-16, вам нужно было вычислить значения UTF-16 и записать их отдельно — например, так:

```
console.log("\uD83D\uDE0A"); // ☺ (Смайлик эмодзи)
```

В ES2015 добавлены *экранированные последовательности кодовой точки Юникода*, позволяющие вместо этого указывать фактическое значение кодовой точки — больше никаких сложных вычислений в формате UTF-16. Вы записываете его в фигурных скобках в шестнадцатеричном формате (как правило, значения Юникода представлены в шестнадцатеричном формате). Улыбающееся лицо с улыбающимися глазами, которое мы использовали, — это U+1F60A, так что:

```
console.log("\u{1F60A}"); // ☺ (Смайлик эмодзи)
```

Метод `String.fromCodePoint`

В ES2015 также добавили эквивалент метода `String.fromCharCode` для кодовой точки (работающий в кодовых единицах): `String.fromCodePoint`. Вы можете передать ему одну или несколько кодовых точек в виде чисел, и он предоставит эквивалентную строку:

```
console.log(String.fromCodePoint(0x1F60A)); // ☺ (Смайлик эмодзи)
```

Метод `String.prototype.codePointAt`

Продолжая тему поддержки кодовых точек, вы можете получить кодовую точку в заданной позиции в строке с помощью `String.prototype.codePointAt`:

```
console.log("☺".codePointAt(0).toString(16).toUpperCase()); // 1F60A
```

Однако это немного сложно: передаваемый вами индекс представлен в *кодовых единицах* («символах» JavaScript), а не в *кодовых точках*. Следовательно, выражение `s.codePointAt(1)` не возвращает вторую кодовую точку в строке, оно возвращает кодовую точку, начинающуюся с индекса «1» строки. Если для первой кодовой точки в строке требуются две кодовые единицы, метод `s.codePointAt(1)` вернет значение конечной суррогатной кодовой единицы этой кодовой точки:

```
const charToHex = (str, i) =>
    "0x" + str.codePointAt(i).toString(16).toUpperCase().padStart(6, "0");
const str = "☺☺"; // Два одинаковых смайлика эмодзи
```

```
for (let i = 0; i < str.length; ++i) {
    console.log(charToHex(str, i));
}
```

Этот код выводит четыре значения (0 × 01F60A, 0 × 00DE0A, 0 × 01F60A, 0 × 00DE0A), потому что каждый смайлик занимает два «символа» в строке, но код только увеличивает счетчик на единицу на каждой итерации. Значение, отображаемое во второй и четвертой позициях — 0 × 00DE0A, представляет собой конечный суррогат пары, определяющей смайлик. Код, вероятно, должен пропустить эти конечные суррогаты и просто перечислить (или найти) фактические кодовые точки в строке.

При запуске цикла с самого начала решение простое: используйте цикл `for-of` вместо цикла `for`. Подробности см. в разделе «Итерация» далее в этой главе.

Если вы попали в середину строки и хотите найти начало ближайшей кодовой точки, это тоже не так уж сложно: найдите «кодовую точку» там, где вы находитесь, и проверьте, попадает ли она в диапазон от 0 × DC00 до 0 × DFFF (включительно). Если да, то это конечный суррогат (предположительно) пары. Поэтому сделайте шаг назад, чтобы перейти к началу пары (повторяя по мере необходимости, чтобы разрешить недопустимые суррогатные пары), или сделайте шаг вперед, чтобы перейти к началу следующей кодовой точки (повторяя по мере необходимости). Также можно проверить наличие изолированных ведущих суррогатов, диапазон от 0 × D800 до 0 × DBFF. Эти два диапазона находятся рядом друг с другом, поэтому вы можете использовать 0 × D800 через 0 × DFFF для проверки, включающей отдельный начальный или конечный суррогат.

Метод `String.prototype.normalize`

Завершаем рассмотрение улучшенной поддержки Юникода. Метод строк `normalize` создает новую «нормализованную» строку, используя одну из форм нормализации, определенных консорциумом Юникода.

В Юникоде одна и та же строка может быть записана несколькими способами. Нормализация — это процесс создания новой строки, записанной в «нормальной» форме (четыре из которых определены). Это может быть важно для сравнений, расширенной обработки и т. д. Давайте рассмотрим это более внимательно.

По-французски буква «c» относится к «твердым» (произносится как русская «к»), если за ней следует буква «a». В словах, где за ней следует «a», но буква должна быть «мягкой» (произносится как буква «s»), под ней добавляется диакритический знак, называемый *седиль* (*cédille*). Вы можете видеть это в самом названии языка: Français. Произносится как «франсе», а не «франке», из-за седиля на букве «c». Когда у буквы «c» есть такой знак, ее творчески называют *c cédille (сеседиль)*.

Главным образом по историческим причинам с cédille («ç») получила свою собственную кодовую точку — U+00E7[61]. Но ее можно также записать как комбинацию буквы «c» (U+0063) и комбинированного знака для седиль (U+0327):

```
console.log("Fran\u0063ais");       // Fran\u0063ais
console.log("Franc\u0327ais"); // Fran\u0063ais
```

[61] Это компьютерная история. Французский был одним из самых ранних языков, поддерживаемых компьютерами, и наличие отдельных символов в наборе символов для букв с диакритическими знаками было простым способом его поддержки.

В зависимости от шрифта они могут выглядеть немного по-разному, но это одно и то же слово, за исключением того, что простое сравнение этого не отражает:

```
const f1 = "Franɜais";
const f2 = "Franc\u0327ais";
console.log(f1 === f2);     // ложь
```

Нормализация это исправляет:

```
console.log(f1.normalize() === f2.normalize()); // истина
```

Это лишь один из вариантов, как строки могут различаться, но при этом кодировать один и тот же текст. Некоторые языки могут применять несколько диакритических знаков к одной и той же «букве». Если порядок меток отличается в одной строке от другой, строки не будут равны при простой проверке, но будут равны в нормализованной форме.

У Юникода есть два основных типа нормализации: каноническая эквивалентность и совместимость. Внутри каждого из типов есть две формы — декомпозированная и составленная. Вот что говорит стандарт Unicode о двух типах нормализации:

> *Каноническая эквивалентность — это фундаментальная эквивалентность между символами или последовательностями символов, которые представляют один и тот же абстрактный символ и которые при правильном отображении всегда должны иметь одинаковый внешний вид и поведение…*
>
> *Эквивалентность совместимости — это более слабый тип эквивалентности между символами или последовательностями символов, которые представляют один и тот же абстрактный символ (или последовательность абстрактных символов), но которые могут иметь различные визуальные проявления или поведение. Визуальные проявления форм, эквивалентности совместимости, обычно составляют подмножество ожидаемого диапазона визуальных проявлений символа (или последовательности символов), которым они эквивалентны. Однако эти варианты форм могут представлять собой визуальное различие, которое в некоторых текстовых контекстах будет значительным, а в других нет. В результате требуется большая осторожность при определении того, когда целесообразно использовать эквивалентность совместимости. Если визуальное различие стилистическое, то для представления информации о форматировании можно использовать разметку или стиль. Однако некоторые символы с декомпозицией совместимости используются в математических обозначениях для представления различий семантического характера; замена использования различных кодов символов форматированием в таких контекстах может вызвать проблемы…*
>
> Приложение № 15 к стандарту Юникода:
> Формы нормализации Юникода

Стандарт предоставляет хороший пример того, когда тип «совместимость» может вызвать проблемы — строка «i^9». В математике это означает «i в степени 9». Но если вы нормализуете эту строку с помощью нормализации совместимости, вы получите строку

«i9» — надстрочный индекс «⁹» (U+2079) был преобразован просто в цифру «9» (U+0039). В математическом контексте эти две строки означают *очень* разные вещи. Канонический (в отличие от совместимого) тип нормализации сохраняет надстрочный характер кодовой точки. Метод `normalize` принимает необязательный аргумент, позволяющий вам контролировать, какая форма используется:

- **NFD** (Форма нормализации D): каноническая декомпозиция. В этой форме строка каноническим образом разбивается на свои наиболее дискретные части. Например, если «Français» использует кодовую точку с cédille, NFD разделит кодовую точку с cédille на отдельную кодовую точку «c» и объединяющую кодовую точку cédille. В строке с несколькими комбинирующими кодовыми точками, влияющими на одну базу, они будут расположены в каноническом порядке.
- **NFC** (Форма нормализации C): каноническая составленная. В этой форме (используется по умолчанию) строка сначала разлагается каноническим способом (NFD), а затем перекомпилируется каноническим способом, используя отдельные кодовые точки, где это уместно. Например, эта форма объединяет букву «c», за которой следует объединение cédille в единую кодовую точку «c cédille».
- **NFKD** (Форма нормализации KD): совместимая декомпозиция. В этой форме строка разбивается на наиболее дискретные части с использованием типа нормализации «совместимость» (тот, что изменил надстрочный индекс «⁹» на простой «9»).
- **NFKC** (Форма нормализации KC): совместимая составленная. В этой форме строка разлагается с использованием типа нормализации совместимости, а затем перекомпилируется с помощью канонической композиции.

Значением по умолчанию, если вы его не указали, будет `NFC`.

В зависимости от варианта использования можно выбрать любую из четырех форм, но для многих вариантов использования каноническая составная форма по умолчанию, вероятно, будет лучшим выбором. Она сохраняет полную информацию о строке, используя при этом канонические кодовые точки для комбинированных форм, которые их содержат, например с cédille.

Вот и все, что касается изменений в работе с Юникодом. Давайте посмотрим на другие варианты улучшения строк.

ИТЕРАЦИЯ

Вы узнали об итерируемых и итераторах в главе 6. В ES2015 и далее строки являются итерируемыми. Итерация посещает каждую кодовую точку (а не каждую кодовую единицу) в строке. Запустите Листинг 10-6.

Листинг 10-6: Простой пример итерации строки — simple-string-iteration-example.js

```
for (const ch of "> <") {
    console.log(`${ch} (${ch.length})`);
}
```

Так будет выглядеть результат:

```
> (1)
☺ (2)
< (1)
```

Помните, что смайлик — это одна кодовая точка, но для него требуются две кодовые единицы UTF-16 («символы» JavaScript). Таким образом, поскольку метод `length` представляет длину в кодовых единицах, вторая итерация выводит «☺ (2)».

Одним из побочных эффектов этого будет то, что в зависимости от вашего варианта использования можно решить изменить способ обычного преобразования строки в массив символов. Идиоматическим способом до ES2015 был метод `str.split("")`, разбивающий строку на массив кодовых единиц. Начиная с ES2015 вместо него можно использовать метод `Array.from(str)`, приводящий к массиву кодовых точек (а не кодовых единиц). Вы изучите метод `Array.from` в главе 11, но вкратце он создает массив перебора передаваемых вами итерируемых и добавления значения каждого итерируемого элемента в массив.

Таким образом, использование метода `Array.from` в строке разбивает ее на массив кодовых точек с помощью итератора строк:

```
const charToHex = ch =>
    "0x" + ch.codePointAt(0).toString(16).toUpperCase().padStart(6, "0");
const show = array => {
    console.log(array.map(charToHex));
};

const str = "> ☺ <";
show(str.split(""));   // ["0x00003E", "0x00D83D", "0x00DE0A", "0x00003C"]
show(Array.from(str)); // ["0x00003E", "0x01F60A", "0x00003C"]
```

Тем не менее, хотя метод `Array.from(str)` может быть лучше (для некоторых определений слова «лучше»), чем метод `str.split("")` в зависимости от вашего варианта использования, он все равно будет разбивать объединяемые кодовые точки, образуя единую воспринимаемую человеком графему. Помните тот слог «ni» (नि) из деванагари? Тот, который требует двух кодовых точек (основы и диакритического знака), но воспринимается как одна графема? Даже разделение на кодовые точки разделит его. Слово *деванагари*, обозначающее деванагари, देवनगरी, содержит пять воспринимаемых графем दे व न ग री, но даже подход с методом `Array.from` приводит к массиву из восьми кодовых точек. Более сложный алгоритм мог бы использовать информацию Юникода о том, какие кодовые точки «расширяют» графему (доступную в базе данных Юникода), но итерация строк не требует такого уровня сложности.

НОВЫЕ СТРОКОВЫЕ МЕТОДЫ

В ES2015 добавили несколько удобных служебных методов к строкам.

Метод `String.prototype.repeat`

Никакого приза за то, что угадаете значение, исходя из названия, вы не получите!

Правильно, `repeat` просто повторяет вызываемую строку заданное количество раз:

```
console.log("n".repeat(3)); // nnn
```

Если вы передадите методу значение 0 или `NaN`, вы получите обратно пустую строку. Если вы передадите методу значение меньше 0 или бесконечность (положительную или отрицательную), вы получите сообщение об ошибке.

Методы `String.prototype.startsWith` и `String.prototype.endsWith`

Методы `startsWith` и `endsWith` предоставляют простой способ проверки того, начинается ли строка с подстроки или заканчивается на нее (с необязательным начальным или конечным индексом):

```
console.log("testing".startsWith("test")); // истина
console.log("testing".endsWith("ing"));    // истина
console.log("testing".endsWith("foo"));    // ложь
```

Оба метода `startsWith` и `endsWith` возвращают значение `true`, если передать им пустую строку (`"foo".startsWith("")`).

Если передать методу `startsWith` начальный индекс, он обрабатывает строку так, как если бы она начиналась с этого индекса:

```
console.log("now testing".startsWith("test"));    // ложь
console.log("now testing".startsWith("test", 4)); // истина
// Индекс 4 ------^
```

Если индекс равен или больше длины строки, результатом вызова будет значение `false` (если вы передаете непустую подстроку), поскольку в строке в этот момент нет ничего, что могло бы соответствовать заданному значению.

Если передать методу `endsWith` конечный индекс, он обрабатывает строку так, как если бы она заканчивалась этим индексом:

```
console.log("now testing".endsWith("test"));    // ложь
console.log("now testing".endsWith("test", 8)); // истина
// Индекс 8 ----------^
```

В этом примере использование индекса 8 заставляет метод `endsWith` работать со строкой, как если бы строка была просто `"now test"`, вместо `"now testing"`.

Передача значения 0 означает, что результатом будет значение `false` (если вы передаете непустую подстроку), поскольку фактически строка для поиска будет пустой.

Проверки при помощи `startsWith` и `endsWith` всегда чувствительны к регистру.

В ES2015–ES2020 (пока) `startsWith` и `endsWith` должны выдавать ошибку, если вы передаете им регулярное выражение, а не строку. Это делается для того, чтобы помешать реализациям предоставлять свое собственное дополнительное поведение для регулярных выражений, чтобы более поздние версии спецификации JavaScript могли определять это поведение.

Метод `String.prototype.includes`

Это еще один случай, когда название в значительной степени говорит вам о том, что вам нужно знать. Метод `includes` проверяет строку, для которой вы его вызываете, чтобы увидеть, включает ли она переданную вами подстроку, необязательно начинающуюся с заданного местоположения в строке:

```
console.log("testing".includes("test"));     // истина
console.log("testing".includes("test", 1));  // ложь
```

Второй вызов в этом примере возвращает ложное значение `false`, потому что начинает проверку по индексу `"1"` строки, и по этой причине пропускает ведущую букву «t», как будто вы просмотрели только `"esting"`, а не `"testing"`.

Если вы передадите пустую подстроку для поиска, результатом будет истинное значение `true`.

Как и в методах `startsWith` и `endsWith`, вы получите ошибки при передаче отрицательного или бесконечного индекса — или при передаче регулярного выражения, а не строки (чтобы будущие версии спецификации могли определять поведение для этого).

Методы `String.prototype.padStart` и `String.prototype.padEnd`

В ES2017 добавили заполнение строк в стандартную библиотеку с помощью методов `padStart` и `padEnd`:

```
const s = "example";
console.log(`|${s.padStart(10)}|`);
// => "|   example|"
console.log(`|${s.padEnd(10)}|`);
// => "|example   |"
```

Вы указываете общую длину нужной строки и при необходимости строку, которую будете использовать для заполнения (по умолчанию используется пробел). Метод `padStart` возвращает новую строку с любым необходимым заполнением в начале строки, чтобы результат получил указанную вами длину; метод `padEnd` вместо этого заполняет окончание строки.

Обратите внимание, что вы указываете общую длину результирующей строки, а не только желаемое количество элементов заполнения. В предыдущем примере, поскольку в слове «`example`» содержалось семь символов, а длина, используемая в коде ранее, равна `10`, добавляются три пробела заполнения.

В этом примере вместо этого используются тире:

```
const s = "example";
console.log(`|${s.padStart(10, "-")}|`);
// => "|---example|"
console.log(`|${s.padEnd(10, "-")}|`);
// => "|example---|"
```

Заполняющая строка может содержать более одного символа. Он повторяется и/или усекается по мере необходимости:

```
const s = "example";
console.log(`|${s.padStart(10, "-*")}|`);
// => "|-*-example|"
console.log(`|${s.padEnd(10, "-*")}|`);
// => "|example-*-|"
console.log(`|${s.padStart(14, "...oooOOO")}|`);
// => "|...oooOexample|"
```

Возможно, вы привыкли видеть «влево» и «вправо» в этом контексте, а не «начало» и «конец». Комитет TC39 решил использовать «start» и «end» (начало и конец), чтобы избежать путаницы, когда строки используются в языковом контексте справа налево (RTL), таком как современный иврит и арабский.

Методы `String.prototype.trimStart` и `String.prototype.trimEnd`

В ES2019 добавили методы `trimStart` и `trimEnd` для строк. Метод `trimStart` обрезает пробелы с начала строки, `trimEnd` — с конца.

```
const s = " testing ";
const startTrimmed = s.trimStart();
const endTrimmed = s.trimEnd();
console.log(`|${startTrimmed}|`);
// => |testing    |
console.log(`|${endTrimmed}|`);
// => |    testing|
```

История этого дополнения несколько интереснее: когда в ES2015 добавили метод `trim` в строки, большинство движков JavaScript (в конце концов, все крупные) также добавили методы `trimLeft` и `trimRight`, хотя они не были в спецификации. При стандартизации этого решения комитет TC39 принял решение использовать имена `trimStart` и `trimEnd`, а не `trimLeft` и `trimRight`, чтобы создать соответствие методам `padStart` и `padEnd` в ES2017. Однако `trimLeft` и `trimRight` указаны в качестве псевдонимов для `trimStart` и `trimEnd` в Приложении Б (Дополнительные возможности ECMAScript для веб-браузеров).

ОБНОВЛЕНИЯ МЕТОДОВ `MATCH`, `SPLIT`, `SEARCH` И `REPLACE`

Строчные методы JavaScript `match`, `split`, `search` и `replace` были сделаны более общими в ES2015. До этого они были тесно связаны с регулярными выражениями: методам `match` и `search` требовалось регулярное выражение или строка, которую можно превратить в него. Методы `split` и `replace` использовали регулярные выражения, если они их получали; в противном случае они приводили свои аргументы к строке.

Начиная с ES2015 можно создавать свои собственные объекты для использования с методами `match`, `search`, `split` и `replace`, а также передавать эти методы вашему объекту, если у него есть определенная функция, которую ищут методы, — метод

с определенным именем. Они ищут имена, относящиеся к хорошо известным символам (вы изучили хорошо известные символы в главе 5):

- `match`: ищет `Symbol.match`
- `split`: ищет `Symbol.split`
- `search`: ищет `Symbol.search`
- `replace`: ищет `Symbol.replace`

Если передаваемый объект содержит соответствующий метод, строчный метод вызывает его, передавая строку, и возвращает результат. То есть он относится к методу вашего объекта.

Давайте используем метод `split` в качестве примера. Начиная с ES2015 строчный метод `split` концептуально выглядит следующим образом (замалчивая некоторые незначительные детали):

```
// В String.prototype
split(separator) {
    if (separator !== undefined && separator !== null) {
        if (separator[Symbol.split] !== undefined) {
            return separator[Symbol.split](this);
        }
    }
    const s = String(separator);
    const a = [];
    // ...разделяет строку на `s`, добавив к `a`...
    return a;
}
```

Как видите, метод `String.prototype.split` передает свой параметр методу `Symbol.split`, если он у него есть. В противном случае он делает то, что всегда делал с разделителем нерегулярных выражений в прошлом. Он по-прежнему поддерживает использование регулярного выражения в качестве разделителя, потому что в регулярных выражениях теперь есть метод `Symbol.split`, поэтому строковый метод `split` относится к нему.

Давайте рассмотрим использование метода `replace` с помощью другого механизма поиска — того, который ищет токены в форме `{{token}}` в строке и заменяет их соответствующими свойствами объекта (Листинг 10-7).

Листинг 10-7: Заменитель нерегулярных выражений — non-regex-replacer.js

```
// Определение заменителя (replacer) токена с помощью настраиваемого
сопоставления токенов
class Replacer {
    constructor(rexTokenMatcher = /\{\{([^}]+)\}\}/g) {
        this.rexTokenMatcher = rexTokenMatcher;
    }

    [Symbol.replace](str, replaceValue) {
        str = String(str);
        return str.replace(
            this.rexTokenMatcher,
                (_, token) => replaceValue[token] || ""
```

```
        );
    }
}
Replacer.default = new Replacer();

// Использование заменителя токенов по умолчанию с помощью `replace`
const str = "Hello, my name is {{name}} and I'm {{age}}.";
const replaced = str.replace(Replacer.default, {
    name: "Марна Gonzales",
    age: 32
});
console.log(replaced); // "Hello, my name is Марна Gonzales and I'm 32."

// Использование пользовательского токена
const str2 = "Hello, my name is <name> and I'm <age>.";
const replacer = new Replacer(/<([^>]+)>/g);
const replaced2 = str2.replace(replacer, {
    name: "Joe Bloggs",
    age: 45
});
console.log(replaced2); // "Hello, my name is Joe Bloggs and I'm 45."
```

Главное, что стоит пронаблюдать при запуске Листинга 10–7, — это вызов строковым методом `replace` метода `Symbol.replace` класса `Replacer`; он больше не просто принимает экземпляры `RegExp` или строки.

Можете выполнить что-то аналогичное с методами `match`, `split` или `search`.

ОТ СТАРЫХ ПРИВЫЧЕК К НОВЫМ

Перед вами несколько старых привычек, которые вам, возможно, захочется обновить.

Используйте шаблонные литералы вместо конкатенации строк (где это уместно)

Старая привычка: Использовать конкатенацию строк для построения строк из переменных в области видимости:

```
const formatUserName = user => {
    return user.firstName + " " + user.lastName + " (" + user.handle + ")";
};
```

Новая привычка: Вероятно, это вопрос стиля, но вместо этого вы можете использовать шаблонный литерал:

```
const formatUserName = user => {
    return `${user.firstName} ${user.lastName} (${user.handle})`;
};
```

Некоторые люди рассматривают идею всегда использовать шаблонные литералы вместо строковых литералов. Однако это не совсем возможно сделать везде: есть еще пара мест, где разрешены только строковые литералы. Самыми масштабными будут имена

свойств в кавычках в инициализаторах (вы можете использовать имя вычисляемого свойства), модуль спецификатора для статических методов `import/export` (глава 13) и `"use strict"`. Но вы можете использовать шаблонные литералы практически в любом другом месте, где вы раньше обычно использовали строковый литерал.

Используйте помеченные функции и шаблонные литералы для DSL вместо пользовательских механизмов заполнения

Старая привычка: Создание собственных механизмов заполнения при создании DSL.

Новая привычка: В ситуациях, когда это имеет смысл, используйте помеченные функции и шаблонные литералы, используя преимущества оценки подстановок, предоставляемой шаблонами.

Используйте строковые итераторы

Старая привычка: Доступ к символам в строках по индексу:

```
const str = "testing";
for (let i = 0; i < str.length; ++i) {
    console.log(str[i]);
}
```

Новая привычка: Если вы хотите обращаться к строке как последовательности кодовых точек, а не кодовых единиц, рассмотрите возможность использования `codePointAt` или `for-of` или других функций, поддерживающих Юникод:

```
const str = "testing";
for (const ch of str) {
    console.log(ch);
}
```

11

Массивы

СОДЕРЖАНИЕ ГЛАВЫ

- Новые возможности массивов
- Типизированные массивы
- Объекты `DataView`

В этой главе вы узнаете о многих новых функциях массивов в ES2015+, включая функции для традиционных массивов и новых типизированных массивов. Прежде чем перейти к сути главы, краткое замечание о терминологии. Есть несколько слов, используемых для обозначения содержимого массивов, два самых популярных из которых — «элементы» (elements) и «записи» (entries). Хотя «элементы», вероятно, используются несколько чаще, чем «записи», в этой книге я использую термин «записи», чтобы избежать путаницы с элементами DOM — и потому, что это название метода для массивов `entries`, используемого для получения итератора для записей в массиве. Но вы очень часто будете встречать термин «элементы», в том числе в большинстве частей спецификации JavaScript. «Элемент» также используется в одном месте в стандартном API для типизированных массивов.

НОВЫЕ МЕТОДЫ МАССИВОВ

В ES2015, ES2016 и ES2019 добавили множество новых методов массива как для создания массивов, так и для доступа и изменения их содержимого.

Метод `Array.of`

Сигнатура:

```
arrayObject = Array.of(value0 [, value1 [, ...]])
```

Метод `Array.of` создает и возвращает массив, содержащий значения, которые вы передаете ему в качестве дискретных аргументов.

Например:

```
const a = Array.of("one", "two", "three");
console.log(a); // ["one", "two", "three"]
```

На первый взгляд это может показаться ненужным, поскольку вы могли бы просто использовать инициализатор массива:

```
const a = ["one", "two", "three"];
console.log(a); // ["one", "two", "three"]
```

Но `Array.of` полезен для *подклассов* массивов, поскольку у них нет литеральной формы:

```
class MyArray extends Array {
    niftyMethod() {
        // ...сделать что-нибудь изящное...
    }
}
const a = MyArray.of("one", "two", "three");
console.log(a instanceof MyArray); // истина
console.log(a); // ["one", "two", "three"]
```

В главе 4 вы узнали, что прототипом функции `myArray` является функция `Array`. Это означает, что `MyArray.of` наследует `Array`. Метод `Array.of` достаточно умен, чтобы посмотреть на `this`, с которым он был вызван, — и, если это конструктор, использовать этот конструктор для создания нового массива (если `this` не конструктор, `Array.of` по умолчанию использует `Array`). Так что без необходимости переопределять `of`, `MyArray.of` создает экземпляр `MyArray`.

Метод `Array.from`

Сигнатура:

```
arrayObject = Array.from(items [, mapFn[, thisArg]])
```

Как `Array.of`, метод `Array.from` создает массив на основе переданных вами аргументов. Но вместо того, чтобы принимать дискретные значения, он принимает любой итеративный или подобный массиву объект[62] в качестве своего первого аргумента и создает массив, используя значения из этого объекта, необязательно применяя к ним функцию сопоставления. Если передать методу `Array.from` значение `null` или `undefined`, он выдаст ошибку. Почти для всего остального, что не является итеративным или массивоподобным, он возвращает пустой массив.

В главе 10 показано, что строки можно повторять. Поэтому метод `Array.from` может создать массив из «символов» (кодовых точек Юникода) в строке:

[62] Массивоподобный объект — это любой объект со свойством `length`, чьи «записи» представляют собой свойства с именами в форме канонической строки целых чисел в диапазоне от `0` до `length - 1`.

```
const str = "123";
const a = Array.from(str);
console.log(a); // ["1", "2", "3"]
```

Перед вами пример построения массива из массивоподобного объекта:

```
const a = Array.from({length: 2, "0": "one", "1": "two"});
console.log(a); // ["one", "two"]
```

(Имена свойств `"0"` и `"1"` могут быть записаны с помощью числовых литералов, но в итоге они будут строками, поэтому здесь я использовал строки для выразительности.)

Метод `Array.from` принимает необязательный второй аргумент `mapFn` — функцию сопоставления, применяемую к каждому значению по мере его добавления в массив. Например, если вы хотите взять строку, содержащую цифры, и получить массив цифр в виде чисел, вы можете передать функцию преобразования в виде `mapFn`:

```
const str = "0123456789";
const a = Array.from(str, Number);
console.log(a); // [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Сигнатуры функции сопоставления: `mapFn(value, index)`, где `value` — это сопоставляемое значение, а `index` — это индекс нового значения в результирующем массиве. Это похоже на аргументы, получаемые обратным вызовом `Array.prototype.map`, но между `from` (при сопоставлении записей) и `map` существуют два серьезных различия:

- Обратный вызов сопоставления `Array.from` не получает третий аргумент, получаемый обратными вызовами `map`, — сопоставленный исходный объект. Были разговоры о включении исходного объекта. Но это не очень полезно, когда речь идет о итерируемом элементе, а не о массиве или массивоподобном объекте, поскольку индексация в нем не сработает. Поэтому команда, создававшая `from`, решила, что лучше не включать исходный объект.
- Метод `map` только вызывает свой обратный вызов для записей, существующих в исходном массиве (пропуская «исчезнувшие» записи в разреженные массивы), при работе с массивоподобным объектом. Метод `from` вызывает свой обратный вызов для каждого индекса в диапазоне от `0` до `length - 1` (включительно), даже если нет соответствующей записи в этом индексе.

Поскольку функция сопоставления получает индекс в качестве второго аргумента, вы должны быть уверены, что избежите классической ловушки pitfall с `map`, заключающейся в передаче функции, которая принимает несколько параметров и может запутаться из-за второго получаемого аргумента. Классический пример — использование функции `parseInt` с `map`. Эта проблема также относится к функции сопоставления `Array.from`:

```
const str = "987654321";
const a = Array.from(str, parseInt);
console.log(a); // [9, NaN, NaN, NaN, NaN, 4, 3, 2, 1]
```

Функция `parseInt` принимает два параметра, вторым из которых является радикс (основание системы исчисления — 2 для двоичной, 10 для десятичной и т. д.). Поэтому она запутывается, когда `map` передает ей индекс, как если бы это был радикс. Значения `NaN` появляются в этом примере, поскольку `parseInt` была вызвана либо с недопустимым параметром радикса, либо с радиксом, в который не вписывалась анализируемая цифра (первый случай сработал, ведь `parseInt` игнорирует его, если передать радикс со значением 0). Например, второй вызов не удался, потому что 1 — это недопустимое значение радикса. Третий вызов не удался, потому что 2 допустимый радикс (двоичный), но цифра «7» недопустима в двоичном формате. Как и в случае с `map`, ответ заключается в использовании стрелочной функции или другого средства обеспечения того, чтобы обратный вызов получал только соответствующие аргументы. В любом случае обычно требуется явное указание радикса при использовании `parseInt`, поэтому:

```
const str = "987654321";
const a = Array.from(str, digit => parseInt(digit, 10));
console.log(a); // [9, 8, 7, 6, 5, 4, 3, 2, 1]
```

Помимо создания массивов из итерируемых и массивоподобных элементов, другим вариантом использования метода `Array.from` будет построение *массивов диапазонов*: массивов, заполненных числами в заданном диапазоне. Например, чтобы создать массив из 100 записей со значениями от 0 до 99, вы можете использовать:

```
const a = Array.from({length: 100}, (_, index) => index);
// Или: const a = Array.from(Array(100), (_, index) => index);
console.log(a); // [0, 1, 2, 3, ... 99]
```

Выражения `{length: 100}` и `Array(100)` создают объекты со свойством `length`, равным `100` (первый — это простой объект, второй — разреженный массив). Метод `Array.from` выполняет обратный вызов сопоставления для каждого индекса в диапазоне от `0` до `99` (включительно), передавая значение `undefined` в качестве первого аргумента[63] и индекс в качестве второго. Поскольку обратный вызов возвращает индекс, результирующий массив содержит значения индекса в своих записях. Это можно обобщить в функцию `rangeArray`:

```
function rangeArray(start, end, step = 1) {
    return Array.from(
        {length: Math.floor(Math.abs(end - start) / Math.abs(step))},
        (_, i) => start + (i * step)
);
}

console.log(rangeArray(0, 5));      // [0, 1, 2, 3, 4]
console.log(rangeArray(6, 11));     // [6, 7, 8, 9, 10]
console.log(rangeArray(10, 20, 2)); // [10, 12, 14, 16, 18]
console.log(rangeArray(4, -1, -1)); // [4, 3, 2, 1, 0]
```

[63] Код в примере получает этот аргумент `undefined` в качестве параметра `_`. Один символ подчеркивания — допустимый идентификатор в JavaScript. Его часто выбирают в качестве имени для не используемого функцией параметра.

Последнее, но не менее важное: `Array.from` принимает третий параметр `thisArg`, определяющий значение `this` в вызовах функции `mapFn`. Итак, если у вас был объект (`example`) с методом (`method`) и вы хотели использовать этот метод в качестве обратного вызова, вы могли бы использовать выражение:

```
const array = Array.from(Array(100), example.method, example);
```

чтобы убедиться, что `this` в вызове `method` ссылается на объект `example`.

Метод `Array.prototype.keys`

Сигнатура:

```
keysIterator = theArray.keys()
```

Метод `keys` возвращает итератор для ключей массива. Ключи массива — это цифры от `0` до `length - 1`. Пример:

```
const a = ["one", "two", "three"];
for (const index of a.keys()) {
    console.log(index);
}
```

Этот код выводит `0`, затем `1`, затем `2`.

Есть несколько аспектов, касающихся метода `keys`, которые стоит отметить:

- Он возвращает итератор, а не массив.
- Несмотря на то что имена элементов массива технически являются строками (поскольку традиционные массивы на самом деле не являются массивами, как вы узнали из раздела «Порядок свойств» главы 5), значения, возвращаемые итератором метода `keys`, представлены числами.
- Все значения индекса в диапазоне `0 <= n < length` возвращаются итератором, даже если массив разрежен.
- Он не включает имена перечисляемых свойств, которые не являются индексами массива, если таковые имеются в массиве.

Сравните эти точки с выражением `Object.keys(someArray)`, который возвращает массив, включает индексы в виде строк, опускает индексы записей, отсутствующие в разреженном массиве, и включает имена других собственных (не унаследованных) перечисляемых свойств, если таковые есть в массиве.

Вот пример разреженного массива и того, как итератор `keys` включает индексы для пропусков:

```
const a = [, "x", , , "y"];
for (const index of a.keys()) {
    console.log(index, index in a ? "present": "absent");
   }
```

Массив a не содержит записей с индексами 0, 2 или 3; в нем есть только записи с индексами 1 и 4. Этот код выводит следующее:

```
0 "absent"
1 "present"
2 "absent"
3 "absent"
4 "present"
```

Метод Array.prototype.values

Сигнатура:

```
valuesIterator = theArray.values()
```

Метод values аналогичен методу keys, но возвращает итератор для значений в массиве, а не для ключей. Это точно такой же итератор, получаемый из самого массива. Пример:

```
const a = ["one", "two", "three"];
for (const index of a.values()) {
    console.log(index);
}
```

Этот код выводит "one", then "two", then "three".

Как и в случае с методом keys, включающим в себя индексы для отсутствующих записей в разреженных массивов, метод values включает значение undefined для недостающих элементов в массиве:

```
const a = [, "x",,, "y"];
for (const value of a.values()) {
    console.log(value);
}
```

Этот код выводит следующее:

```
undefined
"x"
undefined
undefined
"y"
```

В результате при получении undefined в качестве значения из итератора values неизвестно, означает ли это, что в массиве была запись со значением undefined или пропуск в разреженном масиве. Если требуется это знать, метод values не подойдет; возможно, лучше подойдет метод entries — см. следующий раздел.

Метод `Array.prototype.entries`

Сигнатура:

```
entriesIterator = theArray.entries()
```

Метод `entries` — это эффективное сочетание методов `keys` и `values`: он возвращает итератор для записей в массиве, где каждая предоставляемая им запись — обеспечение массива `[index, value]`. Пример:

```
const a = ["one", "two", "three"];
for (const entry of a.entries()) {
    console.log(entry);
}
```

Результат будет следующим:

```
[0, "one"]
[1, "two"]
[2, "three"]
```

При циклическом просмотре записей из `entries` обычно используется присваивание с деструктуризацией (см. главу 7), чтобы получить индекс и значение в виде дискретных переменных или констант:

```
const a = ["one", "two", "three"];
for (const [index, value] of a.entries()) {
    console.log(index, value);
}
```

Вывод результата:

```
0 "one"
1 "two"
2 "three"
```

Как и в случае с `keys` и `values`, итерация включает запись, даже если такая запись не существует в массиве, поскольку он разреженный. Но, в отличие от них, `values` может дифференцировать, получаете ли вы значение `undefined` из-за пропуска или фактическую запись, путем проверки существования ключа (`index`) в массиве:

```
const a = [, undefined,, , "y"];
for (const [index, value] of a.entries()) {
    console.log(index, value, index in a ? "present": "absent");
}
```

Вывод результата:

```
0 undefined "absent"
1 undefined "present"
2 undefined "absent"
```

```
3 undefined "absent"
4 "y" "present"
```

Обратите внимание, что второе значение `undefined` предназначено для существующей записи, в то время как все остальные предназначены для пропусков.

Если у потребителя итератора нет доступа к исходному массиву, то он не может узнать, значение `undefined` в массиве `[index, value]` появилось из-за пропуска или от действительного значения `undefined`. Массивы `[index, value]` из метода `entries` всегда содержат как индекс, так и значение, даже для разреженных записей.

Метод `Array.prototype.copyWithin`

Сигнатура:

```
obj = theArray.copyWithin(target, start [, end])
```

Метод `copyWithin` копирует записи из одной части массива в другую часть массива, устраняя любые потенциальные проблемы с перекрытием и не увеличивая длину массива. Укажите целевой индекс `target` (куда скопировать записи), начальный индекс `start` (с чего начать копирование) — и при необходимости конечный индекс `end` (где остановиться, эксклюзивное значение; по умолчанию это длина массива). Метод возвращает массив, для которого он был вызван (грубо говоря, подробности см. во врезке «Возвращаемое copyWithin значение»). Если какой-либо из аргументов отрицательный, он используется как смещение от конца массива. Например, `start` со значением `-2` в массиве с длиной `6` воспринимается, как `start` со значением `4`, потому что `6-2` равно `4`. Параметр `start` не определен как необязательный, но если не задать ему значение, его фактическим значением станет `0` из-за того, что спецификация обрабатывает интерпретируемое значение `start`.

ВОЗВРАЩАЕМОЕ `COPYWITHIN` ЗНАЧЕНИЕ

При описании метода `copyWithin` я сказал, что он «…возвращает массив, для которого он был вызван». Строго говоря, это неправда. Он принимает значение `this`, с которым был вызван, преобразует его в объект, если он еще не был объектом, выполняет свою работу, а затем возвращает этот объект. В обычном случае вы вызываете его для массива, поэтому `this` ссылается на этот массив, и этот же массив возвращается. Но если вы использовали `call`, `apply` или аналогичные выражения для вызова метода с массивоподобным объектом в качестве значения `this`, метод вернет этот объект, а не массив. Если вы вызовете метод с примитивом в `this`, он преобразует этот примитив в объект и вернет этот объект (если не выдаст ошибку, потому что не смог выполнить запрошенное вами копирование).

Вот пример копирования внутри массива записей, расположенных ближе к концу, в позиции, находящиеся ближе к началу:

```
const a = ["a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k"];
console.log("before", a);
a.copyWithin(2, 8);
console.log("after ", a);
```

Результат будет следующим:

```
before ["a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k"]
after  ["a", "b", "i", "j", "k", "f", "g", "h", "i", "j", "k"]
```

Вызов скопировал записи внутри массива, начиная с индекса 8 и до конца массива (`"i"`, `"j"`, `"k"`), записав их, начиная с индекса 2 (рисунок 11-1).

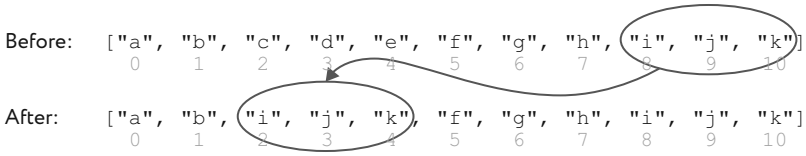


РИСУНОК 11-1

Обратите внимание, что он *переписал* предыдущие записи в индексах 2–4 копиями, а не *вставлял* копии.

Вот пример копирования более ранних записей в более позднее местоположение:

```
const a = ["a", "b", "c", "d", "e", "f", "g"];
console.log("before", a);
a.copyWithin(4, 2);
console.log("after ", a);
```

Вывод результата:

```
before ["a", "b", "c", "d", "e", "f", "g"]
after  ["a", "b", "c", "d", "c", "d", "e"]
```

Смотрим рисунок 11-2.

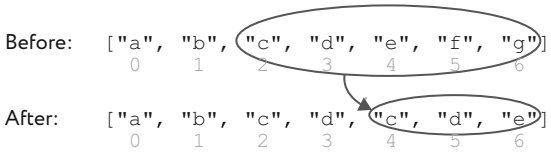


РИСУНОК 11-2

В этом примере следует отметить две вещи:

- Копирование — это не наивный цикл `for`, движущийся вперед. Если бы оно было таким, операция попала бы сама на себя, копируя с `"c"` по `"e"` и позже, используя скопированные `"c"` для последней записи, вместо (верного) использования `"e"`. Вместо этого `copyWithin` гарантирует, что копируемые записи будут такими же, как до начала операции.
- Копирование остановится без расширения массива. В коде нет конечной точки (третьего аргумента), поэтому конечная точка по умолчанию равна длине массива. Операция не копировала `"f"` и `"g"`, потому что это сделало бы массив длиннее. Вот почему на рисунке 11–2, хотя пять исходных записей обведены кружком (поскольку в вызове указаны записи с индексами со 2 по 6), только три были скопированы до того, как метод попал в конец массива.

Это может показаться очень специфической операцией, и так оно и есть. Но это обычная операция в графических приложениях. Она включена в `Array` в первую очередь потому, что она включена в *типизированные массивы* (о которых вы узнаете позже в этой главе), использующиеся (среди прочего) для графических операций. Комитет TC39 принял решение сохранить API массивов и типизированных массивов как можно более похожими. Поскольку у типизированных массивов есть метод `copyWithin`, его оставили и в массивах тоже.

Одна интересная особенность `copyWithin`, которая не сводится к типизированным массивам, заключается в том, как он обрабатывает разреженные массивы (позже вы узнаете, типизированные массивы никогда не бывают разреженными). Метод `copyWithin` «копирует» пропуски, удаляя запись в том месте, в которое должна быть скопирована отсутствующая запись:

```
function arrayString(a) {
    return Array.from(a.keys(), key => {
        return key in a ? a[key]: "*gap*";
    }).join(", ");
}
const a = ["a", "b", "c", "d",, "f", "g"];
console.log("before", arrayString(a));
a.copyWithin(1, 3);
console.log("after ", arrayString(a));
```

Обратите внимание на пропуск (*gap*) в этом массиве, в индексе 4 нет записи (`"e"` отсутствует). Результат будет следующим:

```
before a, b, c, d, *gap*, f, g
after  a, d, *gap*, f, g, f, g
```

Пропуск, находящийся в индексе 4, был скопирован в индекс 2 (а затем пропуск в индексе 4 был заполнен, когда в него было скопировано значение из индекса 6).

Метод `Array.prototype.find`

Сигнатура:

```
result = theArray.find(predicateFn [, thisArg])
```

Метод `find` предназначен для нахождения значения первой соответствующей записи в массиве с использованием функции предиката. Он вызывает функцию предиката для каждой записи в массиве, при необходимости используя полученный параметр `thisArg` в качестве значения `this` для вызовов, останавливающих и возвращающих значение из первой записи, для которого предикат возвращает истинное значение, или возвращающих `undefined`, если метод выходит за пределы диапазона записей. Пример:

```
const a = [1, 2, 3, 4, 5, 6];
const firstEven = a.find(value => value% 2 == 0);
console.log(firstEven); // 2
```

Функции предиката вызывается с тремя аргументами (аналогичными тем, что используются в `forEach`, `map`, `some` и т. д.): значение для этого вызова, его индекс и ссылка на объект, для которого вызван `find` (обычно массив). Метод `find` останавливается в первый раз, когда предикат возвращает истинное значение, не затрагивая последующие записи в массиве.

При вызове метода `find` для пустого массива он всегда возвращает значение `undefined`, поскольку `find` выходит за пределы диапазона записей до того, как предикат возвращает истинное значение (так как предикат никогда не вызывается):

```
const x = [].find(value => true);
console.log(x);          // undefined
```

Изменение массива во время операции `find`, как правило, не лучшая практика. Но, если вы это сделаете, результаты будут четко определены: диапазон затрагиваемых записей определяется до того, как `find` начнет свой цикл. Значение, используемое для записи, — это ее значение на момент, когда запись обрабатывается (они не сохраняются заранее). Это означает, что:

- Если добавить новые записи в конец, они не будут затронуты.
- Если изменить уже использованную запись, она больше не будет обрабатываться.
- Если изменить еще не затронутую запись, ее новое значение будет использоваться при ее обработке.
- Если удалить записи из массива, уменьшив его длину, будут обработаны пропуски в конце с обычным значением для пропусков в массивах — `undefined`.

Для примера рассмотрим следующий код:

```
const a = ["one", "two", "three"];
const x = a.find((value, index) => {
    console.log(`Visiting index ${index}: ${value}`);
    if (index === 0) {
        a[2] = a[2].toUpperCase();
```

```
    } else if (index === 1) {
        a.push("four");
    }
    return value === "four";
});
console.log(x);
```

Результат будет следующим:

```
Visiting index 0: one
Visiting index 1: two
Visiting index 2: THREE
undefined
```

Видно, что запись `"four"` не была обработана, потому что она вне диапазона обрабатываемых к началу работы `find` записей. Но мы видим запись `"THREE"` в верхнем регистре, поскольку она изменилась до обработки. Метод `find` возвращает значение `undefined` (что показано при помощи `console.log(x)` в конце), потому что предикат не возвращает истинное значение, поскольку запись `"four"` не обрабатывалась.

Метод `Array.prototype.findIndex`

Сигнатура:

```
result = theArray.findIndex(predicateFn [, thisArg])
```

Метод `findIndex` точно такой же, как `find` — за исключением того, что `findIndex` возвращает *индекс* записи, для которой функция предикат вернула истинное значение, или значение `-1`, при выходе за диапазон записей.

Пример:

```
const a = [1, 2, 3, 4, 5, 6];
const firstEven = a.findIndex(value => value% 2 == 0);
console.log(firstEven); // 1 - первое четное значение - это число 2
                        // с индексом 1
```

Все остальное то же самое. Функция предиката получает те же три аргумента, диапазон записей, обрабатываемых `findIndex`, определяется заранее, обрабатываемые значения остаются такими, какие они были при их обработке (не сохраняются заранее), если вы уменьшаете длину массива, пропуски обрабатываются в конце, а вызов `findIndex` для пустого массива всегда возвращает значение `-1`.

Метод `Array.prototype.fill`

Сигнатура:

```
obj = theArray.fill(value[, s tart [, end]])
```

Метод `fill` заполняет массив (или массивоподобный объект) и вызывается для полученного значения `value`. При необходимости метод заполняет массив только

в пределах, определяемых индексами, начала `start` (по умолчанию `0`) и конца `end` (эксклюзивный, по умолчанию `length`). Он возвращает массив, для которого был вызван (фактически его возвращаемое значение похоже на метод `copyWithin`; см. врезку ранее).

Если значение `start` или `end` отрицательное, оно используется как смещение от конца массива.

Пример:

```
const a = Array(5).fill(42);
console.log(a); // [42, 42, 42, 42, 42]
```

В этом примере `Array(5)` возвращает разреженный массив со свойством `length`, равным 5, но без записей; затем метод `fill` заполняет массив с заданным значением (`42`).

Общая ловушка Pitfall: Использование объекта в качестве значения заполнения

Предоставляемое значение — это просто значение. В конечном счете в массиве это очень похоже не на вызов `Array.from`, а на следующее:

```
const a = Array(5);
const value = 42;
for (let i = 0; i < a.length; ++i) {
    a[i] = value;
}
console.log(a); // [42, 42, 42, 42, 42]
```

Имея это в виду, как вы думаете, что выводит следующий код?

```
const a = Array(2).fill({});
a[0].name = "Joe";
a[1].name = "Bob";
console.log(a[0].name);
```

Если вы сказали `"Bob"`, вы молодец! Выражение `Array(2).fill({})` помещает *один и тот же* объект в обе записи массива, он не заполняет массив кучей различных объектов. Это означает, что `a[0]` и `a[1]` относятся к одному и тому же объекту. Выражение `a[0].name = "Joe"` устанавливает свойство `name` объекта в значение `"Joe"`, а затем `a[1].name = "Bob"` перезаписывает его со значением `"Bob"`.

Если вы хотите поместить отдельный объект в каждую запись массива, метод `Array.fill`, вероятно, не лучший выбор. Лучше применить сопоставляющий обратный вызов `Array.from`:

```
const a = Array.from({length: 2}, () => ({}));
a[0].name = "Joe";
a[1].name = "Bob";
console.log(a[0].name); // Joe
```

Но если хочется непременно использовать метод `Array.fill`, стоит сначала заполнить массив, а затем использовать метод `map`:

```
const a = Array(2).fill().map(() => ({}));
a[0].name = "Joe";
a[1].name = "Bob";
console.log(a[0].name); // Joe
```

Здесь требуется вызов `fill` (заполняет массив значениями `undefined`), поскольку `map` не обрабатывает несуществующие записи, а `Array(2)` создает массив со свойством `length = 2`, но без записей.

Метод `Array.prototype.includes`

Сигнатура:

```
result = theArray.includes(value [, start])
```

Метод `includes` (добавлен в ES2016) возвращает значение `true`, если полученное значение присутствует в массиве в соответствии с алгоритмом SameValueZero, определенном в спецификации, или значение `false` в противном случае. При необходимости он запускает поиск по предоставленному индексу `start`; если значение `start` отрицательное, оно используется как смещение от конца массива.

Примеры:

```
const a = ["one", "two", "three"];
console.log(a.includes("two"));     // истина
console.log(a.includes("four"));    // ложь
console.log(a.includes("one", 2));  // ложь, "one" находится до индекса 2
```

Распространенное заблуждение, что `includes(value)` — это просто более короткий способ написать `indexOf(value) !== -1`, не совсем корректно. Метод `indexOf` использует алгоритм сравнения строгих равенств (как ===) для проверки значения, но операция SameValueZero (новая в ES2015) отличается от строгого равенства в том, как она справляется с `NaN`. При строгом равенстве значени `NaN` не равно самому себе, при SameValueZero равно:

```
const a = [NaN];
console.log(a.indexOf(NaN) !== -1); // ложь
console.log(a.includes(NaN));       // истина
```

Во всем, кроме `NaN`, операция SameValueZero работает как строгие равенства. Это означает, что отрицательные и положительный ноль равны, следовательно, `[-0].includes(0)` возвращает значение `true`.

Метод `Array.prototype.flat`

Добавленный в ES2019 метод `flat` создает новый «выровненный» массив. Метод получает каждое значение из исходного массива и, если значение представляет собой массив, берет *его* значения для помещения в результат, а не в сам массив:

```
const original = [
    [1, 2, 3],
```

```
    4,
    5,
    [6, 7, 8]
];
const flattened = original.flat();
console.log(flattened);
// => [1, 2, 3, 4, 5, 6, 7, 8]
```

Это в основном заменяет распространенную идиому для выравнивания массивов с помощью `concat`:

```
const flattened = [].concat.apply([], original);
// или
const flattened = Array.prototype.concat.apply([], original);
```

и представляет собой более эффективный способ (хотя обычно это не имеет значения), поскольку не создает и не выбрасывает временные массивы. Однако метод `flat` не проверяет значение `Symbol.isConcatSpreadable` (см. главу 17), так что, если у вас есть массивоподобные объекты с параметром `Symbol.isConcatSpreadable` в значении истины, функция `concat` будет расширять их, а `flat` — нет. Метод `flat` расширяет только фактические массивы. (Если требуется такое поведение, можно продолжать использовать `concat`, возможно, с нотацией расширения: `const flattened = [].concat(...original);`.)

По умолчанию метод `flat` выравнивает только один уровень (как `concat`), поэтому массивы, вложенные за пределы одного уровня, не выравниваются:

```
const original = [
    [1, 2, 3],
    [
        [4, 5, 6],
        [7, 8, 9]
]
];
const flattened = original.flat();
console.log(flattened);
// => [1, 2, 3, [4, 5, 6], [7, 8, 9]];
```

Вы можете задать дополнительный аргумент *глубины*, чтобы указать `flat` выполнять рекурсивное выравнивание до заданной глубины: 1 = только один уровень (по умолчанию), 2 = два уровня и т. д. Вы можете использовать значение `Infinity` для полного выравнивания структуры независимо от ее глубины:

```
const original = [
    "a",
    [
        "b",
        "c",
        [
                "d",
            "e",
            [
                "f",
                "g",
```

```
                [
                    "h",
                    "i"
                ],
            ],
        ],
    ],
    "j"
];
const flattened = original.flat(Infinity);
console.log(flattened);
// => ["a", "b", "c", "d", "e", "f", "g", "h", "i", "j"]
```

Метод `Array.prototype.flatMap`

Также добавленный в ES2019, метод `flatMap` похож на `flat` — за исключением того, что он передает каждое значение через функцию сопоставления (маппинга) перед выравниванием результата и выравнивает только один уровень:

```
const original = [1, 2, 3, 4];
const flattened = original.flatMap(e => e === 3 ? ["3a", "3b", "3c"]: e);
console.log(flattened);
// => [1, 2, "3a", "3b", "3c", 4]
```

Конечный результат — это именно то, что вы получили бы, вызвав `map`, а затем вызвав `flat` для результата (только с одним уровнем выравнивания по умолчанию):

```
const original = [1, 2, 3, 4];
const flattened = original.map(e => e === 3 ? ["3a", "3b", "3c"]: e).flat();
console.log(flattened);
// => [1, 2, "3a", "3b", "3c", 4]
```

Единственное функциональное отличие заключается в том, что `flatMap` выполняет это всего за один проход по массиву, а не за два.

ИТЕРАЦИЯ, РАСШИРЕНИЕ, ДЕСТРУКТУРИЗАЦИЯ

В ES2015 массивы получили некоторые другие функции, они рассматриваются в других местах книги:

- Массивы стали итеративными (см. главу 6).
- Литералы массива могут включать в себя нотацию расширения (см. главу 6).
- Массивы участвуют в деструктуризации (см. главу 7).

СТАБИЛЬНАЯ СОРТИРОВКА МАССИВА

До ES2019 алгоритм сортировки, используемый методом `Array.prototype.sort`, определялся как «не обязательно стабильный». Это означает, что, если две записи считаются равными, их относительные позиции в результирующем массиве все равно могут поменяться местами. (Это также относится к типизированным массивам, таким как

объект `Int32Array`.) Начиная с ES2019 метод `sort` требуется для реализации стабильной сортировки (как для обычных, так и для типизированных массивов).

Например, со старым определением этот код сортирует массив, игнорируя чувствительность к регистру:

```
const a = ["b", "B", "a", "A", "c", "C"];
a.sort((left, right) => left.toLowerCase().localeCompare(right.
toLowerCase()));
console.log(a);
```

было разрешено получить любой из нескольких различных возможных результатов:

```
["a", "A", "b", "B", "c", "C"] - Равные записи не менялись местами
                                 (стабильный)
["A", "a", "B", "b", "C", "c"] - Все равные записи поменялись местами
                                 (нестабильный)
["a", "A", "b", "B", "C", "c"] - Некоторые равные записи поменялись
                                 местами (нестабильный)
["a", "A", "B", "b", "c", "C"] - "
["a", "A", "B", "b", "C", "c"] - "
["A", "a", "b", "B", "c", "C"] - "
["A", "a", "b", "B", "C", "c"] - "
["A", "a", "B", "b", "c", "C"] - "
```

Начиная с ES2019 реализации обязаны регулярно производить только первый результат: значение `"a"` должно быть перед `"A"` (поскольку оно было до сортировки), `"b"` перед `"B"` и `"c"` перед `"C"`.

ТИПИЗИРОВАННЫЕ МАССИВЫ

В этом разделе вы узнаете о *типизированных массивах* — истинных массивах примитивных числовых значений, добавленных в JavaScript в ES2015.

КРАТКИЙ ОБЗОР

Традиционные «массивы» JavaScript, как известно, на самом деле не массивы в обычном определении информатики, то есть в виде непрерывного (все подряд) блока памяти, разделенного на блоки фиксированного размера. Вместо этого традиционные массивы JavaScript — это просто объекты со специальной обработкой имен свойств, соответствующих определению спецификации «индекс массива»[64], со специальным свойством `length`, с литеральным обозначением с использованием квадратных скобок и с методами, унаследованными от `Array.prototype`. Тем не менее движки JavaScript могут свободно оптимизировать там, где оптимизация не противоречит спецификации (и они это делают).

Традиционные массивы JavaScript мощные и полезные, но иногда требуется настоящий массив (особенно при чтении/записи файлов или взаимодействии с графическими

[64] Строка в канонической числовой форме, которая преобразуется в целое число в диапазоне $0 <= n < 2^{32}-1$.

или математическими API). По этой причине в ES2015 добавили в язык истинные массивы в виде *типизированных массивов*.

Типизированные массивы похожи на традиционные массивы JavaScript, за исключением того, что:

- Их входные значения всегда представлены примитивными числовыми значениями: 8-битные целые числа, 32-битные числа с плавающей точкой и т. д.
- Все значения в типизированном массиве относятся к одному и тому же типу, который зависит от типа массива: `Uint8Array`, `Float32Array`, и т. д.
- Их длина зафиксирована: как только массив создан, его длину изменить невозможно[65].
- Их значения хранятся в непрерывном буфере памяти в указанном двоичном формате.
- Типизированные массивы не могут быть разреженными (то есть содержать пропуски в середине), а традиционные массивы могут. Например, с помощью традиционного массива вы можете сделать это:

  ```
  const a = [];
  a[9] = "nine";
  ```

  …и массив будет содержать только *одну* запись с индексом 9, без записей с индексами от 0 до 8:

  ```
  console.log(9 in a); // истина
  console.log(8 in a); // ложь
  console.log(7 in a); // ложь (и т. д.)
  ```

  Это невозможно с типизированным массивом.
- Типизированные массивы могут совместно использовать память (их базовый буфер данных) с другими типизированными массивами даже разных типов.
- Буфер данных типизированного массива может *передаваться* или даже *использоваться совместно* между потоками (например, с веб-воркерами (web worker) в браузере или рабочими потоками в Node.js); вы узнаете об этом в главе 16.
- При получении или установке значения элемента типизированного массива всегда используется какая-либо форма преобразования (кроме случаев использования чисел с объектом `Float64Array`, поскольку числа JavaScript — это данные типа `Float64`).

В Таблице 11-1 перечислены одиннадцать типов типизированных массивов (на основе таблицы «Конструкторы TypedArray» в спецификации). В ней указано имя типа (то есть глобальное имя его функции-конструктора), концептуальное имя его типа значения, сколько байт памяти занимает запись в этом массиве, абстрактную операцию спецификации, преобразующую значения в тип, и (краткое) описание.

[65] Вы узнаете об одном *незначительном* предостережении по этому пункту в главе 16.

Таблица 11-1. Одиннадцать типизированных массивов[66]

Имя	Тип значения	Размер записи	Операция преобразования	Описание
Int8Array	Int8	1	ToInt8	8-битное целое со знаком в дополнительном коде
Uint8Array	Uint8	1	ToUint8	8-битное беззнаковое целое
Uint8ClampedArray	Uint8C	1	ToUint8Clamp	8-битное беззнаковое целое (сжатое преобразование)
Int16Array	Int16	2	ToInt16	16-битное целое со знаком в дополнительном коде
Uint16Array	Uint16	2	ToUint16	16-битное беззнаковое целое
Int32Array	Int32	4		32-битное целое со знаком в дополнительном коде
Uint32Array	Uint32	4		32-битное беззнаковое целое
Float32Array	Float32	4		32-разрядное двоичное с плавающей точкой из IEEE-754
Float64Array	Float64/ тип «number»	8		64-разрядное двоичное с плавающей точкой из IEEE-754
BigInt64Array	BigInt64	8		Новое в ES2020, см. Главу 17
BigUint64Array	BigUint64			

Однако типизированные массивы — это не просто необработанные блоки данных с указателем на них: как и в случае с традиционными массивами, они представляют собой объекты. Все обычные операции с объектами работают с типизированными массивами. У них есть прототипы, методы и свойство `length`; можно поместить в них не вводимые

[66] Числа преобразуются в формат Float32 с использованием правил спецификации IEEE-754-2008 для преобразования 64-разрядных двоичных значений в 32-разрядные двоичные значения с использованием режима «округление до ближайшего, четного или четного».

свойства, точно так же как в традиционные массивы JavaScript; они могут быть итерируемыми и т. д.

Давайте посмотрим на них в действии.

Основное использование

У типизированных массивов нет литеральной формы. Можно создать типизированный массив, вызвав его конструктор или используя методы его конструктора `of` или `from`.

У конструкторов для каждого вида типизированного массива (`Int8Array`, `Uint32Array` и т. д.) одинаковые доступные формы. Вот список конструкторов, использующих `%TypedArray%` в качестве заполнителя для различных конкретных типов (`Int8Array` и т. д.), как это описано в спецификации:

- *`new %TypedArray%()`*: Создает массив со свойством `length` в значении `0`.
- *`new %TypedArray%(length)`*: Создает массив с количеством записей, указанным в `length`; каждая запись изначально получает значение `all-bits-off` (ноль).
- *`new %TypedArray%(object)`*: Создает массив путем копирования значений из полученного объекта с помощью итератора объекта, если он у него есть, или путем обработки объекта как массива и использования свойств массива `length` и индекс для чтения его содержимого.
- *`new %TypedArray%(typedArray)`*: Создает массив путем копирования значений из заданного типизированного массива. Эта операция не проходит через итератор типизированного массива, для повышения эффективности он работает непосредственно с нижележащими буферами. (Когда типы массивов одинаковы, это может быть чрезвычайно эффективная копия в памяти.)
- *`new %TypedArray%(buffer[, start[, length]])`*: Создает массив, используя данный буфер (это описано в более позднем разделе `ArrayBuffer`).

Если вы создаете массив, указывая длину, для входных значений устанавливается значение «все биты на ноль» — значение 0 для всех типов типизированных массивов:

```
const a1 = new Int8Array(3);
console.log(a1); // Int8Array(3): [0, 0, 0]
```

Когда вы присваиваете значения элементам типизированного массива (во время построения или позже, предоставляя объект или другой типизированный массив), механизм JavaScript передает их через функцию преобразования, указанную в Таблице 11-1. Взгляните на пример:

```
const a1 = new Int8Array(3);
a1[0] = 1;
a1[1] = "2";     // Обратите внимание на строку
a1[2] = 3;
console.log(a1); // Int8Array(3): [1, 2, 3] - обратите внимание, что 2 - число
```

В этом примере были преобразованы все три значения, но наиболее очевидно это со значением `"2"`, которое было преобразовано из строки в 8-битное целое число. (`1` и `3` также были преобразованы, по крайней мере теоретически, из стандартного типа

чисел JavaScript — двоичных чисел с плавающей точкой двойной точности IEEE-754 — в 8-битные целые числа.)

Аналогично вот примеры использования `of` и `from`:

```
// Использование `of`:
const a2 = Int8Array.of(1, 2, "3");
console.log(a2); // Int8Array(3): [1, 2, 3] - значение "3" конвертировано в 3
// Использование `from` с массивоподобным объектом:
const a3 = Int8Array.from({length: 3, 0: 1, 1: "2"});
console.log(a3); // Int8Array(3): [1, 2, 0] - значение undefined
                 // конвертировано в 0
// Использование `from` с массивом:
const a4 = Int8Array.from([1, 2, 3]);
console.log(a4); // Int8Array(3): [1, 2, 3]
```

В этом примере при использовании `a3`, поскольку в миссивоподобном объекте нет значения для свойства `"2"`, даже при том, что длина массивоподобного объекта равна `3`, метод `from` получает значение `undefined` при получении свойства `"2"`, а затем конвертирует его в `0`, используя операцию преобразования для `Int8Array`. (И опять же, все эти значения были преобразованы в теории, но это наиболее наглядно при использовании строчных значений и `undefined`.)

Скоро вы увидите более сложный пример, узнав о `ArrayBuffer`. Сначала давайте кратко рассмотрим преобразования значений более подробно.

Подробнее о преобразовании значений

Давайте рассмотрим преобразование значений более подробно.

Операции преобразования, выполняемые при присвоении значений элементам массива, всегда приводят к получению значения, а не к возникновению исключения, если предоставленное значение не может быть преобразовано. Например, присваивание строки `"foo"` записи в `Int8Array` устанавливает, что значение записи равно `0`, вместо того чтобы выбрасывать ошибку. В этом разделе вы узнаете, как выполняется преобразование.

Для массивов с плавающей точкой это довольно просто:

1. При присвоении записи типа `Float64` значение сначала преобразуется в стандартное число JavaScript обычным способом, если это необходимо, а затем сохраняется как есть, поскольку стандартные числа JavaScript представляют собой значения `Float64`.
2. При присвоении записи типа `Float32` значение сначала преобразуется в стандартное число JavaScript, если это необходимо, а затем преобразуется в формат `Float32` с использованием режима округления спецификации IEEE-754–2008 — «округление до ближайшего, четного или четного».

При присвоении записям с целочисленным типом этот процесс сложнее:

1. Входное значение преобразуется в стандартное число JavaScript, если оно еще не является таковым.

2. Если результат шага 1 `NaN` — положительный или отрицательный `0` или минус бесконечность, то используется значение `0` и следующие шаги пропускаются. Если результат шага 1 — положительная бесконечность, значение `0` используется, если только массив не относится к виду `Uint8ClampedArray`. В этом случае используется значение `255`; в обоих случаях следующие шаги пропускаются.
3. Дробные значения сокращаются до нуля.
4. Если массив — это `Uint8ClampedArray`, для определения входного значения используется проверка диапазона:
   - Если значение из шага 3 меньше `0`, используется `0`; если оно больше, чем `255`, используется `255`.
   - В противном случае значение используется как есть.
5. Если массив не сжат:
   - *Деление по модулю* (не *остаток*; см. врезку «Деление по модулю в сравнении с остатком») использует 2^n в качестве делителя должна обеспечить, чтобы значение находилось в общем диапазоне без знака для размера целочисленной записи (где ^{n} — количество битов, например 2^8 у `Int8Array`).
   - Для массивов со знаковыми записями, если значение, полученное в результате *деления по модулю* (которое всегда положительно или равно нулю, поскольку 2^n является положительным), находится за пределами положительного диапазона записи, из него вычитается 2^n и используется результат.

ДЕЛЕНИЕ ПО МОДУЛЮ В СРАВНЕНИИ С ОСТАТКОМ

Преобразование целочисленных значений типизированного массива использует операцию деления по модулю (modulo) на одном из шагов преобразования. Это не та же операция, что и оператор остатка (`%`), хотя этот оператор обычно (но неправильно) называют «оператором modulo». Деление по модулю и остаток — это одна и та же операция, когда оба операнда положительны, но в остальном они могут отличаться, в зависимости от того, какой вариант модуля вы используете. Спецификация определяет свою абстрактную операцию по модулю следующим образом:

> Запись «x по модулю y» (значение y должно быть конечным и не нулевым) вычисляет значение k с тем же знаком, что и y (или ноль), как abs(k) < abs(y) и x – k = q × y для некоторого значения q.
>
> Поскольку операции по модулю, применяемые по отношению к типизированному массиву значений, всегда используют положительное число y (например, y = 28, что равно 256 для `Uint8Array`), в результате этих операций всегда будут положительные числа в пределах полного положительного диапазона возможных значений для размера записи. Для массивов со знаком последующая операция преобразует значения, выходящие за пределы положительного диапазона типа, в отрицательные числа.

Например, присвоенное записи в `Int8Array` число `25.4` становится равным `25`, `-25.4` становится `-25`:

```
const a = new Int8Array(1);
a[0] = 25.4;
console.log(a[0]); // 25
a[0] = -25.4;
console.log(a[0]); // -
```

Для типов без знака (отличных от `Uint8ClampedArray`) преобразование отрицательных значений может показаться неожиданным. Рассмотрим:

```
const a = new Uint8Array(1);
a[0] = -25.4;
console.log(a[0]); // 231
```

Как значение `-25.4` превратилось в `231`?!

Первые части просты: `-25.4` уже представляет собой число, поэтому нет нужды преобразовывать его, а когда происходит сокращение до нуля, вы получаете `-25`. Теперь пришло время для этой операции по модулю (см. врезку ранее). Одним из способов вычисления этого модуля будет следующий фрагмент кода, где `value` — это значение (`-25`) и `max` — это 2^n (2^8, что равно `256` в этом 8-битном примере):

```
const value = -25;
const max = 256;
const negative = value < 0;
const remainder = Math.abs(value) % max;
const result = negative ? max - remainder: remainder;
```

Если проследить код до конца, то в итоге получится `231`. Но есть другой, более прямой способ добраться до результата. 8-битное целое значение *со знаком* `-25`, в дополнительном коде, используемое со знаковыми значениями в типизированных массивах, имеет тот же битовый шаблон, что и 8-битное целое значение *без знака* `231`. В обоих случаях паттерн — `11100111`. Спецификация просто использует математику, а не битовые шаблоны, чтобы добраться до результата.

Объект `ArrayBuffer`: Хранилище для типизированных массивов

Все типизированные массивы используют объект `ArrayBuffer` для хранения своих значений. `ArrayBuffer` — это объект со связанным непрерывным блоком данных фиксированного размера, заданного в байтах. Типизированные массивы считывают и записывают данные в своей блок данных `ArrayBuffer` в зависимости от типа предоставляемых массивом данных: `Int8Array` обращается к байтам в буфере и использует биты как 8-разрядные целые числа со знаком (в дополнительном коде); `Uint16Array` обращается к парам байтов в буфере, используя их как беззнаковые 16-битные целые числа, и т. д. Прямой доступ к данным в буфере осуществить не получится, вы можете получить к ним доступ только через типизированный массив или вид `DataView` (о котором вы узнаете немного позже).

Код в предыдущих примерах создавал типизированные массивы напрямую, без явного создания объекта `ArrayBuffer`. При это будет создан буфер соответствующего размера. Так, например, выражение `new Int8Array(5)` создает пятибайтовый буфер; выражение `new Uint32Array(5)` создает 20-байтовый буфер (поскольку каждая из пяти записей занимает четыре байта памяти).

Получить доступ к буферу, прикрепленному к типизированному массиву, можно с помощью свойства `buffer` массива:

```
const a = new Int32Array(5);
console.log(a. buffer.byteLength); // 20 (байт)
console.log(a.length);              // 5 (записей, каждая из которых
занимает четыре байта)
```

Вы также можете явно создать объект `ArrayBuffer`, а затем передать его в конструктор типизированного массива при создании массива:

```
const buf = new ArrayBuffer(20);
const a = new Int32Array(buf);
console.log(buf.byteLength); // 20 (байт)
console.log(a.length);       // 5 (записей, каждая из которых занимает
четыре байта)
```

Размер, который вы задаете конструктору `ArrayBuffer`, указывается в байтах.

При попытке использовать буфер неподходящего для создаваемого типизированного массива размера, вы получите сообщение об ошибке:

```
const buf = new ArrayBuffer(18);
const a = new Int32Array(buf); // RangeError: длина Int32Array в байтах
                               // должна быть кратна 4
```

(Позже в этой главе вы увидите, как использовать только часть буфера.)

Для помощи в создании буферов нужного размера конструкторы типизированных массивов содержат свойство, доступное для использования при создании буфера: `BYTES_PER_ELEMENT`[67]. Итак, чтобы создать буфер для `Int32Array` с пятью записями, следует сделать так:

```
const buf = new ArrayBuffer(Int32Array.BYTES_PER_ELEMENT * 5);
```

Это стоит делать только если есть какая-то причина для создания буфера отдельно. В противном случае просто вызовите конструктор типизированного массива и используйте свойство результирующего массива `buffer`, если вам нужен доступ к буферу.

Теперь, когда вы увидели объект `ArrayBuffer`, давайте рассмотрим более реальный пример использования типизированных массивов — считывание файла в веб-браузере и проверка, относится ли он к файлам портативной сетевой графики (PNG), см. Листинг 11-1. Вы можете запустить это локально, используя код из Листинга 11-1

[67] Помните, я говорил в начале этой главы, что «элемент» используется в одном месте в стандартной библиотеке JavaScript по отношению к типизированным массивам? Это то самое место — название свойства.

(`read-file-as-arraybuffer.js`) вместе с файлом **read-file-as-arraybuffer.html**; оба доступны в разделе загрузки для этой главы.

Листинг 11-1: Считывание файла в веб-браузере и проверка, относится ли он к PNG — read-file-as-arraybuffer.js

```
const PNG_HEADER = Uint8Array.of(0x89, 0x50, 0x4E, 0x47, 0x0D, 0x0A,
                                 0x1A, 0x0A);
function isPNG(byteData) {
    return byteData.length >= PNG_HEADER.length &&
            PNG_HEADER.every((b, i) => b === byteData[i]);
}
function show(msg) {
    const p = document.createElement("p");
    p.appendChild(document.createTextNode(msg));
    document.body.appendChild(p);
}
document.getElementById("file-input").addEventListener(
    "change",
    function(event) {
        const file = this.files[0];
        if (!file) {
            return;
        }
        const fr = new FileReader();
        fr.readAsArrayBuffer(file);
        fr.onload = () => {
            const byteData = new Uint8Array(fr.result);
            show(`${file.name} ${isPNG(byteData) ? "is": "is not"} a PNG file.`);
        };
        fr.onerror = error => {
            show(`File read failed: ${error}`);
        };
    }
);
```

Файлы PNG начинаются с определенного 8-байтового заголовка. Код в Листинге 11-1 отвечает пользователю, выбирающему файл во вводе `type="file"`, с помощью объекта `FileReader` API-файла, чтобы считать его как `ArrayBuffer` (доступен через свойство `result` объекта `FileReader`), а затем проверить 8-байтовый заголовок необработанных полученных данных. Поскольку код в этом примере работает с байтами без знака, в нем используется массив `Uint8Array`, поддерживаемый буфером из объекта `FileReader`.

Обратите внимание, что `FileReader` не зависит от контекста: он просто снабжает `ArrayBuffer`, предоставляя использующему буфер коду возможность решать, обращаться ли к нему побайтно (как в примере в Листинге 11-1) или в 16-разрядных или в 32-разрядных словах, или даже в комбинации этих факторов. (Позже вы увидите, как один буфер `ArrayBuffer` может использоваться несколькими типизированными массивами, так что вы можете открыть один раздел с помощью, скажем, `Uint8Array`, а другой раздел с помощью `Uint32Array`.)

Порядковый номер (Порядок байтов)

Объекты `ArrayBuffer` хранят байты. Большинство типизированных массивов содержат записи, занимающие несколько байт памяти. Например, каждая запись в массиве `Uint16Array` требует два байта памяти: *байтов высокого порядка* (или «старший байт»), содержащих кратные `256` значения, и *байтов низкого порядка* (или «младший байт»), содержащих кратные `1` значения. Это похоже на «столбцы десяток» и «столбцы единиц», используемых большинством из нас в системе счисления с основанием 10. В значении `258` (`0 × 0102` в шестнадцатеричной системе) старший байт содержит значение `0 × 01`, а младший байт содержит `0 × 02`. Значение старшего байта умножается на `256` и прибавляется значение младшего байта: `1 × 256 + 2` равно `258`.

Пока все хорошо, но в каком порядке должны располагаться байты в памяти? Должен ли первым быть старший или младший байт? Единого ответа нет — используются оба способа. Если значение хранится в порядке старшего/младшего байта, оно находится в *прямом* порядке (от старшего к младшему). Если значение хранится в порядке младшего/старшего байта, оно находится в *обратном* порядке (от младшего к старшему)[68]. См. рисунок 11-3.

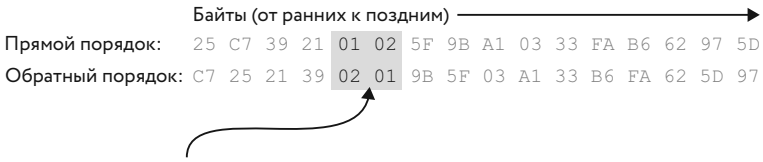


РИСУНОК 11-3

Вы, вероятно, задаетесь вопросом: если используются оба варианта, что определяет, какой из них используется и когда? На этот вопрос существуют два фундаментальных ответа:

- Машинная архитектура компьютера (в частности, архитектура процессора) обычно определяет порядковый номер значений в памяти в этой системе. Если вы прикажете компьютеру сохранить значение `0 × 0102` по определенному адресу памяти, центральный процессор запишет это значение, используя свой собственный порядок байтов. Архитектура x86, используемая основными процессорами Intel и AMD, относится к обратному типу, и поэтому подавляющее большинство настольных компьютеров использует обратный порядок. Архитектура PowerPC, используемая старыми компьютерами Macintosh до того, как Apple перешла на x86, изначально придерживалась прямого порядка, хотя у нее есть переключатель режимов, активирующий обратный порядок.
- Форматы файлов, сетевые протоколы и т. д. должны указывать порядок байтов, чтобы гарантировать корректную обработку в разных архитектурах. В некоторых

[68] Английский вариант названий прямого (Big-endian) порядка и обратного (Little-endian) порядка взяты из книги «Путешествия Гулливера», где лилипуты спорили о том, следует ли разбивать яйцо с острого конца (little-end) или с тупого (big-end). Иногда компьютерные дебаты столь же значимы. — *Прим. пер.*

случаях формат определяет порядок байтов: формат изображения портативной сетевой графики (PNG) использует прямой порядок, как и Протокол управления передачей (TCP в TCP/IP) для целых чисел в заголовках, таких как номера портов. (На самом деле, прямой порядок иногда называют «сетевым порядком байтов».) В других случаях формат или протокол позволяют указывать порядок байтов данных внутри самих данных: формат файла изображения с тегами (TIFF) начинается с двух байтов, представленных либо символами `II` (для «Intel», обратного порядка байтов) или `MM` (для «Motorola», прямого порядка байтов), потому что в 1980-х годах, когда формат разрабатывался, процессоры Intel использовали обратный порядок, а процессоры Motorola использовали прямой. Многобайтовые значения, хранящиеся в файле, используют порядок, определенный этим начальным тегом.

Типизированные массивы используют порядковый номер компьютера, на котором они задействованы. Причина такова: типизированные массивы используются для работы с встроенным API (такими как WebGL), и поэтому данные, отправляемые во встроенный API, должны быть в собственном порядке байтов машины. Код записи данных записывает их в свой типизированный массив (например, `Uint16Array`, `Uint32Array` или аналогичные), и типизированные массивы записывают данные в `ArrayBuffer` в порядке байтов машины, на которой выполняется код. Заботится об этом в коде не придется. Когда вы передаете этот буфер `ArrayBuffer` во встроенный API, он будет использовать собственный порядковый номер платформы, на которой выполняется код.

Однако это не значит, что вам никогда не придется задумываться о порядке байтов. При считывании файла (например PNG) или сетевого потока данных (например TCP-пакета), определенного как находящийся в конкретном порядке (прямой или обратный), вы не можете полагать, что порядок совпадает с платформой, выполняющей код. То есть что вы не можете использовать типизированный массив для доступа к нему. Потребуется другой инструмент — `DataView`.

Вид `DataView`: Необработанный доступ к буферу

Объекты `DataView` предоставляют необработанный доступ к данным в `ArrayBuffer` с методами для чтения этих данных в любой из числовых форм, предоставляемых типизированными массивами (Int8, Uint8, Int16, Uint16 и т. д.), с возможностью чтения многобайтовых данных в прямом или обратном порядках.

Предположим, вы считываете PNG-файл, как это было сделано ранее в коде из Листинга 11-1. PNG определяет, что его различные многобайтовые целочисленные поля находятся в формате прямого порядка, но вы не можете предполагать, что код JavaScript работает на платформе с прямым порядком. (На самом деле, скорее всего, этого не произойдет, поскольку обратный порядок используется большинством настольных, мобильных и серверных платформ.) В Листинге 11-1 это не имело значения, потому что код просто проверял, относится ли файл к типу PNG, проверяя восьмибайтовую подпись по одному байту за раз. В нем не было никаких многобайтовых чисел. Но, предположим, требуется получить размеры изображения в формате PNG? Для этого требуется считывание значений Uint32 в прямом порядке.

Вот тут-то и появляется `DataView`. Вы можете использовать его для считывания значений из буфера `ArrayBuffer`, при этом явно указывая на порядок байтов. Через

мгновение вы просмотрите немного кода, но сначала очень краткое примечание о формате файла PNG.

В Листинге 11-1 показано, что файл PNG начинается с 8-байтовой неизменяемой подписи. После нее PNG представляет собой серию «блоков», где каждый блок находится в этом формате:

- *длина*: Длина сегмента данных блока (Uint32, прямой порядок);
- *тип:* Тип блока (четыре символа из ограниченного набора символов[69], один байт на символ, хотя спецификация рекомендует реализациям рассматривать их как двоичные данные, а не символы);
- *данные:* Данные блока, если таковые имеются (*длина* в байтах, формат варьируется в зависимости от типа блока);
- *crc*: Значение CRC блока (Uint32, прямой порядок).

Спецификация PNG также требует, чтобы первый блок (сразу после 8-байтового заголовка) был блоком «заголовок изображения» («IHDR»), предоставляющим основную информацию об изображении: его ширину и высоту в пикселях, глубину цвета и т. д. Ширина — это первый Uint32 в области данных блока; высота — это второй Uint32. Оба, опять же, в прямом порядке.

Предположим, требуется получить ширину и высоту PNG из `ArrayBuffer`, предоставляемого объектом `FileReader`. Сначала вы можете проверить, что 8-байтовый заголовок PNG правильный, и, возможно, тип первого блока действительно IHDR. Затем, если оба они верны, вы знаете, что ширина равна Uint32 со смещением 16 байт в данных (первые 8 байт — это заголовок PNG, следующие 4 — это длина блока IHDR, а следующие 4 — тип блока IHDR; 8 + 4 + 4 = 16), и что высота — это следующий Uint32 после этого (со смещением в байтах равным 20). Вы можете прочитать эти значения Uint32 в формате прямого порядка даже на машинах с обратным порядком, используя метод `getUint32` из `DataView`:

```
const PNG_HEADER_1 = 0x89504E47; // Прямой порядок, первый Uint32 заголовка PNG
const PNG_HEADER_2 = 0x0D0A1A0A; // Прямой порядок, второй Uint32 заголовка PNG
const TYPE_IHDR = 0x49484452;   // Прямой порядок, тип блока IHDR
// ...
fr.onload = () => {
    const dv = new DataView(fr.result);
    if (dv.byteLength >= 24 &&
        dv.getUint32(0) === PNG_HEADER_1 &&
        dv.getUint32(4) === PNG_HEADER_2 &&
        dv.getUint32(12) === TYPE_IHDR) {
        const width = dv.getUint32(16);
        const height = dv.getUint32(20);
        show(`${file.name} is ${width} by ${height} pixels`);
    } else {
        show(`${file.name} is not a PNG file.`);
    }
};
```

(В загрузках есть работоспособная версия такого кода в файлах **read-png-info.html** и **read-png-info.js**; есть PNG-файл с названием **sample.png**.) Метод `getUint32`

[69] Допустимые символы имени блока представлены подмножеством стандарта ISO 646, в частности буквы A–Z и a–z. Они согласованы во всех вариантах ISO 646 (и такие же, как ASCII и Unicode).

принимает необязательный второй параметр, называемый `littleEndian`. Можно установить значение `true` для этого параметра, и считывание будет происходить в обратном порядке. Значение по умолчанию, если оно не указано (как в этом примере), — это прямой порядок.

Если бы в этом коде применялся объект `Uint32Array`, он использовал бы порядковый номер платформы. Если бы код платформы не предоставлял прямой порядок (и опять же, большинство систем с обратным порядком), код, проверяющий заголовок PNG и тип блока IHDR, потерпел бы неудачу. (В загрузке посмотрите файл **read-png-info-incorrect.html** и **read-png-info-incorrect.js.**) Даже если вы обновили проверки заголовка и типа, чтобы вместо этого использовать байты, и использовали только `Uint32Array` для чтения ширины и высоты, на платформе с обратным порядком вы получите для них совершенно неправильные значения (попробуйте запустить файлы **read-png-info-incorrect2.html** и **read-png-info-incorrect2.js**). Например, с помощью `sample.png` вы получите ответ «sample.png is 3355443200 by 1677721600 pixels» вместо «sample.png is 200 by 100 pixels».

Совместное использование ArrayBuffer массивами

Буфер `ArrayBuffer` может совместно использоваться несколькими типизированными массивами двумя способами:

- Без перекрытия: каждый массив использует только свою собственную часть буфера.
- С перекрытием: массивы используют одну и ту же часть буфера совместно.

Совместное использование без перекрытия

В Листинге 11-2 показан массив `Uint8Array`, использующий первую часть `ArrayBuffer`; массив `Uint16Array` использует все остальное.

Листинг 11-2: Совместное использование ArrayBuffer без перекрытия — sharing-arraybuffer-without-overlap.js

```
const buf = new ArrayBuffer(20);
const bytes = new Uint8Array(buf, 0, 8);
const words = new Uint16Array(buf, 8);
console.log(buf.byteLength); // 20 (20 байт)
console.log(bytes.length);   // 8 (8 байт)
console.log(words.length);   // 6 (шесть двухбайтовых [16 бит] слов = 12 байт)
```

Можно сделать что-то подобное, если вам будет предоставлен буфер с данными, к которым вам нужно получить доступ, и вам нужно получить доступ к первой части в виде байтов без знака, а ко второй части — в виде 16-разрядных слов без знака (в зависимости от порядка байтов платформы).

Чтобы использовать только часть буфера, код использует сигнатуру конечного конструктора, упомянутую (но не описанную) ранее:

```
new %TypedArray% (buffer [, s tart [, length]])
```

Эти параметры:

- *buffer*: Буфер ArrayBuffer для использования.
- *start*: Смещение (в байтах) от начала буфера, с которого следует начать его использование. Значение по умолчанию равно 0 (начинать с начала).
- *length*: Длина для нового типизированного массива в записях (не байтах). По умолчанию задается, сколько записей поместится в оставшуюся часть ArrayBuffer.

При создании массива Uint8Array при помощи кода new Uint8Array(buf, 0, 8) параметр со значением 0 сообщает, что начинать необходимо с начала ArrayBuffer, а параметр со значением 8 указывает, что массив Uint8Array будет длиной в восемь записей. При создании массива Uint16Array с помощью new Uint16Array(buf, 8) параметр со значением 8 указывает, что начинать необходимо со смещением 8, аргумент length не указывается, что значит использование всего пространства до конца буфера ArrayBuffer. Вы можете узнать, какую часть буфера используют массивы, посмотрев на их свойства byteOffset и byteLength.

```
console.log(bytes.byteOffset); // 0
console.log(bytes.byteLength); // 8
console.log(words.byteOffset); // 8
console.log(words.byteLength); // 12
```

Имея в виду, что третий аргумент — это количество *записей*, а не байтов, что бы вы использовали в качестве третьего аргумента конструктора Uint16Array, если бы вы хотели указать его явно, не изменяя то, что делает код?

Если вы сказали 6, вы — молодец! Длина всегда указывается в записях, а не в байтах. Так и тянет сказать 12 (число байтов для использования) или (если бы не имя «length») 20 (со желаемым смещением от начала буфера). Но *length* всегда будет количеством *записей,* согласованным с конструктором *%TypedArray%*(*length*).

Совместное использование с перекрытием

В Листинге 11-3 показаны два массива — Uint8Array и Uint16Array, совместно использующие один и тот же буфер ArrayBuffer (целиком).

Листинг 11-3: Совместное использование ArrayBuffer с перекрытием — sharing-arraybuffer-with-overlap.js

```
const buf = new ArrayBuffer(12);
const bytes = new Uint8Array(buf);
const words = new Uint16Array(buf);
console.log(words[0]); // 0
bytes[0] = 1;
bytes[1] = 1;
console.log(bytes[0]); // 1
console.log(bytes[1]); // 1
console.log(words[0]); // 257
```

Обратите внимание, что `words[0]` начинается с `0`, но затем мы присваиваем `1` для `bytes[0]` и `bytes[1]`, и значение `words[0]` становится `257` (`0x0101` в шестнадцатиричной). Это связано с тем, что два массива используют одно и то же базовое хранилище (буфер `ArrayBuffer`). Следовательно, записав `1` для `byte[0]` и для `byte[1]`, мы записываем `1` в *оба* байта, составляющих единую запись `words[0]`. Как обсуждалось ранее в разделе «Порядковый номер (Порядок байтов)», один из них становится старшим байтом, а другой — младшим байтом.

Поместив значение `1` в оба байта, мы получили 16-битное значение `1 × 256 + 1`, что равно `257` (`0x0101`). Таким образом, от `words[0]` мы получаем такое значение.

Предположим, что вы изменили код так, что вместо `1` будет присваиваться `2` для `bytes[1]`. Какое значение мы получим из `words[0]`?

Если вы сказали: «Как посмотреть», вы — молодец! Вы наверняка получили значение `513` (`0x0201`), поскольку код, скорее всего, был запущен на платформе с обратным порядком байтов, таким образом `bytes[1]` — старший байт, следовательно результат: `2 × 256 + 1 = 513`. Но при использовании платформы с прямым порядком байтов вы получили бы значение `258` (`0x0102`), поскольку `bytes[1]` становится младшим байтом, следовательно, результат: `1 × 256 + 2 = 258`.

Подклассы типизированных массивов

Вы можете подклассировать типизированные массивы (возможно, у вас есть свои собственные пользовательские методы) обычными способами: с использованием синтаксиса `class` или с использованием `Reflect.construct` (об объекте `Reflect` рассказывается в главе 14). Вероятно, это действительно полезно только для добавления дополнительных служебных методов. В общем, лучше всего включать типизированный массив в класс с помощью агрегации (как поле, используемое классом за кулисами), а не с помощью наследования.

Если вы создаете подкласс класса типизированного массива, обратите внимание на ограничения для `map` и `filter`, обсуждаемые в следующем разделе «Стандартные методы массива».

Методы типизированного массива

Типизированные массивы поддерживают большинство обычных методов массива, хотя и не все, и несколько методов, специфичных для типизированных массивов.

Стандартные методы массива

Типизированные массивы реализуют большинство тех же методов, что и традиционные массивы, используя те же алгоритмы — в некоторых случаях с небольшими изменениями, обсуждаемыми в этом разделе.

Однако, поскольку у типизированных массивов фиксированная длина, у них нет ни одного из методов, (потенциально) включающих изменение длины массива: `pop`, `push`, `shift`, `unshift` или `splice`.

Типизированные массивы также не поддерживают `flat`, `flatMap` или `concat`. Применение методов `flat` и `flatMap` не имеет смысла для типизированных массивов, поскольку типизированные массивы не могут содержать вложенных массивов. Аналогично выровненное поведение функции `concat` не применимо к типизированным

массивам. Другие операции `concat` могут быть реализованы с использованием `of` и нотацией расширения (поскольку типизированные массивы итерируемые):

```
const a1 = Uint8Array.from([1, 2, 3]);
const a2 = Uint8Array.from([4, 5, 6]);
const a3 = Uint8Array.from([7, 8, 9]);
const all = Uint8Array.of(...a1, ...a2, ...a3);
console.log(all); // Uint8Array [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Методы прототипа для создания новых массивов (такие как `filter`, `map` и `slice`) создают массив такого же типа, что и вызвавший их типизированный массив. Применение `filter` и `slice`, скорее всего, не приведет к неожиданностям, чего нельзя сказать о методе `map`. Например:

```
const a1 = Uint8Array.of(50, 100, 150, 200);
const a2 = a1.map(v => v * 2);
console.log(a2); // Uint8Array [100, 200, 44, 144]
```

Обратите внимание, что произошло с последними двумя записями: их значения были перенесены, потому что новый массив также относится к типу `Uint8Array` и поэтому не может содержать значения `300` (`150 × 2`) или `400` (`200 × 2`).

В реализации `map`, `filter` и `slice` для типизированных массивов есть еще одно ограничение: если подклассировать типизированный массив, нельзя использовать `Symbol.species` (см. глава 4) для указания методам типа `map` и `slice` создать нетипизированные массивы. Например, это не сработает:

```
class ByteArray extends Uint8Array {
    static get [Symbol.species]() {
        return Array;
    }
}
const a = ByteArray.of(3, 2, 1);
console.log(a.map(v => v * 2));
// => TypeError: Метод%TypedArray%.prototype.map вызван для
//    несовместимого приемника [объект Array]
```

Вы могли бы заставить их создать другой *вид* типизированного массива (хотя это было бы странно), но не нетипизированный массив. Если вам потребуется подкласс, способный сделать это, необходимо будет переопределить методы в подклассе. Например:

```
class ByteArray extends Uint8Array {
    static get [Symbol.species]() {
        return Array;
    }
    map(fn, thisArg) {
        const ctor = this.constructor[Symbol.species];
        return ctor.from(this).map(fn, thisArg);
    }
    // ...и аналогично для `filter`, `slice` и `subarray`
}
const a = ByteArray.of(3, 2, 1);
console.log(a.map(v => v * 2)); // [6, 4, 2]
```

Метод %TypedArray%.prototype.set

Сигнатура:

```
theTypedArray.set(a rray [, offset])
```

Метод set задает несколько значений в типизированном массиве из предоставленного вами «массива» (который может быть типизированным массивом, нетипизированным массивом или массивоподобным объектом, хотя и не просто итерируемым), при необходимости начиная запись с заданного смещения (заданного в записях, а не байтах) внутри типизированного массива. Метод set всегда копирует весь предоставленный вами массив (в исходном массиве нет никаких параметров для выбора диапазона).

Можно использовать set для объединения нескольких типизированных массивов:

```
const all = new Uint8Array(a1.length + a2.length + a3.length);
all.set(a1);
all.set(a2, a1.length);
all.set(a3, a1.length + a2.length);
```

Обратите внимание, что второй set начинает запись сразу после последней записи, написанной первым, а третий — сразу после последней записи, написанной вторым.

Обратите внимание, что для set специально требуется массив, типизированный массив или массивоподобный объект. Он не обрабатывает итерируемые. Чтобы использовать итерируемый элемент с set, сначала разложите его в массив (или используйте метод from соответствующего конструктора массива или типизированного массива и т. д.).

Метод %TypedArray%.prototype.subarray

Сигнатура:

```
newArray = theTypedArray.subarray(begin, end)
```

Метод subarray создает новый типизированный массив для подмножества массива, для которого вы его вызываете. Новый массив совместно использует буфер исходного массива (то есть массивы совместно используют одни и те же данные).

- Параметр begin — это индекс первой записи в исходном массиве, который должен быть использован совместно с подмассивом. Если значение отрицательное, оно используется как смещение от конца массива. Если его не указать, будет использоваться значение 0.
- Параметр end — это индекс первой записи, которую нельзя совместно использовать. Если значение параметра отрицательное, оно используется как смещение от конца массива. Технически он не указан как необязательный, но если его не указать, будет использоваться значение длины массива.

Значения индексов указаны в записях, а не в байтах.

Взгляните на пример:

```
const wholeArray = Uint8Array.of(0, 1, 2, 3, 4, 5, 6, 7, 8, 9);
const firstHalf = wholeArray.subarray(0, 5);
console.log(wholeArray); // Uint8Array [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
console.log(firstHalf);  // Uint8Array [0, 1, 2, 3, 4]
firstHalf[0] = 100;
console.log(wholeArray); // Uint8Array [100, 1, 2, 3, 4, 5, 6, 7, 8, 9]
console.log(firstHalf);  // Uint8Array [100, 1, 2, 3, 4]
const secondHalf = wholeArray.subarray(-5);
console.log(wholeArray); // Uint8Array [100, 1, 2, 3, 4, 5, 6, 7, 8, 9]
console.log(secondHalf); // Uint8Array [5, 6, 7, 8, 9]
secondHalf[1] = 60;
console.log(wholeArray); // Uint8Array [100, 1, 2, 3, 4, 5, 60, 7, 8, 9]
console.log(secondHalf); // Uint8Array [5, 60, 7, 8, 9]
```

Обратите внимание, как присвоение значения `firstHalf[0]` изменяет значение в `wholeArray[0]`, поскольку `firstHalf` и `wholeArray` совместно используют один буфер с самого его начала. Аналогично присвоения значения `secondHalf[1]` изменяет значение в `wholeArray[6]`, поскольку `secondHalf` и `wholeArray` используют один буфер, но `secondHalf` использует только вторую его половину.

ОТ СТАРЫХ ПРИВЫЧЕК К НОВЫМ

Большая часть того, что вы узнали в этой главе, относится к ранее недоступным возможностям. Но все же есть пара привычек, которые вы могли бы изменить.

Используйте find и findIndex для поиска в массивах вместо циклов (где это уместно)

Старая привычка: Поиск записей (или их индексов) в массивах с использованием цикла `for` или метода `some` и т. д.:

```
let found;
for (let n = 0; n < array.length; ++n) {
    if (array[n].id === desiredId) {
        found = array[n];
        break;
    }
}
```

Новая привычка: Рассмотрите возможность использования `find` (или `findIndex`):

```
let found = array.find(value => value.id === desiredId);
```

Используйте для заполнения массивов Array.fill, а не циклы

Старая привычка: Заполнение массива значений при помощи циклов:

```
// Статическое значение
const array = [];
```

```
while (array.length < desiredLength) {
    array[array.length] = value;
}

// Динамическое значение
const array = [];
while (array.length < desiredLength) {
    array[array.length] = determineValue(array.length);
}
```

Новая привычка: Используйте `Array.fill` или `Array.from`:

```
// Статическое значение
const array = Array(desiredLength).fill(value);

// Динамическое значение
const array = Array.from(
    Array(desiredLength),
    (_, index) => determineValue(index)
);
```

Используйте `readAsArrayBuffer` вместо `readAsBinaryString`

Старая привычка: Использовать метод `readAsBinaryString` для экземпляров `FileReader`, работая с результирующими данными при помощи метода `charCodeAt`.

Новая привычка: Используйте метод `readAsArrayBuffer`, работая с данными через типизированные массивы.

12

Карты и множества

СОДЕРЖАНИЕ ГЛАВЫ

- Карты
- Множества
- Слабые карты (WeakMap)
- Слабые множества (WeakSet)

В этой главе вы познакомитесь с коллекциями Map, Set, WeakMap и WeakSet из ES2015+. Map (Карта) — коллекции, хранящие пары ключ/значение, где ключ и значение могут быть (почти) любыми. Set (Множество) — коллекция уникальных значений. «Слабые» коллекции WeakMaps похожи на карты, но ключи — это объекты, и эти объекты удерживаются *слабо* (они могут быть собраны в мусор, что удаляет запись из карты). «Слабые» коллекции WeakSets слабо держат уникальные объекты.

КОЛЛЕКЦИИ MAP ИЛИ КАРТЫ

Часто требуется создать сопоставление одной вещи с другой, например, сопоставление идентификаторов с объектами, содержащими этот идентификатор. В JavaScript мы часто использовали (некоторые говорят, что злоупотребляли) объекты для этих целей. Но поскольку это не то, для чего предназначены объекты, существуют некоторые прагматические проблемы с использованием объектов для общих карт:

- Ключи могут быть только строками (или символами, начиная с ES2015).
- До ES2015 нельзя было рассчитывать на порядок при обработке циклом записей в объекте. В главе 5 описано, что даже несмотря на добавление порядка в ES2015, полагаться на него обычно не очень хорошая идея, поскольку порядок зависит от порядка добавления свойств в объект и от формы ключа свойства (строки в канонической форме целочисленного индекса идут первыми, численно).
- Объекты оптимизируются движками JavaScript при условии, что свойства к ним в основном только добавляются и обновляются, а не удаляются.

- До ES5 невозможно было создать объект без прототипа с такими свойствами, как `toString` и `hasOwnProperty`. Хотя это вряд ли будет конфликтовать с вашими собственными ключами, это все же вызывало легкое беспокойство.

Карты решают все эти проблемы:

- Ключами и значениями может быть вообще любое значение (включая объект)[70].
- Порядок записей определен: это порядок, в котором были добавлены записи (обновление значения записи не изменяет ее место в порядке).
- Движки JavaScript оптимизируют карты иначе, чем объекты, поскольку их варианты использования различны.
- Карты по умолчанию пусты.

Основные операции с картой

Основы работы с картами (создание, добавление, доступ и удаление записей) довольно просты, давайте быстро пройдемся по ним. (Вы можете запустить весь код в этом разделе, используя файл **basic-map-operations.js** из перечня загрузок.)

Чтобы создать карту, используйте конструктор (подробнее об этом позже в этой главе):

```
const m = new Map();
```

Используйте метод `set`, чтобы связать ключ со значением и добавить в коллекцию записи:

```
m.set(60, "sixty");
m.set(4, "four");
```

Ключи в этом примере представлены цифрами `60` и `4` (первый аргумент в каждом вызове `set`). Значения — это строки `"sixty"` и `"four"` (второй аргумент метода).

Метод `set` возвращает карту, так что вы можете связать несколько вызовов вместе, если хотите:

```
m.set(50, "fifty").set(3, "three");
```

Чтобы увидеть, сколько записей находится в карте, используйте ее свойство `size`:

```
console.log(`Entries: ${m.size}`);       // Entries: 4
```

Чтобы получить значение для ключа, используйте метод `get`:

```
let value = m.get(60);
console.log(`60: ${value}`);             // 60: sixty
console.log(`3: ${m.get(3)}`);           // 3: three
```

[70] Здесь есть одно небольшое предостережение; подробности см. в разделе «Равенство ключей».

Метод `get` возвращает значение `undefined`, если для ключа нет записи:

```
console.log(`14: ${m.get(14)}`);          // 14: undefined
```

Все ключи в этом примере до сих пор представлены числами. Если бы вы делали это с объектом, а не с картой, ключи были бы преобразованы в строки:

```
console.log('Look for key "4" instead of 4:');
console.log(`"4": ${m.get("4")}`);        // "4": undefined (ключ - 4,
а не "4")
console.log('Look for key 4:');
console.log(`4: ${m.get(4)}`);            // 4: four
```

Чтобы обновить значение записи, снова используйте `set`; это обновит существующую запись:

```
m.set(3, "THREE");
console.log(`3: ${m.get(3)}`);            // 3: THREE
console.log(`Entries: ${m.size}`);        // Entries: 4 (все еще)
```

Чтобы исключить (удалить) запись, используйте метод `delete`:

```
m.delete(4);
```

Метод `delete` возвращает значение `true`, если запись была удалена, и значение `false`, если не было ни одной записи, соответствующей ключу. Обратите внимание, что `delete` — это *метод* карт, мы не использовали оператор `delete`. `delete m[2]` будет пытаться удалить свойство под названием `"2"` из объекта карты, но записи в карте не относятся к *свойствам* объекта карты — следовательно, ничего не произойдет.

Чтобы проверить, существует ли запись в карте, используйте метод `has`, возвращающий логическое значение (`true`, если в карте есть запись для ключа, и `false`, если нет):

```
console.log(`Entry for 7 ? ${m.has(7)}`); // Entry for 7 ? Ложь
console.log(`Entry for 3 ? ${m.has(3)}`); // Entry for 3 ? Истина
```

Обратите внимание, что записи — это не свойства, поэтому оператор `in` или метод `hasOwnProperty` не используется.

До сих пор все ключи в этом примере были одного типа (числа). Обычно это справедливо и в реальном мире, но ключи в карте не обязательно должны быть одного типа. Вы можете добавить запись со строковым ключом к этому примеру созданной карты, в которой до сих пор были только цифровые ключи, например:

```
m.set("testing", "one two three");
console.log(m.get("testing"));            // one two three
```

Ключи могут быть объектами:

```
const obj1 = {};
m.set(obj1, "value for obj1");
console.log(m.get(obj1));                 // value for obj1
```

Разные объекты всегда представляют собой разные ключи, даже если у них одинаковые свойства. Ключ `obj1` — это простой объект без свойств. Если добавить другую запись, используя в качестве ключа простой объект без свойств, это будет другой ключ:

```
const obj2 = {};
m.set(obj2, "value for obj2");
console.log(`obj1: ${m.get(obj1)}`);    // obj1: value for obj1
console.log(`obj2: ${m.get(obj2)}`);    // obj2: value for obj2
```

Поскольку ключами могут быть (почти) любые значения JavaScript, действительные ключи включают значения `null`, `undefined` и даже `NaN` (которое уверенно не равно ничему, даже самому себе):

```
m.set(null, "value for null");
m.set(undefined, "value for undefined");
m.set(NaN, "value for NaN");
console.log(`null: ${m.get(null)}`);           // null: value for null
console.log(`undefined: ${m.get(undefined)}`); // undefined: value for undefined
console.log(`NaN: ${m.get(NaN)}`);             // NaN: value for NaN
```

Обратите внимание, что метод `get` возвращает значение `undefined` в двух разных ситуациях: при отсутствии соответствующей записи для ключа или если запись есть и ее значение равно `undefined`. (Точно так же, как свойства объекта.) Если вам нужно определить разницу между этими ситуациями, используйте метод `has`.

Чтобы удалить все записи из карты, используйте метод `clear`:

```
m.clear();
console.log(`Entries now: ${m.size}`);   // Entries now: 0
```

Равенство ключей

Ключи в картах могут быть представлены практически любым значением (подробнее об этом чуть позже). Они сравниваются с использованием операции `SameValueZero`, аналогичной строгому равенству (`===`), за исключением того, что `NaN` равно самому себе (что, как известно, в противном случае не так). Это означает, что:

- При попытке сопоставления ключей нет приведения типов: ключ всегда отличается от ключа другого типа (`"1"` и `1` — разные ключи).
- Ключ объекта всегда отличается от ключа любого другого объекта, даже если у них одинаковые свойства.
- `NaN` работает как ключ.

«Практически» во фразе «Ключи в картах могут быть представлены практически любым значением» заключается в том, что ключа карты со значением `-0` (отрицательный ноль) быть не может. Если вы попытаетесь добавить элемент с ключом `-0`, карта будет вместо него использовать `0`. Если просматривать запись с помощью ключа `-0`, метод `get` будет искать запись по значению `0`:

```
const key = -0;
console.log(key);             // -0
const m = new Map();
m.set(key, "value");
const [keyInMap] = m.keys(); // (`keys` возвращает итератор для ключей карты)
console.log(keyInMap);        // 0
console.log(`${m.get(0)}`);   // value
console.log(`${m.get(-0)}`); // value
```

Это делается для того, чтобы избежать двусмысленности, поскольку `0` и `-0` трудно отличить друг от друга. Это разные значения (в соответствии со спецификацией IEEE-754, которой придерживаются числа JavaScript), но строго равные друг другу (`0 === -0` — истина). Функция чисел `toString` возвращает значение `"0"` для обоих, и большинство операций JavaScript обрабатывает их как одно и то же значение. Когда `Map` и `Set` разрабатывались, было много дискуссий по поводу двух нулей (в том числе рассматривался флаг, который можно было бы использовать, чтобы решить, были ли эти значения одинаковыми или разными в данном методе `Map` или `Set`), но значимость этого не оправдывает сложности. Поэтому, чтобы не стрелять себе в ногу, `Map` преобразует `-0` в `0` при попытке использовать его в качестве ключа.

Создание карт из итерируемых

Ранее было показано, что конструктор `Map` создает пустую карту, если вы не передаете ему никаких аргументов. Можно также предоставить записи для карты, передав итеративный набор объектов — обычно массив массивов. Вот простой пример того, как это делается, сопоставляя английские слова с итальянскими:

```
const m = new Map([
    ["one", "uno"],
    ["two", "due"],
    ["three", "tre"]
]);
console.log(m.size);         // 3
console.log(m.get("two"));  // due
```

Однако ни итерируемый объект, ни его записи *не должны* быть массивами: можно использовать любой итерируемый элемент, а для его записей — любой объект со свойствами `"0"` и `"1"`. По сути конструктор `Map` выглядит примерно так:

```
constructor(entries) {
    if (entries) {
        for (const entry of entries) {
            this.set(entry["0"], entry["1"]);
        }
    }
}
```

Обратите внимание, что записи создаются в том порядке, в котором они предоставлены итератором итерируемого элемента.

Если вы передаете конструктору `Map` более одного аргумента, последующие аргументы в настоящее время игнорируются.

Вы можете скопировать карту, просто передав ее в конструктор `Map` (подробнее о том, как это работает, в следующем разделе):

```
const m1 = new Map([
    [1, "one"],
    [2, "two"],
    [3, "three"]
]);
const m2 = new Map(m1);
console.log(m2.get(2)); // two
```

Итерация содержимого карты

Карты — итерируемые элементы. Их итератор по умолчанию создает массив `[ключ, значение]` для каждой записи (именно поэтому вы можете скопировать карту, передав ее конструктору `Map`). Карты также предоставляют метод `keys` для итерации ключей и метод `values` для итерации значений.

Когда вам нужны и ключ, и значение, вы можете использовать итератор по умолчанию. Например, в цикле `for-of` с итеративной деструктуризацией:

```
const m = new Map([
    ["one", "uno"],
    ["two", "due"],
    ["three", "tre"]
]);
for (const [key, value] of m) {
    console.log(`${key} => ${value}`);
}
// one => uno
// two => due
// three => tre
```

Конечно, не обязательно использовать деструктуризацию, можно просто использовать массив в теле цикла:

```
for (const entry of m) {
    console.log(`${entry[0]} => ${entry[1]}`);
}
// one => uno
// two => due
// three => tre
```

Итератор по умолчанию также доступен через метод `entries` (на самом деле, методы `Map.prototype.entries` и `Map.prototype[Symbol.iterator]` относятся к одной и той же функции).

Записи в карте хранятся в том порядке, в котором были созданы. Вот почему в двух предыдущих примерах записи были показаны в том же порядке, в котором они появляются в массиве при вызове конструктора `Map`. Обновление значения записи не приводит к ее перемещению в порядке. Однако, если запись удалить, а затем добавить другую запись с тем же ключом, новая запись будет помещена в «конец» карты (поскольку, как только вы удалили старую запись, она больше не существовала в карте). Вот примеры этих правил:

```
const m = new Map([
    ["one", "uno"],
    ["two", "due"],
    ["three", "tre"]
]);

// Изменение существующей записи
m.set("two", "due (updated)");
for (const [key, value] of m) {
    console.log(`${key} => ${value}`);
}
// one => uno
// two => due (обновленная запись)
// three => tre

// Удаление записи, затем добавление новой с тем же ключом
m.delete("one");
m.set("one", "uno (new)");
for (const [key, value] of m) {
    console.log(`${key} => ${value}`);
}
// two => due (обновленное значение)
// three => tre
// one => uno (новая запись)
```

Порядок применяется ко всем итерируемым элементам карты (`entries`, `keys` и `values`):

```
const m = new Map([
    ["one", "uno"],
    ["two", "due"],
    ["three", "tre"]
]);

for (const key of m.keys()) {
    console.log(key);
}
// one
// two
// three

for (const value of m.values()) {
    console.log(value);
}
// uno
// due
// tre
```

Вы также можете перебирать записи в карте, используя ее метод `forEach`. Он точно такой же, как предоставляемый массивами:

```
const m = new Map([
    ["one", "uno"],
    ["two", "due"],
    ["three", "tre"]
]);
```

```
m.forEach((value, key) => {
    console.log(`${key} => ${value}`);
});
// one => uno
// two => due
// three => tre
```

Как и в случае с массивами, вы можете передать второй аргумент, если хотите контролировать, значение `this` в обратном вызове, и обратный вызов получает три аргумента: обрабатываемое значение, его ключ и карту, по которой вы проходите цикл. (В примере используются только ключ и значение.)

Создание подклассов для карты

Как и другие встроенные элементы, `Map` может быть подклассирован. Например, у встроенного класса `Map` нет функции `filter`, аналогичной `filter` у массивов. А в вашем коде есть карты, требующие частой фильтрации. Хотя вы могли бы добавить свою собственную функцию `filter` в `Map.prototype`, расширение встроенных прототипов может быть проблематичным (особенно в коде библиотеки, а не в коде приложения/страницы). Вместо этого можно использовать подкласс. Например, в Листинге 12-1 создается подкласс `MyMap` с функцией `filter`.

Листинг 12-1: Создание подкласса карты — subclassing-map.js

```
class MyMap extends Map {
    filter(predicate, thisArg) {
        const newMap = new (this.constructor[Symbol.species] || MyMap)();
        for (const [key, value] of this) {
            if (predicate.call(thisArg, key, value, this)) {
                newMap.set(key, value);
            }
        }
        return newMap;
    }
}

// Применение:
const m1 = new MyMap([
["one", "uno"],
["two", "due"],
["three", "tre"]
]);
const m2 = m1.filter(key => key.includes("t"));
for (const [key, value] of m2) {
    console.log(`${key} => ${value}`);
}
// two => due
// three => tre
console.log(`m2 instanceof MyMap ? ${m2 instanceof MyMap}`);
// m2 instanceof MyMap ? true
```

Обратите внимание на использование символа `Symbol.species`, изученного в главе 4. У конструктора `Map` есть геттер `Symbol.species`, возвращающий `this`.

Подкласс `MyMap` не переопределяет это значение по умолчанию, следовательно, его функция `filter` создает экземпляр `MyMap`, допуская при этом дополнительные подклассы (например `MySpecialMap`), чтобы контролировать, что функция `filter` должна создать экземпляр этого подкласса — `MyMap` или `Map` (или что-то еще, хотя это кажется маловероятным).

Производительность

Реализация Map, естественно, зависит от конкретного движка JavaScript, но спецификация требует, чтобы они были реализованы «…с использованием либо хэш-таблиц, либо других механизмов, которые в среднем обеспечивают время доступа, сублинейное по количеству элементов в коллекции». Это означает, что, например, добавление записи должно быть в среднем быстрее, чем поиск по массиву, чтобы увидеть, содержит ли он запись, а затем добавление ее при отсутствии. В коде это означает следующее:

```
map.set(key, value);
```

Требуется, чтобы в среднем этот код был быстрее, чем следующий:

```
const entry = array.find(e => e.key === key);
if (entry) {
    entry.value = value;
} else {
    array.push({key, value});
}
```

МНОЖЕСТВА

Set (Множество) — это коллекции уникальных значений. Все, что вы узнали о ключах карты в предыдущем разделе, применимо к значениям в множестве. Множество может содержать любое значение, кроме `-0`, преобразуемое в `0` при добавлении (точно так же, как ключи карты). Значения сравниваются с использованием операции `SameValueZero` (аналогично ключам карты). Порядок значений в множестве — это порядок, в котором они были добавлены к множеству. Повторное добавление существующего значения не меняет его положения; удаление значения, а затем его повторное добавление меняет положение. Когда вы добавляете значение в множество, оно сначала проверяет, содержится ли уже в нем это значение, и добавляет его, только если это не так.

До множеств для этой цели иногда использовались объекты (сохранение значений в виде имен свойств, преобразованных в строки, с любым значением), но такой способ обладает теми же недостатками, что и использование объектов для карт. В качестве альтернативы вместо множеств часто используются массивы или массивоподобные объекты. Перед добавлением значения производится поиск в массиве, чтобы увидеть, содержит ли он уже добавляемое значение. (Например, jQuery обычно основан на множестве: экземпляр jQuery похож на массив, но добавление того же элемента DOM в экземпляр jQuery не добавляет его повторно.) Если вы когда-либо внедряли библиотеку, подобную jQuery из ES2015 +, то могли бы задуматься об использовании внутреннего множества в ней. Его даже можно сделать подклассом множества, если вас устраивает, что подкласс потенциально может содержать в себе элементы, отличные от элементов DOM,

благодаря принципу подстановки листингов. (jQuery действительно допускает это, хотя это и малоизвестно.)

Основные операции с множеством

Давайте быстро пройдемся по основам множеств: создание, добавление, доступ и удаление записей. (Вы можете запустить весь код в этом разделе, используя файл **basic-set-operations.js** из перечня загрузок.)

Чтобы создать множество, используйте конструктор (подробнее об этом позже в этой главе):

```
const s = new Set(
```

Чтобы добавить записи, используйте метод `add`:

```
s.add("two");
s.add("four");
```

Метод `add` принимает только одно значение. Передача ему более одного значения не приведет к добавлению последующих значений. Но он возвращает множество, поэтому несколько вызовов `add` могут быть объединены в цепочку:

```
s.add("one").add("three");
```

Чтобы проверить, есть ли в наборе заданное значение, используйте метод `has`:

```
console.log(`Has "two" ? ${s.has("two")}`);     // Has "two" ? true
```

Записи множества не являются свойствами, поэтому не стоит использовать оператор `in` или метод `hasOwnProperty`.

Чтобы получить количество записей в множестве, используйте свойство `size`:

```
console.log(`Entries: ${s.size}`);              // Entries: 4
```

Множества, по своей сути, никогда не содержат одно и то же значение дважды. Если вы попытаетесь добавить уже имеющееся значение, оно не будет добавлено:

```
s.add("one").add("three");
console.log(`Entries: ${s.size}`);              // Entries: 4 (все еще)
```

Чтобы исключить (удалить) запись из множества, используйте метод `delete`:

```
s.delete("two");
console.log(`Has "two" ? ${s.has("two")}`);     // Has "two" ? false
```

Чтобы полностью очистить множество, используйте метод `clear`:

```
s.clear();
console.log(`Entries: ${s.size}`);              // Entries: 0
```

Хотя примерами значений в этом разделе были строки, опять же, значения в множестве могут быть практически любым значением JavaScript (только не отрицательным нулем, как в случае с ключами карты), и все они не обязательно должны быть одного типа.

Создание множеств из итерируемых

Конструктор `Set` принимает итерируемый элемент и заполняет набор значениями из итерируемого (по порядку):

```
const s = new Set(["one", "two", "three"]);
console.log(s.has("two")); // истина
```

Естественно, если итерируемый возвращает одно и то же значение дважды, результирующий набор будет содержать его только один раз:

```
const s = new Set(["one", "two", "three", "one", "two", "three"]);
console.log(s.size); // 3
```

Итерация содержимого множества

Множества являются итеративными. Порядок итерации — это порядок, в котором значения были добавлены в множество:

```
const s = new Set(["one", "two", "three"]);
for (const value of s) {
    console.log(value);
}
// one
// two
// three
```

Добавление уже находящегося во множестве значения не меняет его положения в множестве. Полное удаление значения с последующим добавлением обратно выглядит так (поскольку к тому времени, когда вы добавляете его обратно, его больше нет в наборе):

```
const s = new Set(["one", "two", "three"]);
for (const value of s) {
    console.log(value);
}
s.add("one"); // Повторно
for (const value of s) {
    console.log(value);
}
// one
// two
// three

s.delete("one");
s.add("one");
for (const value of s) {
    console.log(value);
}
// two
```

```
// three
// one
```

Поскольку множества можно создать из итерируемых элементов, и множества тоже итерируемые, можно скопировать множество, передав его в конструктор `Set`:

```
const s1 = new Set(["a", "b", "c"]);
const s2 = new Set(s1);
console.log(s2.has("b")); // истина
s1.delete("b");
console.log(s2.has("b")); // истина (тем не менее, удаление из s1
                          // не приводит к удалению из s2)
```

Использование итеративности множества — удобный способ копирования массива при удалении любых повторяющихся записей в нем (классическая функция `unique`, находящаяся в наборах инструментов многих программистов):

```
const a1 = [1, 2, 3, 4, 1, 2, 3, 4];
const a2 = Array.from(new Set(a1));
console.log(a2.length);     // 4
console.log(a2.join(", ")); // 1, 2, 3, 4
```

Тем не менее, если требуются только уникальные значения, возможно, стоит использовать в первую очередь множество, а не массив.

Итератор множества по умолчанию выполняет итерацию его значений. Этот итератор также доступен с помощью метода `values`. Чтобы сделать интерфейсы карт и множеств похожими, этот итератор также доступен с помощью метода `keys`. Фактически выражения `Set.prototype[Symbol.iterator]`, `Set.prototype.values` и `Set.prototype.keys` относятся к одной и той же функции.

Множества также предоставляют метод `entries`, возвращающий массивы с двумя записями (как `entries` из карт), где обе записи содержат значение из множества — как если бы множество было картой, где ключи сопоставляются сами с собой:

```
const s = new Set(["a", "b", "c"]);
for (const value of s) {// или `of s.values()`
    console.log(value);
}
// a
// b
// c
for (const key of s.keys()) {
    console.log(key);
}
// a
// b
// c
for (const [key, value] of s.entries()) {
    console.log(`${key} => ${value}`);
}
// a => a
// b => b
// c => c
```

Создание подклассов для множества

`Set` легко подклассируется. Например, предположим, что требуется метод `addAll`, добавляющий все значения из итерируемого в множество, вместо того чтобы вызывать `add` множество раз, но вы не хотите добавлять его в `Set.prototype`, поскольку вы пишете код библиотеки. В Листинге 12-2 есть простой подкласс множества с добавленным методом.

Листинг 12-2: Создание подкласса множества — subclassing-set.js

```
class MySet extends Set {
    addAll(iterable) {
        for (const value of iterable) {
            this.add(value);
        }
        return this;
    }
}

// Применение
const s = new MySet();
s.addAll(["a", "b", "c"]);
s.addAll([1, 2, 3]);
for (const value of s) {
    console.log(value);
}
// a
// b
// c
// 1
// 2
// 3
```

Как и в случае с картами, если вы собираетесь добавить метод, создающий новое множество, можно использовать шаблон вида для создания нового экземпляра.

Производительность

Как и в случае с картами, реализация множества зависит от конкретного движка JavaScript, но спецификация требует, чтобы они были реализованы «…с использованием либо хэш-таблиц, либо других механизмов, которые в среднем обеспечивают время доступа, сублинейное по количеству элементов в коллекции». Следовательно требуется, чтобы выражение

```
set.add(value);
```

было в среднем быстрее, чем это:

```
if (!array.includes(value)) {
    array.push(value);
}
```

СЛАБЫЕ КАРТЫ (WEAKMAP)

WeakMap (слабые карты) позволяют сохранять значение, связанное с объектом (ключом), не заставляя ключ оставаться в памяти; ключ *слабо удерживается* картой. Если карта будет единственной причиной, по которой объект хранится в памяти, его запись (ключ и значение) автоматически удаляется из карты, оставляя ключевой объект пригодным для сборки мусора. (*Значение* записи, если это объект, обычно сохраняется картой до тех пор, пока запись существует. Только *ключ* удерживается слабо.) Это верно, даже если один и тот же объект используется в качестве ключа более чем в одной слабой карте (или если он хранится в слабом множестве, об этом вы узнаете в следующем разделе). Как только единственные оставшиеся ссылки на ключевой объект находятся в слабой карте (или слабом множестве), движок JavaScript удаляет записи для этого объекта и делает объект доступным для сборки мусора.

Например, если необходимо сохранить информацию, относящуюся к элементу DOM, не сохраняя ее в свойстве самого элемента, можно сохранить ее в слабой карте, используя элемент DOM в качестве ключа и сохраняемую информацию в качестве значения. Если элемент DOM удален из DOM и на него больше ничего не ссылается, запись для этого элемента автоматически удаляется из карты, и память элемента DOM может быть восстановлена (вместе с любой памятью, выделенной для значения записи карты, при условии, что ничто другое не ссылается на это значение отдельно от карты).

Слабые карты не итерируемые

Одним из важных аспектов слабых карт является то, что, если у вас еще нет определенного ключа, вы не сможете получить его из карты. Это связано с тем, что у реализации может быть задержка между моментом, когда ключ недоступен другими средствами, и моментом, когда запись удаляется из карты. Если бы вы могли получить ключ с карты в течение этого времени, это внесло бы неопределенность в код, выполняющий эту операцию. Таким образом, слабые карты не позволяют получить ключ, если у вас его еще нет.

Это оказывает значительное влияние на слабые карты: они не итерируемые. Слабые карты предоставляют очень мало информации о своем содержимом, только следующее:

- `has`: При получении ключа этот метод сообщит вам, есть ли в коллекции запись для этого ключа.
- `get`: При получении ключа этот метод выдаст значение для записи ключа в слабой карте (или значение `undefined`, если у него нет соответствующей записи, как и случае с картой).
- `delete`: При получении ключа этот метод удалит запись этого ключа (если таковая имеется) и вернет флаг: `true`, если запись была обнаружена и удалена, `false`, если запись не найдена.

В слабых картах недоступны ни `size`, ни `forEach`, ни `keys` итератора, ни `values` итератора и т. д. `WeakMap` — это не подкласс `Map`, хотя общие фрагменты их API намеренно отражают друг друга.

Варианты использования и примеры

Давайте рассмотрим несколько вариантов использования слабых карт.

Вариант использования: Закрытая информация

Один из классических вариантов использования слабых карт — приватная информация. Рассмотрим Листинг 12-3.

Листинг 12-3: Использование WeakMap для закрытия информации — private-info.js

```
const Example = (() => {
    const privateMap = new WeakMap();

    return class Example {
        constructor() {
            privateMap.set(this, 0);
        }

        incrementCounter() {
            const result = privateMap.get(this) + 1;
            privateMap.set(this, result);
            return result;
        }

        showCounter() {
            console.log(`Counter is ${privateMap.get(this)}`);
        }
    };
})();

const e1 = new Example();
e1.incrementCounter();
console.log(e1);         // (некое представление объекта)

const e2 = new Example();
e2.incrementCounter();
e2.incrementCounter();
e2.incrementCounter();

e1.showCounter();        // Counter is 1
e2.showCounter();        // Counter is 3
```

Реализация `Example` обернута во встроенную функцию, вызываемую сразу после объявления (IIFE), так что `privateMap` является полностью закрытым только для реализации класса.

ПРИВАТНЫЕ ПОЛЯ КЛАССА

В главе 18 рассказывается о предлагаемых *частных полях класса*, которые будут в ES2021 и предоставят классу возможность содержать действительно приватную информацию (как специфичную для экземпляра, так и статическую). Это можно было бы использовать здесь двумя способами: вы могли бы просто использовать приватное поле для

> счетчика вместо использования слабой карты, или, если у вас была причина для использования слабой карты (возможно, не всем экземплярам нужна приватная информация), вы могли бы сделать саму карту статическим приватным полем (что упростило бы реализацию отказавшись от оболочки IIFE).

Когда экземпляр создается при помощи конструктора `Example`, он сохраняет запись в слабой карте (`privateMap`) со значением `0`, вводимым новым объектом (`this`). Вызов функции `incrementCounter` обновляет эту запись, увеличивая ее значение, а вызов `showCounter` отображает текущее значение. Чтобы увидеть этот код в действии, вероятно, лучше всего запустить его в среде с отладчиком или интерактивной консолью. Обязательно покопайтесь в объекте, выводимом в строке «(некое представление объекта)». Если вы это сделаете, вы нигде не найдете значение счетчика (counter), даже если объект может получить доступ к значению счетчика и даже изменить его. Значение счетчика действительно закрыто для кода внутри функции ограничения области видимости, обернутой вокруг класса `Example`. Никакой другой код не может получить доступ к этой информации, потому что никакой другой код не имеет доступа к `privateMap`.

Вопрос: что произошло бы, если бы вы использовали в коде карту вместо слабой карты?

Верно! Со временем карта становилась бы все больше и больше, поскольку объекты `Example` продолжали бы создаваться, и коллекция не очищалась, даже когда с ними был выполнен весь остальной код, потому что карта сохраняла их в памяти. Но со слабой картой, когда весь остальной код выполняется с объектом `Example` и удаляет ссылку на него, движок JavaScript удаляет запись в слабой карте для этого объекта.

Вариант использования: Хранение информации для объектов, находящихся вне вашего контроля

Другой важный вариант использования слабых карт — хранение информации об объектах, находящихся вне вашего контроля. Предположим, вы имеете дело с API, предоставляющим объекты, и вам нужно отслеживать свою информацию, связанную с этими объектами. Добавление свойств к этим объектам — обычно плохая идея (и API может даже сделать это невозможным, предоставив вам прокси-сервер или замороженный объект).

Слабая карта спешит на помощь! Просто храните информацию, введенную объектом. Поскольку слабая карта слабо удерживает ключ, это не предотвращает сборку мусора для объекта API.

В Листингах 12-4 и 12-5 показан пример отслеживания информации об элементах DOM с использованием элементов DOM в качестве ключей в слабых картах.

Листинг 12-4: Хранение данных для элементов DOM (HTML) — storing-data-for-dom.html

```
<!doctype html>
<html>
<head>
<meta charset="UTF-8">
```

```
<title>Storing Data for DOM Elements</title>
</head>
<style>
.person {
    cursor: pointer;
}
</style>
<body>
<label>
<div id="status"></div>
<div id="people"></div>
<div id="person"></div>
<script src="storing-data-for-dom.js"></script>
</body>
</html>
```

Листинг 12-5: Хранение данных для элементов DOM (JavaScript) — storing-data-for-dom.js

```
(async() => {
    const statusDisplay = document.getElementById("status");
    const personDisplay = document.getElementById("person");
    try {
        // Слабая карта, которая будет содержать информацию, связанную
        // с нашими элементами DOM
        const personMap = new WeakMap();
        await init();

        async function init() {
            const peopleList = document.getElementById("people");
            const people = await getPeople();
            // В этом цикле мы сохраняем переменную person, относящуюся
            // к каждому элементу div в слабой карте, и использующую div
            // в качестве ключа
            for (const person of people) {
                const personDiv = createPersonElement(person);
                personMap.set(personDiv, person);
                peopleList.appendChild(personDiv);
            }
        }
        async function getPeople() {
            // Это дублер для операции, получающей данные person
            // от сервера или аналогичные
            return [
                {name: "Joe Bloggs", position: "Front-End Developer"},
                {name: "Abha Patel", position: "Senior Software Architect"},
                {name: "Guo Wong", position: "Database Analyst"}
            ];
        }

        function createPersonElement(person) {
            const div = document.createElement("div");
            div.className = "person";
            div.innerHTML =
                '<a href="#show" class="remove">X</a>
                <span class="name"></span>';
            div.querySelector("span").textContent = person.name;
```

```
            div.querySelector("a").addEventListener("click", removePerson);
            div.addEventListener("click", showPerson);
            return div;
        }

        function stopEvent(e) {
            e.preventDefault();
            e.stopPropagation();
        }

        function showPerson(e) {
            stopEvent(e);
            // Здесь мы получаем переменную person для отображения,
            // просматривая выбранный элемент в слабой карте
            const person = personMap.get(this);
            if (person) {
                const {name, position} = person;
                personDisplay.textContent = `${name}'s position is: ${position}`;
            }
        }

        function removePerson(e) {
            stopEvent(e);
            this.closest("div").remove();
        }
    } catch (error) {
        statusDisplay.innerHTML = `Error: ${error.message}`;
    }
})();
```

Значения, ссылающиеся на ключ

Пример данных DOM в предыдущем разделе (`storing-data-for-dom.js`) сохранял объектное значение (каждой переменной `person`) в `personMap` для каждого человека, отображая элемент `div`.

Предположим, что объект `person` сослался обратно на элемент DOM. Например, если цикл `for-of` в функции `init` заменил константу `personDiv` на свойство объекта `person`, таким образом:

```
for (const person of people) {
    const person.div = createPersonElement(person);
    personMap.set(person.div, person);
    peopleList.appendChild(person.div);
}
```

Теперь ключ записи слабого массива — это `person.div` и его значение — `person`. Это означает, что значение (`person`) получило ссылку обратно на ключ (`person.div`). Будет ли существование ссылки на ключ объекта `person` хранить ключ в памяти, даже если *только* объект `person` (и `personMap`) ссылается на ключ?

Вы, наверное, догадались: нет, этого не будет. Формулировка спецификации по этому поводу — одна из самых простых для чтения частей. В ней говорится[71]:

[71] `https://tc39.github.io/ecma262/#sec-weakmap-objects`

Если объект, используемый в качестве ключа пары ключ/значение в слабой карте, доступен только путем следования цепочке ссылок, начинающихся внутри этой слабой карты, то эта пара ключ/значение недоступна и автоматически удаляется из слабой карты.

Давайте это докажем. См. Листинг 12-6.

Листинг 12-6: Значение, ссылающееся обратно на ключ — value-referring-to-key.js

```
function log(msg) {
    const p = document.createElement("pre");
    p.appendChild(document.createTextNode(msg));
    document.body.appendChild(p);
}

const AAAAExample = (() => {
    const privateMap = new WeakMap();

    return class AAAAExample {
        constructor(secret, limit) {
            privateMap.set(this, {counter: 0, owner: this});
        }

        get counter() {
            return privateMap.get(this).counter;
        }

        incrementCounter() {
            return ++privateMap.get(this).counter;
        }
    };
})();

const e = new AAAAExample();

let a = [];
document.getElementById("btn-create").addEventListener("click", function(e) {
    const count = +document.getElementById("objects").value || 100000;
    log(`Generating ${count} objects...`);
    for (let n = count; n > 0; --n) {
        a.push(new AAAAExample());
    }
    log(`Done, ${a.length} objects in the array`);
});
document.getElementById("btn-release").addEventListener("click", function(e) {
    a.length = 0;
    log("All objects released");
});
```

Код создает объект `AAAAExample`, на который он ссылается в `e`, и никогда не выпускает его. При нажатии кнопки он создает ряд дополнительных объектов `AAAAExample`, которые запомнил в массиве `a`, а при нажатии на другую кнопку выпускает все объекты массива `a`.

Откройте страницу, используя следующий HTML-код, и файл **value-referring-to-key.js** из Листинга 12-6 (можете использовать файл **value-referring-to-key.html** из загрузок):

```
<label>
    Objects to create:
    <input type="text" id="objects" value="100000">
</label>
<input type="button" id="btn-create" value="Create">
<input type="button" id="btn-release" value="Release">
<script src="value-referring-to-key.js"></script>
```

После открытия этой страницы выполните следующие действия:

1. Откройте инструменты разработчика вашего браузера и перейдите на вкладку **Memory** (Память) или аналогичную ей. На рисунке 12-1 показано, как это выглядело бы в Chrome с закрепленными внизу инструментами разработчика.
2. Нажмите кнопку **Create** (Создать) на веб-странице. Это создает 100 000 объектов (или любое другое число, на которое вы могли его изменить) с помощью конструктора `AAAAExample` и сохраняет их в массиве (`a`).

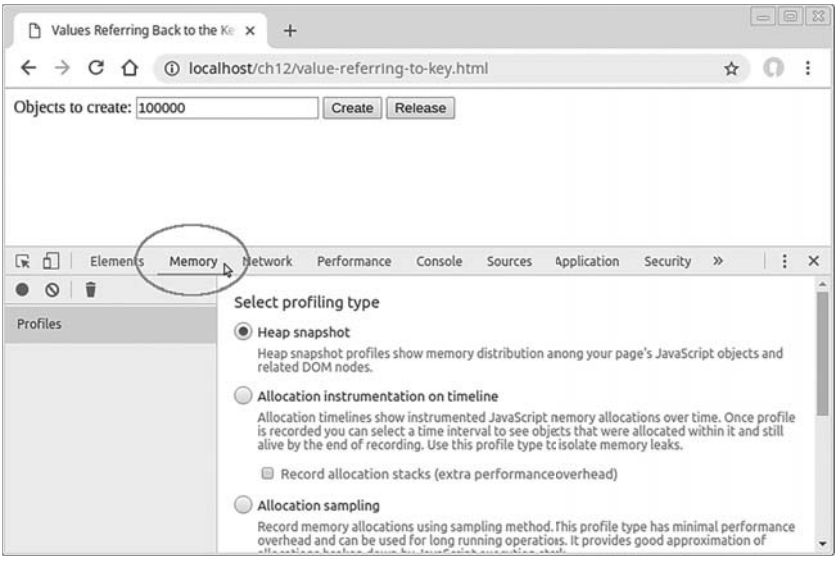


РИСУНОК 12-1

Используя любой предоставляемый вашим браузером механизм, посмотрите, сколько объектов, созданных с помощью конструктора `AAAAExample` (названного так, чтобы он отображался в верхней части алфавитных списков), находится в памяти. В Chrome это можно сделать, нажав кнопку **Take Heap Snapshot** (Сделать снимок кучи) (рисунок 12-2), а затем просмотрев список объектов на экране **Constructor** (Конструктор) (возможно, вам придется потянуть панель под названием **Retainers** (Сохраненные пути) вниз, чтобы просмотреть

содержимое панели **Constructor** (Конструктор)). См. рисунок 12-3. На этом рисунке вы можете видеть, что в памяти находится 100 001 объект `AAAAExample` (начальный, который никогда не освобождается, и 100 000 объектов, созданных в ответ на нажатие кнопки). Возможно, вам будет удобнее ввести «ааа» в поле **Class Filter** (Фильтр классов), чтобы отображалась только строка `AAAAExample` (см. рисунок 12-4).

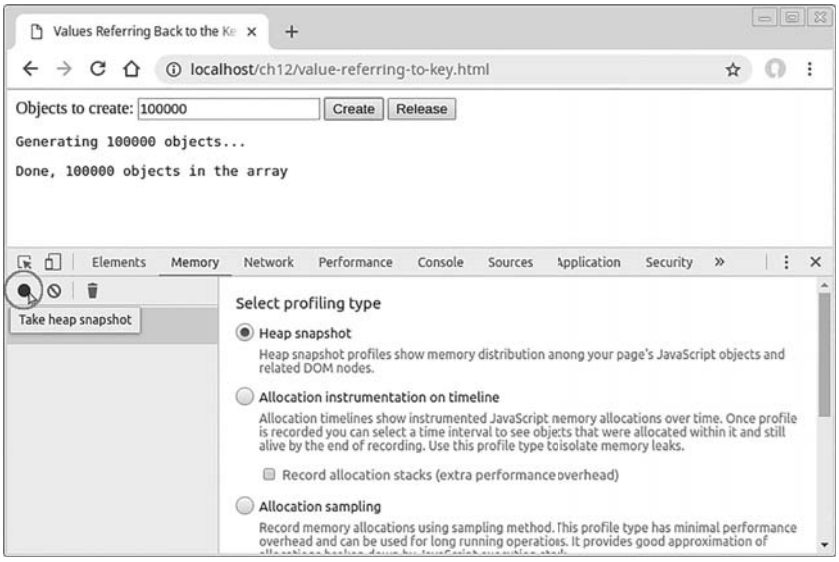


РИСУНОК 12-2

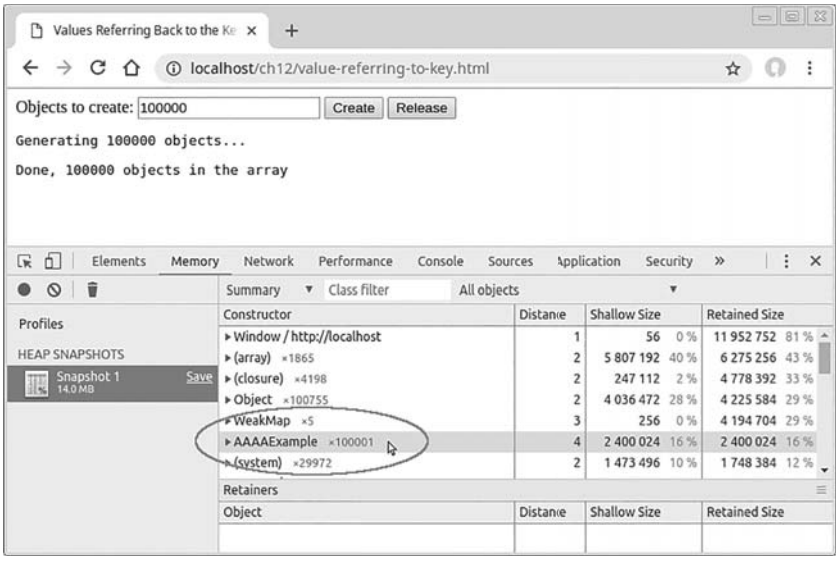


РИСУНОК 12-3

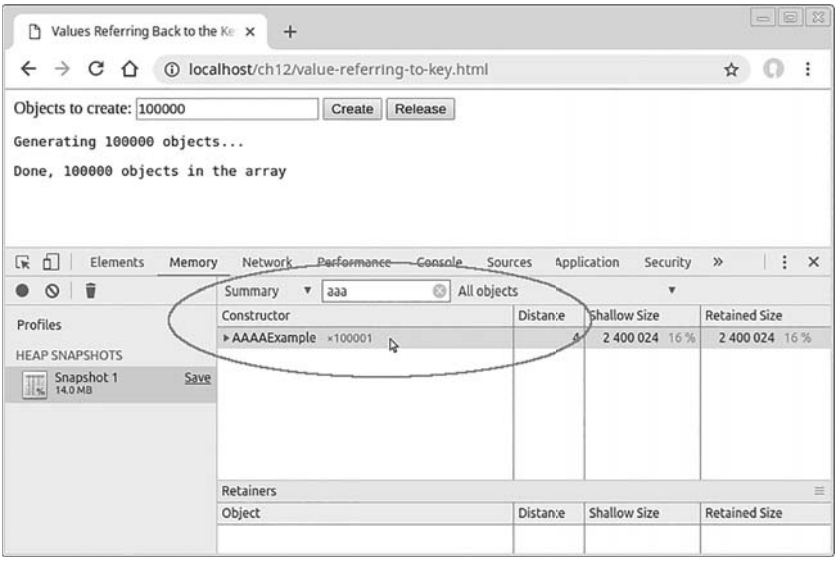


РИСУНОК 12-4

3. Нажмите кнопку **Release** (Освободить) на веб-странице; она удалит объекты из массива.
4. Используя любой предоставляемый вашим браузером механизм, принудительно выполните сборку мусора.
5. В Chrome используется кнопка **Collect Garbage** (Собрать мусор), показанная на рисунке 12-5.
6. Сделайте еще один снимок кучи (или выполните аналогичную операцию) и посмотрите, сколько объектов, созданных с помощью конструктора `AAAAExample`, сейчас существует. Чтобы сделать это в Chrome, вам нужно сделать еще один снимок, а затем снова заглянуть в панель **Constructor** (Конструктор). Если прокручивать страницу достаточно долго, можно найти в списке один объект `AAAAExample` (рисунок 12-6) или просто снова введите «aaa» в поле фильтра класса, чтобы просто отобразить строку `AAAAExample` (рисунок 12-7).

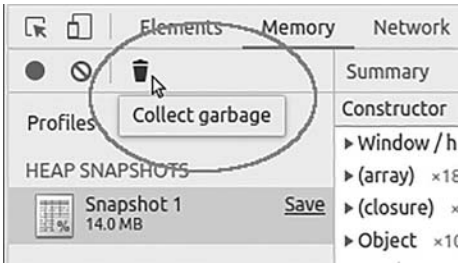


РИСУНОК 12-5

Это демонстрирует, что, хотя объекты значений ссылаются на свои ключевые объекты, они не препятствуют очистке этих записей. Фактически объект значения одной записи может ссылаться на ключевой объект другой записи, значение которого ссылается на ключевой объект первой записи (циклическая ссылка) или на другую запись. Но как только ничто за пределами любой слабой карты больше не ссылается на эти ключевые объекты, движок JavaScript удаляет записи и делает объекты доступными для сборки мусора.

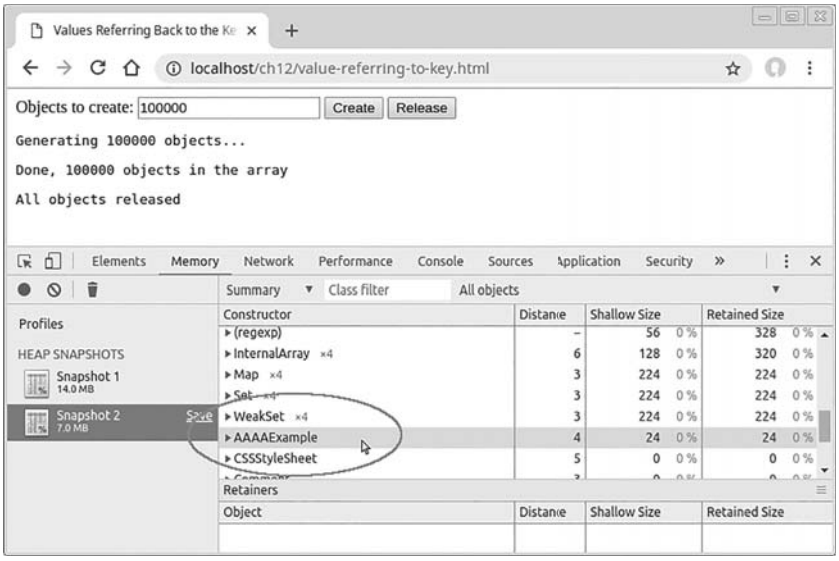


РИСУНОК 12-6

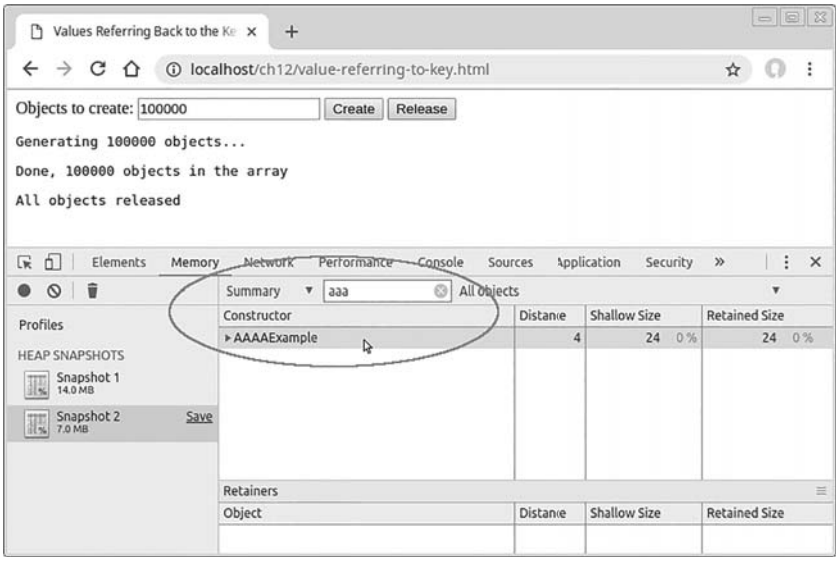


РИСУНОК 12-7

СЛАБЫЕ МНОЖЕСТВА (WEAKSET)

Слабые множества — это множества, эквивалентные слабым картам: множества объектов, нахождение которых в множестве не препятствует очистке объекта. Представьте себе слабые множества как слабые карты, где ключ — это значение в множестве, а значение (концептуально) `true`, что означает «да, объект находится в множестве». Или

представьте себе множества как карты, где ключ и значение — это одно и то же (как предполагает итератор `entries` множества); так тоже можно, хотя у слабого множества нет итератора `entries`.

Слабые множества не являются итерируемыми по той же причине, что и слабые карты. Они только предоставляют методы:

- `add`: добавляет объект в множество.
- `has`: проверяет, находится ли объект в множестве.
- `delete`: исключает (удаляет) объект из множества.

Следовательно, слабые множества — не подклассы множеств. Но, как и в случае со слабыми картами и картами, общие фрагменты их API намеренно согласованы.

Так в чем же смысл слабых множеств? Если нельзя получить доступ к объектам в нем, если у вас уже нет этих объектов, то какой цели он служит?

Ответ — аналогия в первом абзаце этого раздела: вы можете проверить, есть ли у вас объект в множестве. Это полезно для проверки того, что вы ранее видели объект в определенной части вашего кода (код, в котором вы добавили его в множестве). Давайте рассмотрим пару примеров этого общего варианта использования.

Вариант использования: Отслеживание

Предположим, что перед «использованием» объекта вам нужно знать, был ли этот объект когда-либо «использован» в прошлом, но без сохранения этих данных в качестве флага объекта (возможно, потому что флаг объекта может увидеть другой код; или потому, что это не ваш объект). Например, это может быть какой-то одноразовый токен доступа. Слабое множество — простой способ реализовать это, не заставляя объект оставаться в памяти, см. Листинг 12-7.

Листинг 12-7: Одноразовый объект — single-use-object.js

```
const SingleUseObject = (() => {
    const used = new WeakSet();

    return class SingleUseObject {
        constructor(name) {
            this.name = name;
        }
        use() {
            if (used.has(this)) {
                throw new Error(`${this.name} has already been used`);
            }
            console.log(`Using ${this.name}`);
            used.add(this);
        }
    };
})();
const suo1 = new SingleUseObject("suo1");
const suo2 = new SingleUseObject("suo2");
suo1.use();                                  // Применение suo1
try {
    suo1.use();
} catch (e) {
```

```
    console.error("Error: " + e.message); // Error: объект suo1 уже был
                                          // использован
}
suo2.use();                               // Применение suo2
```

Вариант использования: Маркировка

Маркировка — это еще одна форма отслеживания. Предположим, вы разрабатываете библиотеку, предоставляющую объекты, а затем принимающую эти объекты обратно, чтобы что-то с ними сделать. Если библиотеке нужно быть абсолютно уверенной, что возвращаемый объект был создан из кода библиотеки, что он не был подделан кодом, использующим библиотеку, она может использовать для этого слабые множества, см. листинг 12-8.

Листинг 12-8: Принимать только известные объекты — only-accept-known-objects.js

```
const Thingy = (() => {
    const known = new WeakSet();
    let nextId = 1;

    return class Thingy {
        constructor(name) {
            this.name = name;
            this.id = nextId++;
            Object.freeze(this);
            known.add(this);
    }

        action() {
            if (!known.has(this)) {
                throw new Error("Unknown Thingy");
            }
            // Здесь код узнает, что этот объект был создан этим классом
            console.log(`Action on Thingy #${this.id} (${this.name})`);
        }
    }
})();

// В другом коде, использующем его:

// Использование реального объекта
const t1 = new Thingy("t1");
t1.action(); // Действие с Thingy № 1 (t1)
const t2 = new Thingy("t2");
t2.action(); // Действие с Thingy № 2 (t2)

// Попытка использования поддельного объекта
const faket2 = Object.create(Thingy.prototype);
faket2.name = "faket2";
faket2.id = 2;
Object.freeze(faket2);
faket2.action(); // Error: Неизвестный Thingy
```

Класс `Thingy` создает объекты со свойствами `id` и `name`, а затем *замораживает* эти объекты. Поэтому их нельзя изменить никаким способом: их свойства доступны только для чтения и их нельзя изменить, свойства нельзя добавить или удалить, и прототип нельзя изменить.

Позже, если код попытается использовать метод `action` для объекта `Thingy`, метод `action` проверяет множество `known` на предмет того, что объект, к которому он был вызван, — подлинный объект `Thingy`. Если это не так, метод отказывается выполнять свое действие. Поскольку множество слабое, это не предотвращает сбор мусора из объектов `Thingy`.

ОТ СТАРЫХ ПРИВЫЧЕК К НОВЫМ

Перед вами несколько старых привычек, которые, возможно, вам захочется обновить.

Используйте карты вместо объектов для карт общего назначения

Старая привычка: Использовать объекты в качестве карт общего назначения:

```
const byId = Object.create(null); // Следовательно у него нет прототипа
for (const entry of entries) {
    byId[entry.id] = entry;
}
// Позже
const entry = byId[someId];
```

Новая привычка: Используйте карты:

```
const byId = new Map();
for (const entry of entries) {
    byId.set(entry.id, entry);
}
// Позже
const entry = byId.get(someId);
```

Карты лучше подходят для отображения общего назначения и не заставляют ключи быть строками, если они другого типа.

Используйте множества вместо объектов для множеств

Старая привычка: Использовать объекты в качестве псевдомножеств:

```
const used = Object.create(null);
// Пометка чего-то как "used"
used[thing.id] = true;
// Проверяет позже, было ли что-то использовано
if (used[thing.id]) {
    // ...
}
```

Новая привычка: Используйте множества:

```
const used = new Set();
// Пометка чего-то как "used"
used.add(thing.id);          // Или, возможно, просто использование `thing`
                             // непосредственно
// Проверяет позже, было ли что-то использовано
if (used.has(thing.id)) {   // Или, возможно, просто использование `thing`
                            // непосредственно
// ...
}
```

Или, если это уместно, слабое множество:

```
const used = new WeakSet();
// Пометка чего-то как "used"
used.add(thing);
// Проверяет позже, было ли что-то использовано
if (used.has(thing)) {
    // ...
}
```

Используйте слабые карты для хранения личных данных вместо публичных свойств

Старая привычка: Использовать соглашения об именовании, такого как префикс подчеркивания (_), чтобы указать, что свойство объекта приватное и не должно использоваться «другим» кодом:

```
class Example {
    constructor(name) {
        this.name = name;
        this._counter = 0;
}
    get counter() {
        return this._counter;
}
    incrementCounter() {
        return ++this._counter;
}
}
```

Новая привычка: (См. предостережения после примера.) Используйте слабые карты, чтобы данные были действительно приватными (но см. предостережение после кода):

```
const Example = (() => {
    const counters = new WeakMap();

    return class Example {
        constructor(name) {
            this.name = name;
            counters.set(this, 0);
        }
```

```
        get counter() {
            return counters.get(this);
        }
        incrementCounter() {
            const result = counters.get(this) + 1;
            counters.set(this, result);
            return result;
        }
    }
})();
```

Однако у этой новой привычки есть два предостережения:

- Использование слабых карт для этих задач увеличивает сложность кода. В любой конкретной ситуации выигрыш в конфиденциальности может стоить или не стоить затрат на сложность. Многие языки поддерживают «приватные» свойства, но затем в любом случае позволяют получить доступ к этим свойствам. Например, к приватным полям Java можно получить доступ с помощью рефлексии. Таким образом, простое использование соглашения об именовании на самом деле ненамного хуже, чем использование приватных свойств Java (хотя доступ к ним определенно проще, чем использование рефлексии Java для таких задач).
- В скором времени синтаксис `class` (по крайней мере) обеспечит возможность получать частные поля без использования слабых карт; об этом вы узнаете в главе 18. Отказ от привычки использовать слабую карту для приватных данных сейчас может просто настроить вас на дальнейший рефакторинг.

Это будет принцип от случая к случаю. Некоторая информация действительно должна быть должным образом закрытой (из кода; помните, что ничто не останется закрытым от отладчиков). Другая информация, вероятно, будет в сохранности, если вы просто пометите ее условным знаком с надписью «Не использовать».

13

Модули

СОДЕРЖАНИЕ ГЛАВЫ

- Введение в модули
- Инструкции `import` и `export`
- Как загружаются модули
- Динамические импорты: `import()`
- Встряхивание дерева
- Бандлеры
- Объект `import.meta`

В этой главе вы узнаете о модулях ES2015, позволяющих легко разделить код на небольшие поддерживаемые части и объединять их по мере необходимости.

ВВЕДЕНИЕ В МОДУЛИ

В течение многих лет программисты JavaScript либо просто помещали свой код в глобальное пространство имен, либо (поскольку глобальное пространство имен переполнено) помещали свой код в функцию-оболочку (функцию ограничения области действия). Иногда они заставляли эту функцию ограничения области возвращать объект (иногда называемый объектом пространства имен), который они присваивали одной глобальной переменной («паттерн открытый модуль»).

В проектах любого размера программисты могут столкнуться с проблемами, связанными с конфликтами имен, сложными зависимостями и разделением кода на файлы соответствующего размера. Эти проблемы привели к появлению различных (и несовместимых) решений для определения и объединения *модулей* кода, таких как CommonJS (CJS), Асинхронное определение модуля (AMD = Asynchronous Module Definition) и их вариаций. Множество несовместимых стандартов усложняют жизнь программистам, разработчикам инструментов, авторам библиотек и всем, кто пытается использовать модули из разных источников.

К счастью, ES2015 стандартизировал модули для JavaScript, предоставляя *в основном* общий синтаксис и семантику для использования инструментами и авторами. (Мы скоро вернемся к этому «в основном».) В дополнение к предыдущему списку сокращений нативные модули часто называют *модулями ESM* (ESM = **E**CMA**S**cript **M**odule), что отличает их от модулей CJS, AMD и других типов.

ОСНОВЫ МОДУЛЕЙ

В этом разделе представлен краткий обзор модулей, который в последующих разделах будет дополнен более подробной информацией.

Модуль — это единица кода в своей собственной «единице компиляции» (в широком смысле «файл»). У него есть своя собственная область видимости (вместо того, чтобы запускать свой код в глобальной области видимости, как это делают скрипты). Он может загружать другие модули, включая *импорт* объектов (таких как функции и переменные) из этих других модулей. Он может *экспортировать* данные для импорта другими модулями. Группа модулей, импортирующих экспорт друг друга, формирует граф, обычно называемый *деревом модулей* (рисунок 13-1).

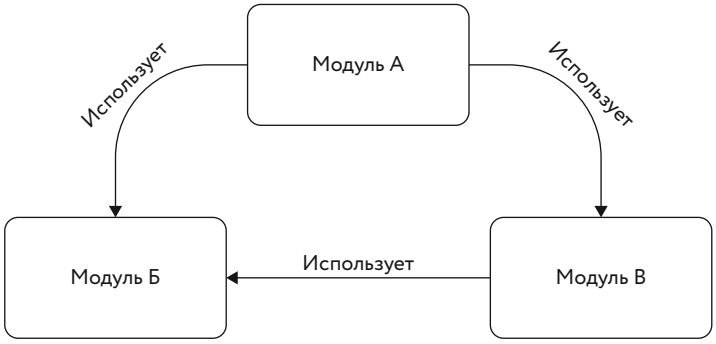


РИСУНОК 13-1

При импорте необходимо указать, из какого модуля производится импорт, используя *спецификатор модуля* — строку, которая указывает, где найти модуль. Движок JavaScript работает с хост-средой для загрузки указанного вами модуля. Движок JavaScript загружает модуль только один раз для каждой базы realm[72]. Если его используют несколько других модулей, все они используют одну и ту же его копию. У модуля может быть *именованный экспорт* и/или один отдельный *по умолчанию*.

[72] В главе 5 говорится, что *realm* — это общий контейнер, в котором находится фрагмент кода, состоящий из глобальной среды, внутренних объектов для этой среды (`Array`, `Object`, `Date` и т. д.), всего кода, загруженного в эту среду, и других битов состояния и т. п. Окно браузера (будь то вкладка, полное окно или iframe) — это база данных realm. Веб-воркер — это база realm, отдельная от базы realm окна, которое его создало.

Чтобы экспортировать что-либо из модуля, используется объявление экспорта, которое начинается с ключевого слова `export`. Вот примеры двух именованных экспортов:

```
export function example() {
    // ...
}
export let answer = 42;
```

Чтобы загрузить модуль и (необязательно) импортировать что-либо из него, используется либо объявление импорта, либо динамический импорт. О динамическом импорте рассказывается дальше в этой главе; сейчас давайте просто рассмотрим объявления импорта (также называемые статическим импортом). В этом случае используется ключевое слово `import`, за ним следует элемент для импорта, затем слово «from», затем строковый литерал спецификатора модуля, указывающий, из какого модуля импортировать, например:

```
import {example} from "./mod.js";
import {answer} from "./mod.js";
// или
import {example, answer} from "./mod.js";
```

Фигурные скобки в этих объявлениях показывают, что они импортируют именованный экспорт. (Подробности вы узнаете через мгновение.)

Вот экспорт по умолчанию (обратите внимание на слово `default`):

```
export default function example() {
    // ...
}
```

У модуля может быть только один экспорт по умолчанию (или ни одного). Его можно импортировать следующим образом (обратите внимание, что здесь нет фигурных скобок; опять же, вы узнаете подробности через мгновение):

```
import example from "./mod.js";
```

Также можно импортировать модуль только для его побочных эффектов, просто не перечисляя ничего для импорта из него:

```
import "./mod.js";
```

Этот код загружает модуль (и модули, от которых он зависит) и запускает его код, но ничего из него не импортирует.

Это наиболее распространенные формы `export` и `import`. Вы узнаете о некоторых дополнительных вариантах и подробностях позже в этой главе.

Только код в модулях, и никакой другой, может использовать эти декларативные формы `export` и `import`. Код вне модуля *может* использовать динамический импорт, о котором рассказывается позже, но не может ничего экспортировать.

Объявления `import` и `export` могут отображаться только в области верхнего уровня модуля; они не могут находиться внутри какой-либо структуры потока управления, такой как цикл или оператор `if`.

```
if (a < b) {
    import example from "./mod.js"; // SyntaxError: Неожиданный
идентификатор
    example();
}
```

Модуль может экспортировать только то, что он объявляет, или повторно экспортировать то, что он импортирует. Например, модуль не может экспортировать глобальную переменную, потому что он ее не объявлял. Модуль не может объявить глобальную переменную, потому что код модуля не выполняется в глобальной области видимости. (Он может добавить свойство к глобальному объекту с помощью присваивания (например, `window.varName = 42` в браузерах) или с помощью свойства `globalThis`, о котором вы узнаете в главе 17, — но это не объявление.)

Завершая этот вихревой обзор: импорт экспорта создает доступную только для чтения «живую» *косвенную привязку* к экспортируемому элементу. Такая привязка доступна только для чтения: модуль может считывать значение импортируемого элемента, в том числе видеть новое значение, если исходный модуль изменяет значение элемента, но не может изменить значение напрямую.

Фух! Давайте рассмотрим эти вопросы более внимательно.

Спецификатор модуля

В предыдущих примерах часть `"./mod.js"` — это спецификатор модуля:

```
import {example} from "./mod.js";
```

При объявлении импорта спецификатор модуля должен быть строковым *литералом*, а не просто выражением, приводящим к строке, поскольку объявления являются статическими формами (движок JavaScript и среда должны иметь возможность интерпретировать их без *запуска* кода). Можно использовать одинарные или двойные кавычки. Кроме того, в спецификации JavaScript почти ничего не говорится о спецификаторах модулей. Она оставляет форму и семантику строк спецификатора модуля на откуп средам размещения. (Именно поэтому во введении использовалось выражение «в основном» в предложении «*…в основном* общий синтаксис и семантику…».) Форма и семантика спецификаторов модулей для Интернета определяются спецификацией HTML, спецификаторы для Node.js определяются с помощью Node.js и т. д. Тем не менее команды, работающие со спецификаторами для основных сред, подобных этим, осознают, что модули создаются для использования в разных средах, и пытаются избежать ненужных различий. На данный момент не задумывайтесь о спецификаторах модулей; вы узнаете о них больше в разделах «Использование модулей в браузерах» и «Использование модулей в Node.js» далее в этой главе.

Базовый именованный экспорт

Вы уже видели пару базовых именованных экспортеров. В этом разделе о них рассказывается больше.

Можно использовать именованный экспорт для экспорта всего, что объявлено в модуле и получило имя: переменных, констант, функций, конструкторов классов и объектов, импортированных из других модулей. Есть несколько различных способов реализовать это. Один из способов — при объявлении и определении просто поместить ключевое слово `export` перед тем, что необходимо экспортировать:

```
export let answer = 42;
export var question = "Life, the Universe, and Everything";
export const author = "Douglas Adams";
export function fn() {
    // ...
}
export class Example {
    // ...
}
```

Этот код создает пять именованных экспортов: `answer`, `question`, `author`, `fn` и `Example`. Обратите внимание, что каждое из этих утверждений/объявлений выглядит точно так, как выглядело бы без экспорта, просто с надписью `export` в начале. Объявление функции и объявление класса по-прежнему являются объявлениями, а не выражениями (и поэтому после них нет точек с запятой).

Другие модули используют эти имена, чтобы указать, что они хотят импортировать из модуля. Например:

```
import {answer, question} from "./mod.js";
```

В этом примере модулю требуется только экспорты `answer` и `question`, поэтому он запрашивает только их, и не запрашивает `author`, `fn` или `Example`.

Из введения следует, что фигурные скобки указывают на именованный экспорт, а не на экспорт по умолчанию. Хотя эти фигурные скобки делают эту часть выражения `import` немного похожей на деструктуризацию объекта (глава 7), это не деструктуризация. Синтаксис импорта полностью отделен от деструктуризации. Он не допускает вложенности, обрабатывает переименование иначе, чем деструктуризацию, и не допускает значений по умолчанию. Это разные вещи с внешне похожим синтаксисом.

Нет необходимости экспортировать что-то в том же месте, где оно создается; можно использовать отдельное объявление экспорта. Вот тот же модуль, что и ранее, использующий отдельное объявление экспорта (в данном случае в конце):

```
let answer = 42;
var question = "Life, the Universe, and Everything";
const author = "Douglas Adams";
function fn() {
    // ...
}
```

```
class Example {
    // ...
}
export {answer, question, author, fn, Example};
```

У вас может получиться несколько отдельных объявлений, например:

```
export {answer};
export {question, author, fn, Example};
```

хотя основной вариант использования отдельного объявления предназначен для стилей кодирования, группирующих все экспортные данные в одном месте.

Объявления экспорта могут находится в конце, в начале или где-то посередине. Вот пример размещения одного объявления в начале:

```
export {answer, question, author, fn, Example};
let answer = 42;
const question = "Life, the Universe, and Everything";
const author = "Douglas Adams";
function fn() {
    // ...
}
class Example {
    // ...
}
```

Стили можно смешивать, хотя это, вероятно, не лучшая практика:

```
export let answer = 42;
const question = "Life, the Universe, and Everything";
const author = "Douglas Adams";
export function fn() {
    // ...
}
class Example {
    // ...
}
export {question, author, Example};
```

Единственное реальное ограничение — имена экспортов должны быть уникальными: нельзя экспортировать `answer` в предыдущем примере дважды, используя имя `answer` для обоих, или попробовать экспортировать переменную с именем `answer`, а также функцию с именем `answer`. Тем не менее *возможно* (хоть и необычно) экспортировать одно и то же дважды, либо переименовав один из экспортов (подробнее об этом в следующем разделе), либо экспортировав что-либо как в качестве именованного экспорта, так и в качестве экспорта по умолчанию. Например, вы можете экспортировать функцию под ее основным именем, а также под псевдонимом.

То, что вы используете — встроенное объявление в конце, объявление в начале или некоторая комбинация, — полностью зависит от стиля. Как обычно, лучше всего быть последовательным в рамках кодовой базы.

Экспорт по умолчанию

Помимо именованного экспорта, у модуля также может дополнительно быть отдельный экспорт по умолчанию. Он аналогичен именованному экспорту с использованием имени `default`, но у него нет собственного выделенного синтаксиса (и не создается локальная привязка с именем `default`). Вы устанавливаете экспорт по умолчанию, добавляя ключевое слово `default` в конце выражения

```
export default function example() {
    // ...
}
```

Другой модуль импортирует его, используя любое понравившееся имя без фигурных скобок:

```
import x from "./mod.js";
```

Если у того, что вы экспортируете, есть имя, оно не используется при экспорте, поскольку эта форма экспортирует значение по умолчанию. Например, при предыдущем экспорте функции с именем `example` другой код может импортировать ее таким образом:

```
import example from "./mod.js";
// или
import ex from "./mod.js"; // Имя импорта не обязательно должно быть "example"
```

но не так, поскольку это не именованный экспорт:

```
// Неверно (для импорта экспорта по умолчанию)
import {example} from "./mod.js";
```

Применение более одного `export default` в модуле является ошибкой: модуль может содержать только один экспорт по умолчанию.

Вы можете объявить что-то, а затем экспортировать его как значение по умолчанию отдельно:

```
function example() {
}
export default example;
```

Существует вторая форма экспорта по умолчанию, позволяющая экспортировать результат любого произвольного выражения:

```
export default 6 * 7;
```

ЭКСПОРТ ОБЪЯВЛЕНИЯ АНОНИМНОЙ ФУНКЦИИ ИЛИ КЛАССА

Это скорее просто любопытство, но обратите внимание, что это:

```
export default function() {/*...*/}
```

экспортируемое объявление функции, а не выражение функции. Экспорт по умолчанию — это единственное место, где можно реализовать объявление функции без имени; результирующая функция получит имя `default`. Поскольку это объявление функции, оно поднимается, как и все другие объявления функций (функция создается до начала пошагового выполнения модуля), хотя это редко имеет значение.

Можно также реализовать анонимное объявление `class` — опять же, только при выполнении экспорта по умолчанию:

```
export default class {/*...*/}
```

Но поскольку это объявление `class`, класс не создается до момента выполнения кодом объявления, до тех пор оно находится в TDZ. (Подробнее о том, как TDZ применяется к модулям, читайте далее в этой главе.)

Эта форма экспорта по умолчанию в точности похожа на экспорт `let`, только без идентификатора, который код модуля мог бы использовать для доступа к нему. Если бы вы могли вызвать переменную `*default*`, то выражение

```
export default 6 * 7;
```

будет идентичным следующему:

```
// Концептуальное представление, недопустимый синтаксис
let *default* = 6 * 7;
export default *default*;
```

но с `*default*`, недоступным для кода модуля. (Вы узнаете, почему я использовал `*default*` позже, когда мы поговорим о привязках.)

Несмотря на то что технически в спецификации есть это произвольное выражение для создания эквивалента переменной `let` (а не `const`), код не может получить ее значение, поэтому оно фактически будет постоянным.

Поскольку эта форма вычисляет выражение, как и экспорт `let`, оно не содержит значения до тех пор, пока в пошаговом выполнении кода не будет достигнуто объявление экспорта. До тех пор оно будет в TDZ.

Использование модулей в браузерах

В приложении с веб-страницей обычно используется один модуль точки входа, чтобы начать загрузку дерева модулей (хотя возможно, что их несколько). Модуль точки входа импортирует данные из других модулей (способных, в свою очередь, импортировать данные из других модулей и т. д.). Чтобы сообщить браузеру, что код элемента `script` представлен модулем, используется выражение `type="module"`:

```
<script src="main.js" type="module" ></script>
```

Давайте рассмотрим простой пример на основе браузера; см. Листинги 13-1, 13-2 и 13-3. Вы можете запустить этот пример, используя файлы из загрузок в любом современном браузере.

Листинг 13-1: Простой пример модуля (HTML) — simple.html

```
<!doctype html>
<html>
<head>
<meta charset="UTF-8">
<title>Simple Module Example</title>
</head>
<body>
<script src="./simple.js" type="module"></script>
</body>
</html>
```

Листинг 13-2: Простой пример модуля (модуль точки входа) — simple.js

```
import {log} from "./log.js";

log("Hello, modules!");
```

Листинг 13-3: Простой пример модуля (модуль log) — log.js

```
export function log(msg) {
    const p = document.createElement("pre");
    p.appendChild(document.createTextNode(msg));
    document.body.appendChild(p);
}
```

При загрузке `simple.html` в браузере, браузер и движок JavaScript работают вместе для загрузки модуля `simple.js`, определяя его зависимость от `log.js`. Загружают и запускают `log.js`, затем запускают код из `simple.js`, который использует функцию `log` из `log.js` для вывода сообщения.

Обратите внимание, что файл **log.js** нигде не указан в HTML-коде. Тот факт, что `simple.js` зависит от него, указывается инструкцией `import`. *Можно* было бы при желании добавить тег `script` для `log.js` перед тегом для `simple.js`, чтобы начать процесс его загрузки раньше, но обычно в этом нет необходимости. (Даже если сделать так, `log.js` все равно будет загружен только один раз.)

Скрипты модуля не задерживают синтаксический анализ

Элемент `script` с выражением `type="module"` не задерживает синтаксический анализ HTML-кода, в то время как скрипт извлекается и выполняется в случае с немодульным тегом `script`. Вместо этого модуль и его зависимости загружаются параллельно с синтаксическим анализом; затем код модуля запускается, когда синтаксический анализ завершен (или когда движок завершает загрузку — в зависимости от того, что происходит последним).

Если это звучит знакомо, то потому, что именно так обрабатывается элемент `script` с атрибутом `defer`. По сути, элементы с выражением `script type="module"` неявно содержат атрибут `defer`. (Указание его явно не имеет никакого эффекта.) Однако есть одно отличие: атрибут `defer` откладывает только скрипты, загружающие свое содержимое с внешнего ресурса через атрибут `src`, а не теги скриптов со встроенным содержимым. А выражение `type="module"` также откладывает скрипты со встроенным содержимым.

Обратите внимание, что, хотя код модуля не запускается до завершения синтаксического анализа HTML, браузер и движок JavaScript работают вместе, чтобы определить зависимости модуля (из его объявлений `import`) и извлекать любые модули, от которых он зависит, параллельно с синтаксическим анализом HTML. Код модуля не должен выполняться для загрузки его зависимостей, поскольку модули поддаются статическому анализу.

Как и в случае с немодульными тегами `script`, можно включить атрибут `async`, чтобы браузер запускал код модуля, как только он будет готов к запуску, даже если синтаксический анализ HTML еще не завершен. Вы можете увидеть, как различные теги `script` обрабатываются на рисунке 13-2, в основе которого лежит схема спецификации WHATWG HTML в разделе, описывающем атрибуты `async` и `defer`[73].

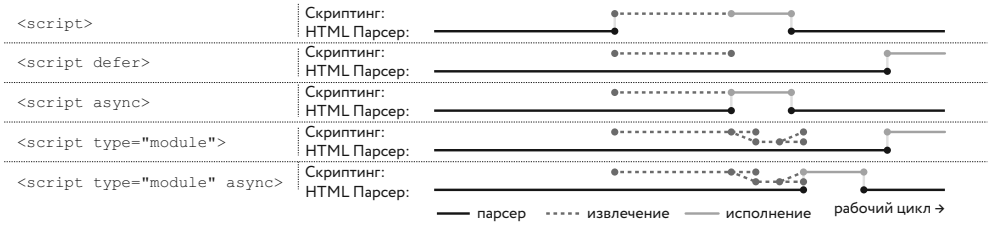


РИСУНОК 13-2

Атрибут `nomodule`

Если вы хотите предоставлять модули в браузерах, которые их поддерживают, и предоставлять немодульные скрипты в браузерах, которые этого не делают (например Internet Explorer), вы можете использовать атрибут `nomodule` для немодульных скриптов. Например:

```
<script type="module" src="./module.js"></script>
<script nomodule src="./script.js"></script>
```

[73] https://html.spec.whatwg.org/multipage/scripting.html#attr-script-defer

К сожалению, Internet Explorer загрузит *оба* файла, хотя будет запускать только файл **script.js.** Safari также делал это в нескольких версиях, как и Edge (даже после того, как в него добавили поддержку модулей), но теперь оба браузера исправлены.

Чтобы обойти это, можно определить модуль/скрипт для загрузки, используя вместо этого встроенный код:

```
<script type="module">
import "./module.js";
</script>
<script nomodule>
document.write('<script defer src="script.js"><\/script>');
</script>
```

(Если вам, как и многим, не нужен метод `document.write`, вы можете всегда использовать методы `createElement` и `appendChild`.)

Спецификаторы модулей в Интернете

На момент написания этой книги спецификация HTML определяет спецификаторы модулей довольно узко, чтобы их можно было расширять с течением времени: спецификаторы модулей — это либо абсолютные URL-адреса, либо относительные URL-адреса, начинающиеся с `/` (слеш или «солидус», как он называется в спецификации), `./` (точка, слеш), или `../` (точка, точка, слеш). Как и URL-адреса в файлах CSS, относительные URL-адреса разрешаются относительно ресурса, в котором находится объявление `import`, а не базового документа. Модуль идентифицируется по его *разрешенному* URL-адресу (таким образом, все разные относительные пути к модулю разрешаются к одному и тому же модулю).

Все спецификаторы в следующем импорте допустимые:

```
import /*...*/ "http://example.com/a.js"; // Абсолютный URL
import /*...*/ "/a.js";                   // Начинается с /
import /*...*/ './b.js';                  // Начинается с ./
import /*...*/ "../c.js";                 // Начинается с ../
import /*...*/ './modules/d.js';          // Начинается с ./
```

Спецификаторы в следующем импорте в настоящее время недопустимы в браузерах, поскольку они не относятся к абсолютным и не начинаются с `/`, `./` или `../` (но это меняется, подробнее ниже):

```
import /*...*/ "fs";
import /*...*/ "f.js";
```

Для модуля, находящегося в том же расположении, что и импортирующий его модуль, требуется префикс `./`, а не просто голое имя (`"./mod.js"`, а не просто `"mod.js"`). Вот почему значение простого имени, такого как `"mod.js"` или `"mod"`, может быть определено позже. Одно из текущих предложений, импорт карт[74], позволяет странице определять сопоставление спецификаторов модулей с URL-адресами, чтобы содержимое

[74] `https://wicg.github.io/import-maps/`

модулей могло использовать голые имена, обеспечивая при этом гибкость содержащей страницы с точки зрения фактического расположения этих модулей. Это предложение все еще находится в стадии разработки, но, похоже, будет продвигаться.

Спецификатор должен включать расширение файла, если вы используете его для своих файлов; расширение или расширения по умолчанию отсутствуют. Используемое расширение зависит от вас, как и в случае с немодульными скриптами. Многие программисты придерживаются.`js`. Однако некоторые выбирают.`mjs`, чтобы отметить тот факт, что файл содержит модуль, а не просто скрипт. Если вы это сделаете, обязательно настройте свой веб-сервер для обслуживания файла с правильным типом MIME `text/javascript`, поскольку у модулей нет своего собственного типа MIME. Как бы то ни было, я считаю, что модули — это новая норма, и поэтому придерживаюсь расширения.`js`.

Помимо следующего раздела, посвященного Node.js, все спецификаторы, показанные в этой главе, определены для Интернета спецификацией HTML.

Использование модулей в Node.js

Платформа Node.js поддерживает собственные модули JavaScript (ESM) в дополнение к оригинальным модулям, подобным CommonJS (CJS)[75]. На момент написания книги поддержка ESM все еще была помечена как «экспериментальная», но она уже довольно далеко продвинулась. В версии v12 (включая выпуск статуса долгосрочного v12) он находится за флагом `--experimental-modules`, но больше не находится за флагом в версии v13 и выше.

ИЗМЕНЕНИЯ МЕЖДУ V8 И V11, V12 И ВЫШЕ

Поддержка ESM платформой Node.js в версиях с v8 по v11 была преимущественно основана на расширениях файлов, в этом разделе описывается более новое поведение в версии 12 и выше.

Поскольку Node.js уже давно поддерживает свою собственную модульную систему на основе CJS, вам необходимо зарегистрироваться при использовании ESM. Это можно сделать одним из трех способов:

1. Разместить `package.json` в своем проекте с полем `type`, содержащим значение `"module"`:

```
{
    "name": "mypackage",
    "type": "module",
    ...other usual fields...
}
```

[75] https://nodejs.org/api/esm.html

2. Использовать.mjs в качестве расширения для файлов вашего модуля ESM.
3. При передаче строки в node (в качестве аргумента для --eval, --print или введя текст в node) указать --input-type=module.

Это относится как к точке входа, которую вы предоставляете в командной строке node, так и к любым модулям, попадающим в код ESM через import.

См. Листинги 13-4, 13-5 и 13-6 для изучения примера с package.json с полем "type": "module". Вы бы запустили пример с v12 так:

```
node --experimental-modules index.js
```

С v13 и выше флаг больше не нужен:

```
node index.js
```

Листинг 13-4: Простой пример модуля Node.js (основная точка входа) — index.js

```
import {sum} from "./sum.js";

console.log(`1 + 2 = ${sum(1, 2)}`);
```

Листинг 13-5: Простой пример модуля Node.js (модуль sum) — sum.js

```
export function sum(...numbers) {
    return numbers.reduce((a, b) => a + b);
}
```

Листинг 13-6: Простой пример модуля Node.js (пакет) — package.json

```
{
    "name": "modexample",
    "type": "module"
}
```

В качестве альтернативы, если удалить поле type из файла **package.json** (или изменить его на «commonjs»), можно использовать ESM, изменив имена на index.mjs и sum.mjs (в названиях файлов и в инструкции import в index).

По умолчанию при импорте модуля из файла с помощью ESM необходимо указать расширение файла в инструкции import. (Подробнее в разделе «Спецификаторы модулей в Node.js».) Обратите внимание, что это в Листинге 13-4 указано как ./sum.js, а не просто ./sum. При импорте встроенных пакетов, таких как fs или пакеты в node_modules, расширение не используется. Используется просто имя модуля:

```
import fs from "fs";

fs.writeFile("example.txt", "Example of using the fs module\n", "utf8",
err => {
    // ...
});
```

Все встроенные модули также предоставляют именованный экспорт, поэтому предыдущий код можно было бы написать так:

```
import {writeFile} from "fs";

writeFile ("example.txt", "Example of using the fs module\n", "utf8", err => {
    // ...
});
```

Независимо от того, используете ли вы `"type": "module"` в файле **package.json**, при импорте файла с расширением **.cjs**, Node.js загрузит его как модуль CJS:

```
import example from "./example.cjs"; // example.cjs должен быть модулем CJS
```

Модули ESM могут импортировать из модулей CJS: значение `exports` в модуле CJS обрабатывается как экспорт по умолчанию. Например, если `mod.cjs` содержит

```
exports.nifty = function() {};
```

модуль ESM импортировал бы его следующим образом:

```
import mod from "./mod.cjs";
```

Затем либо используйте `mod.nifty`, либо присвоение деструктуризации после импорта, чтобы получить `nifty` отдельно:

```
import mod from "./mod.cjs";
const {nifty} = mod;
```

Этот код не может использовать именованную форму экспорта `import {nifty}`, потому что это статическое объявление импорта, а это значит, оно должно быть статически анализируемым. Но экспорт модуля CJS динамический, а не статический: указывается экспорт CJS, присваивая объекту `exports` код рабочего процесса. Поддержка именованного импорта из CJS потребует изменения спецификации JavaScript, чтобы разрешить динамический именованный экспорт. Продолжаются дискуссии о том, следует ли и как это исправить, но на данный момент (или, возможно, навсегда) значение CJS `exports` поддерживается только как экспорт по умолчанию. (Это верно даже при использовании динамического импорта, о котором рассказывается в следующем разделе.)

Модули CJS можно импортировать из модулей ESM только с помощью динамического импорта.

Спецификаторы модулей в Node.js

При импорте файла модуля спецификаторы модуля в Node.js очень похожи на определенные для Интернета по умолчанию: абсолютные или относительные имена файлов (а не URL-адреса) — и обязательно расширение файла. Часть с расширением файла отличается от загрузчика модуля CJS в Node.js, который позволяет отказаться

от расширения, а затем проверяет наличие файла с различными расширениями (.js,.json и т. д.). Однако вы можете включить это поведение, используя аргумент командной строки. В v12 это делается следующим образом:

```
node --experimental-modules --es-module-specifier-resolution=node index.js
```

В v13 флаг немного отличается (и не требуется флаг модулей):

```
node --experimental-specifier-resolution=node index.js
```

Если бы мы использовали флаг, то в инструкции `import` из Листинга 13-4 могло бы отсутствовать расширение.`js`:

```
import {sum} from "./sum"; // Если режим разрешения узла включен с помощью флага
```

Как показано ранее в примере с использованием модуля `fs`, используются пустые имена при импорте встроенных пакетов или установленных в `node_modules` пакетов:

```
import {writeFile} from "fs";
```

Node.js добавляет дополнительные возможности модулей

Платформа Node.js не просто придерживается свойства `type` в файле **package.json** и основных функций, описанных здесь. Ведется множество работ: экспорт карт, различные возможности пакетов и т. д. Большинство из них все еще находятся на относительно ранних стадиях на момент написания этой книги, поэтому я не буду вдаваться в подробности, поскольку детали пока еще плавающая цель. Вы можете узнать о них на сайте Node.js в разделе «Модули ECMAScript» (https://nodejs.org/api/esm.html).

ПЕРЕИМЕНОВАНИЕ ЭКСПОРТА

Экспортируемый из модуля идентификатор не обязательно должен совпадать с идентификатором, используемым для него в коде вашего модуля. Можно переименовать экспорт, используя предложение `as` в объявлении экспорта:

```
let nameWithinModule = 42;
export {nameWithinModule as exportedName};
```

Можно импортировать это в другой модуль, используя имя `exportedName`:

```
import {exportedName} from "./mod.js";
```

Имя `nameWithinModule` предназначено только для использования *внутри* модуля, а не за его пределами.

Вы можете переименовывать экспорт только при применении отдельного экспорта, а не при встроенном экспорте (поэтому не при выполнении выражения `export let nameWithinModule` или аналогичного).

Если необходимо создать псевдоним (например, для такой ситуации, как `trimLeft`/`trimStart`, о которой говорится в главе 10), можно провести встроенный экспорт с одним именем и затем использовать экспорт с переименованием:

```
export function expandStart() {      // expandStart - первичное имя
    // ...
}
export {expandStart as expandLeft}; // expandLeft - псевдоним
```

Этот код для одной функции создает два экспорта: `expandStart` и `expandLeft`.

Можно также использовать синтаксис переименования для создания экспорта по умолчанию, хотя это не лучшая практика:

```
export {expandStart as default}; // Не лучший способ
```

Как вы уже знаете, экспорт по умолчанию аналогичен именованному экспорту с использованием экспортируемого имени `default`, так что это эквивалентно форме, определенной по умолчанию (показана ранее):

```
export default expandStart;
```

ПОВТОРНЫЙ ЭКСПОРТ ЭКСПОРТА ИЗ ДРУГОГО МОДУЛЯ

Модуль может повторно экспортировать экспорт другого модуля:

```
export {example} from "./example.js";
```

Это называется *косвенным экспортом*.

Одно из мест, где это полезно — «сводные» модули. Предположим, вы пишете библиотеку манипуляций с DOM, такую как jQuery. Хотя можно написать ее как единый массивный модуль, содержащий все, что предоставляет библиотека, это означало бы, что любой код, использующий библиотеку, должен ссылаться на этот один модуль, который переносит весь код модуля в память, даже если он не будет использоваться весь. (Возможно, но позже, когда мы поговорим о *встряхивании дерева*.) Вместо этого решения можно разбить библиотеку на более мелкие части, из которых пользователи библиотеки могли бы импортировать требующийся код. Например:

```
import {selectAll, selectFirst} from "./lib-select.js";
import {animate} from "./lib-animate.js";
// ...код с использованием selectAll, selectFirst и animate...
```

Затем для проектов, которые, вероятно, будут использовать все функции библиотеки или которым не требуется загрузка всей библиотеки в память, библиотека может предоставить сводный модуль со всеми его частями, собранными вместе, как `lib.js`:

```
export {selectAll, selectFirst, selectN} from "./lib-select.js";
export {animate, AnimationType, Animator} from "./lib-animate.js";
export {attr, hasAttr} from "./lib-manipulate.js";
// ...
```

Код тогда сможет просто использовать `lib.js` при импорте всех этих функций:

```
import {selectAll, selectFirst, animate} from "./lib.js";
// ...код с использованием selectAll, selectFirst и animate...
```

Однако явное перечисление этих экспортов приводит к проблеме обслуживания: если вы добавляете новый экспорт в `lib-select.js` (например), легко забыть обновить `lib.js` для экспорта нового элемента. Чтобы избежать этого, существует специальная форма со звездочкой (`*`), в которой говорится «экспортировать все именованные экспортные данные из этого другого модуля»:

```
export * from "./lib-select.js";
export * from "./lib-animate.js";
export * from "./lib-manipulate.js";
// ...
```

Выражение `export *` не выдает экспорт модуля по умолчанию, только именованные экспорты.

Повторный экспорт только создает новый экспорт, но не импортирует элемент в область модуля:

```
export {example} from "./mod.js";
console.log(example); // ReferenceError: example не определено
```

Если требуется импортировать элемент, а также экспортировать его, каждая операция выполняется отдельно:

```
import {example} from "./mod.js";
export {example};
console.log(example);
```

При повторном экспорте вы можете изменить имя экспорта, используя ключевое слово `as`:

```
export {example as mod1_example} from "./mod1.js";
export {example as mod2_example} from "./mod2.js";
```

Как почти буквально сказано в синтаксисе, этот код повторно экспортирует `example` из `mod1.js` в виде `mod1_example`, и `example` из `mod2.js` в виде `mod2_example`.

Существует еще один способ повторного экспортирования экспорта другого модуля, о котором вы узнаете в разделе «Экспорт объекта пространства имен другого модуля» далее в этой главе.

ПЕРЕИМЕНОВАНИЕ ИМПОРТА

Предположим, у вас есть два модуля, экспортирующих функцию `example`, и требуется использовать обе эти функции. Или имя, используемое модулем, конфликтует с чем-то в вашем коде. Если вы читали предыдущие разделы о переименовании экспорта, вы,

вероятно, догадались, что импорт можно переименовать таким же образом при помощи предложения `as`:

```
import {example as aExample} from "./a.js";
import {example as bExample} from "./b.js";

// Применение импортов
aExample();
bExample();
```

При импорте экспорта по умолчанию, как вы уже узнали, всегда нужно выбрать свое собственное имя, поэтому предложение `as` неуместно:

```
import someName from "./mod.js";
import someOtherName from "./mod.js";
console.log(someName === someOtherName); // истина
```

Как и в случае с формой переименования `export`, при желании можно использовать форму переименования `import` для импорта экспорта по умолчанию, хотя это не лучшая практика:

```
import {default as someOtherName}; // Не лучший сопособ
```

ИМПОРТ ОБЪЕКТА ПРОСТРАНСТВА ИМЕН МОДУЛЯ

Вместо (или в дополнение) импорта отдельных экспортов можно импортировать объект пространства имен модуля для всего модуля. Объект пространства имен модуля — это объект со свойствами для всего экспорта модуля (если есть экспорт по умолчанию, имя его свойства, конечно, `default`). Так что, если у вас есть `module.js`:

```
export function example() {
    console.log("example called");
}
export let something = "something";
export default function() {
    console.log("default called");
}
```

можно импортировать объект пространства имен модуля, используя форму объявления импорта, которая раньше не описывалась:

```
import * as mod from "./module.js";
mod.example();              // вызов example
console.log(mod.something); // something
mod.default();              // вызов default
```

Выражение `* as mod` означает импортировать объект пространства имен модуля и связать его с локальным идентификатором (биндинг) `mod`.

Объект пространства имен модуля — это не то же самое, что сам модуль. Это отдельный объект, создаваемый при первом запросе (никогда не создается вообще, если его

ничто не запрашивает) со свойствами для всего экспорта модуля. Значения свойств обновляются динамически по мере того, как исходный модуль изменяет значения своего экспорта (если он это делает). После создания объект пространства имен модуля используется повторно, если другие модули также импортируют объект пространства имен модуля. Объект доступен только для чтения: его свойства нельзя записать или добавить новые.

ЭКСПОРТ ОБЪЕКТА ПРОСТРАНСТВА ИМЕН ДРУГОГО МОДУЛЯ

Начиная с нормативного изменения в ES2020, ускорившего разработку предложения[76], модуль может предоставлять экспорт, преобразуемый в объект пространства имен другого модуля. Предположим, у вас есть этот экспорт в `module1.js`:

```
// В module1.js
export * as stuff from "./module2.js";
```

Этот код создает именованный экспорт в `module1.js` и называет его `stuff` при импорте, и импортирует объект пространства имен модуля из `module2.js`. Это означает, что, скажем, в `module3.js` этот импорт:

```
// В module3.js
import {stuff} from "./module1.js";
```

и этот импорт:

```
// В module3.js
import * as stuff from "./module2.js";
```

сделают одно и то же — импортируют объект пространства имен модуля из `module2.js`, при необходимости создадут его и привяжут к локальному имени `stuff` в `module3.js`.

Как и в случае с формами, которые вы изучили ранее в разделе «Повторный экспорт экспорта из другого модуля», это иногда полезно при создании сводных модулей. Это также улучшило симметрию между формами `import` и `export...from`. Так же, как и у формы импорта

```
import {x} from "./mod.js";
```

есть соответствующая форма `export...from`

```
export {x} from "./mod.js";
```

и у формы `import`

```
import {x as v} from "./mod.js";
```

есть соответствующая форма `export...from`

[76] `https://github.com/tc39/proposal-export-ns-from`

```
export {x as v} from "./mod.js";
```

у формы `import`

```
import * as name from "./mod.js";
```

теперь есть соответствующая форма `export...from`

```
export * as name from "./mod.js";
```

ИМПОРТ МОДУЛЯ ТОЛЬКО ИЗ-ЗА ПОБОЧНЫХ ЭФФЕКТОВ

Вы можете импортировать модуль, не импортируя из него ничего, просто чтобы загрузить и запустить его:

```
import "./mod.js";
```

Предположим, что импорт находится в модуле `main.js`. Это условие добавляет `mod.js` в список требуемых `main.js` модулей, но ничего из него не импортирует. Загрузка модуля `mod.js` запускается на верхнем уровне кода (после загрузки всех модулей, от которых он зависит), так что это полезно, когда вам нужно только импортировать модуль для каких-либо побочных эффектов его кода верхнего уровня.

В общем, лучше всего, если у кода верхнего уровня модуля нет никаких побочных эффектов. Но для случайного случая использования, когда уместно создание исключения, это обеспечивает способ запуска кода верхнего уровня модуля.

Ранее в разделе, посвященном атрибуту `nomodule`, вы кратко рассмотрели один возможный вариант использования — предоставление модуля точки входа для браузеров, поддерживающих модули, и немодульной точки входа для не поддерживающих браузеров, без того, чтобы заставлять некоторые браузеры (Internet Explorer, некоторые старые версии Safari и Edge и т. д.) загружать как модульный, так и немодульный код, даже если они будут выполнять только один из них.

ИМПОРТ И ЭКСПОРТ ЗАПИСЕЙ

Когда движок JavaScript анализирует модуль, он создает список *записей импорта* модуля (то, что он импортирует) и список его *записей экспорта* (то, что он экспортирует). Списки предоставляют спецификации возможность описать, как происходит загрузка и связывание модулей (о чем вы узнаете подробнее позже). Ваш код не может получить прямой доступ к этим спискам, но давайте рассмотрим их: они помогут вам понять последующие разделы.

Импорт записей

Список импортируемых записей модуля описывает, что он импортирует. Каждая запись импорта содержит три поля:

- *[[ModuleRequest]]*: строка спецификатора модуля из объявления импорта. Здесь указано, из какого модуля происходит импорт.

- *[[ImportName]]*: название импортируемого элемента. Оно будет — `"*"`, если импортировать объект пространства имен модуля.
- *[[LocalName]]*: имя локального идентификатора (биндинга), который будет использоваться для импортируемого элемента. Часто это то же значение, что и [[ImportName]], но, если для переименования использовалось ключевое слово `as` в объявлении импорта, они могут отличаться.

Чтобы увидеть, как эти поля соотносятся с различными формами объявлений импорта (Таблица 13-1). Она в значительной степени основана на «Таблице 44 (информативной): Сопоставление форм импорта с записями ImportEntry» из спецификации.

Таблица 13-1. Импорт операторов и записей

Форма объявления импорта	[[MODULEREQUEST]]:	[[IMPORTNAME]]	[[LOCALNAME]]
`import v from "mod";`	`"mod"`	`"default"`	`"v"`
`import * as ns from "mod";`	`"mod"`	`"*"`	`"ns"`
`import {x} from "mod";`	`"mod"`	`"x"`	`"x"`
`import {x as v} from "mod";`	`"mod"`	`"x"`	`"v"`
`import "mod";`	Запись `ImportEntry` не создается		

Причина, по которой при импорте модуля только ради его побочных эффектов не создается запись импорта (`import "mod";` в конце таблицы), заключается в том, что в этом списке конкретно указано, какие *вещи* (биндинги) импортирует модуль. Список других модулей, запрашиваемых модулем, — это отдельный список, также созданный во время синтаксического анализа. Модули, импортированные только ради своих побочных эффектов, включены в этот другой список. Он сообщает движку JavaScript, что нужно этому модулю для загрузки.

Экспорт записей

Список экспортируемых записей модуля описывает то, что он экспортирует. Каждая экспортная запись содержит четыре поля:

- *[[ExportName]]*: название экспорта. Имя, используемое другими модулями при импорте. Это название — строка `"default"` для экспорта по умолчанию и `null` для объявления `export * from`, которое повторно экспортирует все из другого модуля (вместо этого используется список экспорта из этого другого модуля).
- *[[LocalName]]*: имя экспортируемого локального идентификатора (биндинга). Оно часто совпадает со значением [[ExportName]], но, если вы переименовали экспорт, они могут отличаться. Оно равно значению `null`, если эта запись предназначена для объявления `export ... from`, которое повторно экспортирует экспорт другого модуля, поскольку локальное имя не задействовано.

Таблица 13-2

Форма объявления экспорта	[[EXPORTNAME]]:	[[MODULEREQUEST]]:	[[IMPORTNAME]]	[[LOCALNAME]]
`export var v;`	`"v"`	`null`	`null`	`"v"`
`export default function f() {}`	`"default"`	`null`	`null`	`"f"`
`export default function () {}`	`"default"`	`null`	`null`	`"*default*"`
`export default 42;`	`"default"`	`null`	`null`	`"*default*"`

`export {x};`	`"x"`
	`null`
	`null`
				`"x"`

`export {v as x};`	`"x"`	`null`	`null`	`"v"`
`export {x} from "mod";`	`"x"`	`"mod"`	`"x"`	`null`
`export {v as x} from "mod";`	`"x"`	`"mod"`	`"v"`	`null`
`export * from "mod";`	`null`	`"mod"`	`"*"`	`null`
`export * as ns from "mod";`	`"ns"`	`"mod"`	`"*"`	`null`

- *[[ModuleRequest]]*: для повторного экспорта это строка спецификатора модуля из объявления `export ... from`. Это значение равно `null` для собственного экспорта модуля.
- *[[ImportName]]:* для повторного экспорта это имя экспорта в другом модуле для повторного экспорта. Часто это то же самое значение, что и [[ExportName]], но, если вы использовали предложение `as` в объявлении о повторном экспорте, они могут отличаться. Это значение равно `null` для собственного экспорта модуля.

Чтобы увидеть, как эти поля соотносятся с различными формами экспортных объявлений, см. Таблицу 13-2. Она в значительной степени основана на «Таблице 46 (информативной): Сопоставление форм экспорта с записями ExportEntry» из спецификации.

Этот список записей сообщает движку JavaScript, что предоставляет этот модуль при загрузке.

ИМПОРТ В РЕЖИМЕ РЕАЛЬНОГО ВРЕМЕНИ И ДОСТУПНОСТИ ТОЛЬКО ДЛЯ ЧТЕНИЯ

Когда вы импортируете что-то из модуля, то получаете доступную только для чтения активную привязку, называемую *косвенной привязкой*, к исходному элементу. Поскольку она доступна только для чтения, ваш код не может присвоить привязке новое значение. Но поскольку это активная привязка, ваш код действительно видит любые новые значения, которые присваивает ей исходный модуль. Например, предположим, что у вас есть `mod.js`, показанный в Листинге 13-7, и `main.js`, показанный в Листинге 13-8 (вы можете запустить их из загрузок, используя предоставленный файл **main.html**).

Листинг 13-7: Простой модуль со счетчиком — mod.js

```
const a = 1;
let c = 0;
export {c as counter};
export function increment() {
    ++c;
}
```

Листинг 13-8: Модуль, использующий модуль счетчика — main.js

```
import {counter, increment as inc} from "./mod.js";
console.log(counter); // 0
inc();
console.log(counter); // 1
counter = 42;         // TypeError: Assignment to constant variable.
```

Как вы можете видеть, код в `main.js` видит изменения `counter`, производимые `mod.js`, но не может установить эти значения. Сообщение об ошибке, показанное в списке, — текущее сообщение V8 (в Chrome); SpiderMonkey (в Firefox) говорит `TypeError: "counter" доступно только для чтения`.

Вы помните из раздела «Привязки: Как работают переменные, константы и другие идентификаторы» главы 2, что эти переменные, константы и другие идентификаторы концептуально являются *привязками* в *объекте среды* (очень похоже на свойства в объекте). У каждой привязки есть имя, флаг, указывающий, является ли она изменяемой (изменяемая = можно изменить ее значение, неизменяемая = нельзя), и текущее значение привязки. Например, этот код:

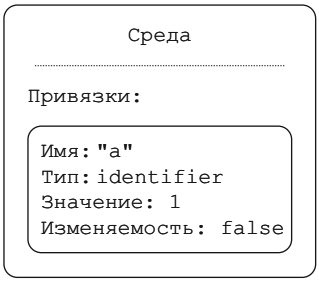


РИСУНОК 13-3

```
const a = 1;
```

создает привязку внутри текущего объекта среды, как показано на рисунке 13-3.

У каждого модуля есть *объект среды модуля*. (Не путайте это с аналогичным названием «объект пространства имен модуля»: это разные вещи.) У объекта среды модуля есть привязки для всего экспорта и импорта этого модуля (но не для его повторного экспорта из других модулей), а также другие привязки верхнего уровня, которые не экспортируются и не импортируются (привязки, используемые только внутри модуля). В объекте среды модуля привязки могут быть либо прямыми (для экспорта и не экспортируемых привязок), либо косвенными (для импорта). Косвенная привязка сохраняет ссылку на исходный модуль (ссылающийся на объект среды этого модуля) и имя привязки в среде этого модуля для использования, вместо того чтобы напрямую сохранять значение привязки.

Когда `main.js` импортирует из `mod.js`, как показано в листингах, среда модуля `main.js` содержит косвенные привязки к `counter` и `inc`, ссылающиеся на модуль `mod.js`, на его привязки к объектам среды `c` и `increment`, как на рисунке 13-4. В косвенных привязках `Module` — это ссылка на модуль, а `Binding Name` — имя привязки в используемой среде этого модуля.

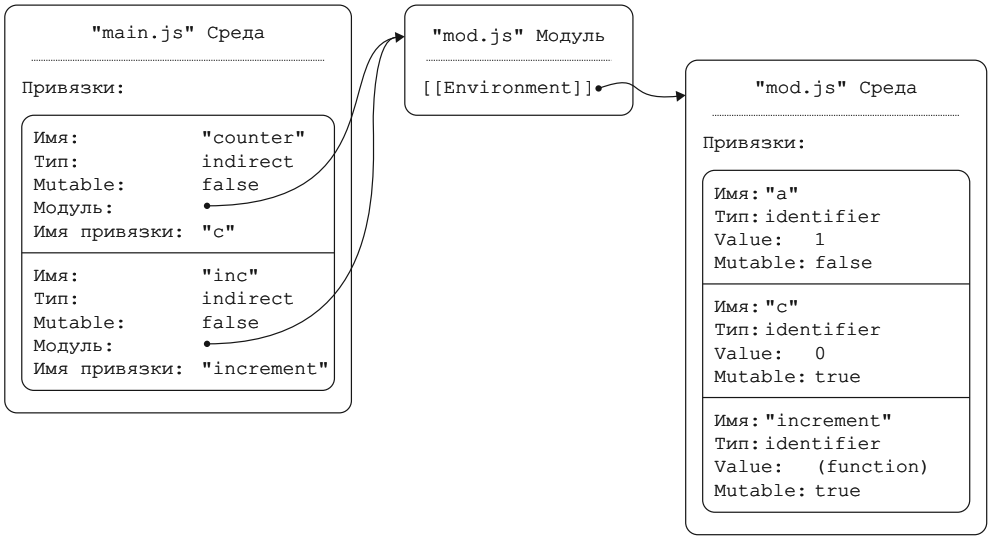


РИСУНОК 13-4

(Вы можете быть удивлены, как `main.js` узнает, что привязка к экспорту `counter`, импортированному из `mod.js`, — это `c`. Движок JavaScript получил эту информацию из списка записей экспорта для `mod.js`, о котором вы узнали в предыдущем разделе. Немного позже вы узнаете об этом подробнее.)

Когда `main.js` считывает значение `counter`, движок JavaScript видит, что объект среды из `main.js` содержит косвенную привязку к `counter`, получает объект среды для `mod.js`, затем получает значение привязки с именем `c` от этого объекта среды модуля и использует ее в качестве результирующего значения.

Этот характер импортируемых привязок, доступных только для чтения, но способных изменяться, также очевиден в свойствах объекта пространства имен модуля: вы можете только читать значения свойств, но не устанавливать их значения.

ОПИСАНИЯ СВОЙСТВ ОБЪЕКТОВ ПРОСТРАНСТВА ИМЕН МОДУЛЕЙ

Если вы используете `Object.getOwnPropertyDescriptor` в свойстве объекта пространства имен модуля для экспорта, возвращаемый дескриптор всегда выглядит следующим образом:

```
{
    value: /*...the value...*/,
    writable: true,
    enumerable: true,
    configurable: false
}
```

Дескриптор не меняется, кроме `value`. Например, дескриптор свойства, представляющего экспорт `const`, выглядит точно так же, как дескриптор свойства, представляющего экспорт `let`. Но хотя в свойстве указано, что оно доступно для записи, при попытке выполнить запись в свойство вы получите сообщение об ошибке, говорящее, что оно доступно только для чтения. (У объекта пространства имен модуля есть специальный внутренний метод [[Set]], предотвращающий установку значения любого свойства.) Может показаться странным, что свойство заявляет о доступности для записи, но при этом вы не можете ничего в него записать. Но ведь оно так помечено по какой-то причине. На самом деле есть пара причин.

- Информация о внутренней структуре модуля не должна раскрываться за пределами модуля, поэтому важно, чтобы экспорт `const` и `let` (или экспорт функций и т. д.) выглядел одинаково. (Может быть и другой аргумент: экспорт const означает, что вы экспортируете не только его существование, но и тот факт, что это `const`, но пока информация скрыта.)

- Если бы свойство было помечено как `writable: false`, можно было бы ошибочно предположить, что значение не может измениться. Но, конечно, значение может измениться, если оно не относится к типу const и код экспортирующего модуля изменяет его. Фактически одна из важных гарантий поведения объекта, предписанных спецификацией, заключается в том, что если было замечено, что у неконфигурируемого, недоступного для записи свойства данных (не свойства-акцессора) есть значение, повторное чтение его позже должно возвращать то же значение. (Это находится в разделе спецификации «Инварианты основных внутренних методов»[77]. Вы узнаете больше об этих гарантиях инвариантного поведения в главе 14.) Ни одному объекту не разрешается нарушать это правило. Таким образом, эти свойства должны быть определены как доступные для записи.

ЭКЗЕМПЛЯРЫ МОДУЛЯ ЗАВИСЯТ ОТ БАЗЫ REALM

Ранее вы узнали (мимоходом), что модуль загружается только один раз для каждой базы realm (окно, вкладка, воркер и т. д.). В частности, движок JavaScript отслеживает загруженные модули в пределах базы realm и повторно использует запрашиваемые более одного раза. Однако это зависит от конкретной базы realm; разные базы не используют общие экземпляры модулей.

Например, если у вас есть окно с окном `iframe` в нем, у главного окна и окна `iframe` будут разные базы данных realm. Если код в обоих окнах загружает модуль `mod.js`, он загружается дважды: один раз в базе realm главного окна и снова в базе окна `iframe`. Эти две копии модуля полностью отделены друг от друга даже больше, чем две глобальных среды в окнах (связь между ними осуществляется через массив `frames` главного окна и переменную `parent` окна `iframe`). Любые модули, загруженные `mod.js`, также загружаются дважды, как и их зависимости и т. д. Модули совместно используются только внутри базы данных realm, а не между такими базами.

КАК ЗАГРУЖАЮТСЯ МОДУЛИ

Модульная система JavaScript разработана таким образом, чтобы простые варианты использования были простыми, но хорошо справлялись со сложными вариантами использования. Для этого модули загружаются в три этапа:

- *Получение и синтаксический анализ:* получение исходного текста модуля и его синтаксический анализ, определение его импорта и экспорта.
- *Создание экземпляра:* создание среды модуля и его привязок, включая привязки для всех его импортов и экспортов.
- *Выполнение:* запуск кода модуля.

[77] `https://tc39.github.io/ecma262/#sec-invariants-of-the-essential-internal-methods`

Чтобы проиллюстрировать этот процесс, мы обратимся к коду в Листингах с 13-9 по 13-12.

Листинг 13-9: HTML-страница для примера загрузки модуля — loading.html

```
<!doctype html>
<html>
<head>
<meta charset="UTF-8">
<title>Module Loading</title>
</head>
<body>
<script src="entry.js" type="module"></script>
</body>
</html>
```

Листинг 13-10: Точка входа загрузки модуля — entry.js

```
import {fn1} from "./mod1.js";
import def, {fn2} from "./mod2.js";

fn1();
fn2();
def();
```

Листинг 13-11: Загрузка модуля mod1 — mod1.js

```
import def from "./mod2.js";

const indentString = " ";

export function indent(nest = 0) {
    return indentString.repeat(nest);
}

export function fn1(nest = 0) {
    console.log(`${indent(nest)}mod1 - fn1`);
    def(nest + 1);
}
```

Листинг 13-12: Загрузка модуля mod2 — mod2.js

```
import {fn1, indent} from "./mod1.js";

export function fn2(nest = 0) {
    console.log(`${indent(nest)}mod2 - fn2`);
    fn1(nest + 1);
}

export default function(nest = 0) {
    console.log(`${indent(nest)}mod2 - default`);
}
```

Как видно из листингов, модуль `entry.js` импортирует из двух других взаимосвязанных модулей: `mod1.js` использует экспорт по умолчанию из `mod2.js`, а `mod2.js`

использует именованные экспорты `fn1` и `indent` из `mod1.js`. Два модуля находятся в циклической взаимосвязи. Большинство ваших модулей не получат подобных циклических отношений, но наличие таких отношений позволяет в примере показать основы того, как они обрабатываются.

При запуске кода из перечня загрузок вы увидите на экране следующий результат:

```
mod1 - fn1
  mod2 - default
mod2 - fn2
  mod1 - fn1
    mod2 - default
mod2 - default
```

Давайте посмотрим, как туда попадает браузер (хост для движка JavaScript в этом примере).

Получение и синтаксический анализ

Когда хост (браузер) видит элемент

```
<script src="entry.js" type="module"></script>
```

он получает `entry.js` и передает исходный текст движку JavaScript для синтаксического анализа в виде модуля. В отличие от немодульных скриптов, модульные скрипты не задерживают синтаксический анализатор HTML, поэтому он продолжает выполняться, пока выполняется работа по получению и загрузке модуля. В частности, скрипт модуля по умолчанию *отложен* (как если бы у него был атрибут `defer`), что означает, что его код не будет выполняться до тех пор, пока анализатор HTML не завершит синтаксический анализ страницы. (Вместо этого можно указать `async`, если требуется, чтобы выполнение происходило как можно скорее, даже до завершения синтаксического анализа HTML.)

Когда браузер передает содержимое `entry.js` движку JavaScript, движок анализирует его и создает для него *запись модуля*. Запись модуля содержит проанализированный код, список модулей, требующихся этому модулю, списки записей импорта и экспорта модуля, о которых вы узнали пару разделов назад,

РИСУНОК 13-5

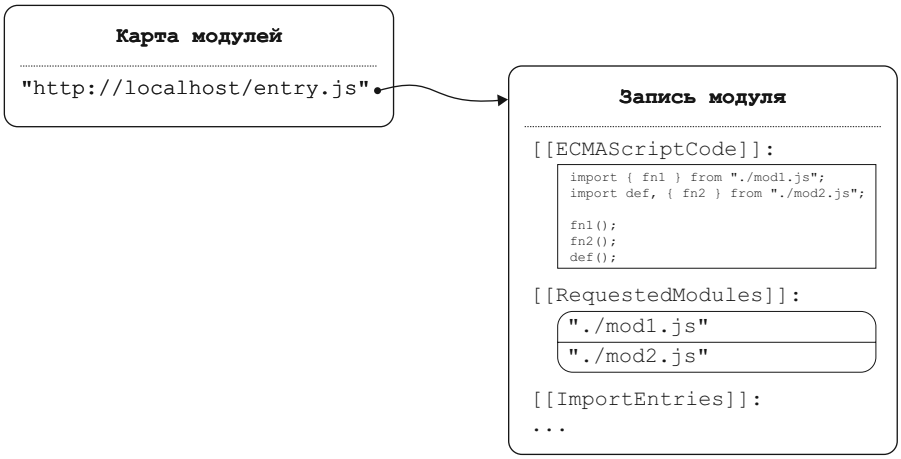


РИСУНОК 13-6

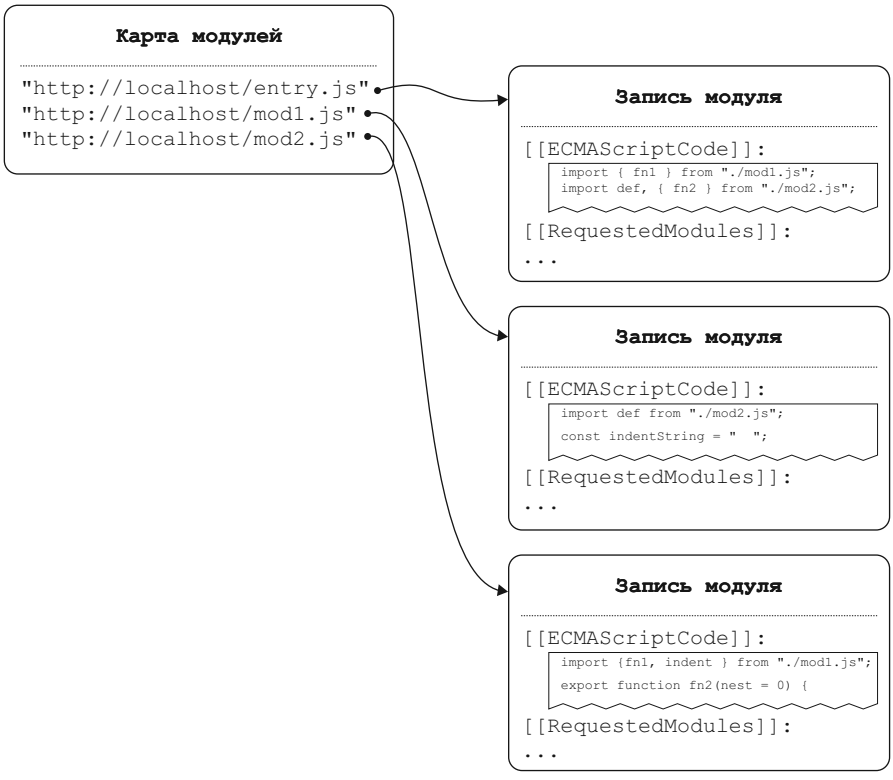


РИСУНОК 13-7

и различные другие учетные данные, такие как статус модуля (где он находится в процессе загрузки и выполнения). Обратите внимание, что вся эта информация была определена статически, просто путем *синтаксического анализа* кода модуля, а не его *выполнения*. Посмотрите на рисунок 13-5, как будут выглядеть записи модуля для ключевых частей модуля `entry.js`. (Имена на рисунке взяты из спецификации, в которой используется соглашение `[[Name]]` — имя в двойных квадратных скобках — для полей в концептуальных объектах.)

Движок JavaScript возвращает эту запись модуля хосту, который сохраняет ее в карте разрешенных модулей (*карта модулей*) в соответствии со своим полностью разрешенным спецификатором модуля (например, `http://localhost/entry.js`, если `loading.html` взято из `http://localhost`). См. рисунок 13-6.

Позже, когда потребуется запись модуля, движок JavaScript запрашивает ее у хоста. Хост ищет ее в карте модуля и возвращает, если она найдена.

На данный момент `entry.js` был извлечен и проанализирован, поэтому движок JavaScript запускает этап *создания экземпляра* `entry.js` (рассматривается в следующем разделе). Первое, что выполняется, — запрос браузеру разрешить `mod1.js` и `mod2.js`, для чего браузер и движок JavaScript взаимодействуют так же, как и с `entry.js`. Как только они будут извлечены и проанализированы, браузер получит карту модулей с записями для всех трех модулей; см. рисунок 13-7.

Информация в записях модуля позволяет движку JavaScript определять дерево модулей, которые необходимо создать и оценить[78]. В этом примере дерево выглядит примерно так, как показано на рисунке 13-8.

Теперь создание экземпляра может начаться как положено.

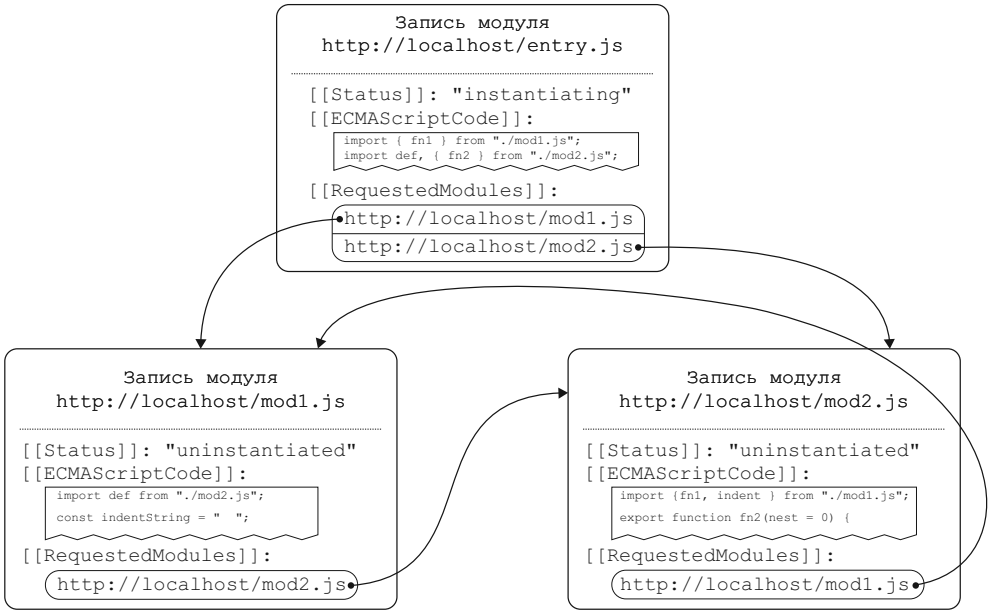


РИСУНОК 13-8

[78] Педантам нравится указывать, что это *граф*, а не дерево, поскольку у него могут быть циклические отношения. Но почти все остальные называют это деревом модулей.

Создание экземпляра

На этом этапе движок JavaScript создает объект среды каждого модуля и привязки верхнего уровня внутри него, включая привязки для всего импорта и экспорта модуля (наряду с любыми другими локальными элементами, которые у него могут быть, — например, `indentString` в `mod1.js`). Для локальных элементов (включая экспортируемые) это прямые привязки. Для импорта это косвенные привязки, подключаемые движком к привязке модуля экспорта для экспорта. Создание экземпляра выполняется с использованием обхода в глубину, так что сначала создаются экземпляры модулей самого низкого уровня. В текущем примере результирующие среды выглядят примерно так, как показано на рисунке 13-9.

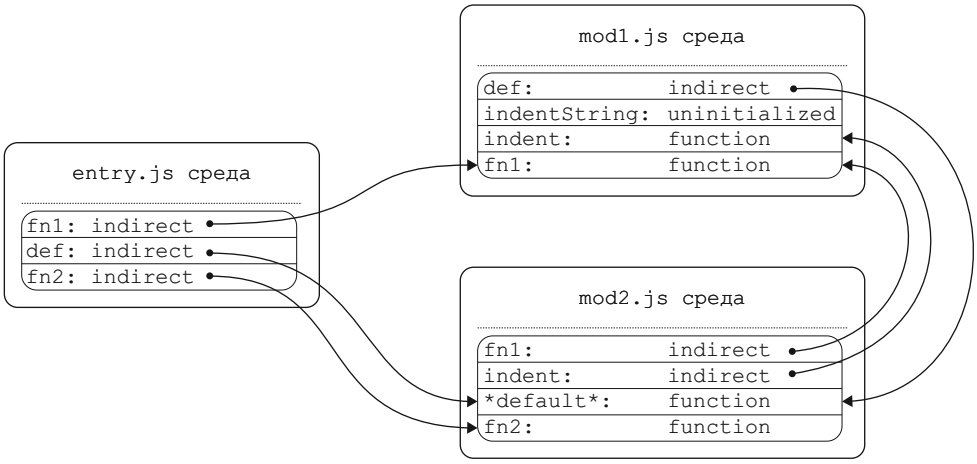


РИСУНОК 13-9

Если внимательно посмотреть на рисунок 13-9, можно задаться вопросом об этой привязке с именем `*default*`. Это имя получает локальная привязка для экспорта по умолчанию, если экспортируемое анонимно (объявление анонимной функции, объявление анонимного класса или результат произвольного выражения). Поскольку экспорт по умолчанию `mod2.js` — это анонимное объявление функции, локальная привязка для него называется `*default*`. (Но, опять же, код в модуле не может использовать эту привязку фактически.)

Создание экземпляра модуля создает его область действия (объект среды и привязки), но не запускает его код. Это означает, что *объявления с возможностью поднятия* (например, все функции в модулях этого примера) обрабатываются, создавая для них функции, но лексические привязки остаются неинициализированными: они находятся во Временной мертвой зоне (TDZ), о которой вы узнали в главе 2. Вот почему на рисунке 13-9 показано, что у `indent`, `*default*` и других привязок есть значения (функции), но `indentString` неинициализирована. Подробнее о TDZ чуть позже.

Звучит ли знакомо представление о создании среды и обработке объявлений с возможностью поднятия до начала пошагового выполнения кода? Верно! Это очень похоже на первую часть вызова функции, которая также создает среду для вызова функции и выполняет всю необходимую работу перед началом пошагового выполнения.

Как только все модули будут созданы, можно приступать к третьему этапу — выполнению.

Выполнение

На этом этапе движок JavaScript запускает код верхнего уровня модулей, опять же в порядке работы в глубину, помечая каждый из них как «выполненный» по мере его выполнения. Это похоже на вторую часть вызова функции — пошаговое выполнение кода. По мере выполнения кода любые неинициализированные привязки верхнего уровня (например `indentString` из `mod1.js`) инициализируются по мере их достижения при выполнении кода.

Модульная система JavaScript гарантирует, что код верхнего уровня каждого модуля выполняется только один раз. Это важно, потому что у модулей могут быть побочные эффекты, хотя в общем случае лучше, если побочных эффектов нет (кроме основного модуля страницы/приложения).

Ранее описывалось, что в браузерах, если элемент `script` для точки входа не получил атрибут `async`, выполнение не начнется до тех пор, пока анализатор HTML не завершит синтаксический анализ документа. Если есть атрибут `async`, выполнение начнется как можно скорее после завершения создания экземпляра, даже если анализатор HTML все еще работает над документом.

Обзор временной мертвой зоны (TDZ)

Модули могут содержать (и импортировать/экспортировать) лексические привязки верхнего уровня (созданные с помощью `let`, `const` и `class`). В главе 2 вы узнали, что лексические привязки создаются при создании объекта среды для области, в которой они отображаются, но не инициализируются до тех пор, пока не будет достигнуто объявление в пошаговом выполнении кода. Между созданием и инициализацией они находятся во *Временной мертвой зоне.* При попытке их использования вы получите сообщение об ошибке:

```
function example() {
    console.log(a); // ReferenceError: a не определено
    const a = 42;
}
example();
```

В этом примере объект среды для вызова `example` создается при входе в `example`. Следовательно, у объекта есть привязка к `a` к локальной `const`, но привязка еще не инициализирована. Когда запускается код и выполняется `console.log`, поскольку привязка к `a` не инициализирована, попытка использовать `a` завершается неудачей.

Не забывайте, что Временная мертвая зона называется *временной*, потому что она относится ко времени между созданием и инициализацией, а не к относительному расположению кода «выше» или «ниже» объявления. Следующий код работает просто отлично:

```
function example() {
    const fn = () => {
```

```
        console.log(a); // 42
    };
    const a = 42;
    fn();
}
example();
```

Он работает, потому что, хотя использующая `a` строка `console.log` находится над строкой переменной в коде, строка с функцией не выполняется до тех пор, пока не будет достигнуто объявление в пошаговом выполнении кода.

Наконец, необходимо помнить, что TDZ относится только к лексическим привязкам, а не к привязкам, созданным с помощью объявлений, проходящих процедуру поднятия — `var` объявленных переменных или объявлений функций. Привязка, созданная с помощью `var`, немедленно инициализируется значением `undefined`, а привязка, созданная объявлением функции, немедленно инициализируется объявляемой функцией.

Как это связано с модулями? TDZ применяется к любому моменту создания объекта среды и получает его привязки, включая среду для модуля. В предыдущих разделах описывается, что объект среды для модуля создается во время создания экземпляра модуля, а его код запускается позже во время выполнения модуля. Таким образом, любые лексические привязки, полученные модулем в своей области верхнего уровня, находятся в TDZ с момента создания экземпляра до тех пор, пока объявление не будет достигнуто во время выполнения.

Все это означает, что если модуль экспортирует лексическую привязку, в некоторых ситуациях это может позволить коду попытаться использовать привязку до ее инициализации. Например, это возможно, если модуль A импортирует из модуля B, а модуль B импортирует из модуля A (прямо или косвенно) — *циклическая зависимость*. Давайте рассмотрим этот вариант.

Циклические зависимости и TDZ

В примере модуля загрузки `mod1.js` и `mod2.js` ссылаются друг на друга — они находятся в циклической зависимостью (простой прямой). Это не проблема для системы модулей JavaScript из-за трехфазного процесса загрузки и выполнения модулей.

Заметьте, что ни `mod1.js`, ни `mod2.js` не используют то, что они импортируют в свой код верхнего уровня — только то, что они импортировали в ответ на вызов функции. Предположим, вы изменили это: измените свою локальную копию `mod2.js`, чтобы добавить вызов функции `console.log` на верхнем уровне, как указано в Листинге 13-13.

Листинг 13-13: Загрузка модуля mod2 (обновленный вариант) — mod2-updated.js

```
import {fn1, indent} from "./mod1.js";

console.log(`${indent(0)}hi there`);

export function fn2(nest = 0) {
    console.log(`${indent(nest)}mod2 - fn2`);
    fn1(nest + 1);
}
```

```
export default function(nest = 0) {
    console.log(`${indent(nest)}mod2 - default`);
}
```

Теперь перезагрузите файл **loading.html**. Вы получите ошибку

```
ReferenceError: indentString is not defined
```

потому что `mod2.js` пытается использовать `indent` до выполнения `mod1.js`. Вызов `indent` работает, поскольку был создан экземпляр `mod1.js` и, следовательно, его объявления, доступные для поднятия, были обработаны. Но когда `indent` пытается использовать `indentString`, происходит ошибка, поскольку `indentString` все еще находится в TDZ.

Теперь попытайтесь изменить объявление `indentString` на `var` вместо `const`, а затем перезагрузите файл **loading.html**. Вы получите другую ошибку

```
TypeError: Cannot read property 'repeat' of undefined
```

Это происходит из-за того, что привязка была инициализирована значением `undefined` во время создания экземпляра, поскольку она объявлена с помощью `var`, и ей еще не была присвоена строка с двумя пробелами, потому что код верхнего уровня в `mod1.js` еще не был выполнен.

Часто циклическая зависимость предполагает, что стоит провести рефакторинг. Но когда вы не можете избежать таких зависимостей, просто помните, что поднятые объявления будут работать нормально (кроме `var` объявленных переменных со значением `undefined`), а другие будут в норме, если они используются только внутри функций (те, что не вызываются непосредственно из кода верхнего уровня), а не в коде модуля верхнего уровня.

ОБЗОР СИНТАКСИСА ИМПОРТА/ЭКСПОРТА

В этой главе вы узнали о нескольких различных вариантах синтаксиса для импорта и экспорта. Давайте подытожим, показав вам все ваши варианты.

Разновидности экспорта

Вот краткое описание различных форм экспорта.

Каждое из следующих действий объявляет локальную привязку (переменную, константу, функцию или класс) и экспорт для нее, используя имя привязки для экспорта:

```
export var a;
export var b = /*...некоторое значение...*/
export let c;
export let d = /*...some value...*/;
export const e = /*...some value...*/;
export function f() {// (а также асинхронные и формы генераторов)
}
export class G() {
}
```

Каждое из следующих действий объявляет экспорт для локальных привязок, объявленных в другом месте модуля (они могут быть объявлены выше или ниже объявления `export`):

```
export {a};
export {b, c};
export {d as delta};      // Экспортируемое имя - `delta`
export {e as epsilon, f}; // Локальная `e` экспортируется как `epsilon`
export {g, h as hotel, i}; // Можно смешивать различные формы
```

Для экспорта каждой привязки используется ее имя, если не указано ключевое слово `as`, в этом случае вместо имени привязки для экспорта используется имя, указанное после `as`. Список экспорта (часть, заключенная в фигурные скобки) может содержать в себе столько элементов, сколько потребуется.

Каждое из нижеперечисленных выражений объявляет экспорт или косвенный экспорт, повторный экспорт одного или нескольких экспортов из текущего модуля `mod.js`, при необходимости выполняя переименование с помощью предложения `as`:

```
export {a} from "./mod.js";
export {b, c} from "./mod.js";
export {d as delta} from "./mod.js";
export {e, f as foxtrot} from "./mod.js"; // Можно реализовать сочетание
                                          // вышеперечисленных выражений
```

Этот код объявляет косвенный экспорт для всех именованных экспортов в `mod.js`:

```
export * from "./mod.js";
```

Каждое из следующих выражений объявляет локальную привязку (переменную, константу, функцию или класс) как экспорт по умолчанию:

```
// Только одно из выражений может появиться в любом полученном модуле,
// поскольку в модуле может быть только один экспорт по умолчанию
export default function a() {/*...*/}
export default class B {/*...*/}
export default let c;
export default let d = "delta";
export default const e = "epsilon";
export default var f;
export default var g = "golf";
```

Каждое из следующих выражений объявляет локальную привязку с именем `*default*` (потому что каждое из них анонимное), к которой ваш код не может получить доступ, поскольку `*default*` — недопустимый идентификатор. Они экспортируют его как экспорт по умолчанию:

```
export default function() {/*...*/}
export default class {/*...*/}
export default 6 * 7; // Любое произвольное выражение
```

Функция, созданная объявлением анонимной функции, получит значение `"default"` в качестве своего имени — как и конструктор класса, созданный объявлением `class` анонимного класса.

Разновидности импорта

Вот краткое описание различных форм импорта.

Этот код импортирует именованный экспорт `example` из `mod.js`, также используя `example` в качестве локального имени:

```
import {example} from "./mod.js";
```

Этот код импортирует именованный экспорт `example` из `mod.js`, используя `e` вместо `example` в качестве локального имени.

```
import {example as e} from "./mod.js";
```

Этот код импортирует экспорт по умолчанию из `mod.js`, используя `example` в качестве локального имени:

```
import example from "./mod.js";
```

Этот код импортирует объект пространства имен для `mod.js`, используя локальное имя `mod`:

```
import * as mod from "./mod.js";
```

Можно комбинировать импорт по умолчанию *либо* с импортом объекта пространства имен модуля, *либо* с именованным импортом. Но, чтобы избежать сложности синтаксического анализа, нельзя комбинировать импорт пространства имен с именованным экспортом; нельзя выполнять все три операции вместе; и если импортируется значение по умолчанию, оно должно быть первым, перед `*` для объекта пространства имен или `{`, чтобы начать список именованных импортов:

```
import def, * as ns from "./mod.js";           // Допустимо
import * as ns, def from "./mod.js";           // Недопустимо, импорт
                                               // по умолчанию должен быть
                                               // первым
import def, {a, b as bee} from "./mod.js";     // Допустимо
import * as ns, {a, b as bee} from "./mod.js"; // Недопустимо, это
                                               // невозможно объединить
```

В конечном счете это импортирует `mod.js` только из-за его побочных эффектов, без импорта каких-либо его экспортов:

```
import "./mod.js";
```

ДИНАМИЧЕСКИЙ ИМПОРТ

Механизм импорта, описанный в текущей главе, представляет собой статический механизм:

- Спецификаторы модуля — это строковые литералы, а не выражения, приводящие к строкам.
- Импорт и экспорт могут быть объявлены только на верхнем уровне модуля, а не в операторах потока управления, таких как `if` или `while`.
- Импорт и экспорт модуля могут быть определены путем синтаксического анализа кода без его запуска (они могут быть *статически проанализированы*).

В результате модуль не может использовать информацию, получаемую во время выполнения, чтобы решить, что импортировать или откуда это берется.

В подавляющем большинстве случаев именно это и требуется. Это позволяет инструментам выполнять очень мощные функции, такие как встряхивание дерева (о котором вы узнаете позже) и автоматическое объединение (определение того, какие модули включать в единый пакет, без отдельного объявления пакета, что является проблемой обслуживания) и т. д.

Однако некоторые варианты использования требуют, чтобы модули определяли, что импортировать или откуда, во время выполнения. По этой причине в ES2020 добавлен динамический импорт[79]. На момент написания книги он широко поддерживался: из основных браузеров его не поддерживали только старая версия Edge (не на базе Chromium) и, конечно же, Internet Explorer.

Динамический импорт модуля

Динамический импорт добавляет новый синтаксис, позволяющий вызывать `import`, как если бы это была функция. При вызове `import` возвращает промис объекта пространства имен модуля:

```
import(/*...some runtime-determined name...*/)
.then(ns => {
    // ...использование `ns`...
})
.catch(error => {
    // Обработка/отчет об ошибке
});
```

Обратите внимание, что, хотя это выглядит как вызов функции, выражение `import(...)` не является вызовом функции; это новый синтаксис, называемый *ImportCall*. В результате нельзя сделать так:

```
// Это не работает
const imp = import;
imp (/*...some runtime-determined name...*/)
.then(// ...
```

[79] https://github.com/tc39/proposal-dynamic-import

Есть две причины этого сбоя:

- Вызов должен содержать контекстную информацию, которую обычные вызовы функций не несут.
- Запрещение его псевдонимов (пример `imp` ранее) позволяет статическому анализу узнать, используется ли динамический импорт в модуле (даже если он не может определить, что конкретно делает этот динамический импорт). Подробнее о том, почему это важно, вы узнаете позже в разделе, посвященном встряхиванию деревьев.

Одно из очевидных приложений для динамического импорта — это плагины. Предположим, вы пишете графический редактор на JavaScript, такой как Electron, Windows Universal Application или аналогичный, и хотите разрешить расширениям предоставлять преобразования или пользовательские перья или что-то подобное. Ваш код может искать плагины в каком-либо расположении плагинов, а затем загружать их с помощью кода, похожего на:

```
// Загружает плагины параллельно, возвращает объект с массивом `plugins`
// загруженных плагинов и массивом `failed` неудачных загрузок (со свой-
ствами
// `error` и `pluginFile`).
async function loadPlugins(editor, discoveredPluginFiles) {
    const plugins = [];
    const failed = [];
    await Promise.all(
        discoveredPluginFiles.map(async (pluginFile) => {
            try {
                const plugin = await import(pluginFile);
                plugin.init(editor);
                plugins.push(plugin);
            } catch (error) {
                failed.push({error, pluginFile});
            }
        })
    )
    return {plugins, failed};
}
```

Или, возможно, вы бы загрузили локализатор для приложения на основе локали:

```
async function loadLocalizer(editor, locale) {
    const localizer = await import(`./localizers/${locale}.js`);
    localizer.localize(editor);
}
```

Динамически загружаемые модули кэшируются так же, как и статически загружаемые.

Импорт модуля динамически запускает процесс загрузки любых статических зависимостей, которые он выражает, и их зависимостей и т. д. точно так же, как это делает загрузка модуля точки входа.

Давайте рассмотрим пример.

Пример динамического модуля

Запустите код из листингов с 13-14 по 13-19, запустив файл **dynamic-example.html.**

Листинг 13-14: Пример динамической загрузки модуля (HTML) — dynamic-example.html

```
<!doctype html>
<html>
<head>
<meta charset="UTF-8">
<title>Dynamic Module Loading Example</title>
</head>
<body>
<script src="dynamic-example.js" type="module"></script>
</body>
</html>
```

Листинг 13-15: Пример динамической загрузки модуля (статическая точка входа) — dynamic-example.js

```
import {log} from "./dynamic-mod-log.js";

log("entry point module top-level evaluation begin");
(async () => {
    try {
        const modName = `./dynamic-mod${Math.random() < 0.5 ? 1: 2}.js`;
        log(`entry point module requesting ${modName}`);
        const mod = await import(modName);
        log(`entry point module got module ${modName}, calling mod.example`);
        mod.example(log);
    } catch (error) {
        console.error(error);
    }
})();
log("entry point module top-level evaluation end");
```

Листинг 13-16: Пример динамической загрузки модуля (статический импорт) — dynamic-mod-log.js

```
log("log module evaluated");
export function log(msg) {
    const p = document.createElement("pre");
    p.appendChild(document.createTextNode(msg));
    document.body.appendChild(p);
}
```

Листинг 13-17: Пример динамической загрузки модуля (динамический импорт 1) — dynamic-mod1.js

```
import {log} from "./dynamic-mod-log.js";
import {showTime} from "./dynamic-mod-showtime.js";

log("dynamic module number 1 evaluated");
export function example(logFromEntry) {
```

```
    log("Number 1! Number 1! Number 1!");
    log(`log === logFromEntry ? ${log === logFromEntry}`);
    showTime();
}
```

Листинг 13-18: Пример динамической загрузки модуля (динамический импорт 2) — dynamic-mod2.js

```
import {log} from "./dynamic-mod-log.js";
import {showTime} from "./dynamic-mod-showtime.js";

log("dynamic module number 2 evaluated");
export function example(logFromEntry) {
    log("Meh, being Number 2 isn't that bad");
    log(`log === logFromEntry ? ${log === logFromEntry}`);
    showTime();
}
```

Листинг 13-19: Пример динамической загрузки модуля (динамическая зависимость от импорта) — dynamicmod-showtime.js

```
import {log} from "./dynamic-mod-log.js";

log("showtime module evaluated");
function nn(n) {
    return String(n).padStart(2, "0");
}
export function showTime() {
    const now = new Date();
    log(`Time is ${nn(now.getHours())}:${nn(now.getMinutes())}`);
}
```

В этом примере у двух динамических модулей одинаковая форма: они просто идентифицируют себя либо как номер 1, либо как номер 2.

При загрузке страницы `dynamic-example.html` вы увидите вывод, подобный этому (на самой странице, а не в консоли):

```
log module evaluated
entry point module top-level evaluation begin
entry point module requesting ./dynamic-mod1.js
entry point module top-level evaluation end
showtime module evaluated
dynamic module number 1 evaluated
entry point module got module ./dynamic-mod1.js, calling mod.example
Number 1! Number 1! Number 1!
log === logFromEntry ? true
Time is 12:44
```

В этом примере был загружен динамический модуль под номером 1, но с таким же успехом он мог быть номером 2.

Давайте пройдемся по тому, что происходит в коде:

- Хост (браузер) и движок JavaScript выполняют процесс получения и синтаксического анализа, создания экземпляра и вычисления точки входа в модуле и модуль `dynamic-mod-log.js`, который он статически импортирует (я просто назову это «модулем log»); это все статическое дерево, только эти два модуля (рисунок 13-10.) Поскольку выполнение осуществляется в первую очередь в глубину, сначала выполняется модуль log («log module evaluated»), а затем — модуль точки входа («entry point module toplevel evaluation begin», и т. д.).

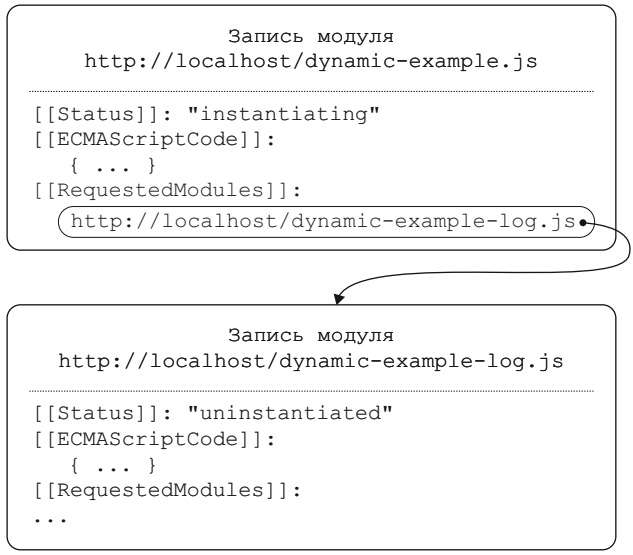


РИСУНОК 13-10

- Во время выполнения модуль точки входа случайным образом выбирает, какой динамический модуль использовать, и вызывает выражение `import()` для его загрузки, используя оператор `await` для ожидания результата. Теперь его выполнение на верхнем уровне завершено; позже, когда промис от `import()` будет выполнен, код продолжит выполнение и будет использовать модуль.
- Хост и движок JavaScript снова запускают процесс получения и синтаксического анализа, создания экземпляра и выполнения, начиная с динамического модуля. У динамического модуля есть два статических импорта: `log` из модуля log и `showTime` из модуля `dynamic-mod-showtime.js` (я буду называть его просто «модуль showtime»). Модуль showtime также импортирует `log` из модуля log. Когда синтаксический анализ завершен, дерево модулей для обработки этого вызова `import` содержит эти три модуля, но оно подключается к существующему дереву (из первоначального статического импорта). См. рисунок 13-11. Обратите внимание, что статус («[[Status]]») модуля журнала получает статус `"evaluated"`, поэтому он уже прошел все три части процесса загрузки и больше не обрабатывается. Однако модуль showtime еще не загружен, поэтому он проходит через процесс с динамическим модулем и, поскольку он находится глубже в дереве загрузки, выполняется первым («showtime module evaluated»).

Затем выполняется выполнение динамического модуля («dynamic module number 1 evaluated»). (Единственная причина, по которой в этом примере используется модуль showtime, — продемонстрировать, чем он отличается от модуля log, потому что модуль log уже был выполнен, но модуль showtime был впервые загружен динамическим модулем.)

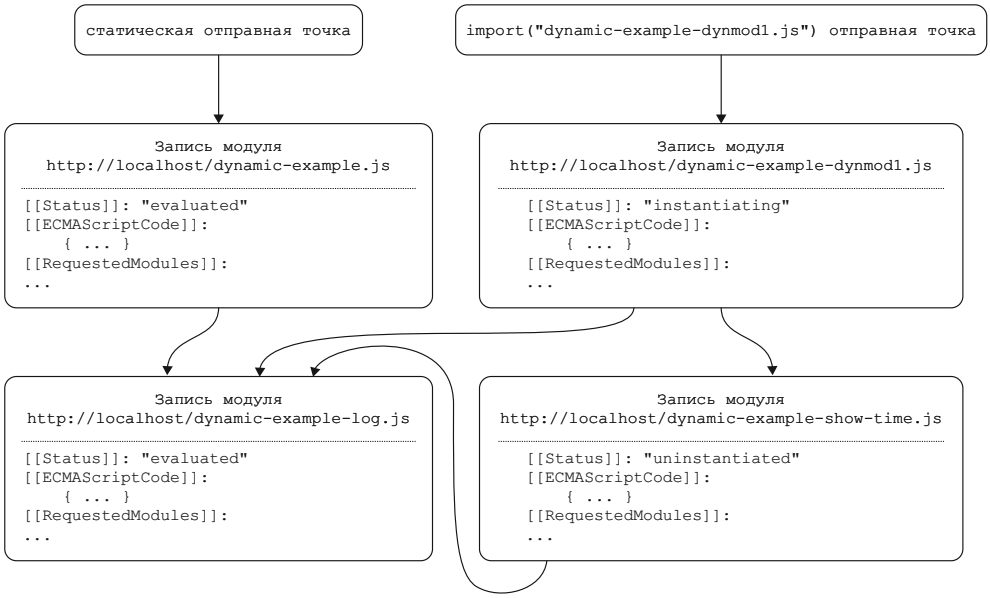


РИСУНОК 13-11

- Процесс загрузки для вызова выражения `import()` завершен, поэтому его промис выполнен с помощью объекта пространства имен модуля для динамического модуля.
- Функция модуля входа `async` продолжает выполнение с оператора `await` («entry point got module./dynamicmod1.js, calling mod.example») и вызывает `mod.example`, передавая его ссылку на функцию `log` из модуля log.
- Выполняется функция `example` динамического модуля, выводящая сообщение, относящееся к конкретному модулю, а затем строку «log === logFromEntry? true». Значение `true` показывает, что статический и динамический процессы работают с одним и тем же набором записей модуля и связанных с ними сред; загружена только одна копия модуля журнала и, следовательно, только одна функция `log`.
- Наконец, динамический модуль вызывает функцию `showTime`.

Динамический импорт в немодульных скриптах

В отличие от статического импорта, динамический импорт также может использоваться в скриптах, а не только в модулях. Посмотрите на Листинги 13–20 и 13–21: они представляют собой другую HTML-страницу и точку входа для более раннего динамического примера (повторное использование оставшихся файлов).

Листинг 13-20: Пример динамической загрузки модуля в скрипте (HTML) — dynamic-example-2.html

```
<!doctype html>
<html>
<head>
<meta charset="UTF-8">
<title>Dynamic Module Loading in Script - Example</title>
</head>
<body>
<script src="dynamic-example-2.js"></script>
</body>
</html>
```

Листинг 13-21: Пример динамической загрузки модуля в скрипте (точка входа) — dynamic-example-2.js

```
(async() => {
    try {
        const {log} = await import("./dynamic-mod-log.js");
        log("entry point module got log");
        const modName = `./dynamic-mod${Math.random() < 0.5 ? 1: 2}.js`;
        log(`entry point module requesting ${modName}`);
        const mod = await import(modName);
        log(`entry point module got module ${modName}, calling mod.example`);
        mod.example(log);
    } catch (error) {
        console.error(error);
    }
})();
```

Как видите, `dynamic-example-2.js` был загружен в виде скрипта, в его элементе `script` нет выражения `type="module"`. Если внимательно рассмотреть его код, он во многом похож на код из `dynamic-example.js`, но использует вызов `import()` вместо статического объявления `import`. При его запуске результат будет таким же — за исключением того факта, что код точки входа не смог зарегистрировать начало и окончание вычисления верхнего уровня, потому что ему пришлось ждать модуля log, поступающего после завершения вычисления верхнего уровня:

```
log module evaluated
entry point module got log
entry point module requesting ./dynamic-mod1.js
showtime module evaluated
dynamic module number 1 evaluated
entry point module got module ./dynamic-mod1.js, calling mod.example
Number 1! Number 1! Number 1!
log === logFromEntry ? true
Time is 12:44
```

«Под капотом» это работает в основном так же — за исключением того, что связь между точкой входа и модулем log больше невозможно определить с помощью статического анализа (хотя, поскольку код использует строковый литерал для вызова `import()`, умный инструмент может решить эту задачу).

Одно ключевое отличие заключается в том, что вместо простого импорта именованного экспорта в `log` из модуля log динамическая версия импортирует весь объект пространства имен для модуля журнала (но использует деструктуризацию, чтобы выбрать только функцию `log`). Напротив, объект пространства имен модуля никогда не создается в статическом примере, потому что его никогда никто не запрашивает.

ВСТРЯХИВАНИЕ ДЕРЕВА

Встряхивание дерева — это форма *устранения мертвого кода*. *Живой код* — это код, который потенциально может быть использован страницей или приложением; *мертвый код* — это код, который определенно не будет использоваться страницей или приложением (в его текущем виде). Встряхивание дерева — это процесс анализа дерева модулей, чтобы избавиться от мертвого кода. (Термин «встряхивание дерева» происходит от образа энергичного встряхивания дерева, чтобы сухая древесина опала, в то время как живая остается на месте.)

Давайте пересмотрим первый пример в этой главе, добавив функцию в используемый им модуль log; см. Листинги с 13-22 по 13-24.

Листинг 13-22: Обновленный простой пример модуля (HTML) — simple2.html

```
<!doctype html>
<html>
<head>
<meta charset="UTF-8">
<title>Revised Simple Module Example</title>
</head>
<body>
<script src="simple2.js" type="module"></script>
</body>
   </html>
```

Листинг 13-23: Обновленный простой пример модуля (модуль точки входа) — simple2.js

```
import {log} from "./log2.js";

log("Hello, modules!");
```

Листинг 13-24: Обновленный простой пример модуля (модуль log) — log2.js

```
export function log(msg) {
    const p = document.createElement("pre");
    p.appendChild(document.createTextNode(msg));
    document.body.appendChild(p);
}
export function stamplog(msg) {
    return log(`${new Date().toISOString()}: ${msg}`);
}
```

Если вы просмотрите код в листингах, то заметите пять моментов, достойных внимания:

- Ничто никогда не запрашивает именованный экспорт `stamplog`.
- Ничто никогда не запрашивает объект пространства имен модуля для модуля log (если что-то запросит, интеллектуальный инструмент встряхивания дерева все равно может выполнить анализ экранирования результирующего объекта, чтобы увидеть, используется ли когда-либо его свойство `stamplog`).
- Ничто в коде верхнего уровня модуля log не использует функцию `stamplog`.
- Ничто в других экспортируемых функциях модуля log или вызываемых ими функциях не использует функцию `stamplog`.
- Используются только статические объявления `import`, а не вызовы `import()`.

Кроме того, вся эта информация может быть определена с помощью *статического анализа* (просто парсинг и проверка кода без его запуска). Это означает, что, если указать пакету JavaScript на `simple.html` и указать ему объединить весь код в оптимизированный файл, он мог бы определить без запуска кода, что функция `stamplog` нигде не использовалась и может быть опущена.

Это и есть встряхивание дерева. Это одна из причин, по которой бандлеры JavaScript не исчезнут в обозримом будущем, несмотря на встроенную поддержку модулей в браузерах. Вот результат использования одного пакета в этом простом примере, в данном случае с использованием Rollup.js и установив для его параметра вывода значение `iife` (выражение функции immediatelyinvoked), но вы получите аналогичные результаты с любым бандлером встряхивания дерева (подробнее в следующем разделе):

```
(function () {
    "use strict";

    function log(msg) {
        const p = document.createElement("pre");
        p.appendChild(document.createTextNode(msg));
        document.body.appendChild(p);
    }

    log("Hello, modules!");

}());
```

Этот блок кода — оптимизированная (но не уменьшенная) комбинация `simple.js` плюс `log.js`. Функцию `stamplog` нигде не найти.

Помните последний из «пяти моментов, достойных внимания»?

- Используются только статические объявления `import`, а не вызовы `import()`.

Теоретически даже однократное использование выражения `import()` с чем-либо, кроме строкового литерала, в любом модуле дерева означает, что инструменты не могут выполнять встряхивание дерева, поскольку не могут доказать, что что-то действительно не используется. На практике разные бандлеры пакетов по-разному оптимизируют влияние динамического импорта и, вероятно, предоставляют параметры конфигурации, позволяющие вам контролировать, насколько агрессивно они пытаются встряхивать дерево.

БАНДЛИНГ (ОБЪЕДИНЕНИЕ)

Хотя модули изначально поддерживаются современными браузерами, люди уже некоторое время используют синтаксис, используя бандлеры JavaScript для преобразования модулей в оптимизированные файлы, работающие в браузерах без поддержки модулей. Такие проекты, как Rollup.js (https://rollupjs.org) и Webpack (https://webpack.js.org/), чрезвычайно популярны и многофункциональны.

В проекте любого размера многие используют бандлер, даже если проект ориентирован только на браузеры с поддержкой встроенных модулей. Встряхивание дерева — одна из причин. Другая заключается в том, что передача всех ресурсов отдельных модулей по протоколу HTTP (даже HTTP/2) по-прежнему потенциально медленнее, чем передача одного ресурса. (Определенно медленнее, если вы застряли на HTTP/1.1.)

Google провел анализ[80], сравнив загрузку собственных модулей с загрузкой пакетов, используя две популярные существующие библиотеки — Moment.js и Three.js, а также множество искусственно созданных модулей, чтобы определить, по-прежнему ли полезно объединение и где находятся узкие места в конвейере загрузки Chrome. Протестированная версия Moment.js[81] использовала 104 модуля с максимальной глубиной дерева 6; протестированная версия Three.js[82] использовала 333 модуля с максимальной глубиной 5.

Основываясь на результатах анализа, Адди Османи и Матиас Байненс, оба из Google, рекомендуют[83]:

- Использовать встроенную поддержку во время разработки.
- Использовать встроенную поддержку в продакшне для небольших веб-приложений с менее чем 100 модулями и неглубоким деревом зависимостей (максимальная глубина менее 5), предназначенных только для современных браузеров.
- Использовать пакетирование для производства чего-либо, превышающего рекомендованный размер, или проектов, ориентированных на браузеры без поддержки модулей.

В любом случае есть компромиссы, особенно если вы по какой-то причине застряли в HTTP 1.1. Полную статью стоит прочитать; скорее всего, со временем они добавят ссылки на обновленные исследования.

МЕТАДАННЫЕ ИМПОРТА

Иногда модулю может потребоваться узнать что-то о себе, например, с какого URL-адреса или пути он был загружен, является ли он «основным» модулем (в средах, где есть один основной модуль, например Node.js) и т. д. Модули ES2015 в настоящее время не предоставляют никакого способа для модуля получить эту информацию. ES2020 добавляет способ — `import.meta`[84]. Это специальный объект модуля, содержащий его свойства.

---

[80] https://docs.google.com/document/d/1ovo4PurT_1K4WFwN2MYmmgbLcr7v6DRQN67ESVA-wq0/pub

[81] https://momentjs.com/

[82] https://threejs.org/

[83] https://developers.google.com/web/fundamentals/primers/modules

[84] https://github.com/tc39/proposal-import-meta

Сами свойства задаются хостом и будут варьироваться в зависимости от среды (например, браузер или Node.js).

В веб-среде свойства `import.meta` определяются спецификацией HTML в разделе `HostGetImportMetaProperties`[85]. На данный момент определено только одно свойство — `url`. Это строка, дающая полностью разрешенный URL-адрес модуля. Например, загруженный из `http://localhost` импорт `import.meta.url` в модуль `mod.js` будет содержать значение `"http://localhost/mod.js"`. Node.js также поддерживает `url`[86], предоставляя абсолютный `file:` URL-адрес модуля. Со временем, вероятно, в одну или обе среды будут добавлены дополнительные свойства.

Объект `import.meta` создается при первом обращении к нему посредством координации между движком JavaScript и хостом. Поскольку объект зависит от модуля, он в значительной степени принадлежит модулю, и это отражается в том факте, что он никоим образом не заблокирован: вы можете добавлять к нему свойства для своих собственных целей (хотя неясно, зачем вам это может понадобиться) или даже изменить свойства по умолчанию, с которыми он поставляется (если только хост не запрещает это).

МОДУЛИ ВОРКЕРОВ

По умолчанию веб-воркеры загружаются как классические скрипты, а не как модули. Они могут загружать другие скрипты через `importScripts`, но это старомодный способ, когда связь со скриптом загрузки осуществляется через глобальную среду воркера. Но веб-воркеры могут быть модулями, так что они могут использовать объявления импорта и экспорта.

Аналогичным образом воркеры Node.js теперь также могут быть модулями ESM.

Давайте рассмотрим каждый из них.

Загрузка веб-воркера в качестве модуля

Чтобы загрузить воркер как модуль и чтобы он получил все обычные преимущества модуля, используется параметр `type` в объекте `options`, переданном в качестве второго аргумента конструктору `Worker`:

```
const worker = new Worker("./worker.js", {type: "module"});
```

Можно запустить воркер таким образом из модуля или классического скрипта. Кроме того, в отличие от классических рабочих скриптов, вы можете запускать модули воркеров из разных источников при условии, что им предоставляется информация о совместном использовании ресурсов из разных источников (CORS), позволяющая запускающему источнику использовать их.

Если браузер поддерживает воркеры в качестве модулей, воркер будет загружен как модуль. На момент написания этой книги в начале 2020 года поддержка браузером модулей воркеров не получила широкого распространения, хотя Chrome и другие

[85] `https://html.spec.whatwg.org/multipage/webappapis.html#hostgetimportmetaproperties`

[86] `https://nodejs.org/api/esm.html#esm_import_meta`

браузеры, основанные на проекте Chromium (Chromium, более новые версии Edge), поддерживают их.

В Листингах с 13-25 по 13-27 показан пример загрузки рабочего элемента в виде модуля.

Листинг 13-25: Веб-воркер в виде модуля (HTML) — worker-example.html

```
<!doctype html>
<html>
<head>
<meta charset="UTF-8">
<title>Web Worker Module Example</title>
</head>
<body>
<script>
const worker = new Worker("./worker-example-worker.js", {type: "module"});
worker.addEventListener("message", e => {
    console.log(`Message from worker: ${e.data}`);
});
</script>
</body>
</html>
```

Листинг 13-26: Веб-воркер в виде модуля (воркер) — worker-example-worker.js

```
import {example} from "./worker-example-mod.js";

postMessage(`example(4) = ${example(4)}`);
```

Листинг 13-27: Веб-воркер в виде модуля (модуль) — worker-example-mod.js

```
export const example = a => a * 2;
```

Загрузка воркера Node.js в качестве модуля

В Node.js поддержка модулей ESM распространяется и на потоки воркера. Применяются те же правила, что и для других способов выполнения кода. Вы можете запустить рабочий процесс в качестве модуля, имея настройку `type: "module"` в ближайшем файле **package.json**, или выдав воркеру расширение `.mjs`.

Воркер находится в собственной базе realm

При создании воркер помещается в новую базу realm: у него есть своя собственная глобальная среда, свои собственные внутренние объекты и т. д. Если воркер является модулем, любые загружаемые им модули будут загружены в пределах этой базы realm отдельно от модулей, загруженных в других базах realm. Например, в браузере, если модуль в главном окне загружает `mod1.js` и запускает воркер, который также загружает `mod1.js`, модуль `mod1.js` загружается дважды: один раз в базу realm окна, второй раз в базу realm воркера. Если воркер загружает другой модуль (скажем, `mod2.js`), который также импортирует из `mod1.js`, этот другой модуль и модуль воркера совместно используют общую копию `mod1.js`; но они *не могут* использовать совместно копию, загружаемую главным окном (рисунок 13-12).

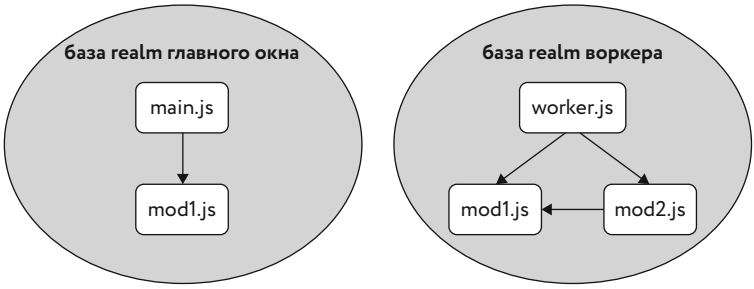


РИСУНОК 13-12

Базы realm изолированы не только от базы realm главного окна, но и друг от друга. Поэтому, если вы запустите десять воркеров, все они загрузят модуль `mod1.js`, и он будет загружен десять раз (один раз на каждого воркера). Поскольку каждый воркер находится в своей собственной базе realm, у каждого есть свое собственное дерево модулей.

Это касается веб-браузеров и Node.js.

ОТ СТАРЫХ ПРИВЫЧЕК К НОВЫМ

Перед вами несколько старых привычек, которые вы могли бы пересмотреть.

Используйте модули вместо псевдопространств имен

Старая привычка: Использование объекта «пространства имен» для предоставления единственного общедоступного символа для вашего кода (например, в библиотеке, как глобальный объект `$` в jQuery):

```
var MyLib = (function(lib) {
    function privateFunction() {
        // ...
}
    lib.publicFunction = function() {
        // ...
};
    return lib;
})(MyLib || {});
```

Новая привычка: Используйте модули:

```
function privateFunction() {
    // ...
}
export function publicFunction() {
    // ...
}
```

Примечание: не экспортируйте объекты со свойствами/методами, которые предназначены для индивидуального использования. Вместо этого используйте именованный экспорт.

Используйте модули вместо обертывания кода в функции области видимости

Старая привычка: Использование «функции области видимости» вокруг вашего кода, чтобы избежать создания глобальных переменных:

```
(function() {
    var x = /* ... */;
    function doSomething() {
        // ...
}
    // ...
})();
```

Новая привычка: Используйте модули, поскольку область верхнего уровня модуля не является глобальной областью:

```
let x = /* ... */;
function doSomething() {
    // ...
}
// ...
```

Используйте модули, чтобы избежать создания мегалитических файлов кода

Старая привычка: Наличие одного огромного файла кода позволяет избежать необходимости беспокоиться об объединении файлов меньшего размера для продакшна. (Но у вас ведь не было этой привычки, правда? Правда?)

Новая привычка: Используйте различные модули правильного размера со статически объявленными зависимостями между ними. Вопрос о том, какой размер является правильным, будет вопросом стиля и того, что нужно согласовать с вашей командой. Помещение каждой функции в отдельный модуль, вероятно, излишне. Но, с другой стороны, один массивный модуль, экспортирующий одно, другое и третье, вряд ли достаточно модульный.

Конвертируйте CJS, AMD и другие модули в ESM

Старая привычка: Использовать CJS, AMD и другие модули (поскольку не было собственного формата модуля JavaScript).

Новая привычка: Используйте ESM, преобразуя ваши старые модули (либо одним большим скачком, либо, что более вероятно, медленно с течением времени).

Не изобретайте велосипед, используйте хорошо обслуживаемый бандлер

Старая привычка: Любой из нескольких специализированных механизмов для объединения ваших отдельных файлов, используемых для разработки, в объединенный файл (или небольшой набор файлов) для продакшна.

Новая привычка: Используйте хорошо обслуживаемый, поддерживаемый сообществом бандлер.

14

Рефлексия — объекты Reflect и Proxy

СОДЕРЖАНИЕ ГЛАВЫ

- Объект `Reflect`
- Объекты `Proxy`

В этой главе вы узнаете об объекте `Reflect`, предоставляющем множество полезных функций для создания объектов и взаимодействия с ними, и о `Proxy`, предоставляющем окончательный шаблон «фасад» для JavaScript — важный для предоставления API для клиентского кода шаблон.

Объекты `Reflect` и `Proxy` предназначены для совместного использования, хотя каждый из них может использоваться отдельно. В этой главе мы кратко познакомимся с объектом `Reflect`, но потом более подробно рассмотрим его в контексте изучения `Proxy`.

ОБЪЕКТ REFLECT

Объект `Reflect` был добавлен в ES2015 с различными методами, соответствующими основным операциям, которые выполняются над объектами: получение и установка значения свойства, получение и установка прототипа объекта, удаление свойства из объекта и т. д.

Ваша первая мысль, вероятно, такова: «Зачем для этого нужны служебные функции? Мы уже можем делать эти вещи, просто используя операторы непосредственно с объектом». Это отчасти верно, и в тех случаях, когда это не так, обычно есть функция (или комбинация функций) для `Object`, выполняющая эту задачу. Однако объект `Reflect` вносит несколько изменений в таблицу:

- Он предоставляет тонкую функцию-оболочку для всех основных операций с объектами, вместо того чтобы некоторые из них были синтаксическими,

а другие — функциями `Object` или комбинациями функций `Object`. Это означает, что вы можете передавать основные операции с объектами без необходимости писать свою собственную оболочку вокруг таких вещей, как операторы `in` или `delete`.
- Его функции предоставляют возвращаемые значения для успеха/неудачи в случаях, когда эквивалентные функции `Object` вместо этого выдают ошибки (подробнее об этом чуть позже).
- Как вы узнаете из раздела о прокси, он предлагает функции, идеально соответствующие каждой «ловушке», к которой может подключиться прокси-объект, реализуя поведение по умолчанию для ловушки, упрощая правильную реализацию ловушек, которые только изменяют поведение, а не заменяют его полностью.
- Он предоставляет операцию, недоступную каким-либо другим способом за пределами синтаксиса `class`: `Reflect.construct` с аргументом `newTarget`.

У `Reflect` и `Object` есть несколько функций с одинаковыми именами, в основном с одним и тем же назначением:

- `defineProperty`
- `getOwnPropertyDescriptor`
- `getOwnPropertyDescriptor`
- `getPrototypeOf`
- `setPrototypeOf`
- `preventExtensions`

Они выполняют в основном одни и те же задачи, но между ними есть небольшие различия:

- В общем случае версия `Reflect` выдаст ошибку, если вы передадите ей не-объект, где она ожидает объект. А версия `Object` будет (во многих, но не во всех случаях) либо принудительно привязывать примитив к объекту и работать с результатом, либо просто игнорировать вызов. (Исключением является метод `Object.defineProperty`, который выбросит ошибку, если вы передаете ему не-объект, как это делают функции `Reflect`.)
- В общем случае версия модификации функции `Reflect` возвращает флаг успеха/неудачи, тогда как версия `Object` возвращает передаваемый для изменения объект (и если вы передаете ему примитив, либо возвращает объект, приведенный к примитиву, либо просто возвращает примитив как есть, если вызов ничего не делает с примитивами). Версия `Object` обычно выдает ошибку при сбое.

У `Reflect` также есть отдельная функция `ownKeys`, внешне похожая на `Object.keys`. Но у нее два существенных отличия, о которых вы узнаете чуть позже.

Хотя большинство функций `Reflect` в первую очередь полезны для реализации объектов прокси, давайте рассмотрим некоторые, полезные и сами по себе.

Метод Reflect.apply

Метод `Reflect.apply` — это служебная функция, выполняющая то же, что и метод `apply` функции. Он вызывает функцию с определенным значением `this` и списком аргументов, предоставленным в виде массива (или массивоподобного объекта):

```
function example(a, b, c) {
    console.log(`this.name = ${this.name}, a = ${a}, b = ${b}, c = ${c}`);
}

const thisArg = {name: "test"};
const args = [1, 2, 3];
Reflect.apply(example, thisArg, args); // this.name = test, a = 1, b, = 2, c = 3
```

В этом примере можно заменить вызов `Reflect.apply` вызовом `example.apply`:

```
example.apply(thisArg, args); // this.name = test, a = 1, b, = 2, c = 3
```

Однако у метода `Reflect.apply` есть несколько преимуществ:

- Он работает, даже если свойство `apply` в `example` было переопределено или заменено чем-то отличным от `Function.prototype.apply`.
- Небольшая вариация на этот счет: он работает в действительно неясном случае, когда прототип функции был изменен (например, на `Object.setPrototypeOf`), так что функция больше не наследует от `Function.prototype`, и поэтому у нее нет свойства `apply`.
- Он работает с любым *вызываемым* объектом (любым объектом с внутренней операцией [[Call]]), даже если это не настоящая функция JavaScript. Их стало меньше, но не так давно «функции», предоставляемые хостом, часто не были настоящими функциями JavaScript и у них не было свойства `apply` и т. п. (Некоторые из них все еще существуют.)

Метод Reflect.construct

Метод `Reflect.construct` создает новый экземпляр при помощи функции конструктора, как это делает оператор `new`. Тем не менее метод `Reflect.construct` поддерживает две новые функциональные возможности, которых нет у оператора `new`:

- Он принимает аргументы конструктора в виде массива (или массивоподобного объекта).
- Это позволяет установить `new.target` для чего-то другого, кроме вызываемой функции.

Давайте сначала рассмотрим аспект аргументов, поскольку он довольно прост. Предположим, у вас есть аргументы, которые вы хотите использовать с конструктором (`Thing`) в массиве. В ES5 было неудобно вызывать конструктор с аргументами из массива. Самым простым способом было создать объект самостоятельно, а затем вызвать конструктор как обычную функцию с помощью `apply`:

```
// В спецификации ES5
var o = Thing.apply(Object.create(Thing.prototype), theArguments);
```

Это работает в спецификации ES5 при условии, что конструктор `Thing` использует `this`, с которым он был вызван (это обычное дело), а не создает свой собственный объект.

В ES2015 + есть два способа сделать это — синтаксис расширения и метод `Reflect.construct`:

```
// В спецификации ES2015+
let o = new Thing(...theArguments);
// или
let o = Reflect.construct(Thing, theArguments);
```

Поскольку синтаксис расширения проходит через итератор, он выполняет больше работы, чем метод `Reflect.construct`, получающий доступ к свойствам `length`, `0`, `1` и т. д. аргументов «массива» непосредственно (не то, чтобы это обычно имело значение). Это означает, что расширение работает с итеративными не массивоподобными объектами, с которыми не работает `Reflect.construct`. А `Reflect.construct` работает с массивоподобными не итерируемыми объектами, в которых не работает расширение.

Второй возможностью, предоставляемой методом `Reflect.construct`, но отсутствующей у `new`, стал контролируемый синтаксис `new.target`. В первую очередь он необходим при использовании `Reflect` с прокси (более подробно в следующем разделе), но его также можно применять для создания экземпляров, относящихся к подтипам таких встроенных классов `Error` или `Array` (который, как известно, не может быть правильно подклассирован с использованием только функций ES5), без использования синтаксиса `class`. Некоторые программисты не любят использовать ключевые слова `class` или `new`, предпочитая создавать объекты другими способами. Но им все еще может потребоваться создать свои собственные экземпляры `Error`, `Array` или `HTMLElement` (для веб-компонентов), имеющих другой прототип (с дополнительными функциями). Если вы не хотите использовать `class` и `new` для этого, `Reflect.construct` вполне подойдет:

```
// Определение функции, создающей пользовательские ошибки
function buildCustomError(...args) {
    return Reflect.construct(Error, args, buildCustomError);
}
buildCustomError.prototype = Object.assign(Object.create(Error.prototype), {
    constructor: buildCustomError,
    report() {
        console.log(`this.message = ${this.message}`);
    }
});

// Применение метода
const e = buildCustomError("error message here");
console.log("instanceof Error", e instanceof Error);
e.report();
console.log(e);
```

Выделенная строка вызывает объект `Error`, но передает его в `buildCustomError` для параметра `new.target`. Это означает, что методу `buildCustomError`.

prototype присваивается вновь созданный объект, а не Error.prototype. Результирующий экземпляр наследует пользовательский метод report.

Метод Reflect.ownKeys

Функция Reflect.ownKeys внешне похожа на Object.keys, но возвращает массив *всех* собственных ключей свойств объекта, включая неисчислимые и те, которые именованы символами, а не строками. Несмотря на схожесть названий, это больше похоже на сочетание методов Object.getOwnPropertyNames и Object.getOwnPropertySymbols, чем на Object.keys.

Методы Reflect.get и Reflect.set

Эти функции получают и устанавливают свойства объекта, но с удобной дополнительной возможностью. Если свойство, к которому осуществляется доступ, — это свойство-*акцессор*, вы можете управлять значением this, находящимся в вызове акцессора. По сути они представляют собой версию свойства-акцессора Reflect.apply.

Установка this для объекта, отличного от того, от которого вы получаете свойство-акцессор, может показаться странной. Но, если поразмышлять, в главе 4 уже было описано нечто, выполняющее такую операцию, — super. Рассмотрим этот базовый класс, вычисляющий произведение своих параметров построения, и подкласс, который принимает этот результат и удваивает его:

```
class Product {
    constructor(x, y) {
        this.x = x;
        this.y = y;
}
    get result() {
        return this.x * this.y;
}
}
class DoubleProduct extends Product {
    get result() {
        return super.result * 2;
}
}

const d = new DoubleProduct(10, 2);
console.log(d.result);          // 40
```

Акцессору result подкласса DoubleProduct необходимо использовать метод super.result для запуска акцессора result из класса Product, а не подкласса DoubleProduct, потому что если будет использована версия из подкласса DoubleProduct, он будет вызывать себя рекурсивно и станет причиной переполнения стека. Но при вызове акцессора result в версии из класса Product ему необходимо убедиться, что this задан для экземпляра (d1), и, следовательно, используются правильные значения для x и y. Объект, из которого поступает акцессор (Product.prototype — прототип прототипа d1), и объект, который должен быть this в вызове акцессора (d1), — это разные объекты.

Чтобы сделать это без ключевого слова `super`, необходимо использовать метод `Reflect.get`. Без применения `Reflect.get` вам потребовалось бы получить дескриптор свойства для `result` и вызвать его метод `get` при помощи `get.call/apply` или `Reflect.apply`, что очень неуклюже:

```
get result() {
    const proto = Object.getPrototypeOf(Object.getPrototypeOf(this));
    const descriptor = Object.getOwnPropertyDescriptor(proto, "result");
    const superResult = descriptor.get.call(this);
    return superResult * 2;
}
```

Гораздо проще реализовать это при помощи `Reflect.get`:

```
get result() {
    const proto = Object.getPrototypeOf(Object.getPrototypeOf(this));
    return Reflect.get(proto, "result", this) * 2;
}
```

Конечно, вам не обязательно делать это в данном конкретном случае, потому что у вас есть ключевое слово `super`. Метод `Reflect.get` позволяет справляться с ситуациями, когда у вас нет `super` (например, как показано далее, в обработчике прокси) или там, где использование `super` недопустимо.

У `Reflect.get` следующая сигнатура:

```
value = Reflect.get(target, propertyName[, receiver]);
```

- `target` — объект, из которого можно получить свойство.
- `propertyName` — имя получаемого свойства.
- `receiver` — необязательный объект для использования в качестве `this` во время вызова акцессора, если это свойство-акцессор.

Он возвращает значение свойства.

Метод `Reflect.get` получает дескриптор свойства для параметра `propertyName` из параметра `target`, и, если дескриптор предназначен для свойства данных, возвращает значение этого свойства. Но если дескриптор предназначен для акцессора, метод `Reflect.get` вызывает функцию акцессор, используя параметр `receiver` в качестве `this` (как это делал «неуклюжий» код выше).

Метод `Reflect.set` так же, как метод `Reflect.get`, только задавая свойство, а не получая его. У него следующая сигнатура:

```
result = Reflect.set(target, propertyName, value[, receiver]);
```

Параметры `target`, `propertyName` и `receiver` точно такие же, `value` — задаваемое значение. Функция возвращает значение `true`, если значение было задано, `false`, если нет.

Оба варианта особенно удобны при работе с прокси; об этом вы узнаете далее в этой главе.

Другие функции `Reflect`

Остальные функции `Reflect` перечислены в Таблице 14-1.

Таблица 14-1. Другие функции Reflect

`defineProperty`	Похожа на `Object.defineProperty`, но возвращает значение `true` в случае успешного выполнения, или `false`, а не ошибку, в случае неудачи.
`deleteProperty`	Функциональная версия оператора `delete` — за исключением того, что она всегда возвращает `true` в случае успеха или `false` при неудачном выполнении, даже в строгом режиме (тогда как оператор `delete` в строгом режиме при сбое выдает ошибку).
`getOwnPropertyDescriptor`	Похожа на `Object.getOwnPropertyDescriptor` — за исключением того, что она выбрасывает исключение, если вы передаете ей не-объект (а не осуществляет приведение).
`getPrototypeOf`	Похожа на `Object.getPrototypeOf` — за исключением того, что она выбрасывает исключение, если вы передаете ей не-объект (а не осуществляет приведение).
`has`	Функциональная версия оператора `in` (`has` также проверяет прототипа, *в отличие* от `hasOwnProperty`).
`isExtensible`	Похожа на `Object.isExtensible` — за исключением того, что при получении не-объекта она выбрасывает исключение, а не просто выбрасывает значение `false`.
`preventExtensions`	Похожа на `Object.preventExtensions` — за исключением: 1) если вы передаете ей не-объект, она выбрасывает ошибку, а не просто ничего не делает и возвращает вам это значение; 2) она возвращает значение `false`, если операция завершается неудачей (вместо того, чтобы выбросить ошибку).
`setPrototypeOf`	Похожа на `Object.setPrototypeOf` — за исключением: 1) если вы передаете ей не-объект, она выдает ошибку, а не просто ничего не делает и возвращает вам это значение; 2) она возвращает значение `false`, если операция завершается неудачей (вместо того, чтобы выдавать ошибку).

ОБЪЕКТ PROXY

Прокси-объекты — это объекты, которые можно использовать для перехвата основных операций с объектами на пути к целевому объекту. При создании прокси можно определить обработчики для одной или нескольких *ловушек* для обработки операций, таких как получение значения свойства или определение нового свойства.

Существует множество вариантов использования прокси:

- Запись операций, выполняемых с объектом.
- Превращение чтения/записи несуществующих свойств в ошибку (вместо возврата значения `undefined` или создания свойства).

- Обеспечение границы между двумя битами кода (например, API и его потребителем).
- Создание представлений изменяемых объектов, доступных только для чтения.
- Сокрытие информации в объекте или создание впечатления, что объект содержит больше информации, чем на самом деле.

…и многое другое. Поскольку прокси позволяют подключаться к основным операциям и (в большинстве случаев) изменять их, существует очень мало ограничений на то, что вы можете с ними делать.

Давайте рассмотрим простой пример; см. Листинг 14-1.

Листинг 14-1: Начальный пример прокси — initial-proxy-example.js

```
const obj = {
    testing: "abc"
};
const p = new Proxy(obj, {
    get(target, name, receiver) {
        console.log(`(getting property '${name}')`);
        return Reflect.get(target, name, receiver);
    }
});
console.log("Getting 'testing' directly...");
console.log(`Got ${obj.testing}`);
console.log("Getting 'testing' via proxy...");
console.log(`Got ${p.testing}`);
console.log("Getting non-existent property 'foo' via proxy...");
console.log(`Got ${p.foo}`);
```

Код в Листинге 14-1 создает прокси, определяющий обработчик для ловушки `get` (получение значения свойства). Когда вы запустите это, вы увидите:

```
Getting 'testing' directly...
Got abc
Getting 'testing' via proxy...
(getting property 'testing')
Got abc
Getting non-existent property 'foo' via proxy...
(getting property 'foo')
Got undefined
```

Есть несколько моментов, на которые стоит обратить внимание:

- Чтобы создать прокси, конструктору `Proxy` передается целевой объект и объект с *обработчиками ловушек*. В примере определен один обработчик ловушки: для операции `get`.
- Операции, выполняемые непосредственно над целевым объектом, не запускают прокси: это делают только операции, выполняемые через прокси. Обратите внимание, что сообщения «`(getting property 'testing')`» нет между сообщениями «`Getting 'testing' directly...`» и «`Got abc`».

- Ловушка `get` не специфична для одного отдельного свойства: все обращения к свойствам прокси проходят через нее, даже обращения к несуществующим свойствам, таким как `foo` в конце примера.

Даже если прокси могли бы только прослушивать основные операции, они были бы чрезвычайно полезны, но они также могут изменить результат операции или даже полностью пресечь операцию. В предыдущем примере обработчик вернул значение базового объекта из обработчика без изменений, но он мог его изменить. Запустите код в Листинге 14-2, чтобы посмотреть пример.

Листинг 14-2: Простой пример прокси с изменением — simple-mod-proxy-example.js

```
const obj = {
    testing: "abc"
};
const p = new Proxy(obj, {
    get(target, name, receiver) {
        console.log(`(getting property '${name}')`);
        let value = Reflect.get(target, name, receiver);
        if (value && typeof value.toUpperCase === "function") {
            value = value.toUpperCase();
        }
        return value;
    }
});
console.log("Getting directly...");
console.log(`Got ${obj.testing}`);
console.log("Getting via proxy...");
console.log(`Got ${p.testing}`);
```

В примере из Листинга 14-2 обработчик изменяет значение перед его возвратом (если у него есть метод `toUpperCase`), в результате чего получается следующее:

```
Getting directly...
Got abc
Getting via proxy...
(getting property 'testing')
Got ABC
```

Для каждой основной операции с объектом существует прокси-ловушка; обработчики для ловушек могут делать практически все (о некоторых ограничениях вы узнаете чуть позже). В Таблице 14-2 далее приведен список имен ловушек и основных операций, которые они перехватывают (имена — это имена основных операций в спецификации). Мы рассмотрим ловушки более детально, как только разберемся с основами.

В ES2015 первоначально определили 14-ю ловушку — `enumerate`, которая была спровоцирована инициализацией цикла `for-in` для прокси (и соответствующей функцией `Reflect.enumerate`). Но ее удалили в ES2016 из-за проблем с производительностью и наблюдаемостью, поднятых разработчиками движка JavaScript: когда они углубились в детали ее реализации, оказалось, что она может быть эффективной и содержать заметные побочные эффекты (например, помогает определить, имеете ли вы дело

с прокси или нет), или это может быть неэффективно. (Это пример того, почему новый процесс улучшения, о котором вы узнали в главе 1, представляется хорошей идеей; что-то не попадает в спецификацию, пока не будет создано несколько реализаций, позволяющих устранить такого рода проблемы до того, как функция попадет в спецификацию.)

Если не указать обработчик для ловушки, прокси-сервер перенаправит операцию непосредственно на свой целевой объект. Вот почему код в Листинге 14-2 нужен только для определения обработчика ловушки `get`.

Мы подробно рассмотрим каждую ловушку позже в этой главе, но давайте начнем с примера одного варианта использования прокси, демонстрирующего все ловушки — прокси, регистрирующий все основные операции над объектом.

Таблица 14-2. Ловушки прокси

Имя ловушки	Имя операции	Вызывается, когда
`apply`	`[[Call]]`	…прокси используется в качестве функции при вызове функции (доступно только при «проксировании» функций)
`construct`	`[[Construct]]`	…прокси используется в качестве конструктора (доступно только при «проксировании» конструктора)
`defineProperty`	`[[DefineOwnProperty]]`	…свойство было определено или переопределено для прокси (включая ранее заданное значение, если это свойство относится к данным)
`deleteProperty`	`[[Delete]]`	…свойство удаляется из прокси
`get`	`[[Get]]`	…значение свойства восстанавливается из прокси
`getOwnPropertyDescriptor`	`[[GetOwnProperty]]`	…дескриптор свойства восстанавливается из прокси (это происходит значительно чаще, чем вы думаете)
`getPrototypeOf`	`[[GetPrototypeOf]]`	…восстанавливается прототип прокси
`has`	`[[HasProperty]]`	…существование свойства проверяется с помощью прокси (например, с использованием `in` или аналогичного оператора)
`isExtensible`	`[[IsExtensible]]`	…прокси проверяется на возможность расширения (т. е. у него не было расширений)
`ownKeys`	`[[OwnPropertyKeys]]`	…восстанавливаются собственные имена свойств прокси
`preventExtensions`	`[[PreventExtensoins]]`	…расширения запрещены для прокси
`set`	`[[Set]]`	…задается значение свойства для прокси
`setPrototypeOf`	`[[StePrototypeOf]]`	…задается прототип прокси

Пример: Регистрирующий прокси

В этом разделе вы увидите регистрирующий прокси, записывающий все основные операции, выполняемые с объектом. В примере указано, какая ловушка была вызвана, и показаны значения параметров, с которыми она была вызвана. Весь код в этом примере находится в разделе загрузки (`logging-proxy.js`), который необходимо запустить локально, чтобы увидеть результаты, но давайте рассмотрим его по частям.

Чтобы сделать вывод более понятным, мы будем отслеживать имена объектов, а затем при выводе на экран сообщения, в котором говорится, что объект был значением параметра, мы будем использовать его имя, а не показывать содержимое объекта. (Есть еще одна причина, по которой этот пример делает это: когда объект, который нам нужно регестрировать, представлен прокси-объектом, регистрация его содержимого вызовет прокси-ловушки, а в случае с некоторыми из этих ловушек это вызовет рекурсию, ведущую к переполнению стека.)

Код начинается с регистрирующей функции `log`:

```
const names = new WeakMap();
function log(label, params) {
    console.log(label + ": " + Object.getOwnPropertyNames(params).map(key => {
        const value = params[key];
        const name = names.get(value);
        const display = name ? name: JSON.stringify(value);
        return `${key} = ${display}`;
    }).join(", "));
}
```

Карта `names` — это то, что код использует для отслеживания имен объектов. Например, этот код:

```
const example = {"answer": 42};
names.set(example, "example");
log("Testing 1 2 3", {value: example});
```

будет выводить имя «example», а не содержимое объекта:

```
Testing 1 2 3: value = example
```

После функции `log` код продолжается определением объекта `handlers` с функциями-обработчиками для всех ловушек:

```
const handlers = {
    apply(target, thisValue, args) {
        log("apply", {target, thisValue, args});
        return Reflect.apply(target, thisValue, args);
    },
    construct(target, args, newTarget) {
        log("construct", {target, args, newTarget});
        return Reflect.construct(target, args, newTarget);
    },
    defineProperty(target, propName, descriptor) {
        log("defineProperty", {target, propName, descriptor});
```

```
        return Reflect.defineProperty(target, propName, descriptor);
    },
    deleteProperty(target, propName) {
        log("deleteProperty", {target, propName});
        return Reflect.deleteProperty(target, propName);
    },
    get(target, propName, receiver) {
        log("get", {target, propName, receiver});
        return Reflect.get(target, propName, receiver);
    },
    getOwnPropertyDescriptor(target, propName) {
        log("getOwnPropertyDescriptor", {target, propName});
        return Reflect.getOwnPropertyDescriptor(target, propName);
    },
    getPrototypeOf(target) {
        log("getPrototypeOf", {target});
        return Reflect.getPrototypeOf(target);
    },
    has(target, propName) {
        log("has", {target, propName});
        return Reflect.has(target, propName);
    },
    isExtensible(target) {
        log("isExtensible", {target});
        return Reflect.isExtensible(target);
    },
    ownKeys(target) {
        log("ownKeys", {target});
        return Reflect.ownKeys(target);
    },
    preventExtensions(target) {
        log("preventExtensions", {target});
        return Reflect.preventExtensions(target);
    },
    set(target, propName, value, receiver) {
        log("set", {target, propName, value, receiver});
        return Reflect.set(target, propName, value, receiver);
    },
    setPrototypeOf(target, newProto) {
        log("setPrototypeOf", {target, newProto});
        return Reflect.setPrototypeOf(target, newProto);
    }
};
```

Определение в примере сделано таким подробным как для наглядности, так и для того, чтобы функция `log` могла отображать имена значений параметров, которые получают обработчики ловушек. (Если бы мы не хотели этого делать, мы могли бы подключить все в простом цикле, поскольку у объекта `Reflect` есть методы, имена которых совпадают с именами прокси-ловушек и которые ожидают именно тех аргументов, которые предоставляют ловушки.) Далее в коде определяется простой класс счетчика:

```
class Counter {
    constructor(name) {
        this.value = 0;
        this.name = name;
    }
```

```
    increment() {
        return ++this.value;
    }
}
```

Затем он создает экземпляр класса, оборачивает вокруг него прокси и сохраняет эти два объекта в карте `names`:

```
const c = new Counter("counter");
const cProxy = new Proxy(c, handlers);
names.set(c, "c");
names.set(cProxy, "cProxy");
```

Теперь код начинает выполнять операции с прокси, вызывая ловушки. Сначала он получает начальное значение `value` счетчика через прокси:

```
console.log("---- Getting cProxy.value (before increment):");
console.log(`cProxy.value (before) = ${cProxy.value}`);
```

Вы уже видели прокси ловушку `get`, поэтому результат вас не удивит:

```
---- Getting cProxy.value (before increment):
get: target = c, propName = "value", receiver = cProxy
cProxy.value (before) = 0
```

Вторая строка показывает, что ловушка `get` была вызвана с объектом `c` в качестве параметра `target`, `"value"` для `propName` и сам прокси `cProxy` для *объекта получателя* `receiver`. (Объект `receiver` будет описан в следующем разделе.) Третья строка показывает нам, что возвращенное значение было `0`, что имеет смысл, поскольку счетчик начинается с `0`.

Далее код вызывает функцию `increment` на прокси-сервере:

```
console.log("---- Calling cProxy.increment():");
cProxy.increment();
```

Результат этого кода может показаться более удивительным:

```
---- Calling cProxy.increment():
get: target = c, propName = "increment", receiver = cProxy
get: target = c, propName = "value", receiver = cProxy
set: target = c, propName = "value", value = 1, receiver = cProxy
getOwnPropertyDescriptor: target = c, propName = "value"
defineProperty: target = c, propName = "value", descriptor = {"value":1}
```

Сработали четыре различных типа ловушек, одна из них дважды!

Первая часть достаточно проста: `cProxy.increment()` указывает [[Get]] искать свойство `increment` в прокси, поэтому мы видим, что ловушка `get` при этом сработала. Затем вспомните о реализации `increment`:

```
increment() {
    return ++this.value;
}
```

Значение `this` внутри вызова функции `increment` в выражении `cProxy.increment()` равно `cProxy`, следовательно, `this` в `++this.value` — это прокси: сначала движок выполняет [[Get]] для `value` (получая значение `0`), затем [[Set]] для `value` (устанавливая значение `1`).

Вы, вероятно, думаете: «Пока все хорошо, но что метод `getOwnPropertyDescriptor` здесь делает? И почему использован `defineProperty`?!»

Реализация операции по умолчанию [[Set]] для обычных объектов[87] разработана таким образом специально для обеспечения прокси. Сначала она проверяет, является ли свойство свойством данных или акцессором (эта проверка выполняется напрямую, без прохождения через прокси). Если устанавливаемое свойство — акцессор, вызывается функция сеттера. Но если это свойство данных, его значение устанавливается путем получения дескриптора свойства через прокси при помощи [[GetOwnProperty]], установки его значения `value`, а затем применяется [[DefineOwnProperty]] с измененным дескриптором для его обновления. (Если свойство еще не существует, [[Set]] определяет дескриптор свойства данных для свойства и использует [[DefineOwnProperty]] для его создания.)

Может показаться странным использовать [[DefineOwnProperty]] для установки значения, но это гарантирует, что *все* изменяющие свойство операции (его возможность записи, расширяемость, возможность конфигурации или даже его значение), проходят через центральную операцию — [[DefineOwnProperty]]. (Вы увидите остальные свойства, когда мы продолжим рассмотрение примера.)

Код продолжает выполнение с:

```
console.log("---- Getting cProxy.value (after increment):");
console.log(`cProxy.value (after) = ${cProxy.value}`);
```

что отображает обновленное значение:

```
---- Getting cProxy.value (after increment):
get: target = c, propName = "value", receiver = cProxy
cProxy.value (after) = 1
```

На данный момент этот пример привел к срабатыванию четырех из тринадцати ловушек: `get`, `set`, `getOwnPropertyDescriptor` и `defineProperty`.

Далее код использует `Object.keys` для получения собственных ключей свойств `cProxy` с перечисляемыми строковыми именами:

```
console.log("---- Getting cProxy's own enumerable string-named keys:");
console.log(Object.keys(cProxy));
```

Это запускает ловушку `ownKeys`, а затем, поскольку `Object.keys` должен проверять, являются ли ключи перечислимыми, прежде чем включать их в массив, он получает дескрипторы свойств для обоих возвращаемых им ключей:

```
---- Getting cProxy's own enumerable string-named keys:
ownKeys: target = c
```

[87] https://tc39.es/ecma262/#sec-ordinaryset

```
getOwnPropertyDescriptor: target = c, propName = "value"
getOwnPropertyDescriptor: target = c, propName = "name"
["value", "name"]
```

Далее код удаляет свойство `value`:

```
console.log("---- Deleting cProxy.value:");
delete cProxy.value;
```

Операция по удалению свойства проста, поэтому срабатывает только ловушка `deleteProperty`:

```
---- Deleting cProxy.value:
deleteProperty: target = c, propName = "value"
```

Затем он проверяет, существует ли еще свойство `value`, используя оператор `in`:

```
console.log("---- Checking whether cProxy has a 'value' property:");
console.log(`"value" in cProxy ? ${"value" in cProxy}`);
```

Это запускает ловушку `has`. Поскольку прокси не предотвратил удаление мгновение назад, у объекта больше нет свойства `value`:

```
---- Checking whether cProxy has a 'value' property:
has: target = c, propName = "value"
"value" in cProxy ? false
```

Далее код получает прототип объекта:

```
console.log("---- Getting the prototype of cProxy:");
const sameProto = Object.getPrototypeOf(cProxy) === Counter.prototype;
console.log(`Object.getPrototypeOf(cProxy) === Counter.prototype ?
${sameProto}`);
```

Это запускает ловушку `getPrototypeOf`:

```
---- Getting the prototype of cProxy:
getPrototypeOf: target = c
Object.getPrototypeOf(cProxy) === Counter.prototype ? true
```

Помните, что прокси передают операции своим целевым элементам (если только не вмешивается обработчик ловушек), поэтому возвращается прототип `c`, а не прокси. (На самом деле, поскольку прокси ведут себя таким образом, им вообще не присваивается прототип, а у конструктора `Proxy` нет свойства `prototype`.) Чтобы вызвать следующую ловушку, код устанавливает прототип объекта:

```
console.log("---- Setting the prototype of cProxy to Object.prototype:");
Object.setPrototypeOf(cProxy, Object.prototype);
```

И действительно, запускается обработчик ловушки `setPrototypeOf`:

```
---- Setting the prototype of cProxy to Object.prototype:
setPrototypeOf: target = c, newProto = {}
```

Затем код проверяет, по-прежнему ли `Counter.prototype` является прототипом `cProxy`:

```
console.log("---- Getting the prototype of cProxy again:");
const sameProto2 = Object.getPrototypeOf(cProxy) === Counter.prototype;
console.log(`Object.getPrototypeOf(cProxy) === Counter.prototype ?
            ${sameProto2}`);
```

Это не так, потому что код только что изменил его на `Object.prototype` минуту назад:

```
---- Getting the prototype of cProxy again:
getPrototypeOf: target = c
Object.getPrototypeOf(cProxy) === Counter.prototype ? false
```

Далее код проверяет расширяемость:

```
console.log("---- Is cProxy extensible?:");
console.log(`Object.isExtensible(cProxy) (before) ?
            ${Object.isExtensible(cProxy)}`);
```

Проверка расширяемости — это действительно простая операция; требуется всего один вызов обработчика ловушки `isExtensible`:

```
---- Is cProxy extensible?:
isExtensible: target = c
Object.isExtensible(cProxy) (before) ? true
```

Далее код предотвращает расширения:

```
console.log("---- Preventing extensions on cProxy:");
Object.preventExtensions(cProxy);
```

запускается ловушка `preventExtensions`:

```
---- Preventing extensions on cProxy:
preventExtensions: target = c
```

Затем код проверяет, объект по-прежнему расширяемый или нет:

```
console.log("---- Is cProxy still extensible ?:");
console.log(`Object.isExtensible(cProxy) (after) ?
            ${Object.isExtensible(cProxy)}`);
```

Объект больше не расширяемый, поскольку ловушка не помешала операции:

```
---- Is cProxy still extensible?:
isExtensible: target = c
Object.isExtensible(cProxy) (after) ? false
```

До этого момента показано 11 из 13 ловушек. Оставшиеся две — это ловушки, имеющие смысл только для прокси с объектами функций. Поэтому пример кода создает функцию и оборачивает вокруг нее прокси (и регистрирует это в `names`, чтобы код знал, что нужно регистрировать их имена, а не содержимое):

```
const func = function() {console.log("func ran");};
const funcProxy = new Proxy(func, handlers);
names.set(func, "func");
names.set(funcProxy, "funcProxy");
```

Затем происходит вызов функции:

```
console.log("---- Calling funcProxy as a function:");
funcProxy();
```

которая вызывает обработчик ловушки `apply`:

```
---- Calling funcProxy as a function:
apply: target = func, thisValue = undefined, args = []
func ran
```

И затем он демонстрирует последнюю ловушку `construct`, вызывая функцию в качестве конструктора:

```
console.log("---- Calling funcProxy as a constructor:");
new funcProxy();
```

в результате чего:

```
---- Calling funcProxy as a constructor:
construct: target = func, args = [], newTarget = funcProxy
get: target = func, propName = "prototype", receiver = funcProxy
func ran
```

Это работает, поскольку все традиционные функции могут использоваться в качестве конструкторов. Но если вы попытаетесь вызвать таким образом не-конструктор, такой как стрелочная функция или метод, ловушка не сработает, потому что прокси вообще не получит внутреннюю функцию [[Construct]]; вместо выполнения попытка просто завершится неудачей. Далее в примере демонстрируется создание стрелочной функции и прокси для нее:

```
const arrowFunc = () => {console.log("arrowFunc ran");};
const arrowFuncProxy = new Proxy(arrowFunc, handlers);
names.set(arrowFunc, "arrowFunc");
names.set(arrowFuncProxy, "arrowFuncProxy");
```

а затем попытка вызвать прокси как функцию и как конструктор:

```
console.log("---- Calling arrowFuncProxy as a function:");
arrowFuncProxy();
console.log("---- Calling arrowFuncProxy as a constructor:");
try {
    new arrowFuncProxy();
```

```
} catch (error) {
    console.error(`${error.name}: ${error.message}`);
}
```

что даст в результате:

```
---- Calling arrowFuncProxy as a function:
apply: target = arrowFunc, thisValue = undefined, args = []
arrowFunc ran
---- Calling arrowFuncProxy as a constructor:
```

```
TypeError: arrowFuncProxy is not a constructor
```

Наконец, помните, что функции — это объекты, поэтому все остальные ловушки применимы и к ним. Пример заканчивается получением свойства `name` стрелочной функции:

```
console.log("---- Getting name of arrowFuncProxy:");
console.log(`arrowFuncProxy.name = ${arrowFuncProxy.name}`);
```

что вызывает ловушку `get`:

```
---- Getting name of arrowFuncProxy:
get: target = arrowFunc, propName = "name", receiver = arrowFuncProxy
arrowFuncProxy.name = arrowFunc
```

Ловушки прокси

Вы видели все ловушки в действии в примере с регистрацией. В этом разделе будет подробно рассмотрена каждая ловушка и все ее ограничения.

Общие возможности

Обработчик прокси-ловушки может делать практически все, что ему заблагорассудится, хотя некоторые из них получили определенные ограничения. В общем случае обработчик ловушек может:

- обрабатывать саму операцию, никогда не затрагивая целевой объект/функцию;
- наладить операцию (иногда только в определенных пределах);
- отклонить операцию, вернув флаг ошибки или выбросив сообщение об ошибке;
- предоставлять любые понравившиеся побочные эффекты (например, операторы регистрации в примере с регистрацией), поскольку обработчик ловушек — это произвольный код.

Там, где обработчики ловушек встречаются с ограничениями, они существуют для обеспечения *необходимого инвариантного поведения*[88], которое ожидается от всех объектов — даже таких экзотических, как прокси. Пределы конкретных ловушек указаны в следующих разделах.

[88] https://tc39.github.io/ecma262/#sec-invariants-of-the-essential-internal-methods

Ловушка `apply`

Ловушка `apply` предназначена для внутренней операции [[Call]] над вызываемыми объектами (например функциями). В прокси будет находиться только операция [[Call]], если целевой объект, который он проксирует, содержит ее (в противном случае попытка вызвать прокси приведет к ошибке типа, поскольку он не доступен для вызова). Поэтому ловушка `apply` применяется только к прокси, цель которых может быть вызвана.

Обработчик ловушки `apply` принимает три аргумента:

- `target`: цель проксирования;
- `thisValue`: значение, используемое в качестве `this` при вызове;
- `args`: массив аргументов для вызова.

Его возвращаемое значение используется в качестве возвращаемого значения операции вызова (независимо от того, действительно ли она вызывала целевую функцию).

Обработчик ловушки может выполнить или не выполнить базовый вызов, и может вернуть любое желаемое значение (или выдать ошибку). В отличие от некоторых других обработчиков ловушек, нет никаких ограничений на то, что может сделать обработчик `apply`.

Ловушка `construct`

Ловушка `construct` предназначена для внутренней операции [[Construct]] конструкторов. В прокси будет находиться только операция [[Construct]], если целевой объект, который он проксирует, содержит ее (в противном случае попытка вызвать прокси приведет к ошибке типа, поскольку он не доступен для вызова), поэтому ловушка `construct` применяется только к прокси, цель которых представлена конструктором (либо конструктором `class`, либо традиционной функцией, которые можно использовать в качестве конструктора).

Ловушка `construct` принимает три аргумента:

- `target`: цель проксирования;
- `args`: массив аргументов для вызова;
- `newTarget`: значение для выражения `new.target` (о котором говорилось в главе 4).

Его возвращаемое значение используется как результат операции конструктора (независимо от того, была она выполнена или нет).

Он может делать *почти* все, что ему заблагорассудится. Единственное ограничение для обработчика ловушки `construct` заключается в том, что, если он возвращает что-то (а не выдает ошибку), это что-то должно быть объектом (а не `null` или примитивом). Если обработчик возвращает значение `null` или примитив, прокси выбросит ошибку.

Ловушка `defineProperty`

Ловушка `defineProperty` предназначена для внутренней операции [[DefineOwnProperty]] объекта. Как вы видели в примере с регистрацией, [[DefineOwnProperty]] используется не только тогда, когда что-то вызывает метод `Object.defineProperty` (или `Reflect.defineProperty`) для объекта, вызов также возможен, когда задается значение свойства данных или когда свойство создается с помощью присваивания.

Ловушка `defineProperty` принимает три аргумента:

- `target`: цель проксирования;
- `propName`: имя определяемого/переопределяемого свойства;
- `descriptor`: дескриптор для применения.

Ожидается, что она вернет значение `true` при успешном выполнении (определение свойства было успешным или уже соответствовало существующему свойству) или значение `false` при ошибке. (Истинноподобные и ложноподобные значения будут приводиться по мере необходимости.)

Обработчик ловушек может отклонить изменение (либо вернуть значение `false`, либо выбросить ошибку), настроить дескриптор свойства перед его применением и т. д. — или любую другую задачу, выполняемую всеми ловушками.

Если обработчик ловушки возвращает значение `true` (успех), на него накладываются некоторые ограничения. Он не может лгать, говоря, что операция прошла успешно, когда очевидно, что операция не была выполнена. В частности, выдается ошибка, если сообщается об успешном завершении, и когда обработчик возвращает результат, свойство:

- …не существует, и целевой элемент не расширяемый;
- …не существует, и дескриптор помечал свойство как не настраиваемое;
- …существует и настраивается, но дескриптор помечал свойство как не настраиваемое;
- …существует, и применение к нему дескриптора приведет к ошибке (например, если свойство существует и не настраивается, но дескриптор менял свою конфигурацию).

Эти правила существуют из-за ограничения *необходимого инвариантного поведения* объектов, упомянутого ранее. Тем не менее они по-прежнему оставляют достаточно гибкости. Например, ловушка может сказать, что вызов для установки значения свойства сработал, когда значение свойства не является (в данный момент) значением, которое предположительно было установлено — даже если это свойство данных, доступное для записи.

Нет никаких ограничений для обработчика ловушки, когда он возвращает значение `false` (сбой) — даже если свойство существует точно так, как его описывает дескриптор (что обычно означает, что возвращается значение `true`).

Как вы узнали ранее, ловушка `defineProperty` срабатывает при установке значения свойства данных (в отличие от свойства-акцессора), но это не делает ловушку `set` избыточной. Ловушка `set` срабатывает и при установке значения свойств-акцессоров, тогда как `defineProperty` — нет.

В Листинге 14-3 показана простая ловушка `defineProperty`, запрещающая делать любое существующее свойство целевого объекта недоступным для записи.

Листинг 14-3: Пример ловушки `defineProperty` — defineProperty-trap-example.js

```
const obj = {};
const p = new Proxy(obj, {
    defineProperty(target, propName, descriptor) {
        if ("writable" in descriptor && !descriptor.writable) {
```

```
            const currentDescriptor =
                Reflect.getOwnPropertyDescriptor(target, propName);
            if (currentDescriptor && currentDescriptor.writable) {
                return false;
            }
        }
        return Reflect.defineProperty(target, propName, descriptor);
    }
});
p.a = 1;
console.log(`p.a = ${p.a}`);
console.log("Trying to make p.a non-writable...");
console.log(
    `Result of defineProperty: ${Reflect.defineProperty(p, "a",
                                {writable: false})}`
);
console.log("Setting pa.a to 2...");
p.a = 2;
console.log(`p.a = ${p.a}`);
```

Обратите внимание, что в Листинге 14–3 при использовании прокси код использует выражение `Reflect.defineProperty`, а не `Object.defineProperty`, чтобы попытаться пометить свойство, недоступное для записи. Можете ли вы догадаться, почему код написан именно так? Это связано с одним из основных различий между изменяющими элементы версиями функций `Object` и `Reflect`…

Это потому, что я хотел показать вам результат `false`, и тот факт, что свойство `a` все еще поддерживало возможность записи после вызова `defineProperty`. Если бы я использовал `Object.defineProperty`, была бы выброшена ошибка, а не возврат значения `false`, следовательно, мне пришлось бы писать блок `try/catch`. То, что вы используете в подобной ситуации, зависит от того, хотите ли вы получать возвращаемое значение или ошибку; существуют варианты использования для обоих вызовов.

Одно заключительное примечание: если ловушка была спровоцирована вызовом `defineProperty` (либо версией `defineProperty`, либо `Reflect`), ловушка не получит прямую ссылку на дескриптор объекта, переданного в эту функцию; вместо этого он получает объект, созданный специально для ловушки, у которого есть только действительные имена свойств. Таким образом, даже если бы дополнительные, не относящиеся к дескриптору свойства были включены в переданный функции результат, ловушка не получила бы их.

Ловушка deleteProperty

Ловушка `deleteProperty` предназначена для внутренней операции [[Delete]] объекта, удаляющей свойство из объекта.

Обработчик ловушки `deleteProperty` принимает два аргумента:

- `target`: цель проксирования;
- `propName`: имя удаляемого свойства.

Ожидается возвращение значения `true` в случае успеха и `false` в случае ошибки, как и в случае с оператором `delete` в свободном режиме (в строгом режиме неудачное выполнение `delete` приводит к выбросу ошибки). Истинноподобные и ложноподобные значения будут приводиться по мере необходимости.

Обработчик может отказаться удалять свойство (возвращая значение `false` или выбрасывая ошибку) или выполнять то, что могут выполнять другие ловушки.

Он не может возвращать значение `true`, если свойство существует в целевом объекте и не настраивается, так как это нарушило бы один из основных инвариантов. Это приводит к возникновению ошибки.

В Листинге 14-4 показана ловушка `deleteProperty`, которая отказывается удалять свойство `value`.

Листинг 14-4: Пример ловушки deleteProperty — deleteProperty-trap-example.js

```
const obj = {value: 42};
const p = new Proxy(obj, {
    deleteProperty(target, propName, descriptor) {
        if (propName === "value") {
            return false;
        }
        return Reflect.deleteProperty(target, propName, descriptor);
    }
});
console.log(`p.value = ${p.value}`);
console.log("deleting 'value' from p in loose mode:");
console.log(delete p.value); // ложь
(() => {
    "use strict";
    console.log("deleting 'value' from p in strict mode:");
    try {
        delete p.value;
    } catch (error) {
        // TypeError: 'deleteProperty' для прокси: ловушка вернула
        // ложное значение для свойства 'value'
        console.error(error);
    }
})();
```

Ловушка `get`

Как вы помните, ловушка `get` предназначена для внутренней операции [[Get]] объекта: получение значения свойства.

Обработчик ловушки `get` принимает три аргумента:

- `target`: цель проксирования;
- `propName`: имя свойства;
- `receiver`: объект, получивший вызов [[Get]].

Возвращаемое значение используется как результат операции [[Get]] (значение, которое код, обращающийся к свойству, видит по его значению).

У свойства `receiver` есть значение, когда оно является свойством-акцессором, а не свойством данных: его значение в таком случае было бы `this` во время вызова акцессора, если бы не было обработчика ловушек. Часто `receiver` представлен в виде прокси, но он может быть объектом, использующим прокси в качестве прототипа. В конце концов, прокси — это объекты, и ничто не мешает использовать прокси в качестве прототипа. В зависимости от варианта использования при передаче вызова

`Reflect.get` можно отказаться от передачи объекту receiver, так что `this` внутри акцессора становится целевым элементом; или заменить на другую цель и т. д.

Ловушка `get` может делать *почти* все, что ей заблагорассудится, изменять значение и т. д. — за исключением того, что, как и другие ловушки, она не может нарушать определенные необходимые инварианты, и это означает, что выдается ошибка, если она:

- …возвращает значение, не совпадающее со значением базового свойства целевого элемента, если это свойство — неконфигурируемое, доступное только для чтения;
- …возвращает значение, отличное от `undefined`, если свойство цели — неконфигурируемый акцессор, предназначенный только для записи (есть только сеттер, а геттер отсутствует).

Вы уже видели несколько примеров `get`; нет необходимости писать еще один. Но `get` снова появляется позже в разделе «Пример: Скрытие свойств».

Ловушка `getOwnPropertyDescriptor`

Ловушка `getOwnPropertyDescriptor` предназначена для внутренней операции [[GetOwnProperty]] объекта, получающей объект-дескриптор для свойства из объекта. Как вы узнали из примера регистрации, [[GetOwnProperty]] вызывается в ряде мест во время обработки других операций с внутренними объектами, а не только тогда, когда код использует `Object.getOwnPropertyDescriptor` или `Reflect.getOwnPropertyDescriptor`.

Обработчик ловушки `getOwnPropertyDescriptor` принимает два аргумента:

- `target`: цель проксирования;
- `propName`: имя свойства.

Ожидается, что обработчик вернет объекту дескриптора свойства или значение `undefined`, если свойство не существует; возврат чего-либо еще (включая значение `null`) приведет к ошибке. Обычно вы получаете объект дескриптора из `Reflect.getOwnPropertyDescriptor`, но также можно вручную создать объект дескриптора или изменить то, что вы получаете обратно от этого вызова (в определенных пределах). Перед вами список свойств объекта дескриптора свойств, использующихся в различных комбинациях в зависимости от свойства:

- `writable`: `true`, если свойство доступно для записи, `false`, если нет (только свойства данных, а не свойства-акцессора). Значение по умолчанию `false` при отсутствии значения;
- `enumerable`: `true`, если свойство перечисляемое, `false`, если нет. Значение по умолчанию `false` при отсутствии значения;
- `configurable`: `true`, если свойство изменяемое, `false`, если нет. Значение по умолчанию `false` при отсутствии значения;
- `value`: значение свойства, если это свойство данных; в противном случае значение отсутствует;
- `get`: функция-геттер для свойства-акцессора (не может быть объединена с `value` или `writable`);
- `set`: функция-сеттер для свойства-акцессора (не может быть объединена с `value` или `writable`).

Объект дескриптора, возвращаемый ловушкой, не возвращается непосредственно в код, запрашивающий дескриптор; вместо этого создается новый объект дескриптора, берущий только допустимые свойства из предоставленного. Любые другие свойства тихо игнорируются.

У этой ловушки есть некоторые ограничения на то, что она может сделать, чтобы сохранить необходимые инварианты. Сообщение об ошибке выдается, если:

- …обработчик возвращает значение `undefined`, и у целевого объекта есть свойство, а целевой объект не расширяемый;
- …обработчик возвращает значение `undefined`, и у целевого объекта есть свойство, и оно неизменяемое;
- …обработчик возвращает дескриптор для неконфигурируемого свойства, но это свойство либо не существует, либо настраиваемое;
- …обработчик возвращает дескриптор для неконфигурируемого свойства, но это свойство либо не существует, а целевой объект не расширяемый.

Основной вариант использования ловушки `getOwnPropertyDescriptor`, вероятно, скрывает свойство, полученное целевым объектом из кода при помощи прокси. Однако это нетривиально и включает в себя обработчики для нескольких ловушек. См. позже в разделе «Пример: Скрытие свойств».

Ловушка `getPrototypeOf`

Ловушка `getPrototypeOf` предназначена для внутренней операции [[GetPrototypeOf]] объекта. Она срабатывает, когда используется функция `getPrototypeOf` объекта `Object` или `Reflect` (напрямую или косвенно через геттер `Object.prototype.__proto__` в веб-браузерах), или когда какой-либо внутренней операции необходимо получить прототип прокси-сервера. Ловушка не вызывается, когда [[Get]] используется с прокси, даже если выполняется цепочка прототипов, поскольку у прокси нет свойства, потому что вызов [[Get]] перенаправляется на целевой объект (в обычном случае). Поэтому операция [[GetPrototypeOf]] используется, когда следование цепочке прототипов в этой точке вызывается на целевом объекте, а не на прокси-сервере.

Она получает только один аргумент:

- `target`: цель проксирования.

Ожидается, что ловушка вернет объект или значение `null`. Она может возвращать любой желаемый объект, если только цель не является нерасширяемой; в этом случае вы должны вернуть прототип целевого элемента.

В Листинге 14-5 показан объект прокси, скрывающий прототип целевого объекта, даже несмотря на то, что прототип используется для разрешения свойств.

Листинг 14-5: Пример ловушки getPrototypeOf — getPrototypeOf-trap-example.js

```
const proto = {
    testing: "one two three"
};
const obj = Object.create(proto);
const p = new Proxy(obj, {
    getPrototypeOf(target) {
```

```
        return null;
    }
});
console.log(p.testing);                 // one two three
console.log(Object.getPrototypeOf(p)); // null
```

Ловушка `has`

Ловушка `has` предназначена для внутренней операции [[hasProperty]] объекта, которая определяет, обладает ли объект заданным свойством (сам по себе или через его прототип).

Она получает два аргумента:

- `target`: цель проксирования;
- `propName`: имя свойства.

Ожидается, что она вернет значение `true`, если свойство есть (напрямую или через свой прототип), или значение `false`, если нет (правдоподобные и лжеподобные значения приводятся по типу).

Вероятно, вы можете догадаться, основываясь на ограничениях предыдущих ловушек, каковы ограничения для ловушки `has`:

- Возврат значения `false` для существующего и не настраивающегося свойства приводит к ошибке.
- Возврат значения `false` для существующего в целевом нерасширяющемся объекте свойства приводит к ошибке.

Обработчику *разрешено* возвращать значение `true` для *несуществующего* свойства, даже для нерасширяемого целевого объекта.

Очевидные вещи, связанные с ловушками `has`, — это сокрытие свойств, которыми обладает объект, или утверждение, что объект обладает свойствами, которых у него нет. Для изучения примера см. раздел «Пример: Скрытие свойств» далее в этой главе.

Ловушка `isExtensible`

Ловушка `isExtensible` предназначена для внутренней операции [[IsExtensible]] объекта — проверки того, является ли объект расширяемым (то есть над ним не выполнялась операция [[PreventExtensions]]).

Обработчик ловушки `isExtensible` принимает только один аргумент:

- `target`: цель проксирования.

Ожидается, что обработчик вернет значение `true`, если объект расширяемый, или значение `false`, если нет (правдоподобные и лжеподобные значения приводятся по типу). Это одна из самых ограниченных ловушек: она должна возвращать то же значение, которое вернул бы сам целевой объект. Следовательно, эта ловушка полезна только для побочных эффектов, таких как регистрация данных из предыдущего примера.

Ловушка `ownKeys`

Ловушка `ownKeys` предназначена для внутренней операции [[OwnPropertyKeys]] объекта, которая создает массив собственных ключей свойств объекта, включая неисчисляемые и те, которые именованы символами, а не строками.

Обработчик ловушки `ownKeys` принимает только один аргумент:

- `target`: цель проксирования.

Ожидается, что он вернет массив или массивоподобный объект (он не может просто возвращать итерируемый элемент, объект должен массивоподобным).

Ошибка выбрасывается, если обработчик ловушки возвращает массив, который:

- …содержит дубликаты;
- …содержит записи, не являющиеся строковыми или символьными;
- …содержит какие-либо отсутствующие или дополнительные записи, если целевой объект не расширяемый;
- …не содержит запись для неконфигурируемого свойства, существующего в целевом объекте.

Это означает, что обработчик может скрывать свойства (при условии, что они настраиваемые и целевой объект расширяемый) или включать дополнительные свойства (при условии, что целевой объект расширяемый).

Одним из распространенных вариантов использования `ownKeys` — это скрытие свойств, как вы узнаете из раздела «Пример: Скрытие свойств».

Ловушка `preventExtensions`

Ловушка `preventExtensions` предназначена для внутренней операции [[PreventExtensions]] объекта, которая помечает объект как нерасширяемый.

Ее обработчик получает только один аргумент:

- `target`: цель проксирования.

Ожидается, что обработчик вернет значение `true` в случае успешного выполнения или значение `false`, если появляется ошибка (правдоподобные и лжеподобные значения приводятся по типу). Это ошибка, если он возвращает значение `true`, а целевой объект расширяемый. Однако разрешено возвращать значение `false`, если целевой объект нерасширяемый.

Обработчик ловушки может предотвратить превращение целевого объекта в нерасширяемый, возвращая значение `false`, как в Листинге 14-6.

Листинг 14-6: Пример ловушки preventExtensions — preventExtensions-trap-example.js

```
const obj = {};
const p = new Proxy(obj, {
    preventExtensions(target) {
        return false;
}
});
console.log(Reflect.isExtensible(p));      // истина
console.log(Reflect.preventExtensions(p)); // ложь
console.log(Reflect.isExtensible(p));      // истина
```

Ловушка `set`

Ловушка `get` предназначена для внутренней операции [[Set]] объекта: установление значения свойства. Она срабатывает, когда задается значение свойства данных или свойства-акцессора. Как показано ранее, если обработчик ловушки разрешает операцию, а устанавливаемое свойство — свойство данных, ловушка `defineProperty` также будет запущена, чтобы установить значение `value` свойства данных.

Обработчик ловушки `set` получает четыре аргумента:

- `target`: цель проксирования;
- `propName`: имя свойства;
- `value`: устанавливаемое значение;
- `receiver`: получающий операцию объект.

Ожидается, что обработчик вернет значение `true` в случае успешного выполнения или значение `false`, если появляется ошибка (правдоподобные и лжеподобные значения приводятся по типу). Ограничения необходимых инвариантных операций означают, что это ошибка, если обработчик возвращает значение `true` и:

- …свойство существует в целевом объекте, является неконфигурируемым, недоступным для записи свойством данных, и его значение не соответствует заданному значению;
- …свойство существует в целевом объекте, является неконфигурируемым свойством-акцессором и не содержит функцию сеттер.

Обратите внимание, что `set` может предотвратить установку значений даже для неконфигурируемых свойств и нерасширяемых целевых объектов, возвращая значение `false`. Пример будет показан в разделе «Пример: Скрытие свойств» далее в этой главе.

Ловушка `setPrototypeOf`

Ловушка `setPrototypeOf` предназначена для внутренней операции [[SetPrototypeOf]] объекта, она (я уверен, это удивит вас) задает прототип объекта.

Ее обработчик получает два аргумента:

- `target`: цель проксирования;
- `newProto`: устанавливаемый прототип.

Ожидается, что обработчик вернет значение `true` в случае успешного выполнения или значение `false` при ошибке (правдоподобные и лжеподобные значения приводятся по типу). Чтобы обеспечить необходимые инварианты, он не может возвращать значение `true`, если целевой объект нерасширяемый, если только устанавливаемый прототип уже не является прототипом цели. Однако он может отказаться устанавливать новый прототип, как показано в Листинге 14-7.

Листинг 14-7: Пример ловушки setPrototypeOf — setPrototypeOf-trap-example.js

```
const obj = {foo: 42};
const p = new Proxy(obj, {
    setPrototypeOf(target, newProto) {
```

```
        // Возвращает значение false, если только `newProto`
        // уже не является прототипом `target`
        return Reflect.getPrototypeOf(target) === newProto;
    }
});
console.log(Reflect.getPrototypeOf(p) === Object.prototype); // истина
console.log(Reflect.setPrototypeOf(p, Object.prototype));    // истина
console.log(Reflect.setPrototypeOf(p, Array.prototype));     // ложь
```

Пример: Скрытие свойств

В этом разделе рассматривается пример скрытия свойств. Скрыть свойства в неизменяемом объекте довольно просто. А вот когда операции над объектом могут изменить скрытое свойство — гораздо сложнее.

Прежде чем мы начнем, стоит отметить, что скрытие свойств редко бывает необходимым. У большинства языков с декларативно приватными свойствами есть черный ход (через рефлексию), который авторы могут использовать, если им действительно потребуется получить доступ к частной информации. (Хотя, что интересно, собственные приватные поля JavaScript, о которых вы узнаете в главе 18, этого не делают.) Поэтому для большинства случаев использования обычно достаточно простого соглашения об именовании или небольшой документации с надписью «не обращаться к этому свойству». Для ситуаций, когда этого недостаточно, есть (как минимум) четыре варианта:

- Создавайте свои методы как замыкания в конструкторе, а не помещайте их в прототип, и сохраняйте «свойства» в переменных/параметрах в области видимости вызова конструктора.
- Используйте слабую карту, как показано в главе 12, чтобы свойства, которые вы хотите скрыть, вообще не были в объекте.
- Используйте приватные поля (о которых вы узнаете в главе 18), когда они попадают в реализации (на момент написания этой книги они находятся на этапе 3) или с помощью транспиляции.
- Используйте прокси, чтобы скрыть свойства.

Вариант использования прокси полезен, если по какой-то причине нельзя использовать один из трех других, возможно, потому, что нельзя изменить реализацию проксируемого объекта.

Давайте посмотрим на это в действии. Предположим, у нас есть простой класс счетчика, подобный этому:

```
class Counter {
    constructor(name) {
        this._value = 0;
        this.name = name;
    }
    increment() {
        return ++this._value;
    }
    get value() {
        return this._value;
    }
}
```

Код, использующий этот класс, может непосредственно наблюдать и изменять `_value`:

```
const c = new Counter("c");
console.log("c.value before increment:");
console.log(c.value);                       // 0
console.log("c._value before increment:");
console.log(c._value);                      // 0
c.increment();
console.log("c.value after increment:");
console.log(c.value);                       // 1
console.log("c._value after increment:");
console.log(c._value);                      // 1
console.log("'_value' in c:");
console.log('_value' in c);                 // истина
console.log("Object.keys(c):");
console.log(Object.keys(c));                // ["name", "_value"]
c._value = 42;
console.log("c.value after changing _value:");
console.log(c.value);                       // 42
```

(Вы можете запустить этот код из файла **not-hiding-properties.js** в разделе загрузки.)

Предположим, требуется скрыть свойство `value`, чтобы его нельзя было напрямую наблюдать или изменять с помощью кода, использующего экземпляр класса счетчика, и вы не можете или не хотите использовать для этого один из других механизмов. Для этого вам нужно подключиться к нескольким операциям:

- `get`, чтобы возвращать значение `undefined` для свойства вместо его значения;
- `getOwnPropertyDescriptor`, чтобы возвращать значение `undefined` вместо дескриптора свойства для него;
- `has`, чтобы не сообщать о существовании свойства;
- `ownKeys` — по той же причине;
- `defineProperty`, чтобы не позволить задавать свойство (непосредственно, либо при помощи `Object.defineProperty` или `Reflect.defineProperty`) и, следовательно, запретить изменять перечисляемость свойства и т. д.;
- `deleteProperty`, чтобы свойство нельзя было удалить.

Нет необходимости использовать ловушку `set`, поскольку, как вы узнали ранее, все изменяющие свойство данных операции в итоге проходят через ловушку `defineProperty`. (Если бы вы скрывали свойство-акцессор, то вам нужно было бы использовать ловушку `set`: только изменения свойств данных проходят через `defineProperty`.) Давайте начнем с этого и посмотрим, что у нас получится:

```
function getCounter(name) {
    const p = new Proxy(new Counter(name), {
        get(target, name, receiver) {
            if (name === "_value") {
                return undefined;
            }
            return Reflect.get(target, name, receiver);
        },
        getOwnPropertyDescriptor(target, propName) {
```

```
            if (name === "_value") {
                return undefined;
            }
            return Reflect.getOwnPropertyDescriptor(target, propName);
        },
        defineProperty(target, name, descriptor) {
            if (name === "_value") {
                return false;
            }
            return Reflect.defineProperty(target, name, descriptor);
        },
        has(target, name) {
            if (name === "_value") {
                return false;
            }
            return Reflect.has(target, name);
        },
        ownKeys(target) {
            return Reflect.ownKeys(target).filter(key => key !== "_value");
        }
    });
    return p;
}
```

Похоже, этот код должен сработать, не так ли? Давайте проделаем с ним ту же серию операций, что и раньше, но с использованием выражения `const p = getCounter("p")` вместо `const c = new Counter("c")`:

```
const p = getCounter("p");
console.log("p.value before increment:");
console.log(p.value);                       // 0
console.log("p._value before increment:");
console.log(p._value);                      // undefined
p.increment();                              // Выбрасывается ошибка!
```

Все начинается хорошо, но затем вызов `increment` терпит неудачу (попробуйте это, запустив файл **hiding-properties-1.js** из загрузок) и указывает на строку `++this._value;`:

```
TypeError: 'defineProperty' on proxy: trap returned falsish for property
'_value'
```

Почему прокси-сервер был вызван для этой строки *внутри* кода `Counter`? Давайте посмотрим, что происходит, когда выполняется `p.increment()`:

- [[Get]] вызывается для объекта, чтобы получить свойство `increment`, поскольку его имя не «`_value`», наш обработчик ловушки `get` возвращает функцию при помощи `Reflect.get`.
- [[Call]] вызывается для `increment` с `this` в значении… в=заметили проблему, правда? `this` получает значение прокси, а не целевой объект. Ведь был вызов `p.increment()`, поэтому во время вызова операции `increment` переменная `p` присваивается `this`.

Так как же все-таки заставить `increment` работать с целевым объектом, а не с прокси? Есть два варианта: обернуть функцию вокруг `increment` или… обернуть в *прокси* (данный раздел, в конце концов, о прокси: свойства `name`, `length` и т. п. отражаются оболочкой). Но создавать новый прокси (или оболочку) чуть ли не для каждого вызова функции довольно неэффективно. В главе 12 описан удобный метод хранения карты объектов, не заставляя ключ оставаться в памяти — слабые карты. Код может использовать слабую карту с исходной функцией в качестве ключа и ее прокси в качестве значения. Можно изменить обработчик `get` на прокси функции по мере необходимости, запоминая эти прокси для повторного использования:

```
function getCounter(name) {
    const functionProxies = new WeakMap();
    const p = new Proxy(new Counter(name), {
        get(target, name) {
            if (name === "_value") {
                return undefined;
            }
            let value = Reflect.get(target, name);
            if (typeof value === "function") {
                let funcProxy = functionProxies.get(value);
                if (!funcProxy) {
                    funcProxy = new Proxy(value, {
                        apply(funcTarget, thisValue, args) {
                            const t = thisValue === p ? target: thisValue;
                            return Reflect.apply(funcTarget, t, args);
                        }
                    });
                    functionProxies.set(value, funcProxy);
                    value = funcProxy;
                }
            }
            return value;
        },
        // ...никаких изменений в других обработчиках ловушек...
    });
    return p;
}
```

Обратите внимание, как прокси использует исходный целевой объект (экземпляр `Counter`, `target`), а не прокси при вызове метода `increment`. Это делает свое дело: попробуйте запустить `hiding-properties-2.js` из загрузок. Как видите, `increment` все еще работает, но доступ к `_value` извне ограничен.

По-прежнему можно вызвать сбой этого прокси-счетчика, получив `increment` из прототипа счетчика, а не из прокси (файл **hiding-properties-fail.js** в загрузках):

```
const {increment} = Object.getPrototypeOf(p);
increment.call(p);
// => Выбрасывает ошибку TypeError: 'defineProperty' у прокси: ловушка
// возвращает ложь...
```

или путем обертывания прокси вокруг прокси, или использовать прокси-счетчик в качестве прототипа (поскольку в обоих случаях проверка `thisValue === p` в ловушке `apply`

больше не будет истинной), но базовый вариант использования обрабатывается правильно. Можно предусмотреть случай использования прокси в качестве прототипа (путем зацикливания прототипов `thisValue`, чтобы увидеть, является ли прокси одним из них), но вы вряд ли сможете обойти проблему проксированного прокси.

Вывод здесь, помимо «очень тщательно проверяйте свои прокси», заключается в том, что прокси — это мощные, но сложные в реальном использовании объекты.

Отключаемые прокси

В начале этой главы описывалось, что прокси-серверы полезны для обеспечения границы между двумя битами кода, такими как API и его потребители. *Отключаемый прокси* особенно полезен для этого, потому что вы можете отключить предоставленный вами объект (прокси), когда придет время. Отключение прокси делает две важные вещи:

- Все операции с отключенным прокси завершаются ошибкой. Теория привратника во всей красе!
- Освобождается ссылка прокси на свой целевой объект. Это значит, что, хотя потребляющий код все еще может ссылаться на прокси, процедура сборки мусора может удалить целевой объект. Это сводит к минимуму воздействие на память отключенных прокси, которые надежно удерживаются потребительским кодом. Например, быстрый тест показывает, что отозванный прокси составляет всего 32 байта в движке Chrome V8 (конечно, это может измениться), а целевой объект может быть довольно большим.

Чтобы создать отключаемый прокси, вызывается метод `Proxy.revocable` (не стоит применять `new Proxy`). Метод `Proxy.revocable` возвращает объект со свойством `proxy` (прокси) и методом `revoke` (для отключения прокси). См. Листинг 14-8.

Листинг 14-8: Пример отключаемого прокси — revocable-proxy-example.js

```
const obj = {answer: 42};
const {proxy, revoke} = Proxy.revocable(obj, {});
console.log(proxy.answer); // 42
revoke();
console.log(proxy.answer); // TypeError: Невозможно выполнить 'get' для
                           // отозванного прокси
```

Обратите внимание, что после отзыва прокси попытка его использования завершилась неудачей.

В примере из Листинга 14-8 не указаны обработчики ловушек, но это сделано просто для краткости. В таком коде могут быть те же обработчики с тем же поведением, что и у прокси, которые вы видели на протяжении всей этой главы.

ОТ СТАРЫХ ПРИВЫЧЕК К НОВЫМ

Предоставляемые `Reflect` и `Proxy` — принципиально новые функции для языка JavaScript и, как правило, они решают новые проблемы, а не предоставляют новые

решения старых. Но все же есть пара конструкций, возможность изменения которых стоит рассмотреть.

Используйте прокси, а не полагайтесь на потребляющий код, чтобы не изменять объекты API

Старая привычка: Предоставление объектов API непосредственно потребляющему коду.

Новая привычка: Предоставить прокси — возможно, отключаемые. Это позволяет контролировать доступ потребляющего кода к объекту, при необходимости включая (если вы предоставляете отключаемый прокси) отмену всего доступа к объекту.

Используйте прокси для отделения кода реализации от инструментального кода

Старая привычка: Смешивание кода инструментария (кода, предназначенного для определения шаблонов использования объекта, производительности и т. д.) с кодом реализации (гарантирующим, что объект выполняет свою работу правильно).

*Новая привычка: И*спользуйте прокси для добавления уровня инструментария, оставляя собственный код объекта не загроможденным.

15

Обновления регулярных выражений

СОДЕРЖАНИЕ ГЛАВЫ

- Свойство `flags`
- Флаги `y`, `u` и `s`
- Именованные группы захвата
- Утверждения ретроспективной проверки
- Экранирование кодовой точки Юникода
- Экранирование свойства Юникода

В этой главе рассказывается о многих новых функциях регулярных выражений, добавленных в ES015 и ES2018, включая свойство `flags` для экземпляров, отражающее все флаги для экземпляра; именованные группы захвата для более удобочитаемых и удобных в обслуживании регулярных выражений; ретроспективные проверки; и экранирование свойств Юникода, предоставляющее доступ к мощному сопоставлению классов.

СВОЙСТВО `FLAGS`

В ES2015 регулярные выражения получили свойство-акцессор `flags`, которое возвращает строку, содержащую флаги для выражения. До его появления единственный способ узнать, какие флаги есть у объекта `RegExp`, заключался в просмотре отдельных свойств, отражающих его индивидуальные флаги (`rex.global`, `rex.multiline` и т. д.), или использовать его метод `toString` и заглянуть в конец строки. Свойство `flags` делает их доступными непосредственно в виде строки:

```
const rex = /example/ig;
console.log(rex.flags); // "gi"
```

Спецификация определяет, что флаги предоставляются в алфавитном порядке, независимо от того, как они были указаны при создании выражения: `gimsuy`. (Три новых флага `s`, `u` и `y` описаны в следующих разделах.) Можно увидеть порядок `ig`, применяемый в примере, в котором создано выражение с флагами, но при отображении свойства `flags` вывод выглядит как `gi`.

НОВЫЕ ФЛАГИ

В ES2015 и ES2018 комитет TC39 добавил новые флаги режима регулярных выражений:

- `y`: «Липкий» флаг (ES2015) означает, что регулярное выражение совпадает только с индексом `lastIndex` объекта регулярного выражения в строке (он не ищет совпадение далее в строке).
- `u`: Флаг «Юникод» (ES2015) включает различные, отключенные по умолчанию, функции Юникода.
- `s`: Флаг «все точки» (ES2018) делает токен «любого символа» (`.`) соответствующим терминаторам строк.

Давайте рассмотрим каждый из них более подробно.

Липкий флаг (y)

Флаг `y` означает, что при выполнении регулярного выражения по строке механизм JavaScript не выполняет поиск по строке, а только проверяет соответствие в строке, начинающейся с индекса объекта регулярного выражения `lastIndex`. См. Листинг 15-1.

Листинг 15-1: Пример липкого флага — sticky-example.js

```
function tryRex(rex, str) {
    console.log(`lastIndex: ${rex.lastIndex}`);
    const match = rex.exec(str);
    if (match) {
        console.log(`Match:     ${match[0]}`);
        console.log(`At:        ${match.index}`);
    } else {
        console.log("No match");
    }
}
const str = "this is a test";

// Не-липкий, ищет строку:
tryRex(/test/, str);
// lastIndex: 0
// Match:     test
// At:        10

// Липкий, не выполняет поиск, совпадает только с `lastIndex`:
const rex1 = /test/y;  // `rex.lastIndex` по умолчанию 0
tryRex(rex1, str);
// lastIndex: 0
// Нет совпадения
```

```
const rex2 = /test/y;
rex2.lastIndex = 10;  // Задает место в строке, которое мы хотим сопоставить
tryRex(rex2, str);
// lastIndex: 10
// Match:     test
// At:        10
```

Запустив код из Листинга 15-1, можно увидеть, что регулярное выражение без липкого флага (`/test/`) просматривает строку в поисках соответствия: `lastIndex`, равный `0`, но регулярное выражение находит слово «test» с индексом 10. С липким флагом (`/test/y`) выражение не найдет «test», когда `lastIndex` равен `0`, так как «test» не находится в строке под индексом 0. Но код находит его при `lastIndex`, равном 10, потому что строка «test» находится в строке «this is a test» под индексом 10.

Это удобно при перемещении токену за токеном по строке и проверке совпадений с набором возможных шаблонов токенов (регулярных выражений), например, при синтаксическом анализе. Чтобы сделать это до добавления липкого флага, требовалось использовать привязку ^ (начало ввода) в начале выражения и отсечь уже обработанные символы из строки перед выполнением сопоставления, чтобы совпадение было в начале строки. Липкий флаг проще и делает процесс более эффективным, позволяя вам избежать создания этих усеченных строк.

Можно посмотреть, установлен ли липкий флаг, просмотрев свойство `flags` или проверив свойство `sticky` выражения: оно хранит значение `true`, если флаг установлен.

Флаг Юникода (u)

В ES2015 улучшили поддержку JavaScript для Юникода во многих областях (см. главу 10 об улучшениях, касающихся строк), в том числе в регулярных выражениях. Однако, чтобы избежать создания проблем для существующего кода, новые функции Юникода в регулярных выражениях по умолчанию отключены. Включаются они с помощью флага `u`. О функциях, для которых требуется флаг `u`, рассказывается в разделе «Функциональные возможности Юникода» далее в этой главе.

Можно проверить, установлен ли флаг Юникода, просмотрев свойство `flags` или проверив свойство `unicode` выражения: оно хранит значение `true`, если флаг установлен.

Флаг «все точки» (s)

В ES2018 добавили флаг `s` («все точки» или «dotAll») в регулярные выражения JavaScript. Из-за множества вариантов регулярных выражений (включая выражения JavaScript) людей часто сбивает с толку тот факт, что маркер «любой символ» (`.`) не совпадает с символами терминаторов строки, такими как `\r` и `\n` (в Юникоде добавлено еще два — `\u2028` и `\u2029`). «Все точки» — это распространенное решение, изменяющее поведение символа «`.`» так, чтобы оно соответствовало терминаторам строк. До ES2018 JavaScript не поддерживал этот флаг, заставляя людей использовать обходные пути, такие как `[\s\S]` (все, что относится или не относится к пробелом), `[\w\W]` (все, что относится или не относится к «символам слова») или очень специфичный для JavaScript `[^]` (пустой класс отрицаемых символов; «не ничего» равно «что-нибудь») и т. д.

Пример флага «все точки» приведен в Листинге 15-3.

Листинг 15-2: Пример флага dotAll — dotAll-example.js

```
const str = "Testing\nAlpha\nBravo\nCharlie\nJavaScript";
console.log(str.match(/.[A-Z]/g));  // ["aS"]
console.log(str.match(/.[A-Z]/gs)); // ["\nA", "\nB", "\nC", "aS"]
```

В этом примере можно увидеть, что без флага `s` совпадает только «aS» в строке «JavaScript», потому что буква «a» соответствует «`.`», а буква «S» соответствует диапазону `[A-Z]`. С флагом `s` также совпадают терминаторы строки перед «A» в «Alpha», «B» в «Bravo» и «C» в «Charlie».

Можно проверить, установлен ли флаг «все точки» для выражения, проверив свойство `flags` и свойство `dotAll`, хранящее значение `true`, если флаг установлен.

ИМЕНОВАННЫЕ ГРУППЫ ЗАХВАТА

ES2018 добавил *именованные группы захвата* в регулярные выражения JavaScript, добавив их к существующим анонимным группам захвата. Именованная группа захвата записывается в такой форме:

```
(?<name>pattern)
```

Название группы помещается в угловые скобки сразу после вопросительного знака (`?`) в начале группы. Именованная группа захвата работает так же, как анонимные группы захвата: следовательно, она доступна в результатах совпадений (`result[1]` и т. д.) как обратная ссылка в выражении (`\1` и т. д.) и в токенах замены при использовании с методом `replace` (`$1` и т. д.). Но также можно ссылаться на именованную группу, используя ее имя, как показано в следующих разделах.

Основная функциональность

Именованная группа захвата появляется в обычном месте в результате сопоставления, а также на новом объекте `groups` внутри результата в качестве свойства с использованием имени группы.

Например, посмотрите на результат с помощью этой анонимной группы захвата:

```
const rex = /testing (\d+)/g;
const result = rex.exec("This is a test: testing 123 testing");
console.log(result[0]);     // testing 123
console.log(result[1]);     // 123
console.log(result.index);  // 16
console.log(result.input);  // This is a test: testing 123 testing
```

Поскольку сопоставление выполнено успешно, результатом будет расширенный массив с полным совпадающим текстом в индексе 0, группой захвата в индексе 1, индексом совпадения в свойстве `index` и входными данными для операции сопоставления в качестве свойства `input`.

Давайте вместо этого использовать именованную группу захвата с именем `number`:

```
const rex = /testing (?<number>\d+)/g;
const result = rex.exec("This is a test: testing 123 testing");
console.log(result[0]);     // testing 123
console.log(result[1]);     // 123
console.log(result.index);  // 16
console.log(result.input);  // This is a test: testing 123 testing
console.log(result.groups); // {"number": "123"}
```

Значение группы захвата по-прежнему отображается под индексом 1, но обратите внимание на новое свойство `groups` — объект со свойствами для каждой именованной группы захвата (в данном случае только для одной — `number`).

У нового объекта `groups` нет прототипа, как если бы он был создан с помощью `Object.create(null)`.

Вот поэтому у него нет каких-либо свойств, даже `toString` и `hasOwnProperty`, которые объекты наследуют от `Object.prototype`. Таким образом, вам не нужно беспокоиться о возможных конфликтах между названиями ваших именованных групп захвата и свойствами, определенными `Object.prototype`. (Когда вы запускали предыдущий пример, в зависимости от среды вы могли видеть что-то в выходных данных, указывающее на то, что у `groups` прототип `null`.)

Наличие имен для групп захвата действительно полезно. Предположим, вы проводите синтаксический анализ даты в формате США `mm/dd/yyyy`. С анонимными группами захвата можно поступить так, как описано в Листинге 15-3.

Листинг 15-3: Парсинг даты в формате США при помощи анонимных групп захвата — anon-capture-groups1.js

```
const usDateRex =
    /^(\d{1,2})[-\/](\d{1,2})[-\/](\d{4})$/;
function parseDate(dateStr) {
    const parts = usDateRex.exec(dateStr);
    if (parts) {
        let year  = +parts[3];
        let month = +parts[1] - 1;
        let day   = +parts[2];
        if (!isNaN(year) && !isNaN(month) && !isNaN(day)) {
            if (year < 50) {
                year += 2000;
            } else if (year < 100) {
                year += 1900;
            }
            return new Date(year, month, day);
        }
    }
    return null;
}
function test(str) {
    let result = parseDate(str);
    console.log(result ? result.toISOString(): "invalid format");
}

test("12/25/2019"); // Проходит парсинг; показывает дату
test("2019/25/12"); // Не проходит парсинг; отображает invalid format (
                    // недействительный формат)
```

Такое решение не дотягивает до идеала, поскольку вы должны помнить, что год находится в индексе 3 (`parts[3]`), месяц — в индексе 1, а день — в индексе 2. Предположим, вы решили улучшить выражение, попросив код сначала попробовать `yyyy-mm-dd`, а затем вернуться к формату США, если в первом случае совпадения не было. Положение становится немного удручающим, как можно видеть в Листинге 15-4.

Листинг 15-4: Добавление второго формата даты при помощи анонимных групп захвата — anon-capture-groups2.js

```
const usDateRex =
    /^(\d{1,2})[-\/](\d{1,2})[-\/](\d{4})$/;
const yearFirstDateRex =
    /^(\d{4})[-\/](\d{1,2})[-\/](\d{1,2})$/;
function parseDate(dateStr) {
    let year, month, day;
    let parts = yearFirstDateRex.exec(dateStr);
    if (parts) {
        year  = +parts[1];
        month = +parts[2] - 1;
        day   = +parts[3];
    } else {
        parts = usDateRex.exec(dateStr);
        if (parts) {
            year  = +parts[3];
            month = +parts[1] - 1;
            day   = +parts[2];
        }
    }
    if (parts && !isNaN(year) && !isNaN(month) && !isNaN(day)) {
        if (year < 50) {
            year += 2000;
        } else if (year < 100) {
            year += 1900;
        }
        return new Date(year, month, day);
    }
    return null;
}
function test(str) {
    let result = parseDate(str);
    console.log(result ? result.toISOString(): "invalid format");
}

test("12/25/2019"); // Проходит парсинг; показывает дату
test("2019-12-25"); // Проходит парсинг; показывает дату
test("12/25/19");   // Не проходит парсинг; отображает invalid format
                    // (недействительный формат)
```

Это решение работает, но требуется выбрать значения разных групп захвата в зависимости от того, какое регулярное выражение дало совпадение, на основе их порядка в выражении с индексами в массиве совпадений. Такой путь делает код несуразным и трудным для чтения, а также неудобным и подверженным ошибкам в обслуживании.

Предположим, что в исходной версии использовались именованные группы захвата, как в Листинге 15-5.

Листинг 15-5: Парсинг даты в формате США при помощи именованных групп захвата — named-capturegroups1.js

```
const usDateRex =
    /^(?<month>\d{1,2})[-\/](?<day>\d{1,2})[-\/](?<year>\d{4})$/;
function parseDate(dateStr) {
    const parts = usDateRex.exec(dateStr);
    if (parts) {
        let year  = +parts.groups.year;
        let month = +parts.groups.month - 1;
        let day   = +parts.groups.day;
        if (!isNaN(year) && !isNaN(month) && !isNaN(day)) {
            if (year < 50) {
                year += 2000;
            } else if (year < 100) {
                year += 1900;
            }
            return new Date(year, month, day);
        }
    }
    return null;
}
function test(str) {
    let result = parseDate(str);
    console.log(result ? result.toISOString(): "invalid format");
}

test("12/25/2019"); // Проходит парсинг; показывает дату
test("12/25/19");   // Не проходит парсинг; отображает invalid format
                    // (недействительный формат)
```

Это уже лучше, так как нет необходимости запоминать порядок групп, можно просто ссылаться на них по их именам (`parts.groups.year` и т. п.). Но добавление другого формата значительно проще; см. Листинг 15-6, отмечая выделенные части, представляющие единственные изменения (кроме примера использования) из Листинга 15-5.

Листинг 15-6: Добавление второго формата даты при помощи именованных групп захвата — named-capturegroups2.js

```
const usDateRex =
    /^(?<month>\d{1,2})[-\/](?<day>\d{1,2})[-\/](?<year>\d{4})$/;
const yearFirstDateRex =
    /^(?<year>\d{4})[-\/](?<month>\d{1,2})[-\/](?<day>\d{1,2})$/;
function parseDate(dateStr) {
    const parts = yearFirstDateRex.exec(dateStr) || usDateRex.exec(dateStr);
    if (parts) {
        let year  = +parts.groups.year;
        let month = +parts.groups.month - 1;
        let day   = +parts.groups.day;
        if (!isNaN(year) && !isNaN(month) && !isNaN(day)) {
            if (year < 50) {
                year += 2000;
            } else if (year < 100) {
                year += 1900;
            }
```

```
            return new Date(year, month, day);
        }
    }
    return null;
}
function test(str) {
    let result = parseDate(str);
    console.log(result ? result.toISOString(): "invalid format");
}

test("12/25/2019"); // Проходит парсинг; показывает дату
test("2019-12-25"); // Проходит парсинг; показывает дату
test("12/25/19");   // Не проходит парсинг; отображает invalid format
                    // (недействительный формат)
```

Гораздо проще!

Обратные ссылки

Именованные группы делают *обратные ссылки* более понятными и простыми в обслуживании.

Возможно, вы знаете, что можно включить обратную ссылку, чтобы позже в выражении сопоставить значение предыдущей группы захвата. Например, следующее выражение соответствует тексту, заключенному в двойные или одинарные кавычки, с помощью группы захвата (`(["'])`) для начальной кавычки и обратной ссылки (`\1`) для конечной кавычки:

```
const rex = /(["']).+?\1/g;
const str = "testing 'a one', \"and'a two\", and'a three";
console.log(str.match(rex)); // ["'a one'", "\"and'a two\""]
```

Обратите внимание, как обратная ссылка гарантировала, что у первой записи были одинарные кавычки с обеих сторон, а вторая запись содержала двойные кавычки с обеих сторон.

Именованная группа захвата может прояснить, на что вы ссылаетесь. Именованные обратные ссылки представлены в виде `\k<имя>`:

```
const rex = /(?<quote>["']).+?\k<quote>/g;
const str = "testing 'a one', \"and'a two\", and'a three";
console.log(str.match(rex)); // ["'a one'", "\"and'a two\""]
```

Теперь вам не нужно подсчитывать группы захвата, чтобы узнать, на что ссылается обратная ссылка. Именование упрощает задачу.

Вы могли бы подумать: «Эй, подождите минутку, этот текст `\k<имя>` для обратной ссылки уже что-то значил!» И вы правы: это излишне экранированный `k`, за которым следует текст `<имя>`. Поскольку такое выражение вполне может существовать в работающем коде (люди часто изолируют символы без необходимости, хотя `k` было бы странно изолировать), формат `\k<имя>` определяет именованную обратную ссылку только в том случае, если в выражении есть именованные группы захвата. В противном случае оно возвращается к своему старому значению. Поскольку в новых выражениях, написанных

с учетом именованных групп захвата, могут быть только именованные группы захвата (последовательность `(?<имя>x)` была синтаксической ошибкой до их добавления), можно безопасно интерпретировать именованную обратную ссылку как именованную обратную ссылку в выражении, содержащем именованные группы захвата.

Наконец: хотя для ясности лучше всего использовать именованную обратную ссылку с именованной группой захвата, вы можете ссылаться на именованную группу захвата со старой анонимной формой (например, `\1`), потому что именованные группы захвата похожи на анонимные, плюс поддерживают дополнительные функции, связанные с именем.

Заменяющие токены

При выполнении замены регулярным выражением (обычно при помощи метода `String.prototype.replace`) в дополнение к знакомому способу обращения к группам захвата с помощью токенов, таких как `$1`, `$2` и т. д., также можно использовать именованные токены в виде `$<имя>`. Например, если требуется преобразовать даты в строку из формата `yyyy-mm-dd` в общий европейский формат `dd/mm/yyyy`:

```
const rex = /^(?<year>\d{2}|\d{4})[-\/](?<month>\d{1,2})[-\/]
(?<day>\d{1,2})$/;
const str = "2019-02-14".replace(rex, "$<day>/$<month>/$<year>");
   console.log(str); // "14/02/2019"
```

Как и в случае с обратными ссылками и результатами сопоставления, вы также можете использовать анонимную форму (`$1` и т. п.) с именованной группой, если хотите.

УТВЕРЖДЕНИЯ РЕТРОСПЕКТИВНОЙ ПРОВЕРКИ

В ES2018 к регулярным выражениям добавлены *утверждения ретроспективной проверки*, как *позитивная ретроспективная проверка* (поиск значения X, которое следует за Y — [(?<=Y)X]), так и *негативная ретроспективная проверка* (поиск значения X, перед которым стоит *не* Y — [(?<! Y)X]). Они уравновешивают утверждения *опережающей* проверки, существовавшие в JavaScript в течение многих лет.

В отличие от некоторых других языков, утверждения ретроспективной проверки в JavaScript не ограничены конструкциями фиксированной длины; вы можете использовать всю мощь регулярных выражений JavaScript в ретроспективных проверках.

Ключевой аспект утверждений ретроспективной проверки, реализованных в JavaScript, — то, что они проверяются справа налево вместо обычной обработки регулярных выражений слева направо. Мы вернемся к этому вопросу в следующих разделах.

Давайте рассмотрим утверждения ретроспективной проверки немного подробнее. Вы можете найти и запустить все примеры в этом разделе (по порядку) в файле **lookbehind.js** в разделе загрузки.

Позитивная ретроспективная проверка

Позитивная ретроспективная проверка имеет вид `(?<=Y)`, где `Y` — это то, что нужно искать. Например, чтобы сопоставить число, следующее за знаком фунта стерлингов (£ — валюта, используемая в Великобритании), без сопоставления со знаком фунта, вы можете использовать позитивную ретроспективную проверку, чтобы утверждать, что он должен быть там:

```
const str1 = "We sold 10 cases for £20 each, and 5 cases for £12.99 each";
const rex1 = /(?<=£)[\d.]+/g;
console.log(str1.match(rex1)); // ["20", "12.99"]
```

(Для упрощения примера здесь просто используется выражение `[\d.]+` для сопоставления чисел, что не является строгим.) Обратите внимание, что `10` и `5` не попали в совпадения, потому что у них не было знака фунта.

Чтобы выполнить сопоставление, концептуально движок находит совпадение для части, не содержащей утверждения (`[\d.]+`), а затем применяет поисковый запрос, беря каждую часть выражения в поисковом запросе и проверяя его на соответствие тексту перед совпадением часть за частью, двигаясь справа налево. В этом примере в ретроспективной проверке есть только одна часть (£), но выражение может быть сложнее:

```
const str2 = 'The codes are: 1E7 ("blue fidget"), 2G9 ("white flugel"),' +
             'and 17Y7 ("black diamond")';
const rex2 = /(?<=\d+[a-zA-Z]\d+ \(").+?(?="\))/g;
console.log(str2.match(rex2));
   // => ["blue fidget", "white flugel", "black diamond"]
```

В этом выражении есть:

- Позитивная ретроспективная проверка, соответствующая формату кода (1E7, 2G9 и т. д.) и открывающая комбинации скобки и кавычки: `(?<=\d+[a-zA-Z]\d+ \(")`
- Выражение, соответствующее описанию: `.+?`
- Позитивная ретроспективная проверка, соответствующая закрывающей комбинации кавычки и скобки: `(?="\))`

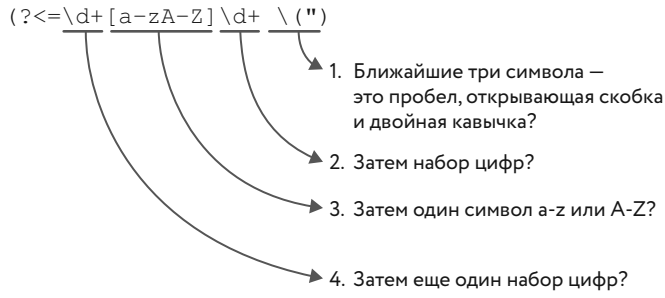


РИСУНОК 15-1

Движок находит совпадение для «.+?», затем проверяет справа налево, соответствует ли текст непосредственно перед совпадением различным частям ретроспективной проверки (?<=\d+[a-zA-Z]\d+ \("), как на рисунке 15-1. (Опять же, движок может быть в состоянии оптимизировать это, но это создает хорошую мысленную модель для работы.)

Негативная ретроспективная проверка

Негативные ретроспективные проверки имеют вид (?<! Y), где Y — это то, чего не должно быть. Итак, если вы хотите сопоставить 10 и 5 из предыдущего примера вместо сопоставления цифр после знаков фунта, ваша первая мысль может заключаться в том, чтобы просто изменить (?<=£) на (?<!£):

```
const str3 = "We sold 10 cases for £20 each, and 5 cases for £ 12.99 each";
const rex3 = /(?<!£)[\d.]+/g;
console.log(str3.match(rex3)); // ["10", "0", "5", "2.99"]
```

А? Почему нашлось сопоставление дополнительным числам?

Если вы не ожидали такого результата, задумайтесь на мгновение, почему (например) 0 в £20 отмечено как совпадение.

Верно! Потому что 0 в £20 не следует сразу за знаком £ (символ прямо перед ним — это 2), как и 2.99 в строке £12.99 (символ прямо перед 2.99 — это 1).

Таким образом, вам нужно будет добавить цифры и десятичные дроби к негативной ретроспективной проверке (опять же, это не так строго, как хотелось бы в продакшн коде, чтобы сохранить простоту примера):

```
const str4 = "We sold 10 cases for £20 each, and 5 cases for £ 12.99 each";
const rex4 = /(?<![£\d.])[\d.]+/g;
console.log(str4.match(rex4)); // ["10", "5"]
```

Как и в случае с позитивной ретроспективной проверкой, негативная проверка также обрабатывается по частям, справа налево.

Жадность проявляется в принципе справа налево в ретроспективных проверках

В ретроспективной проверке, использующей жадные квантификаторы, жадность распределяется справа налево, а не слева направо, как обычно. Это естественным образом вытекает из обработки ретроспективной проверки справа налево.

Можно наблюдать это при наличии одной или нескольких групп захвата в вашей ретроспективной проверке:

```
const behind = /(?<=(?<left>\w+)(?<right>\w+))\d$/;
const behindMatch = "ABCD1".match(behind);
console.log(behindMatch.groups.left);
// => "A"
console.log(behindMatch.groups.right);
// => "BCD"
```

Обратите внимание, что «левый» \w+ получил только один символ, но «правый» \w+ получил все остальные — жадность была справа налево. За пределами ретроспективной проверки (в том числе и при опережающей проверке) жадность проявляется слева направо:

```
const ahead = /\d(?=(?<left>\w+)(?<right>\w+))/;
const aheadMatch = "1ABCD".match(ahead);
console.log(aheadMatch.groups.left);
// => "ABC"
console.log(aheadMatch.groups.right);
// => "D"
```

Это неотъемлемая часть природы поиска справа налево в JavaScript.

Ссылки и нумерация групп захвата

Несмотря на поведение обработки ретроспективных проверок справа налево, нумерация групп захвата внутри них остается прежней (порядок, в котором они начинаются в регулярном выражении, слева направо). В более раннем примере жадности использовались именованные группы захвата и ссылки на них по имени. Давайте рассмотрим тот же пример с анонимными группами захвата, ссылаясь на них по их положению:

```
const behindAgain = /(?<=(\w+)(\w+))\d$/;
const behindMatchAgain = "ABCD1".match(behindAgain);
console.log(behindMatchAgain[1]);
// => "A"
console.log(behindMatchAgain[2]);
// => "BCD"
```

Жадность была справа налево, но номера групп по-прежнему присваивались слева направо. Это делается для простоты нумерации. Группы захвата нумеруются в соответствии с тем, где они находятся в выражении, а не в порядке их обработки.

Хотя в результате они упорядочены таким образом, они все равно *выполняются* справа налево. Можно увидеть это с помощью обратных ссылок. За пределами ретроспективной проверки нельзя с пользой ссылаться на группу, если ссылка находится слева от группы:

```
const rex = /\1\w+(["'])/; // Не имеет смысла
```

Это не синтаксическая ошибка, но значение ссылки (\1) ничему не будет соответствовать, потому что, когда оно использовалось выражением, захват еще не был выполнен.

По той же самой причине при ретроспективной проверке ссылка не может быть полезной *справа* от захвата:

```
const referring1 = /(?<=(["'])\w+\1) X/; // Не имеет смысла
console.log(referring1.test("'testing'X"));
// => ложь
```

Вместо этого, поскольку обработка выполняется справа налево, вы помещаете захват справа и ссылаетесь на него слева:

```
const referring2 = /(?<=\1\w+(["']))X/;
console.log(referring2.test("'testing'X"));
// => истина
```

Это верно независимо от того, представлено ли выражение анонимной группой захвата или именованной:

```
const referring3 = /(?<=\k<quote>\w+(?<quote>["']))X/;
console.log(referring3.test("'testing'X"));
// => истина
```

Термин «обратная ссылка» все еще используется. Думайте о слове «обратная» в словосочетании «обратная ссылка» как о том, чтобы оглядываться назад при *обработке* выражения, а не оглядываться в *определении* выражения.

ФУНКЦИОНАЛЬНЫЕ ВОЗМОЖНОСТИ ЮНИКОДА

В главе 10 вы узнали, что обработка Юникода в JavaScript заметно улучшилась в ES2015 и далее. Это улучшение распространяется и на регулярные выражения. Чтобы включить новые функции, регулярное выражение должно получить флаг `u`. Это позволит сохранить обратную совместимость для существующих регулярных выражений, которые могут использовать новый синтаксис (возможно, непреднамеренно или без необходимости). Например, одна из новых синтаксических возможностей присваивает значение для последовательности `\p` и `\P`, где раньше были лишние экранирования букв `p` и `P` соответственно. Простое изменение значения привело бы к нарушению любого регулярного выражения, написанного с ненужными экранирующими символами для `p` или `P`. Но можно придать им новые значения в выражении, созданном с помощью нового флага `u`.

Экранирование кодовой точки

Старая экранирующая последовательность Юникода, `\uNNNN`, определяет единственную кодовую *единицу* UTF16. Однако, как описывается в главе 10, кодовая единица может быть только половиной суррогатной пары. Например, чтобы сопоставить эмодзи «улыбающееся лицо с улыбающимися глазами» (U +1F60A) с помощью экранирующей последовательности (вместо того, чтобы просто использовать эмодзи буквально — ☺), вам нужны две базовые экранирующие последовательности Юникода, представляющие две кодовые единицы UTF-16 для этого эмодзи (0xD83D и 0xDE0A):

```
const rex = /\uD83D\uDE0A/;
const str = "Testing: ☺";
console.log(rex.test(str)); // истина
```

Вычислять кодовые единицы UTF16 для кодовой точки Юникода трудно.

Начиная с ES2015 регулярное выражение, использующее флаг u, может использовать *кодовую точку экранирующей последовательности* вместо: фигурная скобка (`{`) после `\u`, затем кодовая точка в шестнадцатеричном виде, за которой следует закрывающая фигурная скобка (`}`):

```
const rex = /\u{1F60A}/u;
const str = "Testing: ☺";
console.log(rex.test(str)); // истина
```

Экранирование кодовых точек работает не только само по себе: их можно использовать в классах символов, чтобы соответствовать *диапазону* кодовых точек. Следующее соответствует всем кодовым точкам в блоке «Эмотиконы» Юникода[89]: `/[\u{1F600}-\u{1F64F}]`.

Экранирование свойства Юникода

Стандарт Юникод не просто присваивает символам числовое значение, но и предоставляет огромное количество информации о самих символах. Например, стандарт Юникода может сообщить вам, что символ «î» относится к латинскому шрифту, буквенному, а не числовому, и не представляет собой знак препинания; он может сообщить вам, что символ «¿» — это общий для нескольких скриптов знак препинания; и т. д. Эти различные вещи называются *свойствами Юникода*. Начиная с ES2018 регулярное выражение, использующее флаг `u`, может включать *экранирование свойств Юникода* для сопоставления символов по их свойствам Юникода. Поскольку это достаточно свежее решение, как всегда, требуется проверить его поддержку в ваших целевых средах.

Существует несколько типов свойств: два, относящиеся к регулярным выражениям JavaScript, — это *двоичные свойства*, которые (как следует из названия) являются либо истинными, либо ложными, и *перечисляемые свойства*, поддерживающие список возможных значений. Например, выражение `\p{Alphabetic}` использует двоичное свойство `Alphabetic` для соответствия любому символу, считающемуся алфавитным по стандарту Юникода:

```
const rex1 = /\p{Alphabetic}/gu;
const s1 = "Hello, I'm James.";
console.log(s1.match(rex1));
// => ["H", "e", "l", "l", "o", "I", "m", "J", "a", "m", "e", "s"]
```

Как можно заметить, экранирование начинается с `\p{` и заканчивается}, со свойством, которое должно совпадать внутри фигурных скобок.

(Все примеры в этом разделе можно найти и запустить в файле **unicode-property-escapes.js** в загрузках.)

Выражение `\p` означает *позитивное* совпадение свойств Юникода. Для *негативного* (например, соответствие всех *неалфавитных* символов) используется заглавная `P` вместо строчной (это согласуется с другими типами экранирования, как, например, `\d` для цифр и `\D` для не-цифр):

```
const rex2 = /\P{Alphabetic}/gu;
const s2 = "Hello, I'm James.";
console.log(s2.match(rex2));
// => [",", " ", "\"", " ", "."]
```

[89] https://en.wikipedia.org/wiki/Emoticons_(Unicode_block)

Можно также использовать указанные псевдонимы (например, `Alpha` вместо `Alphabetic`). Значения логических свойств и доступные для использования псевдонимы перечислены в таблице «Псевдонимы свойств в двоичном Юникоде и их канонические имена свойств»[90] в спецификации.

Есть три доступных для использования перечисляемых свойства:

- `General_Category` (псевдоним: `gc`). Самое основное общее свойство символов, категоризация символов Юникода на буквы, знаки препинания, символы, знаки, цифры, разделители и другие (с различными подкатегориями). Значения и псевдонимы `General_Category`, доступные для использования, перечислены в таблице «Псевдонимы значений и канонические значения для свойства `General_Category` Юникода»[91] в спецификации. Более подробная информация о свойстве `General_Category` находится в Техническом стандарте Юникода № 18[92].
- `Script` (псевдоним: `sc`). Присваивает символу отдельную категорию скрипта, такую как `Latin`, `Greek`, `Cyrillic` и т. д. Категорию `Common` для символов, используемых во многих скриптах. `Inherited` для символов, используемых во многих скриптах, которые наследуют свой скрипт от предыдущего базового символа. `Unknown` для различных кодовых точек, не попадающих в классификацию скриптов. Значения и псевдонимы, доступные для использования, перечислены в спецификации в таблице «Псевдонимы значений и канонические значения для скрипта свойств Юникода и Script_Extensions». Более подробная информация о свойстве `Script` находится в Техническом стандарте Юникода № 24 в разделе «Свойства скриптов»[93].
- `Script_Extensions` (псевдоним: `scx`). Присваивает набор категорий скриптов символам (а не просто `Common`), чтобы более точно указать скрипты, в которых встречается символ. Допустимые имена и псевдонимы значений те же, что и для `Script`. Более подробная информация о свойстве `Script` находится в Техническом стандарте Юникода № 24 в разделе «Свойство Script_Extensions»[94].

Например, чтобы найти все символы греческого алфавита в строке:

```
const rex3 = /\p{Script_Extensions=Greek}/gu;
const s3 = "The greek letters alpha (α), beta (β), and gamma (γ)
            are used...";
console.log(s3.match(rex3));
// => ["α", "β", "γ"]
```

(Можно использовать псевдоним `scx`: `/\p{scx=Greek}/gu`.)

Наиболее полезное перечисляемое свойство — это `General_Category`, поэтому есть сокращенная форма: вы можете оставить только часть `General_Category=`.

---

[90] https://tc39.es/ecma262/#table-binary-unicode-properties

[91] https://tc39.es/ecma262/#table-unicode-general-category-values

[92] https://unicode.org/reports/tr18/#General_Category_Property

[93] https://unicode.org/reports/tr24/#Script

[94] https://unicode.org/reports/tr24/#Script_Extensions

Например, выражения `\p{General_Category=Punctuation}` и `\p{Punctuation}` ищут пунктуацию в строке:

```
const rex4a = /\p{General_Category=Punctuation}/gu;
const rex4b = /\p{Punctuation}/gu;
const s4 = "Hello, my name is Pranay. It means \"romance\" in Hindi.";
console.log(s4.match(rex4a));
// => [",", "'", ".", "\"", "\"", "."]
console.log(s4.match(rex4b));
// => [",", "'", ".", "\"", "\"", "."]
```

Возможно, вы думаете: «Но разве не так указывается двоичное свойство, например, `Alphabetic`?» Да, так. Поскольку нет перекрытия между допустимыми значениями/псевдонимами `General_Category` и именами/псевдонимами двоичных свойств, эта ситуация не станет двусмысленной для анализатора регулярных выражений, хотя и может быть непонятной для будущих читателей кода, если они не знают псевдоним. Использовать ли сокращенную форму для свойств `General_Category` — вопрос стиля.

Имена и значения свойств чувствительны к регистру; `Script` — допустимое имя свойства, `script` — нет.

Если вы привыкли к подобным функциям в других вариантах регулярных выражений (regex), обратите внимание, что экранирование свойств Юникода в JavaScript довольно строгое и узконаправленное (во всяком случае, на данный момент). Спецификация не поддерживает различные сокращенные или альтернативные формы, а также дополнительные свойства Юникода (и это не позволяет движкам поддерживать их в качестве дополнения). Например:

- Отбросить фигурные скобки. В некоторых вариантах регулярных выражений `\p{L}` (использование псевдонима для значения `Lower` в свойстве `General_Category`) может быть записано как `\pL`. Такой синтаксис не поддерживается в JavaScript.
- Использовать «`:`» вместо «=» при указании имени и значения. Некоторые варианты регулярных выражений позволяют писать `\p{scx: Greek}` в дополнение к варианту `\p{scx=Greek}`. В JavaScript это должен быть знак равенства (=).
- Позволять использовать «is» в разных местах. Некоторые варианты регулярных выражений позволяют добавлять «is» к именам или значениям свойств, например `\p{Script=IsGreek}`. Это не допускается в JavaScript.
- Использовать свойства `General_Category`, `Script` и `Script_Extensions`, отличные от двоичных свойств. Некоторые варианты регулярных выражений поддерживают другие свойства, такие как `Name`. По крайней мере, на данный момент JavaScript этого не поддерживает.

Добавление новой функции экранирования свойств с небольшой, четко определенной областью действия и строгими правилами упрощает ее добавление разработчикам движка и использование пользователями. Дополнительные функции всегда могут быть добавлены последующими предложениями, если они получат достаточную поддержку для прохождения процесса.

Приведенные до сих пор примеры были немного надуманными; с точки зрения реального использования полезен один из примеров из предложения экранирования Юникода[95] — версия `\w`, поддерживающая Юникод. Как известно, `\w` довольно англоцентрично и определяется как `[A-Za-z0-9_]`. И такое экранирование не соответствует всем символам, встречающимся во многих обычных для английского языка словах, которые были заимствованы из других языков (таких как «résumé» или «naïve»), не говоря уже о словах, написанных на других языках. Версия, поддерживающая Юникод, как описано в Техническом стандарте Юникода № 18[96]:

```
[\p{Alphabetic}\p{Mark}\p{Decimal_Number}\p{Connector_Punctuation}
 \p{Join_Control}]
```

или использование сокращенных псевдонимов:

```
[\p{Alpha}\p{M}\p{digit}\p{Pc}\p{Join_C}]
```

См. полный пример предложения `\w` с поддержкой Юникода в Листинге 15-7.

Листинг 15-7: Версия \w, поддерживающая Юникод — unicode-aware-word.js

```
// Из: https://github.com/tc39/proposal-regexp-unicode-property-escapes
// (Изменено для использования сокращенных свойств по соображениям
// форматирования страницы)
const regex = /([\p{Alpha}\p{M}\p{digit}\p{Pc}\p{Join_C}]+)/gu;
const text = `
Amharic: የኔ ማንጠቢያ መኪና በዓሣዎች ተሞልቷል
Bengali: আমার হভারক্রাফ্ট কুঁচে মাছ-এ ভরা হয়ে গছে
Georgian: ჩემი ხომალდი საჰაერო ბალიშზე სავსეა გველთევზებით
Macedonian: Моето летачко возило е полно со јагули
Vietnamese: Tàu cánh ngầm của tôi đầy lươn
`;
let match;
while (match = regex.exec(text)) {
    const word = match[1];
    console.log(`Matched word with length ${word.length}: ${word}`);
}
// Результат:
// Matched word with length 7: Amharic
// Matched word with length 2: የኔ
// Matched word with length 6: ማንጠቢያ
// Matched word with length 3: መኪና
// Matched word with length 5: በዓሣዎች
// Matched word with length 5: ተሞልቷል
// Matched word with length 7: Bengali
// Matched word with length 4: আমার
// Matched word with length 11: হভারক্রাফ্ট
// Matched word with length 5: কুঁচে
// Matched word with length 3: মাছ
// Matched word with length 1: এ
// Matched word with length 3: ভরা
```

[95] https://github.com/tc39/proposal-regexp-unicode-property-escapes

[96] http://unicode.org/reports/tr18/#word

```
// Matched word with length 3: হয়ে
// Matched word with length 4: গেছে
// Matched word with length 8: Georgian
// Matched word with length 4: ჩემი
// Matched word with length 7: ხომალდი
// Matched word with length 7: საჰაერო
// Matched word with length 7: ბალიშზე
// Matched word with length 6: სავსეა
// Matched word with length 12: გველთევზებით
// Matched word with length 10: Macedonian
// Matched word with length 5: Моето
// Matched word with length 7: летачко
// Matched word with length 6: возило
// Matched word with length 1: е
// Matched word with length 5: полно
// Matched word with length 2: со
// Matched word with length 6: јагули
// Matched word with length 10: Vietnamese
// Matched word with length 3: Tàu
// Matched word with length 4: cánh
// Matched word with length 4: ngầm
// Matched word with length 3: của
// Matched word with length 3: tôi
// Matched word with length 3: đầy
// Matched word with length 4: l n
```

ОТ СТАРЫХ ПРИВЫЧЕК К НОВЫМ

Существуют различные старые привычки, которые при желании можно заменить на новые. Но есть одно условие: ваши целевые среды должны поддерживать новые функции (или они могут быть транспилированы; на момент написания этой книги в Babel были плагины для большинства функций в этой главе).

Используйте липкий флаг (y) вместо создания подстрок и использования «^» при синтаксическом анализе

Старая привычка: При сопоставлении регулярного выражения со строкой в указанной позиции разделить строку в этой позиции, чтобы использовать привязку начала ввода (^):

```
const digits = /^\d+/;
// ...тогда где-то у вас есть `pos`...
let match = digits.exec(str.substring(pos));
if (match) {
    console.log(match[0]);
}
```

Новая привычка: Используйте липкий флаг (y) без разделения строки и без символа ^:

```
const digits = /\d+/y;
// ...тогда где-то у вас есть `pos`...
digits.lastIndex = pos;
let match = digits.exec(str);
```

```
if (match) {
    console.log(match[0]);
}
```

Используйте флаг «все точки» (s) вместо использования обходных путей для сопоставления всех символов (включая разрывы строк).

Старая привычка: Использование различных обходных путей для сопоставления всех символов, таких как `[\s\S]` или `[\d\D]`, или очень специфичный для JavaScript вариант `[^]`:

```
const inParens = /\(([\s\S]+)\)/;
const str =
`This is a test (of
line breaks inside
parens)`;
const match = inParens.exec(str);
console.log(match ? match[1]: "no match");
// => "of\nline breaks inside\nparens"
```

Новая привычка: Используйте флаг «все точки» (`s`) и «`.`»:

```
const inParens = /\((.+)\)/s;
const str =
`This is a test (of
line breaks inside
parens)`;
const match = inParens.exec(str);
console.log(match ? match[1]: "no match");
// => "of\nline breaks inside\nparens"
```

Используйте именованные группы захвата вместо анонимных

Старая привычка: Использовать несколько анонимных групп захвата (потому что у вас не было выбора) и тщательно следить за использованием правильных индексов в результате сопоставления, в ссылках на захват в самом регулярном выражении и т. д.:

```
// Если вы измените это, обязательно измените назначение деструктуризации позже!
const rexParseDate = /^(\d{2}|\d{4})-(\d{1,2})-(\d{1,2})$/;
const match = "2019-02-14".match(rexParseDate);
if (match) {
    // Зависит от порядка захвата регулярного выражения!
    const [, year, month, day] = match;
    console.log(`day: ${day}, month: ${month}, year: ${year}`);
} else {
    console.log("no match");
}
   // => "day: 14, month: 02, year: 2019"
```

Новая привычка: Использовать именованные группы захвата (`(?<captureName>content)`) и именованные свойства объекта `groups` (`match.`

`groups.captureName`) или именованные ссылки в регулярном выражении (`\k{captureName}`), и т. д.:

```
const rexParseDate = /^(?<year>\d{2}|\d{4})-(?<month>\d{1,2})-(?<day>\d{1,2})$/;
const match = "2019-02-14".match(rexParseDate);
if (match) {
    const {day, month, year} = match.groups;
    console.log(`day: ${day}, month: ${month}, year: ${year}`);
} else {
    console.log("no match");
}
// => "day: 14, month: 02, year: 2019"
```

Используйте ретроспективные проверки вместо различных обходных путей

Старая привычка: Различные обходные пути (ненужные захваты и т. д.), потому что в JavaScript раньше не было ретроспективных запросов.

Новая привычка: Используйте мощный инструмент ретроспективных проверок от JavaScript.

Используйте экранирование кодовой точки вместо суррогатных пар в регулярных выражениях

Старая привычка: Использование суррогатных пар в регулярных выражениях, иногда заметно усложняя их, вместо использования кодовых точек (потому что выбора не было). Например, чтобы сопоставить как блок Юникода «Эмотиконы» (упомянутый ранее), так и блок «Дингбат»[97] в регулярном выражении:

```
const rex = /(?:\uD83D[\uDE00-\uDE4F]|[\u2700-\u27BF])/;
```

Обратите внимание, что в таком случае требуется чередование, потому что для этого необходимо обработать суррогатную пару для Эмотикона, а затем отдельный диапазон для Дингбатов.

Новая привычка: Используйте экранирование кодовой точки:

```
const rex = /[\u{1F600}-\u{1F64F}\u{1F680}-\u{1F6FF}]/u;
```

Обратите внимание, что теперь это односимвольный класс с парой диапазонов в нем.

Используйте шаблоны Юникода вместо обходных путей

Старая привычка: Работа над отсутствием шаблонов Юникода с большими трудными в обслуживании классами символов, выбирающими соответствующие диапазоны Юникода.

Новая привычка: Используйте экранирование свойств Юникода (сначала убедитесь, что ваша целевая среда или транспилятор поддерживают их).

---

[97] https://en.wikipedia.org/wiki/Dingbat#Dingbats_Unicode_block

16

Совместно используемая память

СОДЕРЖАНИЕ ГЛАВЫ

- Совместное использование памяти между потоками (`SharedArrayBuffer`)
- Объект `Atomics`

В этой главе вы узнаете о функции совместного использования памяти JavaScript (в ES2017+), позволяющей совместно использовать память между потоками, и об объекте Atomics, который можно использовать для выполнения низкоуровневых операций с общей памятью, а также для приостановки и возобновления потоков на основе событий в общей памяти.

ВВЕДЕНИЕ

В течение большей части десятилетия или около того, когда JavaScript в браузерах не был однопоточным (благодаря веб-воркерам), не было никакого способа совместно использовать (разделить) память между потоками в браузере. Первоначально потоки могли отправлять друг другу сообщения с содержащимися в них данными, но данные были скопированы. Когда данных много или их приходится часто отправлять туда и обратно, это проблема. Несколько лет спустя, когда были определены типизированные массивы (глава 11), первоначально вне JavaScript, а затем добавлены в ES2015, во многих случаях стало возможным передавать данные из одного потока в другой без их копирования, но отправляющий поток должен был отказаться от доступа к отправляемым данным. В ES2017 это изменилось с помощью SharedArrayBuffer. При помощи SharedArrayBuffer код JavaScript, выполняемый в одном потоке, может совместно использовать память с кодом JavaScript, выполняемым в другом потоке. Прежде чем перейти к тому, как работает совместно используемая память в JavaScript, давайте рассмотрим вопрос о том, нужно ли вам вообще ее использовать.

ЗДЕСЬ ВОДЯТСЯ ДРАКОНЫ!

Или действительно ли вам нужна совместно используемая память?

Совместное использование памяти между потоками открывает множество проблем синхронизации данных, с которыми программистам, работающим в типичных средах JavaScript, раньше не приходилось сталкиваться. Мы могли бы потратить целую книгу на тонкости совместно используемой памяти: гонки данных, переупорядоченные хранилища, ускоренное чтение, кэширование потоков процессора и т. д. В этой главе у нас есть место только для рассмотрения самых основ этих проблем. Для того чтобы использовать совместно используемую память в реальном мире, вам необходимо досконально разобраться в этих проблемах — или, по крайней мере, знать и строго придерживаться лучших практик и шаблонов для их предотвращения. В противном случае вы потратите часы или дни на выслеживание малозаметных, часто трудно воспроизводимых ошибок, или, что еще хуже, вы этого *не сделаете*, а затем получите отчеты об ошибках, которые практически невозможно воспроизвести на местах. Развитие этих знаний и/или освоение этих передовых методов требует значительных затрат времени. Погружение в совместно используемую память — это не то, что нужно делать легкомысленно.

В большинстве случаев программистам не требуется совместно используемая память, отчасти благодаря *передаваемым* элементам — объектам, которые вы можете *передавать* между потоками, а не копировать их. Передаваться могут различные объекты (включая некоторые типы изображений и холсты в браузерах), в том числе `ArrayBuffer`. Поэтому, если у вас есть данные, которые вам нужно передавать туда и обратно между двумя потоками, вы можете избежать накладных расходов на их копирование, *передав* их. Например, следующий код создает массив `Uint8Array` и передает его потоку воркера, передавая его базовые данные через буфер, а не копируя их:

```
const MB = 1024 * 1024;
let array = new Uint8Array(20 * MB);
// ...заполняется 20 МБ данных...
worker.postMessage({array}, [array.buffer]);
```

Первый аргумент — это данные для отправки (массив); второй аргумент — массив передаваемых данных для передачи, а не для копирования. Поток воркера получает клон *объекта* массива, но его *данные* передаются: клон повторно использует данные *исходного* массива, поскольку буфер `ArrayBuffer` массива указан в списке передаваемых элементов. Это похоже на передачу эстафетной палочки в гонке: отправляющий поток передает буфер (эстафетную палочку) потоку воркера, который принимает его и работает с ним. После вызова `postMessage` массив отправляющего потока утрачивает доступ к буферу: `array` в этом примере фактически становится массивом нулевой длины.

Когда воркер получает массив, он может использовать данные и работать с ними, не беспокоясь о том, что другие потоки работают с ними, а затем при необходимости может передать их обратно в исходный поток (или какой-либо другой).

Такая передача данных туда и обратно довольно эффективна и позволяет избежать целых классов проблем, которые могут возникнуть при буквальном разделении базового буфера между потоками. Прежде чем использовать совместно используемую память, стоит спросить себя, будет ли достаточно переноса. Вы можете сэкономить себе *много* времени и избавиться от хлопот.

ПОДДЕРЖКА БРАУЗЕРА

И последнее, прежде чем мы перейдем к описанию совместно используемой памяти. В январе 2018 года браузеры отключили разделяемую память в ответ на факторы уязвимости Spectre и Meltdown в процессорах. Chrome вернул поддержку некоторых вариантов использования в июле того же года на платформах, где была активна функция изоляции сайтов. Но к концу 2019 года благодаря напряженной работе большого списка людей появился более общий подход, который внедрили в браузеры с начала 2020 года. Теперь снова можно совместно использовать память, но только между *безопасными контекстами*[98].

Если вас не беспокоит вопрос «зачем?», а только вопрос «как?» — короткая версия ответа заключается в том, что для совместного использования памяти вам нужно сделать две вещи: А) отправлять ваши документы и скрипты безопасно или локально (то есть через `https`, `localhost` или их аналоги) и Б) включите в них эти два HTTP-заголовка:

```
Cross-Origin-Opener-Policy: same-origin
Cross-Origin-Embedder-Policy: require-corp
```

(Первый заголовок не нужен для содержимого, предназначенного для фреймов, или для скриптов, но это нормально, если он все же есть.)

Если вас не интересуют подробности, можете перейти к следующему разделу прямо сейчас.

(Короткая пауза…)

Вы еще здесь? Хорошо!

Вкратце контекст — это концепция веб-спецификации для контейнера окна/вкладки/фрейма или воркера (например, веб-/выделенного или сервисного воркера). Контекст, содержащий окно, используется повторно при выполнении навигации в окне, даже при переходе от одного источника к другому. Базы данных realm DOM и JavaScript отбрасываются, а новые создаются во время навигации, но все еще в том же контексте, по крайней мере, обычно.

Браузеры группируют связанные контексты вместе в *контекстные группы*. Например, если окно содержит `iframe`, контексты для окна и `iframe` обычно находятся в одном контексте группы, потому что они могут взаимодействовать друг с другом (с помощью `window.parent`, `window.frames` и т. д.). Аналогично контексты для окна и открываемого им всплывающего окна традиционно находятся в одной и той же контекстной группе — опять же потому, что они могут взаимодействовать. Если у контекстов разное происхождение, их взаимодействие ограничено по умолчанию, но все же возможности есть.

Почему контексты и контекстные группы имеют такое важное значение? Потому что, как правило, все контексты в группе контекстов используют память в рамках одного и того же процесса операционной системы. Современные браузеры — это многопроцессорные приложения, обычно с одним общим координирующим процессом, а затем отдельным процессом для каждой контекстной группы. Такая архитектура делает браузер более надежным: сбой в одной контекстной группе не влияет ни на другие, ни на процесс координации, поскольку они находятся в совершенно разных процессах операционной

[98] `https://w3c.github.io/webappsec-secure-contexts/`

системы. Но все, что находится в одной и той же контекстной группе, обычно находится в одном и том же процессе.

Именно здесь в историю вступают факторы Spectre и Meltdown.

Spectre и Meltdown — это аппаратные уязвимости. Они используют предсказание ветвлений в современных процессорах таким образом, что код может получить доступ ко *всей* памяти в текущем процессе, минуя любые проверки доступа в процессе. Они не ограничиваются браузерами, но в браузерах совместно используемая память и высокоточные таймеры позволили использовать эти уязвимости в коде JavaScript. Это означает, что вредоносный код может считывать *любую* память в процессе, даже память для другого окна или веб-воркера. Он может обойти любые проверки доступа внутри процесса, которые может поддерживать браузер, например, между контекстами из разных источников.

Решение состоит в том, чтобы гарантировать, что содержимое ресурсов (скрипты, документы) не было подделано (требуя, чтобы они доставлялись безопасно или локально), и разрешить ресурсам ограничивать, в какие контекстные группы они могут быть загружены (через заголовки).

Давайте взглянем на эти заголовки.

Первый заголовок — `Cross-Origin-Opener-Policy`[99] — позволяет окну верхнего уровня гарантировать, что оно находится только в контекстной группе с другими окнами верхнего уровня из того же источника, у которых то же значение заголовка, что и у него (обычно `same-origin`). При навигации на верхнем уровне, если источник и заголовок нового содержимого не совпадают с предыдущим, браузер создает новый контекст в новой контекстной группе для хранения содержимого нового ресурса.

Второй заголовок — `Cross-Origin-Embedder-Policy`[100] — позволяет ресурсу указать, что он может быть загружен только из того же окна источника или окна, которое ресурс явно разрешает с помощью заголовка Cross-Origin Resource Policy[101] (CORP) или Cross-Origin Resource Sharing[102] (CORS). Это гарантирует, что ресурс не может быть встроен в контейнер, которому он не доверяет.

При совместном применении они защищают от использования совместно используемой памяти и высокоточных таймеров для работы Spectre или Meltdown для доступа к данным, к которым код не должен иметь доступа. Если вы хотите узнать об этом более подробно, я рекомендую статью «COOP and COEP explained»[103] Артура Янка, Чарли Рейса и Анны ван Кестерен.

ОСНОВЫ СОВМЕСТНО ИСПОЛЬЗУЕМОЙ ПАМЯТИ

В этом разделе вы узнаете некоторые основы совместно используемой памяти: что такое *критические секции*; как создавать, разделять и использовать совместно используемую память; и как использовать блокировку для защиты критических разделов кода с помощью *блокировок* и *условных переменных*.

---

[99] `https://docs.google.com/document/d/1Ey3MXcLzwR1T7aarkpBXEwP7jKdd2NvQdgYvF8_8scI/edit`

[100] `https://wicg.github.io/cross-origin-embedder-policy/`

[101] `https://fetch.spec.whatwg.org/#cross-origin-resource-policy-header`

[102] `https://fetch.spec.whatwg.org/#http-cors-protocol`

[103] `https://docs.google.com/document/d/1zDlfvfTJ_9e8Jdc8ehuV4zMEu9ySMCiTGMS9y0GU92k/edit`

Критические секции, блокировки и условные переменные

Критическая секция — это код, выполняемый в потоке, ему необходимо получить доступ (чтение и/или запись) к общей памяти *атомарным* способом относительно любых действий, выполняемых параллельными потоками, которые совместно используют память. Например, в критической секции чтение одной и той же памяти дважды всегда должно приводить к одному и тому же значению:

```
const v1 = shared[0];
const v2 = shared[0];
// В критической секции `v1` и `v2` ДОЛЖНЫ содержать одно и то же значение
```

Аналогично в критической секции, если код считывает данные из общей памяти, обновляет значение и записывает обратно в общую память, не должно существовать возможности, что он перезаписывает другое значение, записанное туда другим потоком за это время:

```
const v = shared[0] + 1;
shared[0] = v; // НЕ ДОЛЖНО перезаписываться значение, записанное другим
               // потоком после считывания выше (в критической секции)
```

Блокировка — это средство защиты критических секций путем предоставления эксклюзивного доступа к общей памяти потоку, использующему блокировку. Блокировка может быть получена только одним потоком одновременно, и только этому потоку разрешен доступ к совместно используемой памяти. (Блокировку иногда называют *мьютексом*, что означает «взаимное исключение».)

Условная переменная предоставляет потокам, использующим данную блокировку, возможность ожидать, пока условие станет истинным, и уведомлять друг друга об этом. Когда поток получает уведомление о том, что условие истинно, он может попытаться получить блокировку, к которой относится переменная условия, чтобы получить возможность выполнить работу теперь, когда условие истинно (или, по крайней мере, теперь, когда оно *было* истинным всего мгновение назад — как с путешествием во времени, времена глагола могут быть немного сложными в параллельном программировании).

В простой ситуации вам нужна только одна условная переменная для блокировки, и эти две вещи иногда объединяются: условие — «блокировка доступна». Этого было бы достаточно, если бы, например, у вас было несколько потоков, которым нужно было выполнять одну и ту же работу с общим буфером.

Но ситуации часто бывают более сложными, требуя более одной условной переменной для одной и той же блокировки и совместно используемой памяти. Один из классических способов применения совместно используемой памяти — распределение работы между потоками через очередь: поток-производитель помещает части работы в очередь, а потоки-потребители берут работу из очереди и выполняют ее. Очередь — это совместно используемая память. Размещение работы в очереди — критическая секция кода: чтобы сделать это, не нарушая очередь, потоку необходим временный эксклюзивный доступ к очереди. У вас не может быть двух потоков, изменяющих внутреннее состояние очереди одновременно, потому что они будут мешать изменениям друг друга. Извлечение работы из очереди — еще одна критическая секция по той же причине. Потоки используют

блокировку для защиты этих критических секций, чтобы очередь не изменялась более чем одним потоком одновременно. Пока все хорошо.

Если в очереди нет работы, потоки-потребители должны ждать, пока в нее не будет добавлена работа. Поскольку размер очереди ограничен (допустим, она может содержать 100 записей); если она заполнена, потоку-производителю приходится ждать, пока она не освободится, чтобы добавить в нее часть работы. Это два разных условия, связанных с одной и той же блокировкой: условие «в очереди есть работа», которую ожидают потребители, и условие «в очереди есть место», которого ожидает производитель.

Хотя было бы возможно реализовать эту очередь производителя/потребителя только с одним условием («блокировка доступна»), это было бы неэффективно по нескольким причинам:

- Если бы существовало только одно условие, потоку-производителю и потокам-потребителям пришлось бы ожидать выполнения этого условия по разным причинам. Хотя, когда это условие становится истинным, это может означать, а может и не означать, что потоки могут выполнять свою работу. Например, если четыре воркера ждут работы и добавляется одна часть работы, все они получают уведомление и соревнуются, чтобы получить блокировку, а затем тот, кто получил блокировку, берет работу, в то время как остальные трое возвращаются к ожиданию. Как только этот потребитель забирает работу из очереди, он снимает блокировку, уведомляя всех других потребителей, что заставляет их мчаться, чтобы получить блокировку, даже если в очередь не было добавлено никакой новой работы — это второе уведомление было для потока производителя, чтобы сообщить ему о свободном месте в очереди, но не для потоков-потребителей.
- Если бы существовало только одно условие, все потоки, использующие блокировку, должны были быть уведомлены о том, что условие стало истинным. Напротив, если есть два отдельных условия («в очереди есть работа» и «в очереди есть место»), только *один* поток должен получить уведомление, когда какое-либо из этих условий становится истинным. Нет необходимости будить все потоки-потребители, когда условие «в очереди есть работа» становится истиной: все они будут бороться за блокировку, и только один из них может выиграть эту гонку. И вообще нет смысла пробуждать какой-либо поток-потребитель для условия «в очереди есть место» — только поток-производитель ожидает это условие.

Вот почему блокировки и условные переменные — это отдельные понятия, но тесно связанные между собой. Далее в этой главе вы увидите пример именно такой ситуации между производителем и потребителем.

Создание совместно используемой памяти

Чтобы создать массив с совместно используемой памятью, сначала необходимо создать буфер `SharedArrayBuffer`, а затем использовать его при создании массива. Поскольку размер `SharedArrayBuffer` указывается в байтах, а не в записях, обычно нужное количество записей умножают на свойство `BYTES_PER_ELEMENT` конструктора

массива[104]. Например, для создания совместно используемого массива `Uint16Array` с пятью записями:

```
const sharedBuf = new SharedArrayBuffer(5 * Uint16Array.BYTES_PER_ELEMENT);
const sharedArray = new Uint16Array(sharedBuf);
```

Чтобы совместно использовать этот массив с веб-воркером в браузере, можно включить его в сообщение с помощью `post Message`. Например, предполагая, что `worker` относится к веб-воркеру, вы можете совместно использовать предыдущий `sharedArray` следующим образом:

```
worker.postMessage(sharedArray);
```

Воркер получает совместно используемый массив в качестве свойства `data` объекта `event` для события `message`. Обычно вместо того, чтобы просто отправлять массив, имеет смысл отправить объект с указанием того, о чем идет речь в сообщении, делая массив свойством этого объекта:

```
worker.postMessage({type: "init", sharedArray});
```

Этот код отправляет объект со свойствами `type` (`"init"`) и `sharedArray` (совместно используемый массив). Воркер будет получать этот объект в качестве свойства `data` для события. (Подробнее о том, что на самом деле используется совместно, чуть позже.)

Давайте сведем все это вместе в простой пример. В Листинге 16-1 показан код основного потока, совместно использующий массив `Uint16Array` с потоком веб-воркера.

Листинг 16-1: Базовое применение SharedArrayBuffer (основной поток) — basic-SharedArrayBuffer-main.js

```
const sharedBuf = new SharedArrayBuffer(5 * Uint16Array.BYTES_PER_ELEMENT);
const sharedArray = new Uint16Array(sharedBuf);
const worker = new Worker("./basic-SharedArrayBuffer-worker.js");
let counter = 0;
console.log("initial: " + formatArray(sharedArray));
worker.addEventListener("message", e => {
    if (e.data && e.data.type === "ping") {
        console.log("updated: " + formatArray(sharedArray));
        if (++counter < 10) {
            worker.postMessage({type: "pong"});
        } else {
            console.log("done");
        }
    }
});
worker.postMessage({type: "init", sharedArray});
```

[104] *Записи* часто называются *элементами*, что объясняет, почему свойство получило имя `BYTES_PER_ELEMENT`. Я использую слово *записи* в этой книге, чтобы избежать путаницы с элементами DOM.

```
function formatArray(array) {
    return Array.from(
        array,
        b => b.toString(16).toUpperCase().padStart(4, "0")
    ).join(" ");
}
```

В Листинге 16-2 показан код потока воркера.

Листинг 16-2: Базовое применение SharedArrayBuffer (воркер) — basic-SharedArrayBuffer-worker.js

```
let shared;
let index;
const updateAndPing = () => {
    ++shared[index];
    index = (index + 1) % shared.length;
    this.postMessage({type: "ping"});
};
this.addEventListener("message", e => {
    if (e.data) {
        switch (e.data.type) {
            case "init":
                shared = e.data.sharedArray;
                index = 0;
                updateAndPing();
                break;
            case "pong":
                updateAndPing();
                break;
        }
    }
});
```

Когда воркер получает сообщение «init», он запоминает общий массив с его переменной `shared`.

В этот момент основной поток и воркер совместно используют память для этого массива (рисунок 16-1).

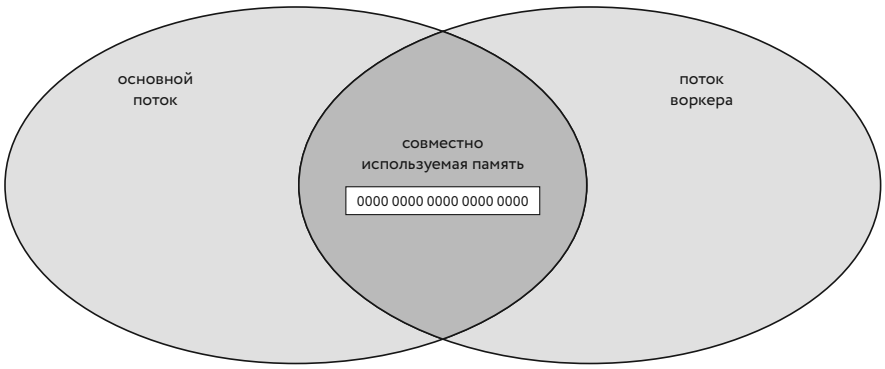


РИСУНОК 16-1

Затем воркер устанавливает свое значение `index` равным 0 и вызывает свою функцию `updateAndPing`. Функция `updateAndPing` приращивает запись в позиции `index` в массиве, приращивает `index` (оборачивание обратно к `0` в конце), а затем отправляет сообщение «ping» в основной поток. Поскольку запись, обновляемая рабочим потоком, находится в совместно используемой памяти, обновление видят и воркер, и основной поток (рисунок 16-2).

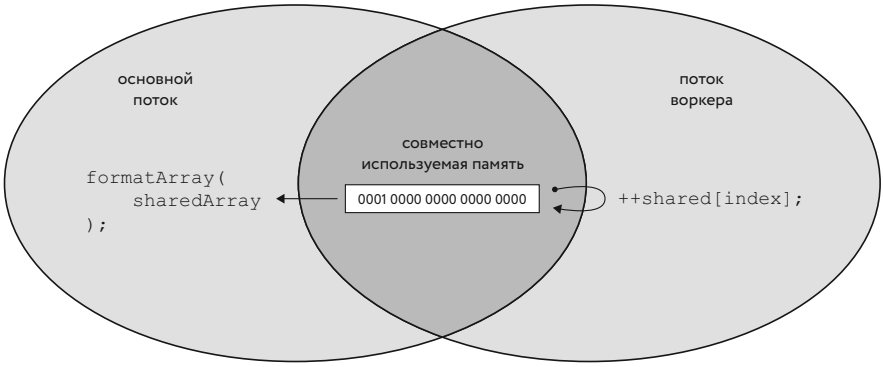


РИСУНОК 16-2

Основной поток отвечает на сообщение «ping», увеличивая свой счетчик и отправляя сообщение «pong». Поток воркера отвечает на «pong», снова вызывая функцию `updateAndPing`. И так далее, пока счетчик основного потока не достигнет 10 и он перестанет отправлять «ping». Каждое сообщение воркеру заставляет его увеличивать следующую запись в общем массиве, завершая ее, когда она достигает конца.

Запустите этот код в своем браузере, используя файлы листингов, и файл **basic-SharedArrayBuffer-example.html** из загрузок. Помните, что файлы необходимо включать с веб-сервера (вы не можете просто открыть файл.html из файловой системы), они должны обслуживаться безопасно или локально и включаться с заголовками, о которых вы узнали в разделе «Поддержка браузера» ранее. При желании можете использовать версию, размещенную на веб-сайте книги, вместо того чтобы запускать свою собственную копию: **https://thenewtoys.dev/bookcode/live/16/basic-SharedArrayBuffer-example.html**. В консоли вы должны увидеть такой вывод от основного потока (при условии, что в вашем браузере доступен буфер `SharedArrayBuffer`):

```
initial: 0000 0000 0000 0000 0000
updated: 0001 0000 0000 0000 0000
updated: 0001 0001 0000 0000 0000
updated: 0001 0001 0001 0000 0000
updated: 0001 0001 0001 0001 0000
updated: 0001 0001 0001 0001 0001
updated: 0002 0001 0001 0001 0001
updated: 0002 0002 0001 0001 0001
updated: 0002 0002 0002 0001 0001
updated: 0002 0002 0002 0002 0001
updated: 0002 0002 0002 0002 0002
done
```

Выходные данные показывают, что основной поток видит обновления, вносимые воркером, в совместно используемой памяти.

Если вы использовали совместно используемую память и несколько потоков в других средах, вам может быть интересно: как мы *узнаем*, что обновления готовы к чтению основным потоком? Что делать, если обновления находятся в кэше для конкретного потока (в виртуальной машине JavaScript или даже в кэше процессора для каждого потока)?

Ответ для этого конкретного примера заключается в том, что `postMessage` определяется как граница синхронизации. Поскольку поток воркера уведомляет основной поток о том, что он выполнил свою работу, используя `postMessage`, а основной поток пытается прочитать результат только тогда, когда получает это сообщение, мы получаем гарантию, что запись завершена до того, как произойдет чтение, и в любых поточных кэшах устаревшее содержимое признано недействительным (так что попытки прочитать их не приведут к считыванию старых значений; вместо этого будет извлечено новое значение). Таким образом, основной поток может безопасно считывать массив, зная, что любые обновления в нем, когда сообщение было опубликовано, будут видны. Этот пример также основан на том факте, что воркер не будет изменять массив снова, пока основной поток не отправит ему «ping», поэтому основному потоку не нужно беспокоиться о чтении незавершенной записи (подробнее об этом чуть позже).

Метод `postMessage` — не только граница синхронизации; далее в этой главе вы узнаете о нескольких границах синхронизации (`Atomics.wait` и `Atomics.notify`).

В примере массив использовал весь буфер `SharedArrayBuffer`, но он мог бы использовать только его часть и дать возможность другому массиву использовать оставшуюся часть буфера для другой цели. Вот пример массива `Uint8Array`, использующий первую половину буфера `SharedArrayBuffer`, и `Uint32Array`, использующий последнюю половину:

```
const sab = new SharedArrayBuffer(24);
const uint8array  = new Uint8Array(sab, 0, 12);
const uint32array = new Uint32Array(sab, 12, 3);
```

Длина буфера составляет 24 байта. Массив `Uint8Array` занимает первые 12 байт для 12 записей длиной в один байт, а массив `Uint32Array` — оставшиеся 12 байт для трех записей длиной в четыре байта.

Предположим, однако, что нам нужно было всего 10 записей для `Uint8Array`, а не 12. Вы можете подумать, что мы могли бы сделать буфер на два байта меньше и просто переместить `Uint32Array` так, чтобы он начинался с индекса 10 в буфере. Но это не сработает:

```
const sab = new SharedArrayBuffer(22);
const uint8array  = new Uint8Array(sab, 0, 10);
const uint32array = new Uint32Array(sab, 10, 3);
// => RangeError: начальное смещение Uint32Array должно быть кратно 4
```

Проблема заключается в *выравнивании памяти*. Современные компьютерные процессоры могут считывать и записывать память блоками размером более отдельного байта. Этот процесс намного эффективнее, когда блок выровнен в памяти, то есть когда местоположение блока в памяти равномерно делится на размер блока.

Адрес	Значение
0 × 0000	0 × 08
0 × 0001	0 × 10
0 × 0002	0 × 27
0 × 0003	0 × 16
...	

РИСУНОК 16-3

См. рисунок 16-3, на котором показан ряд байтов. Например, при чтении 16-битного значения гораздо эффективнее считывать его из местоположения 0 × 0000 или 0 × 0002, чем из 0 × 0001, из-за того, как оптимизированы инструкции процессора. По этой причине экземпляры `SharedArrayBuffer` всегда ссылаются на блоки, выровненные с наибольшей степенью детализации центрального процессора (это обрабатывается движком JavaScript), и вы можете создать многобайтовый типизированный массив, используя смещение в буфер, выровненный по типу массива. Таким образом, можно создать массив `Uint8Array` с любым смещением в буфере, но массив `Uint16Array` должен быть со смещением 0, 2, 4 и т. д. Аналогично `Uint32Array` должен применяться со смещением 0, 4, 8 и т. д.

СОВМЕСТНО ИСПОЛЬЗУЕТСЯ ПАМЯТЬ, НО НЕ ОБЪЕКТЫ

Совместное использование памяти касается только *памяти*, используемой совместно. Объекты-оболочки (`Shared ArrayBuffer` и любые типизированные массивы, использующие его) не используются совместно. Например, в предыдущем примере, который передавал массив `Uint16Array` при помощи буфера `SharedArrayBuffer` из основного потока в воркер, объекты `Uint16Array` и `SharedArrayBuffer` не использовались совместно. Вместо этого на получающей стороне были созданы новые объекты `Uint16Array` и `SharedArrayBuffer`, а затем подключены к базовому блоку памяти буфера `SharedArrayBuffer` отправляющего потока. См. рисунок 16-4.

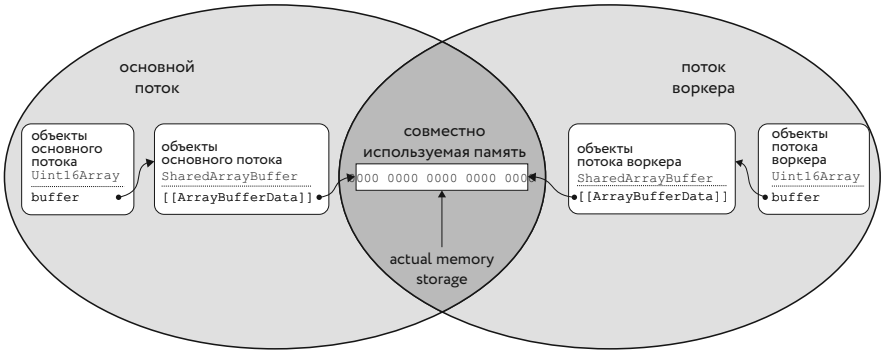


РИСУНОК 16-4

При желании можно легко доказать это себе: добавьте пользовательские свойства к объектам буфера и массива, а затем убедитесь, что вы не видите эти пользовательские свойства в воркере. Изучите Листинги 16-3 и 16-4, которые вы можете запустить с помощью файла **objects-not-shared.html** из загрузок.

Листинг 16-3: Объекты не используются совместно (основной поток) — objects-not-shared-main.js

```
const sharedBuf = new SharedArrayBuffer(5 * Uint16Array.BYTES_PER_ELEMENT);
const sharedArray = new Uint16Array(sharedBuf);
sharedArray[0] = 42;
sharedBuf.foo = "foo";
sharedArray.bar = "bar";
const worker = new Worker("./objects-not-shared-worker.js");
console.log("(main) sharedArray[0] = " + sharedArray[0]);
console.log("(main) sharedArray.buffer.foo = " + sharedArray.buffer.foo);
console.log("(main) sharedArray.bar = " + sharedArray.bar);
worker.postMessage({type: "init", sharedArray});
```

Листинг 16-4: Объекты не используются совместно (воркер) — objects-not-shared-worker.js

```
this.addEventListener("message", e => {
    if (e.data && e.data.type === "init") {
        let {sharedArray} = e.data;
        console.log("(worker) sharedArray[0] = " + sharedArray[0]);
        console.log("(worker) sharedArray.buffer.foo =
                    " + sharedArray.buffer.foo);
        console.log("(worker) sharedArray.bar = " + sharedArray.bar);
    }
});
```

Результат при запуске примера выглядит следующим образом:

```
(main) sharedArray[0] = 42
(main) sharedArray.buffer.foo = foo
(main) sharedArray.bar = bar
(worker) sharedArray[0] = 42
(worker) sharedArray.buffer.foo = undefined
(worker) sharedArray.bar = undefined
```

Вы можете видеть, что, хотя память, используемая буфером, используется совместно, объект буфера и объект массива, использующие ее, не используются совместно.

УСЛОВИЯ ГОНКИ, ВЫШЕДШИЕ ИЗ СТРОЯ ХРАНИЛИЩА, УСТАРЕВШИЕ ЗНАЧЕНИЯ, ЗНАЧЕНИЯ С РАЗРЫВАМИ, ТИРИНГ И МНОГОЕ ДРУГОЕ

И снова: здесь водятся драконы! Мы очень кратко рассмотрим некоторых драконов, но помните, что целые книги написаны о правильном обращении с совместно используемой между потоками памятью.

Давайте рассмотрим один сценарий. Предположим, у вас есть массив `Uint8Array`, использующий буфер `SharedArrayBuffer` для его хранения, и вы установили для первых двух записей определенные значения:

```
const sharedBuf = new SharedArrayBuffer(10);
const sharedArray = new Uint8Array(sharedBuf);
```

```
sharedArray[0] = 100;
sharedArray[1] = 200;
```

Настроив его, вы поделились им с другим потоком. Теперь оба потока запущены, и ради этого примера вы не выполнили никакой синхронизации или координации между ними. В какой-то момент основной поток записывает в эти две записи (сначала индекс 0, затем индекс 1):

```
sharedArray[0] = 110;
sharedArray[1] = 220;
```

В то же время рабочий поток считывает эти две записи в противоположном порядке (сначала индекс 1, затем индекс 0):

```
console.log(`1 is ${sharedArray[1]}`);
console.log(`0 is ${sharedArray[0]}`);
```

Вполне возможно, что поток воркера может вывести:

```
1 is 220
0 is 110
```

Достаточно просто. Воркер прочитал значения после того, как основной поток выполнил свое обновление, и воркер увидел обновленные значения.

В равной степени возможно, что поток воркера может вывести:

```
1 is 200
0 is 100
```

Возможно, воркер прочитал значения непосредственно перед тем, как основной поток выполнил свое обновление. Но также возможно, что воркер прочитал их *после* того, как основной поток выполнил свое обновление, но все еще видел старые значения. Чтобы максимизировать производительность, операционная система, центральный процессор и/или движок JavaScript могут сохранять кэшированную копию небольшой части памяти для каждого потока в течение короткого промежутка времени. (Или не очень короткого.) Поэтому, если вы не обеспечите синхронизацию памяти, может возникнуть такая ситуация, что воркер видит старые значения даже после записи новых значений.

Однако вот действительно сложная проблема — воркер может также вывести следующее:

```
1 is 220
0 is 100
```

Взгляните на этот последний вывод еще раз. Как воркер может видеть значение `220` (обновленное значение в `sharedArray[1]`), но *потом* видеть значение `100` (*исходное* значение в `sharedArray[0]`)?! Основной поток записывает значение в `sharedArray[0]` перед записью нового значения в `sharedArray[1]`, а воркер считывает `sharedArray[1]` перед считыванием `sharedArray[0]`!

Ответ заключается в том, что как чтение, так и запись могут быть переупорядочены компилятором JavaScript или процессором в целях оптимизации. Внутри потока такое изменение порядка никогда не бывает очевидным, но когда вы используете память совместно между потоками, это может стать заметным при неправильной синхронизации потоков.

В зависимости от архитектуры платформы, на которой выполняется ваш код, в дополнение к устаревшим полным значениям или другим подобным проблемам операция считывания может *прерваться*, считав только *часть* незавершенной записи. Предположим, поток записывает запись в массив `Float32Array` (одна запись = четырем байтам). Поток, считывающий эту запись, может считывать часть старого значения (например, первые два байта) и часть нового значения (вторые два байта). Аналогично в ситуации с массивом `Float64Array` (одна запись равна восьми байтам) может случиться так, что будут считаны первые четыре байта старого значения и вторые четыре байта нового значения. Проблема также может возникнуть при считывании многобайтовых значений с помощью `DataView`. Это *не* проблема ни для одного из массивов с целочисленным типом, таких как `Int32Array`: спецификация требует, чтобы эти операции были *без разрывов*. Один из авторов предложения, Ларс Т. Хансен, в ответ на этот вопрос сказал, что эта гарантия присутствует на всех соответствующих аппаратных средствах, на которых, вероятно, будут реализованы движки JavaScript, поэтому предложение включило гарантию в спецификацию. (Движок, реализованный на оборудовании, не предоставляющем гарантию, должен был бы сам это гарантировать.)

Это лишь некоторые запутывания в кажущемся логичным коде, когда задействованы совместно используемая память и несколько потоков.

Решение (если вам нужно записывать данные в одном потоке и считывать в другом) состоит в том, чтобы обеспечить некоторую форму синхронизации между потоками, работающими с одной и той же общей памятью. Ранее вы видели одну форму, специфичную для среды браузера, — `postMessage`. Существует также способ, определенный самим JavaScript — объект `Atomics`.

ОБЪЕКТ ATOMICS

Чтобы справиться с гонками данных, устаревшими считываниями, неупорядоченными записями, разрывами и т. д., JavaScript предоставляет объект `Atomics` («Атомика»). Он предлагает высокоуровневые и низкоуровневые инструменты для работы с совместно используемой памятью согласованным, последовательным и синхронизированным способами. Методы, предоставляемые объектом `Atomics`, налагают порядок на операции, а также гарантируют непрерывность операций чтения, изменения и записи, как показано далее в этой главе. Объект `Atomics` также обеспечивает передачу сигналов между потоками, что гарантирует преимущество синхронизации (подобно тому, которое браузеры предоставляют при помощи `postMessage`).

«АТОМИКА»?

Возможно, вы задаетесь вопросом: «Почему это называется «Атомикой» (`Atomics`)? Это ведь программирование, а не физика. Объект `Atomics`

> назван так потому, что он поддерживает *атомарные операции*: операции, которые, как кажется со стороны, выполняются за один шаг, без возможности для других потоков взаимодействовать с данными операции во время ее выполнения; операция *неделима*. Хотя в физике давно известно, что атомы не являются неделимыми, слово «атом» первоначально означало именно это (от греческого *atomos*, означающего «неразрезанный, необрезанный; неделимый»), поэтому оно было использовано Джоном Далтоном около 1805 года для обозначения того, что он считал неделимым компонентом части природных элементов, таких как железо и кислород.

В примере предыдущего раздела основной поток содержал `sharedArray` со значением `100` в индексе 0 и значением `200` в индексе 1, примерно так:

```
sharedArray[0] = 100;
sharedArray[1] = 200;
```

а потом сделал значения такими:

```
sharedArray[0] = 110;
sharedArray[1] = 220;
```

В том разделе вы узнали, что другие потоки, считывающие эти записи, могут считывать устаревшие значения или даже видеть, что значения записываются не по порядку. Чтобы гарантировать, что эти записи были замечены другими потоками, совместно использующими массив, и просмотрены по порядку, основной поток может использовать метод `Atomics.store` для хранения новых значений. `Atomics.store` принимает массив, индекс записи для записи значения и значение для записи:

```
Atomics.store(sharedArray, 0, 110);
Atomics.store(sharedArray, 1, 220);
```

Аналогично поток воркера будет использовать `Atomics.load` для извлечения значений:

```
console.log(Atomics.load(sharedArray, 1));
console.log(Atomics.load(sharedArray, 0));
```

Теперь, если основной поток выполняет свои записи до того, как воркер выполняет свои считывания, воркер гарантированно увидит обновленные значения. Воркер также гарантированно не увидит обновление индекса 1 до обновления индекса 0. Использование методов `Atomics` гарантирует, что эти операции не будут переупорядочены компилятором JavaScript или процессором. (Однако, если поток воркера считывает значения в другом порядке, он все равно сможет увидеть обновленное значение с индексом 0 и исходное значение с индексом 1, если время было выбрано правильно и он выполнял свои считывания после того, как основной поток осуществил запись в индекс 0, но до того, как основной поток осуществил запись в индекс 1.)

`Atomics` также предлагает методы для обычных операций. Предположим, что у вас есть несколько воркеров, каждый из которых должен периодически увеличивать счетчик в общей памяти. У вас может возникнуть соблазн написать:

```
// Неверно
const old = Atomics.load(sharedArray, index);
Atomics.store(sharedArray, index, old + 1);
```

Проблема с этим кодом заключается в том, что существует разрыв между операциями `load` и `store`. В пределах этого промежутка другой поток может прочитать и записать эту запись (даже при использовании `Atomics`, потому что это происходит между двумя вызовами с его использованием). Если один и тот же код выполняется в двух воркерах, у вас может быть гонка, подобная той, что показана на рисунке 16-5.

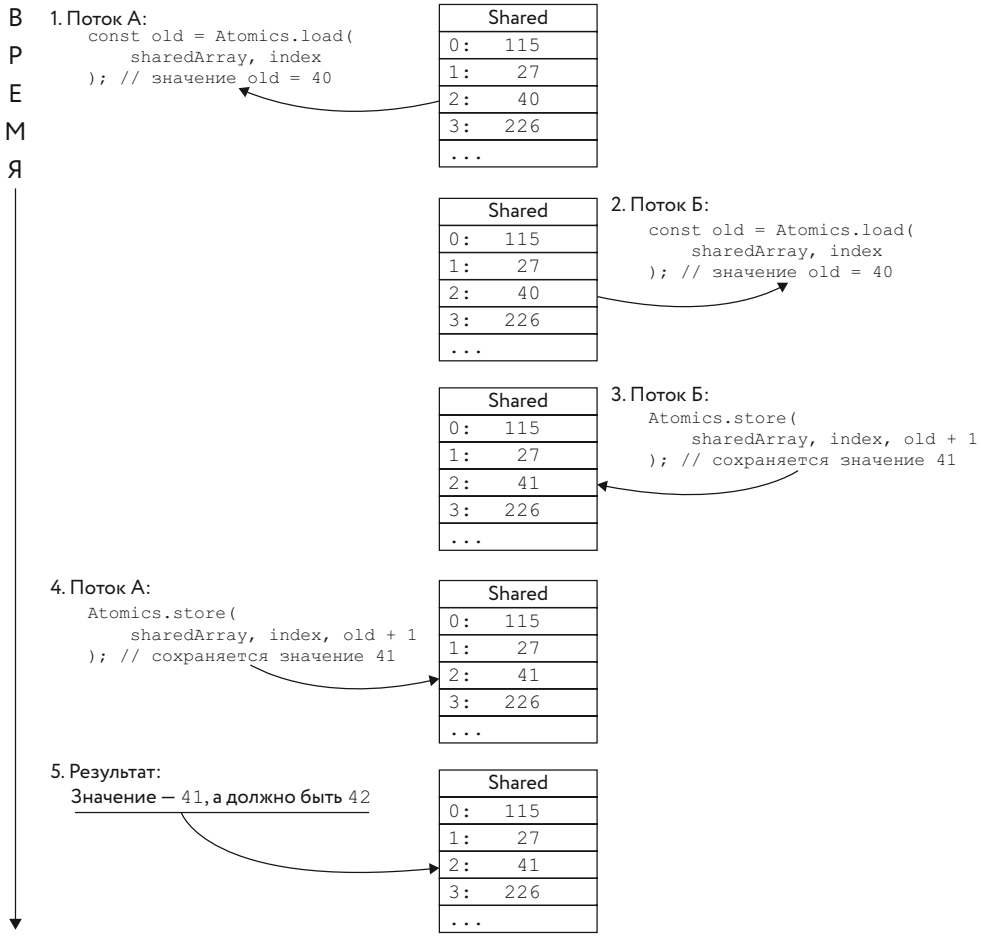


РИСУНОК 16-5

Приращение воркера Б было перезаписано приращением воркера А, в результате чего счетчик получил неправильное значение.

Ответ? Метод `Atomics.add`. Он принимает массив, индекс записи и значение. Считывает значение записи по этому индексу, добавляет к нему заданное значение и сохраняет результат обратно в запись (все как в атомарной операции), и возвращает старое значение записи:

```
oldValue = Atomics.add(sharedArray, index, 1);
```

Другой поток не может обновить значение по заданному индексу после того, как метод `add` считывает старое значение и до того, как `add` записывает новое. Гонка между потоком А и потоком Б, описанная ранее, не может произойти.

`Atomics` предлагает несколько операций, работающих точно так же, как метод `add` — за исключением операции, выполняемой со значением записи: они принимают значение, изменяют значение записи в соответствии с операцией, сохраняют результат, и возвращают старое значение записи (таблица 16-1).

Таблица 16-1

Метод	Операция	Версия Atomic:
`Atomics.add`	Сложение	`array[index] = array[index] + newValue`
`Atomics.sub`	Вычитание	`array[index] = array[index] - newValue`
`Atomics.and`	Битовое «И»	`array[index] = array[index] & newValue`
`Atomics.or`	Битовое «ИЛИ»	`array[index] = array[index] | newValue`
`Atomics.xor`	Битовое Исключающее «ИЛИ»	`array[index] = array[index] ^ newValue`

Низкоуровневые возможности объекта Atomics

Продвинутым алгоритмам с совместным использованием памяти, возможно, потребуется пойти дальше простых методов `load`, `store`, `add` и т. д. Объект `Atomics` включает в себя строительные блоки для этих алгоритмов.

Операция «атомарного обмена» — один из довольно классических строительных блоков многопоточных процессов: атомарная замена одного значения на другое. Эта операция обеспечивается выражением `Atomics.exchange(array, index, value)`: оно считывает значение в индексе (`index`) массива (`array`), записывает полученное значение (`value`) в него и возвращает предыдущее значение. Все это проделывается как атомарная операция.

Операция «атомарного сравнения и обмена» — это еще один классический строительный блок: атомарный обмен одного значения на другое, но только тогда, когда старое значение соответствует ожидаемому значению. Она обеспечивается выражением `Atomics.compareExchange(array, index, expectedValue, replacementValue)`: оно считывает старое значение из записи в массиве и заменяет его значением `replacementValue` *только* в том случае, если оно совпадает

со значением `expectedValue`. Оно возвращает старое значение (независимо от того, совпало оно или нет). Метод `compareExchange` удобен для реализации блокировок:

```
if (Atomics.compareExchange(locksArray, DATA_LOCK_INDEX, 0, 1) === 0) {
    //                                                    ↑  ↑      ↑
    //                                                    |  |      |
    //                Ожидается, что значение будет 0 - /   |      |
    //                       Если да, записать значение 1 - /       |
    //    Проверка, что предыдущее значение было ожидаемым (0) - /
    // Получение блокировки, выполнение работы с данными, защищенными
    // блокировкой
    for (let i = 0; i < dataArray.length; ++i) {
        dataArray[i] *= 2;
    }
    // Высвобождение блокировки
    Atomics.store(locksArray, DATA_LOCK_INDEX, 0);
}
```

В этом примере массив `locksArray` используется в качестве привратника для массива `dataArray`: потоку разрешен доступ/изменение `dataArray` только если ему удастся атомарно *изменить* запись в индексе `DATA_LOCK_INDEX` с `0` на `1`. В этом примере, получив блокировку, код удваивает значение каждой записи в массиве; программист знает, что он может сделать это безопасно из-за блокировки. Завершив обновление массива, код атомарно записывает значение `0` обратно в индекс блокировки, чтобы снять блокировку.

Важный момент: эта блокировка работает только в том случае, если вы убедитесь, что весь код, способный получить доступ к `dataArray`, использует одну и ту же блокировку. Использование метода `compareExchange` таким образом гарантирует, что текущий поток «владеет» общим массивом и что массив обновлен (никаких ожидающих изменений, хранящихся в кэшах CPU для каждого потока, и т. д.).

Зачем вам может понадобиться написать свою собственную подобную блокировку вместо использования одной из функциональных возможностей `Atomics` (например, вместо использования `Atomics.load` и подобных ему в циклах `for`)? Тому есть две причины:

- *Чтобы убедиться, что вы обрабатываете весь массив атомарно.* Метод `Atomics.load` и подобные ему гарантируют, что их отдельная операция обрабатывается атомарно. К другим частям массива может одновременно обращаться другой код, включая даже попытку работать с одной и той же записью почти одновременно.
- *Для повышения эффективности.* При получении доступа к нескольким записям гораздо эффективнее провести блокировку один раз для всей операции, чем заставлять `Atomics` выполнять синхронизацию и блокировку для каждой отдельной операции.

Давайте представим, что код выше использовал только вызовы `Atomics` для изменения массива `dataArray`:

```
for (let i = 0; i < dataArray.length; ++i) {
    let v;
```

```
    do {
        v = Atomics.load(dataArray[i]);
    } while (v !== Atomics.compareExchange(dataArray, i, v, v * 2));
}
```

Это немного отличается от предыдущего кода. С версией `locksArray` весь массив `dataArray` обрабатывается как единое целое; с этой второй версией два потока могут одновременно работать с разными частями массива. (Это может быть полезно или нет, в зависимости от вашего варианта использования, и, если вы специально не собираетесь разрешать этот параллельный доступ, скорее всего, это станет ошибкой, ожидаемо сбивающей вас с толку.)

Эта вторая версия кода также более сложна. Обратите внимание, что каждая итерация должна получить цикл `do-while` при обновлении своей записи, поскольку другой код может изменить значение записи между получением значения и попыткой сохранить обновленное значение. (Вспомните рисунок 16–5: нельзя просто использовать методы `Atomics.load` и `Atomics.store`, потому что потоки могут начать гонку.)

Наконец, эта вторая версия также требует большого количества отдельных низкоуровневых операций блокировки/разблокировки (не менее двух на запись в массиве `dataArray`, больше, если требуется выполнение цикла `do-while`, потому что другой поток также обновляет массив).

Использование Atomics для приостановки и возобновления потоков

Начиная с ES2017 поток можно приостановить в процессе выполнения работы, а затем возобновить для продолжения выполнения этой работы. Большинство сред не позволяет приостанавливать основной поток[105], но позволяют приостанавливать потоки воркеров. Объект `Atomics` предоставляет методы для выполнения таких операций.

Чтобы приостановить поток, вызывается метод `Atomics.wait` для записи в общем массиве `Int32Array`:

```
result = Atomics.wait(theArray, index, expectedValue, timeout);
```

Например:

```
result = Atomics.wait(sharedArray, index, 42, 30000);
```

В этом примере метод `Atomics.wait` считывает значение записи `sharedArray[index]` и, если значение равно `42`, приостанавливает поток. Поток будет приостановлен до тех пор, пока что-то не возобновит его или не наступит тайм-аут (тайм-аут в примере составляет 30000 мс — 30 секунд). Если вы отключите тайм-аут,

[105] Во всяком случае, не через этот механизм. Устаревшие функции, такие как `alert`, `confirm` и `prompt` в браузерах, приостанавливают JavaScript в потоке пользовательского интерфейса до тех пор, пока отображаемое ими модальное диалоговое окно не будет отклонено. Хотя разработчики браузеров постепенно утрачивают абсолютное блокирование, они используются для обеспечения. (Например, Chrome больше не блокирует предупреждение `alert` в фоновой вкладке.)

значение по умолчанию будет равно `Number.Infinity`, то есть вечно ждать возобновления.

Метод `Atomics.wait` возвращает строку с указанием результата:

- `"ok"`, если поток был приостановлен, а затем возобновлен (вместо ожидания тайм-аута);
- `"timed-out"`, если поток был приостановлен и возобновлен по истечению тайм-аута;
- `"not-equal"`, если поток не был остановлен, поскольку значение в массиве не совпало с заданным.

Чтобы возобновить поток, ожидающий этой записи массива, вызывается метод `Atomics.notify`:

```
result = Atomics.notify(theArray, index, numberToResume);
```

Например:

```
result = Atomics.notify(sharedArray, index, 1);
```

Число, которое вы вводите (`1` в примере), — это количество ожидающих возобновления потоков. В примере запрашивается возобновление только одного потока, даже если несколько потоков ожидают этой записи. Метод `Atomics.notify` возвращает количество фактически возобновленных потоков (`0`, если ожидающих не было).

Поток не будет обрабатывать никаких дальнейших заданий из своей очереди заданий, пока он приостановлен. Это происходит из-за семантики выполнения до завершения JavaScript. Поток не может получить следующее задание из своей очереди, пока не завершит текущее задание, и он может этого сделать, если был приостановлен в процессе выполнения работы. Вот почему основной поток в большинстве сред невозможно приостановить. Важно, чтобы он продолжал обрабатывать задания. (Например, основной поток в браузерах представлен потоком пользовательского интерфейса. Приостановка потока пользовательского интерфейса приведет к тому, что пользовательский интерфейс перестанет отвечать на запросы.)

В следующем разделе вы увидите пример приостановки/возобновления потоков.

ПРИМЕР СОВМЕСТНО ИСПОЛЬЗУЕМОЙ ПАМЯТИ

Пример в этом разделе объединяет различные сведения, полученные вами об общей памяти и методах `Atomics` на протяжении всей этой главы. Он реализует очередь производителя/потребителя, упомянутую ранее в разделе «Основы совместно используемой памяти», задачей которой (в этом примере) будет вычисление хэша (MD5, SHA-1 и т. д.) блока памяти. Осуществляется совместное использование памяти между основным потоком, потоком-производителем, заполняющим блок для хэширования и помещающим его в очередь, и несколькими потоками-потребителями, которые берут блоки из очереди и хэшируют их содержимое, отправляя результат в основной поток. (На самом деле существуют две очереди: одна для блоков, ожидающих хэширования, а другая для блоков, доступных для загрузки с новыми данными.) Используются критические секции,

защищенные блокировками; условные переменные для сигнализации, когда условия становятся истинными; метод `Atomics.wait` для приостановки потоков, ожидающих, пока условие станет истинным; и метод `Atomics.notify` для уведомления (возобновления) потоков, когда условие становится истинным.

Давайте начнем с многоразовых классов `Lock` и `Condition`; см. Листинг 16-5.

Листинг 16-5: Классы Lock и Condition — lock-and-condition.js

```
// Lock и Condition навеяны работой Ларса Т. Хансена в этом репозитории
github:
// https://github.com/lars-t-hansen/js-lock-and-condition
// API у них не такой, как у тех, что уже есть, но внутренняя работа
// очень похожа. Но если вы обнаружите ошибку, считайте, что она моя,
// а не Хансена.

/**
 * Служебная функция, используемая `Lock` и `Condition` для проверки
 * аргументов их конструкторов и определенных методов, получающих аргументы,
 * принимаемые этой функцией.
 *
 * @param   {SharedArrayBuffer} sab         Буфер для проверки.
 * @param   {number}            byteOffset  Смещение для проверки.
 * @param   {number}            bytesNeeded Количество байтов, необходимых
 *                                           для этого смещения.
 * @throws  {Error} Если какое-либо из требований не будет выполнено.
 */
function checkArguments(sab, byteOffset, bytesNeeded) {
    if (!(sab instanceof SharedArrayBuffer)) {
        throw new Error("`sab` must be a SharedArrayBuffer");
    }
    // Смещение должно быть целым числом, определяющим позицию в буфере,
    // кратным четырем, потому что мы используем Int32Array, чтобы получить
    // возможность использовать `Atomics.compareExchange`. Он должен
    // содержать по крайней мере заданное количество байтов, доступных
    // в этой позиции.
    if ((byteOffset|0) !== byteOffset
        || byteOffset % 4 !== 0
        || byteOffset < 0
        || byteOffset + bytesNeeded > sab.byteLength
    ) {
        throw new Error(
            ```byteOffset`` must be an integer divisible by 4 and identify a ` +
            `buffer location with ${bytesNeeded} of room`
        );
    }
}

// Значения состояния, используемые `Lock` и `Condition`; они ДОЛЖНЫ быть
// значениями 0, 1 и 2, как показано ниже, иначе реализация `unlock`
// завершится неудачей (имена приведены для наглядности чтения):
const UNLOCKED = 0;     // Блокировка снята, доступен для получения.
const LOCKED = 1;       // Блокировка активирована, нет ожидающих потоков.
const CONTENDED = 2;    // Блокировка заблокирована, и может быть по крайней
                        // мере один поток, ожидающий уведомления о том, что
                        // блокировка была снята.
```
```

```
/** * Класс, реализующий блокировку с использованием заданной
 * `SharedArrayBuffer` области.
 * Вы создаете новую блокировку в SAB, используя `new Lock` для
 * инициализации SAB и получаете экземпляр `Lock` для доступа к нему.
 * Чтобы использовать существующий `Lock` в SAB, необходимо использовать
 * `Lock.deserialize` для десериализации объекта, созданного `serialize`,
 * **А НЕ** `new Lock`.
 */
export class Lock {
    // Примечания по реализации:
    //
    // Состояние блокировки - это единственная 32-разрядная запись
    // в SharedArrayBuffer (при помощи Int32Array). Эта запись должна быть
    // инициализирована в известное состояние, прежде чем ее можно будет
    // использовать для блокировки (`Lock.initialize`). После инициализации
    //запись может получить значение `UNLOCKED`, `LOCKED` или `CONTENDED`.
    //
    // Этот класс не делает никаких попыток гарантировать, что код,
    // снимающий блокировку - это код, первым получивший блокировку.
    // Это делается для упрощения и повышения производительности.

/**
 * Создает блокировку в заданном буфере `SharedArrayBuffer` и возвращает
 * экземпляр `Lock, способный использовать блокировку или быть
 * сериализованным для передачи другому коду для использования
 * (через `Lock.deserialize`).
 * @param   {SharedArrayBuffer} sab         Буфер для использования.
 * @param   {number}            byteOffset  Смещение области
 *                                          в используемом буфере. Объект
 *                                          `Lock`использует
 *                                          `Lock.BYTES_NEEDED` байт.
 */
constructor(sab, byteOffset) {
    checkArguments(sab, byteOffset, Lock.BYTES_NEEDED);
    this.sharedState = new Int32Array(sab);
    this.index = byteOffset / 4; // смещение в байтах => Int32Array index
    Atomics.store(this.sharedState, this.index, UNLOCKED);
}

/**
 * Получение `SharedArrayBuffer`, используемого этим экземпляром `Lock`.
 */
get buffer() {
    return this.sharedState.buffer;
}

/**
 * Получение блокировки. Ожидание будет вечным, до успешного результата.
 * Не может быть использован основным потоком большинства сред (включая
 * браузеры), поскольку потоку может понадобиться перейти в состояние
 * ожидания, а основному потоку не разрешено делать это во многих средах.
 */
lock() {
    const {sharedState, index} = this;
    // Попытка получить блокировку, заменив существующее значение `UNLOCKED`
    // на `LOCKED`. `compareExchange` возвращает значение, которое было
    // в заданном индексе до того, как был произведен обмен, независимо
```

```
    // от того, был произведен обмен или нет.
    let c = Atomics.compareExchange(
                sharedState,
                index,
                UNLOCKED,   // Если запись содержит значение `UNLOCKED`,
                LOCKED       // заменить его на `LOCKED`.
    );
    // Если `c` содержит значение `UNLOCKED`, этот поток получает блокировку.
    // Если нет, цикл повторяется до тех пор, пока это не произойдет.
    while (c !== UNLOCKED) {
        // Подождите, если `c` уже содержит значение `CONTENDED или если
        // попытка этого потока заменить `LOCKED` на `CONTENDED` не обнаружила,
        // что в это время была произведена замена значения на `UNLOCKED`.
        const wait =
            c === CONTENDED
            ||
            Atomics.compareExchange(
                sharedState,
                index,
                LOCKED,     // Если запись содержит значение `LOCKED`,
                CONTENDED   // заменить его на `CONTENDED`.
            ) !== UNLOCKED;
        if (wait) {
            // Вводите состояние ожидания только в том случае, если значение
            // на момент начала ожидания этого потока равно `CONTENDED`.
            Atomics.wait(sharedState, index, CONTENDED);
        }
            // Поток попадает сюда одним из трех способов:
            // 1. Он ожидал и получил уведомление, или
            // 2. Он попытался заменить значение `LOCKED` на `CONTENDED`
            // и обнаружил, что там уже находится значение `UNLOCKED`.
            // 3. Он попытался подождать, но значение, которое было там,
            // когда он начал бы ожидать, не было `CONTENDED`.
            // Попытка заменить значение `UNLOCKED` на `CONTENDED`.
            c = Atomics.compareExchange(sharedState, index, UNLOCKED, CONTENDED);
    }
}

    /**
     * Снятие блокировки
     */
    unlock() {
        const {sharedState, index} = this;
        // Вычитание единицы из текущего значения в состоянии и получение
        // старого значения, которое было там. Это преобразует `LOCKED`
        // в `UNLOCKED` или `CONTENDED` в `LOCKED`(или при ошибочном вызове,
        // когда блокировка не заблокирована) преобразует `UNLOCKED` в `-1`.
        const value = Atomics.sub(sharedState, index, 1);
        // Если старое значение было `LOCKED`, все готово: теперь значение
        // `UNLOCKED`.
        if (value !== LOCKED) {
            // Старое значение не было `LOCKED`. Обычно это означает, что
            // значение было `CONTENDED` и один или несколько потоков могут
            // ожидать снятия блокировки. Сделайте так и уведомите до одного
            // потока.
            Atomics.store(sharedState, index, UNLOCKED);
            Atomics.notify(sharedState, index, 1);
            // максимальное количество потоков для уведомления ^
```

```
        }
    }

    /**
     * Сериализует этот объект `Lock` в объект, который можно использовать
     * с `postMessage`.
     * @returns возвращает объект для использования с `postMessage`.
     */
    serialize() {
        return {
            isLockObject: true,
            sharedState: this.sharedState,
            index: this.index
        };
    }

    /**
     * Десериализует объект из `serialize` обратно в пригодный для
     * использования `Lock`.
     * @param {object} obj Сериализованный объект `Lock`
     * @returns возвращает экземпляр `Lock`.
     */
    static deserialize(obj) {
        if (!obj || !obj.isLockObject ||
            !(obj.sharedState instanceof Int32Array) ||
            typeof obj.index !== "number"
            ) {
            throw new Error("`obj` is not a serialized `Lock` object");
        }
        const lock = Object.create(Lock.prototype);
        lock.sharedState = obj.sharedState;
        lock.index = obj.index;
        return lock;
    }
}
Lock.BYTES_NEEDED = Int32Array.BYTES_PER_ELEMENT; // Блокировка использует
                                                  // только одну запись

/**
 * Класс, реализующий условные переменные с использованием заданной
 * блокировки и заданной области`SharedArrayBuffer`
 * (другая область в том же буфере, которую использует `Lock`).
 *
 * Новая условная переменная создается в SAB с помощью `new Condition`,
 * который устанавливает для области SAB начальные значения и возвращает
 * объект `Condition`,его можно использовать для доступа к условной
 * переменной. Другой код может использовать условную переменную с помощью
 * `Condition.deserialize` (**А НЕ** `new Condition`)для объекта, созданного
 * с помощью `serialize`,чтобы получить доступ к условной переменной.
 */
export class Condition {
    /**
     * Создает объект `Condition`, использующий заданную `Lock` и состояние
     * информация в заданной области буфера, которую использует `Lock`. Эта
     * область должна быть инициализирована в какой-то точке при помощи
     * `Condition.initialize` ине должна перекрывать области, используемые
     * `Lock` или любымдругим условием `Condition`.
     * @param   {Lock}      lock        Используемая блокировка.
```

```
 * @param   {number}    byteOffset  Смещение области в
 *                                  используемом `Lock` буфере.
 *                                  Будет использовать
 *                                  `Condition.BYTES_NEEDED`байты,
 *                                  начиная со смещения.
 */
constructor(lock, byteOffset, noInit = false) {
    if (!(lock instanceof Lock)) {
        throw new Error("`lock` must be a `Lock` instance");
    }
    const sab = lock.buffer;
    checkArguments(sab, byteOffset, Condition.BYTES_NEEDED);
    this.sharedState = new Int32Array(sab);
    this.index = byteOffset / 4; // смещение в байтах => Int32Array index
    this.lock = lock;
    Atomics.store(this.sharedState, this.index, 0);
}


/**
 * Разблокирует блокировку `Lock` этого условия `Condition`
 * и ожидает получения уведомления об этом условии.
 * Вызывающий код должен содержать блокировку.
 */
wait() {
    const {sharedState, index, lock} = this;
    const sequence = Atomics.load(sharedState, index);
    lock.unlock();
    Atomics.wait(sharedState, index, sequence);
    lock.lock();
}

/**
 * Уведомить один ожидающий поток. Обычно код делает это, когда условие
 * принимает значение "true" (какое бы значение ни имелось для
 * конкретного состояния). Вызывающий код должен содержать блокировку.
 */
notifyOne() {
    const {sharedState, index} = this;
    Atomics.add(sharedState, index, 1);     // Перемещение
                                            // последовательности
                                            // на единицу
    Atomics.notify(sharedState, index, 1);  // 1 = уведомить один поток
}

/**
 * Уведомлять все ожидающие потоки. Обычно код делает это, когда условие
 * принимает значение "true" (какое бы значение ни имелось для
 * конкретного состояния). Вызывающий код должен содержать блокировку.
 */
notifyAll() {
    const {sharedState, index} = this;
    Atomics.add(sharedState, index, 1);     // Перемещение
                                            // последовательности
                                            // на единицу
    Atomics.notify(sharedState, index);     // Отсутствие 3-го аргумента
                                            // = уведомить всепотоки
}
```

```
    /**
     * Сериализует этот объект `Condition` в объект, который можно
     * использовать с `postMessage`.
     * @returns возвращает объект для использования с `postMessage`.
     */
    serialize() {
        return {
            isConditionObject: true,
            sharedState: this.sharedState,
            index: this.index,
            lock: this.lock.serialize()
        };
    }

    /**
     * Десериализует объект из `serialize` обратно
     * в пригодный для использования `Condition`.
     *
     * @param   {object}    obj Сериализованный объект `Condition`
     * @returns возвращает экземпляр `Lock`.
     */
    static deserialize(obj) {
        if (!obj || !obj.isConditionObject ||
        !(obj.sharedState instanceof Int32Array) ||
        typeof obj.index !== "number"
        ) {
        throw new Error("`obj` is not a serialized `Condition` object");
        }
        const condition = Object.create(Condition.prototype);
        condition.sharedState = obj.sharedState;
        condition.index = obj.index;
        condition.lock = Lock.deserialize(obj.lock);
        return condition;
    }
}
Condition.BYTES_NEEDED = Int32Array.BYTES_PER_ELEMENT;
```

Класс `Lock` (блокировка) предоставляет простую базовую блокировку. Блокировка создается внутри буфера `SharedArrayBuffer` с использованием конструктора `Lock`, передавая ему `SharedArrayBuffer` и смещение, при котором создается блокировка:

```
const lock = new Lock(sab, offset);
```

Конструктор `Lock` задает информацию о начальном состоянии для блокировки в `SharedArrayBuffer` и возвращает экземпляр `Lock` для использования. Поскольку блокировка использует экземпляры `Int32Array`, смещение должно быть кратно 4, чтобы убедиться, что записи в массиве выровнены (иначе вы получите ошибку `RangeError`; см. обсуждение выравнивания в разделе «Создание совместно используемой памяти» ранее в этой главе). Класс `Lock` сообщает, сколько байтов буфера `SharedArrayBuffer` ему требуется, при помощи свойства под названием `Lock.BYTES_NEEDED` (на случай, если вы храните другие данные в буфере или вам нужно сделать буфер достаточно большим для блокировки).

Поскольку точка блокировки должна быть совместно используемой между потоками, для совместного использования `Lock` требуется вызвать его метод `serialize` для получения простого объекта. Сделать это можно при помощи метода `postMessage`:

```
worker.postMessage({type: "init", lock: lock.serialize()});
```

В получающем потоке используется `Lock.deserialize` для получения экземпляра `Lock`, используемого в этом потоке:

```
const lock = Lock.deserialize(message.lock);
```

Процессы отправки и получения экземпляра `Lock` будут использовать одни и те же базовые данные в буфере `SharedArrayBuffer`.

Чтобы *получить* блокировку (то есть *стать ее владельцем*) и не дать ее получить любому другому потоку, вызывается метод `lock`:

```
lock.lock();
```

Метод `lock` будет ждать бесконечно долго, пока не получит блокировку. (Более сложный класс `Lock` может обеспечить механизм тайм-аута.) Поскольку он заставляет текущий поток ждать (при помощи `Atomics.wait`), его нельзя вызвать в основном потоке в большинстве сред; вы должны использовать поток воркера. Этот пример будет использовать `lock` в потоках производителя и потребителя в коде, показанном позже.

Как только код становится владельцем блокировки, он может продолжить работу со своим кодом критической секции. После завершения он снимет блокировку при помощи метода `unlock`:

```
lock.unlock();
```

Класс `Condition` (условие) работает аналогичным способом. Класс `Condition` создается при помощи конструктора `Condition`. Получая экземпляр `Lock`, конструктор должен использовать смещение внутри буфера `SharedArrayBuffer`, где он должен хранить информацию о состоянии условия:

```
const myCondition = new Condition(lock, conditionOffset);
```

Обратите внимание, что конструктору класса `Condition` передается экземпляр класса `Lock`. Это связано с тем, что условие должно использовать ту же блокировку, что и другие условия или код, используемые с той же совместно используемой памятью. В примере потока производителя/потребителя у очереди есть единственная блокировка, используемая для основных операций очереди, таких как ввод значения и извлечение значения, а также для условий, использующих эти операции, таких как «в очереди есть место» и «в очереди есть работа».

Как и в случае с `Lock`, класс `Condition` использует массив `Int32Array`, следовательно, предоставляемое вызову конструктора смещение должно быть кратным 4. Сколько места требуется в буфере `SharedArrayBuffer`, можно узнать при помощи свойства `Condition.BYTES_NEEDED`.

Чтобы заставить поток ожидать условия, вызывается метод условия `wait`. Поток должен стать владельцем блокировки, используемой условием при вызове `wait`. Это может показаться удивительным, но имеет смысл, когда вы видите, как используются условия. Например, в общей очереди, такой как используемая в этом примере, операция «put» (помещение значения в очередь) — это критическая секция, поскольку ей необходимо получить доступ к данным очереди и обновить их. Таким образом, поток, выполняющий операцию «put», должен стать владельцем блокировки, прежде чем он сможет выполнить свою работу. Но если очередь заполнена, потоку приходится ждать, пока условие «в очереди есть место» станет истинным. Для этого требуется снять блокировку, дождаться уведомления о том, что условие выполнено, а затем повторно установить блокировку. Метод `wait` класса `Condition` обрабатывает выполнение этой работы соответствующим образом, поэтому должен вызываться только тогда, когда поток уже получил блокировку. Вот упрощенный пример:

```
put(value) {
    this.lock.lock();
    try {
        while (/*...очередь заполнена...*/) {
            this.hasRoomCondition.wait();
        }
        // ...поместить значение в очередь...
    } finally {
        this.lock.unlock();
    }
}
```

(Обратите внимание, что, `this.lock` должна быть той же блокировкой, которую использует `this.hasRoomCondition` — блокировка, защищающая очередь в целом.) Поскольку неизбежно наступает момент, когда условие становится истинным, но ожидающий поток еще не получил блокировку, он не может предположить, что после того, как у него появится блокировка, условие все еще будет истинным. Другой поток может проникнуть в память за это время! Вот почему код `wait` во фрагменте `put` выше расположен в цикле `while`, а не просто в операторе `if`.

Итак, как это условие становится истинным? В случае условия «в очереди есть место» другой поток должен выполнить операцию, удаляющую элемент из очереди, а затем *уведомить* любой ожидающий поток о том, что в очереди снова есть место. Это было бы в методе очереди «take», который выглядел бы примерно так:

```
take() {
    this.lock.lock();
    try {
        // ...убедиться, что очередь не пуста...
        const value = /* ...получение следующего значения из очереди,
        освободив место... */;
        this.hasRoomCondition.notifyOne();
        return value;
    } finally {
        this.lock.unlock();
    }
}
```

Метод `notifyOne` класса `Condition` возобновляет один поток, ожидающий этого условия, если таковое имеется, используя метод `Atomics.notify` (см. Листинг 16-6).

Эти предыдущие фрагменты замалчивают второе условие, используемое очередью — «в очереди есть работа». Оно также управляется методами `put` и `take`, только в обратном направлении (`take` ожидает, `put` уведомляет). Давайте рассмотрим полную реализацию очереди с блокировкой совместно используемой памяти (Листинг 16-6).

Листинг 16-6: Реализация LockingInt32Queue — locking-int32-queue.js

```
// Реализация кольцевого буфера, частично взятая из
// https://www.snellman.net/blog/archive/2016-12-13-ring-buffers/
// Обратите внимание, что он основан на оборачивании переноса Uint32
с 2**32-1 на 0
// при увеличении.

import {Lock, Condition} from "./lock-and-condition.js";

// Индексы верхнего и нижнего индексов в массиве `indexes` из
// `LockingInt32Queue`.
const HEAD = 0;
const TAIL = 1;

/**
 * Проверяет заданную вместимости очереди, выдает ошибку, если она
 * недопустима.
 *
 * @param {number} capacity Размер очереди
 */
function validateCapacity(capacity) {
    const capLog2 = Math.log2(capacity);
    if (capLog2 !== (capLog2-0)) {
        throw new Error(
            "`capacity` must be a power of 2 (2, 4, 8, 16, 32, 64, etc.)"
        );
    }
}

/**
 * Преобразует заданное число в Uint32 (а затем обратно в число), применяя
 * оборачивающее стандартное 32-разрядное целое число без знака.
 * Например, -4294967294 преобразует в 2.
 * @param   {number}    n   Число для преобразования.
 * @returns Возвращает результат.
 */
function toUint32(n) {
    // Оператор сдвига вправо без знака преобразует свои операнды в значения
    // Uint32, таким образом, сдвиг `n` на 0 мест преобразует его в Uint32
    // (а затем обратно в число)
    return n >>> 0;
}

// Служебные функции для очереди LockingInt32
// Это будут полезные приватные методы, когда они будут развиваться дальше.

/**
 * Получает размер заданной очереди (количество неиспользованных записей в ней).
```

```
 * **ДОЛЖЕН** вызываться только из кода, удерживающего блокировку очереди.
 * @param   {LockingInt32Queue} queue   Очередь для получения размера.
 * @returns Возвращает количество неиспользованных записей в очереди.
 */
function size(queue) {
    // Преобразование в Uint32 (кратко) обрабатывает необходимое
    // оборачивание, так что, когда головной индекс равен (например) 1,
    // а хвостовой индекс равен (например) 4294967295, результат равен 2
    // (есть запись в индексе 4294967295% вместимости, а также одна
    // в индексе 0), а не -4294967294.
    return toUint32(queue.indexes[HEAD] - queue.indexes[TAIL]);
}

/**
 * Определяет, заполнена ли данная очередь.
 * **ДОЛЖЕН** вызываться только из кода, удерживающего блокировку очереди.
 *
 * @param   {LockingInt32Queue} queue   Проверяемая очередь.
 * @returns возвращает `true`, если очередь заполнена, `false`, если нет.
 */
function full(queue) {
    return size(queue) === queue.data.length;
}

/**
 * Определяет, заполнена ли данная очередь.
 * **ДОЛЖЕН** вызываться только из кода, удерживающего блокировку очереди.
 *
 * @param   {LockingInt32Queue} queue   Проверяемая очередь.
 * @returns возвращает `true`, если очередь заполнена, `false`, если нет.
 */
function empty(queue) {
    return queue.indexes[HEAD] === queue.indexes[TAIL];
}

/**
 * Проверяет значение, чтобы убедиться, что оно допустимо для очереди.
 * Выдает ошибку, если значение недопустимо.
 *
 * @param {number} v Проверяемое значение.
 */
function checkValue(v) {
    if (typeof v !== "number" || (v|0) !== v) {
        throw new Error(
            "Queue values must be integers between -(2**32) and 2**32-1,
            inclusive"
        );
    }
}
/**
 * Помещает заданное значение в заданную очередь.
 * Вызывающий абонент **ДОЛЖЕН** проверить наличие места в очереди
 * перед вызовом этого метода, при этом очередь заблокирована.
 * **ДОЛЖЕН** вызываться только из кода, удерживающего блокировку очереди.
 *
 * @param   {LockingInt32Queue} queue   Очередь, в которую помещается значение.
 * @param   {number}            value   Помещаемое значение.
 */
```

```
function internalPut(queue, value) {
    queue.data[queue.indexes[HEAD]% queue.data.length] = value;
    ++queue.indexes[HEAD];
}

/**
 * Очередь значений Int32 с соответствующим экземпляром `Lock`, чей метод
 * `put` блокируется до тех пор, пока в очереди не останется места,
 * и чей метод `take` блокируется до тех пор, пока
 * в очереди не появится запись для возврата.
 */
export class LockingInt32Queue {
    /**
     * Получает количество байт, требуемое для реализации этой очереди
     * в пределах буфера `SharedArrayBuffer` для поддержки очереди заданной
     * вместимости.
     *
     * @param   {number}    capacity    Желаемая вместимость очереди.
     */
    static getBytesNeeded(capacity) {
        validateCapacity(capacity);
        const bytesNeeded = Lock.BYTES_NEEDED +
                            (Condition.BYTES_NEEDED * 2) +
                            (Uint32Array.BYTES_PER_ELEMENT * 2) +
                            (Int32Array.BYTES_PER_ELEMENT * capacity);
        return bytesNeeded;
    }

    /**
     * Создает новую очередь с заданной вместимостью, необязательно
     * используя заданный буфер `SharedArrayBuffer` с заданным смещением
     * в байтах (если не задано, создается новый буфер
     * соответствующего размера). Конструктор предназначен **только** для
     * создания новой очереди, а не использования существующей. Для
     * существующей используйте `LockingInt32Queue.deserialize` на объекте,
     * возвращенном методом `serialize` экземпляра очереди.
     *
     * @param   {number}            capacity        Максимальное количество
     *                                               содержащихся в очереди
     *                                               записей.
     * @param   {SharedArrayBuffer} sab             Буфер `SharedArrayBuffer`,
     *                                               в котором должна
     *                                               поддерживаться очередь.
     *                                               Если вы предоставляете
     *                                               этот аргумент,
     *                                               используйте статический
     *                                               метод `getBytesNeeded`,
     *                                               чтобы получить
     *                                               количество требуемых
     *                                               очерели байтов.
     * @param    {number}           byteOffset      Смещение в байтах внутри
     *                                               SAB где должна храниться
     *                                               информация о состоянии
     *                                               очереди.
     * @param    {number[]}         initialEntries  Необязательные записи
     *                                               для предварительного
     *                                               заполнения очереди.
     */
```

```
    constructor(capacity, sab = null, byteOffset = 0, initialEntries = null) {
        const bytesNeeded = LockingInt32Queue.getBytesNeeded(capacity);
        if (sab === null) {
            if (byteOffset !== 0) {
                throw new Error(
                    "`byteOffset` must be omitted or 0 when `sab` is " +
                    "omitted or `null`"
                );
            }
            sab = new SharedArrayBuffer(byteOffset + bytesNeeded);
        }
        // Смещение должно быть целым числом, кратным четырем, определяющим
        // позицию в буфере, потому что мы используем Int32Array, чтобы
        // получить возможность использовать `Atomics.compareExchange`.
        // Он должен содержать по крайней мере заданное количество
        // байтов, доступных в этой позиции.
        if ((byteOffset|0) !== byteOffset
            || byteOffset % 4 !== 0
            || byteOffset < 0
            || byteOffset + bytesNeeded > sab.byteLength
        ) {
            throw new Error(
                `\`byteOffset\` must be an integer divisible by 4 and ` +
                `identify a buffer location with ${bytesNeeded} of room`
        );
    }

    // Создание блокировки и условия
    this.byteOffset = byteOffset;
    let n = byteOffset;
    this.lock = new Lock(sab, n);
    n += Lock.BYTES_NEEDED;
    this.hasWorkCondition = new Condition(this.lock, n);
    n += Condition.BYTES_NEEDED;
    this.hasRoomCondition = new Condition(this.lock, n);
    n += Condition.BYTES_NEEDED;
    // Создание индексов и массивов данных
    this.indexes = new Uint32Array(sab, n, 2);
    Atomics.store(this.indexes, HEAD, 0);
    Atomics.store(this.indexes, TAIL, 0);
    n += Uint32Array.BYTES_PER_ELEMENT * 2;
    this.data = new Int32Array(sab, n, capacity);
    if (initialEntries) {
        if (initialEntries.length > capacity) {
            throw new Error(
                `\`initialEntries\` has ${initialEntries.length} entries, ` +
                `queue only supports ${capacity} entries`
            );
        }
        for (const value of initialEntries) {
            checkValue(value);
            internalPut(this, value);
        }
    }
}
    /**
     * Вместимость этой очереди.
     */
```

```
get capacity() {
    return this.data.length;
}

/**
 * Блокирует очередь, помещает в нее заданное значение и снимает
 * блокировку. Вечно ожидает блокировки (без тайм-аута) и пока в очереди
 * не появится место для новой записи.
 *
 * @param   {number}   value  Помещаемое значение. Должно быть числом,
 *                            представленным 32-разрядным целым числом
 *                            без знака.
 * @returns Возвращает размер очереди сразу после того, как это значение
 *          было помещено (хотя может быть устаревшим в тот момент,
 *          когда вызывающий получает его).
 */
put(value) {
    checkValue(value);
    this.lock.lock();
    try {
        // Если очередь заполнена, дождитесь, когда условие "в очереди
        // есть место"получит истинное значение. Поскольку ожидание
        // снимает и повторно получает блокировку, после ожидания еще
        // раз проверьте, заполнена ли очередь в случае, если другой
        // поток проник в нее и использовал пространство, которое стало
        // доступно до того, как поток смог ее использовать.
        while (full(this)) {
            this.hasRoomCondition.wait();
        }
        internalPut(this, value);
        const rv = size(this);
        // Уведомляет один поток, ожидающий в `take`, если таковой
        // имеется, о том, что в очереди есть
        // доступная на данный момент работа.
        this.hasWorkCondition.notifyOne();
        return rv;
    } finally {
        this.lock.unlock();
    }
}

/**
 * Блокирует очередь, принимает из нее следующее значение, разблокирует
 * очередь и возвращает значение.
 * Вечно ожидает блокировки и появления в очереди * хотя бы одной записи.
 */
take() {
    this.lock.lock();
    try {
        // Если очередь пуста, дождитесь, когда условие "в очереди есть
        // работа"получит истинное значение. Поскольку ожидание снимает
        // и повторно получает блокировку, после ожидания еще раз
        // проверьте, свободна ли очередь в случае, если другой поток
        // поток проник в нее и взял работу, которая была только что
        // добавлена перед тем, как этот поток смог ее использовать.
        while (empty(this)) {
            this.hasWorkCondition.wait();
        }
```

```
            const value = this.data[this.indexes[TAIL]% this.data.length];
            ++this.indexes[TAIL];
            // Уведомить один поток, ожидающий в `put`, если таковой
            // имеется, что сейчас есть место для
            // новой записи в очереди
            this.hasRoomCondition.notifyOne();
            return value;
        } finally {
            this.lock.unlock();
        }
    }

    /**
     * Сериализует этот объект `LockingInt32Queue` в объект, который можно
     * использовать с `postMessage`.
     * @returns возвращает объект для использования с `postMessage`.
     */
    serialize() {
        return {
            isLockingInt32Queue: true,
            lock: this.lock.serialize(),
            hasWorkCondition: this.hasWorkCondition.serialize(),
            hasRoomCondition: this.hasRoomCondition.serialize(),
            indexes: this.indexes,
            data: this.data,
            name: this.name
        };
    }

    /**
     * Десериализует объект из `serialize` обратно в пригодный для
     * использования `Lock`.
     *
     * @param   {object} obj Сериализованный объект `Lock`
     * @returns возвращает экземпляр `Lock`.
     */
    static deserialize(obj) {
        if (!obj || !obj.isLockingInt32Queue ||
        !(obj.indexes instanceof Uint32Array) ||
        !(obj.data instanceof Int32Array)
    ) {
            throw new Error(
                "`obj` is not a serialized `LockingInt32Queue` object"
            );
        }
        const q = Object.create(LockingInt32Queue.prototype);
        q.lock = Lock.deserialize(obj.lock);
        q.hasWorkCondition = Condition.deserialize(obj.hasWorkCondition);
        q.hasRoomCondition = Condition.deserialize(obj.hasRoomCondition);
        q.indexes = obj.indexes;
        q.data = obj.data;
        q.name = obj.name;
        return q;
    }
}
```

`LockingInt32Queue` — простая очередь кольцевого буфера, реализованная в совместно используемой памяти с использованием одной блокировки `Lock` и двух условий `Condition`: «в очереди есть работа» (`hasWorkCondition`) и «в очереди есть место» (`hasRoomCondition`). Для создания и совместного использования он придерживается того же подхода, что и `Lock` и `Condition`, используя конструктор для создания очереди и совместного использования ее с помощью `serialize` и `deserialize`. Поскольку объем занимаемой памяти зависит от того, насколько большой должна быть очередь, предоставляется метод `LockingInt32Queue.getBytesNeeded`, который сообщает, сколько байт требуется для запрошенной вами емкости очереди.

Поскольку очередь может быть создана основным потоком (который не может «ждать»), и основному потоку может потребоваться предварительно заполнить очередь некоторыми значениями, конструктор может дополнительно принять начальные значения для очереди. Это показано позже, когда основной поток в Листинге 16-7 создаст очередь доступных буферов.

Чтобы поместить работу в очередь, поток вызывает метод `put`. Код очереди обрабатывает получение блокировки, ожидает места в очереди, помещает значения в очередь и уведомляет любой поток, который может ожидать наличие работы в очереди (при помощи `hasWorkCondition`). Аналогично, чтобы получить значение из очереди, код вызывает метод `take`, а код очереди обрабатывает блокировку и ожидание, необходимые для получения значения из очереди и уведомления любого возможного ожидающего потока о том, что в очереди теперь есть место (при помощи `hasRoomCondition`).

В Листингах 16-7 и 16-8 есть основная точка входа для этого примера и очень маленький модуль разнообразных утилит. Основная точка входа создает восемь буферов данных и две очереди с емкостью для идентификаторов всех восьми буферов: одна очередь для буферов, готовых к заполнению данными для хэширования (`availableBuffersQueue`), и другая для буферов, которые заполнены и готовы к хэшированию их содержимого (`pendingBuffersQueue`).

Затем создается поток-производитель и четыре потока-потребителя; им отправляются буферы и сериализованные очереди вместе с другой информацией об инициализации. Добавляется прослушиватель сообщений в очереди потребителей, чтобы он мог получать вычисляемые ими хэши. Наконец, через секунду производителю указывается, что необходимо прекратить производство новых значений и сказать потребителям, чтобы они тоже остановились.

Пока не беспокойтесь о флаге `fullspeed`: мы вернемся к этому немного позже.

Листинг 16-7: Пример основного модуля — example-main.js

```
// В этом примере используется один воркер-производитель и несколько
// воркеров-потребителей для вычисления хэшей буферов данных.
// Работа управляется с помощью двух очередей. Все очереди и
// буферы данных содержатся в одном `SharedArrayBuffer`. Производитель
// получает идентификатор доступного буфера из `availableBuffersQueue`,
// заполняет этот буфер случайными данными, а затем добавляет идентификатор
// буфера в `pendingBuffersQueue` для обработки потребителями. Когда
// потребитель берет идентификатор буфера из ожидающей очереди, он вычисляет
// хэш, затем помещает идентификатор буфера обратно в доступную очередь
// и отправляет хэш в основной поток.
```

```
import {log, setLogging} from "./example-misc.js";
import {LockingInt32Queue} from "./locking-int32-queue.js";
const fullspeed = location.search.includes("fullspeed");
setLogging(!fullspeed);

// Вместимость очередей - const capacity = 8;
const capacity = 8;

// Размер каждого буфера данных - const dataBufferLength = 4096;
const dataBufferLength = 4096;

// Количество буферов должно быть не менее вместимости очереди
const dataBufferCount = capacity;

// Размер SAB, который нам понадобится; обратите внимание, что, поскольку
// буферы данных представляют собой байтовые массивы, нет необходимости
// умножать значение на Uint8Array.BYTES_PER_ELEMENT.
const bufferSize = (LockingInt32Queue.getBytesNeeded(capacity) * 2) +
                   (dataBufferLength * dataBufferCount);

// Количество создаваемых потребителей
const consumerCount = 4;

// Количество полученных от потребителей хэшей
let hashesReceived = 0;

// Создание SAB, буферов данных и очередей. Опять же, поскольку буферы
// данных представляют собой байтовые массивы, в этом коде не нужно
// использовать `Uint8Array.BYTES_PER_ELEMENT`.
let byteOffset = 0;
const sab = new SharedArrayBuffer(bufferSize);
const buffers = [];
for (let n = 0; n < dataBufferCount; ++n) {
    buffers[n] = new Uint8Array(sab, byteOffset, dataBufferLength);
    byteOffset += dataBufferLength;
}
const availableBuffersQueue = new LockingInt32Queue(
    capacity, sab, byteOffset, [...buffers.keys()]
    //                          ^-- Изначально доступны все буферы
);
byteOffset += LockingInt32Queue.getBytesNeeded(capacity);
const pendingBuffersQueue = new LockingInt32Queue(
    capacity, sab, byteOffset // Изначально пустой
);

// Обработка сообщения, отправленного потребителем.
function handleConsumerMessage({data}) {
    const type = data && data.type;
    if (type === "hash") {
        const {consumerId, bufferId, hash} = data;
        ++hashesReceived;
        log(
            "main",
            `Hash for buffer ${bufferId} from consumer${consumerId}: ${hash}, ` +
            `${hashesReceived} total hashes received`
        );
    }
}
```

```
// Создаются производитель и потребители, заставьте их начать
const initMessage = {
    type: "init",
    availableBuffersQueue: availableBuffersQueue.serialize(),
    pendingBuffersQueue: pendingBuffersQueue.serialize(),
    buffers,
    fullspeed
};
const producer = new Worker("./example-producer.js", {type: "module"});
producer.postMessage({...initMessage, consumerCount});
const consumers = [];
for (let n = 0; n < consumerCount; ++n) {
    const consumer = consumers[n] =
            new Worker("./example-consumer.js", {type: "module"});
    consumer.postMessage({...initMessage, consumerId: n});
    consumer.addEventListener("message", handleConsumerMessage);
}

// Сообщение производителю прекратить производство новой работы через одну
// секунду
setTimeout(() => {
    producer.postMessage({type: "stop"});
    setLogging(true);
    const spinner = document.querySelector(".spinner-border");
    spinner.classList.remove("spinning");
    spinner.role = "presentation";
    document.getElementById("message").textContent = "Done";
}, 1000);
// Показать, что основной поток не заблокирован
let ticks = 0;
(function tick() {
    const ticker = document.getElementById("ticker");
    if (ticker) {
        ticker.textContent = ++ticks;
        setTimeout(tick, 10);
    }
})();
```

Листинг 16-8: Пример модуля «Разное» — example-misc.js

```
let logging = true;
export function log(id, message) {
    if (logging) {
        console.log(String(id).padEnd(10, " "), message);
    }
}
export function setLogging(flag) {
    logging = flag;
}
```

Поток-производитель находится в Листинге 16–9. Он берет буферы и очереди из своего метода инициализации и запускает цикл, принимая доступный идентификатор буфера из `availableBuffersQueue` (возможно, ожидающий, если потребители отстают), заполняя этот буфер случайными данными (для этого примера), а затем помещая его идентификатор в `pendingBuffersQueue`. Примерно раз в 500 мс он останавливает цикл и использует метод `setTimeout`, чтобы запланировать повторный запуск почти

немедленно. Это делается для того, чтобы поток мог получать любое сообщение, отправленное ему через метод `postMessage`. Когда поток получает сообщение `stop`, он заполняет `pendingBuffersQueue` значениями флага вместо идентификаторов буфера, чтобы сообщить потребителям об остановке.

Листинг 16-9: Пример потока-производителя — example-producer.js

```
import {log, setLogging} from "./example-misc.js";
import {LockingInt32Queue} from "./locking-int32-queue.js";

// Основной флаг для определения того, должен ли этот производитель
// продолжать работать. Устанавливается
// с помощью `actions.init` (вызывается сообщением `init`
// из основного потока), очищается
// сообщением `stop` или при получении идентификатора буфера, равного -1.
let running = false;

// Идентификатор, используемый этим производителем при вызове `log`,
// задается с помощью `actions.init`
let logId = "producer";

// Используемые очереди и буферы, а также количество запущенных
// основных потребителей (устанавливается
// с помощью `actions.init`).
let availableBuffersQueue;
let pendingBuffersQueue;
let buffers;
let consumerCount;
let fullspeed;

// Заполняются буферы до тех пор, пока у флага `running`
// исчезнет значение true или не наступит время
// для кратковременного перехода в цикл событий,
// чтобы получать любые ожидающие сообщения.
function fillBuffers() {
    const yieldAt = Date.now() + 500;
    while (running) {
        log(logId, "Taking available buffer from queue");
        const bufferId = availableBuffersQueue.take();
        const buffer = buffers[bufferId];
        log(logId, `Filling buffer ${bufferId}`);
        for (let n = 0; n < buffer.length; ++n) {
            buffer[n] = Math.floor(Math.random() * 256);
        }
        log(logId, `Putting buffer ${bufferId} into queue`);
        const size = pendingBuffersQueue.put(bufferId);
        if (Date.now() >= yieldAt) {
            log(logId, "Yielding to handle messages");
            setTimeout(fillBuffers, 0);
            break;
        }
    }
}

// Обработка сообщений, осуществление соответствующих действий
const actions = {
```

```
    // Инициализировать производителя на основе данных в сообщении
    init(data) {
        ({consumerCount, buffers, fullspeed} = data);
        setLogging(!fullspeed);
        log(logId, "Running");
        running = true;
        availableBuffersQueue =
            LockingInt32Queue.deserialize(data.availableBuffersQueue);
        pendingBuffersQueue =
            LockingInt32Queue.deserialize(data.pendingBuffersQueue);
        fillBuffers(data);
    },
    // Остановка этого потока-производителя
    stop() {
        if (running) {
            running = false;
            log(logId, "Stopping, queuing stop messages for consumers");
            for (let n = 0; n < consumerCount; ++n) {
                pendingBuffersQueue.put(-1);
            }
            log(logId, "Stopped");
        }
    }
}
self.addEventListener("message", ({data}) => {
    const action = data && data.type && actions[data.type];
    if (action) {
        action(data);
    }
});
```

Код потребителя в Листинге 16-10 очень похож на код производителя. Просто берутся идентификаторы буферов из `pendingBuffersQueue` (возможно, ожидая, пока один из них станет доступным), вычисляется «хэш» (в данном случае простой XOR с произвольной задержкой для имитации более сложной функции хэширования). Идентификатор буфера помещается в `availableBuffersQueue`, и хэш отправляется в основной поток.

Потребитель останавливается, когда видит идентификатор буфера, равный –1.

Листинг 16-10: Пример потока-потребителя — example-consumer.js

```
import {log, setLogging} from "./example-misc.js";
import {LockingInt32Queue} from "./locking-int32-queue.js";

// Основной флаг для определения того, должен ли этот потребитель продолжать
// работать. Устанавливается сообщением `init`, очищается сообщением `stop`
// или при получении идентификатора буфера, равного -1.
let running = false;

// Идентификатор, используемый этим потребителем при вызове `log`,
// задается `init`
let logId;

// Идентификатор этого потребителя, а также используемые очереди
// и буферы (задаются с помощью `init`)
```

```
let consumerId = null;
let availableBuffersQueue;
let pendingBuffersQueue;
let buffers;
let fullspeed;

// Функция "now", которую мы будем использовать для определения
// времени ожидания операций очереди
const now = typeof performance !== "undefined" && performance.now
? performance.now.bind(performance)
: Date.now.bind(Date);

// Массив, используемый для ожидания внутри `calculateHash`, см. ниже
const a = new Int32Array(new SharedArrayBuffer(Int32Array.BYTES_PER_ELEMENT));

// Вычисляется хэш для данного буфера.
function calculateHash(buffer) {
    // Реальное вычисление хэша, такое как SHA-256 или даже MD5, заняло бы
    // гораздо больше времени, чем в коде ниже, поэтому после выполнения
    // базового хеша XOR (который не является надежным хешем, это просто
    // для простоты), этот код ожидает несколько миллисекунд, чтобы
    // предотвратить полную перегрузку основного потока сообщениями.
    // В реальном коде, вероятно, никто не стал бы этого делать, поскольку
    // смысл разгрузки работы на воркеры предназначен для перемещения
    // работы, занимающей довольно много времени вне основного потока.
    const hash = buffer.reduce((acc, val) => acc ^ val, 0);
    if (!fullspeed) {
        Atomics.wait(a, 0, 0, 10);
    }
    return hash;
}

// Обрабатывает буферы до тех пор, пока у флага `running`
// исчезнет значение true или не наступит время
// для кратковременного перехода в цикл событий,
// чтобы получать любые ожидающие сообщения.
function processBuffers() {
    const yieldAt = Date.now() + 500;
    while (running) {
        log(logId, "Getting buffer to process");
        let waitStart = now();
        const bufferId = pendingBuffersQueue.take();
        let elapsed = now() - waitStart;
        log(logId, `Got bufferId ${bufferId} (elapsed: ${elapsed})`);
        if (bufferId === -1) {
            // Это флаг от производителя, который должен остановить этот
            // потребитель
            actions.stop();
            break;
}
        log(logId, `Hashing buffer ${bufferId}`);
        const hash = calculateHash(buffers[bufferId]);
        postMessage({type: "hash", consumerId, bufferId, hash});
        waitStart = now();
        availableBuffersQueue.put(bufferId);
        elapsed = now() - waitStart;
        log(logId, `Done with buffer ${bufferId} (elapsed: ${elapsed})`);
        if (Date.now() >= yieldAt) {
```

```
            log(logId, `Yielding to handle messages`);
            setTimeout(processBuffers, 0);
            break;
            }
        }
}

// Обработка сообщений, осуществление соответствующих действий
const actions = {
    // Инициализация этого потребителя данными из сообщения
    init(data) {
        ({consumerId, buffers, fullspeed} = data);
        setLogging(!fullspeed);
        logId = `consumer${consumerId}`;
        availableBuffersQueue =
            LockingInt32Queue.deserialize(data.availableBuffersQueue);
        pendingBuffersQueue =
            LockingInt32Queue.deserialize(data.pendingBuffersQueue);
        log(logId, "Running");
        running = true;
        processBuffers();
    },
    // Остановка этого потока-потребителя
    stop() {
        if (running) {
            running = false;
            log(logId, "Stopped");
        }
    }
}
self.addEventListener("message", ({data}) => {
    const action = data && data.type && actions[data.type];
    if (action) {
        action(data);
    }
});
```

Используя файл **example.html** из перечня загрузок, запустите пример с открытой консолью, используя последнюю версию Chrome. Вы увидите, что пример выполняется чуть более секунды и вычисляет несколько сотен хэшей буфера, эффективно планируя работу между различными потоками воркеров с помощью функций общей памяти, ожидания и уведомлений JavaScript. Как видно по волчку и счетчику, которые показывает основной поток, все это делается без привязки пользовательского интерфейса браузера (спасибо воркерам).

ЗДЕСЬ ВОДЯТСЯ ДРАКОНЫ! (СНОВА)

Или: Нет, серьезно, вам действительно нужна совместно используемая память?

Стоит повторить: совместное использование памяти между потоками открывает множество проблем синхронизации данных, с которыми программистам, работающим в типичных средах JavaScript, раньше не приходилось сталкиваться.

В большинстве случаев совместное использование памяти не требуется, отчасти благодаря переносимым ресурсам. В Листингах с 16-11 по 16-13 есть альтернативная

реализация одного и того же блока хэширования, как в примере выше, с использованием метода `postMessage`, передачей буферов памяти вместо совместно используемой памяти и методами `Atomics.wait`/`Atomics.notify`. Нет необходимости в блокировках, условных переменных или `LockingInt32Queue` (он не использует эти файлы; он повторно использует файл `example-misc.js`), а код в остальных файлах намного проще и понятнее.

Листинг 16-11: Пример postMessage основного потока — pm-example-main.js

```
// Это версия example-main.js с использованием `postMessage` + передаваемые
// элементы

import {log, setLogging} from "./example-misc.js";

const fullspeed = location.search.includes("fullspeed");
setLogging(!fullspeed);

// Вместимость очередей (которая также является количеством имеющихся у нас
// буферов данных: именно так на самом деле ограничены очереди в этом примере)
const capacity = 8;

// Размер каждого буфера данных - const dataBufferLength = 4096;
const dataBufferLength = 4096;

// Количество буферов должно быть не менее вместимости очереди
const dataBufferCount = capacity;

// Количество создаваемых потребителей
const consumerCount = 4;

// Количество полученных от потребителей хэшей
let hashesReceived = 0;

// Флаг для определения того, запущен ли процесс
// (производителю и потребителям этот флаг больше не нужен,
// они просто отвечают на то, что им посылают)
let running = false;

// Создаются буферы данных и очереди (которые могут быть простыми массивами,
// поскольку только этот поток получает к ним доступ)
const buffers = [];
const availableBuffersQueue = [];
for (let id = 0; id < dataBufferCount; ++id) {
    buffers[id] = new Uint8Array(dataBufferLength);
    availableBuffersQueue.push(id);
}
const pendingBuffersQueue = [];

// Обработка сообщений, осуществление соответствующих действий
const actions = {
    hash(data) {
        // Получение хэша от потребителя
        const {consumerId, bufferId, buffer, hash} = data;
        buffers[bufferId] = buffer;
        availableBuffersQueue.push(bufferId);
        availableConsumersQueue.push(consumerId);
```

```
        ++hashesReceived;
        log(
            "main",
            `Hash for buffer ${bufferId} from consumer${consumerId}: ` +
            `${hash}, ${hashesReceived} total hashes received`
        );
        if (running) {
            sendBufferToProducer();
            sendBufferToConsumer();
        }
    },
    buffer(data) {
        // Получение буфера от производителя
        const {buffer, bufferId} = data;
        buffers[bufferId] = buffer;
        pendingBuffersQueue.push(bufferId);
        sendBufferToProducer();
        sendBufferToConsumer();
    }
};
function handleMessage({data}) {
    const action = data && data.type && actions[data.type];
    if (action) {
        action(data);
    }
}

// Создаются производитель и потребители, заставьте их начать
const initMessage = {type: "init", fullspeed};
const producer = new Worker("./pm-example-producer.js", {type: "module"});
producer.addEventListener("message", handleMessage);
producer.postMessage(initMessage);
const availableConsumersQueue = [];
const consumers = [];
for (let consumerId = 0; consumerId < consumerCount; ++consumerId) {
    const consumer = consumers[consumerId] =
        new Worker("./pm-example-consumer.js", {type: "module"});
    consumer.postMessage({...initMessage, consumerId});
    consumer.addEventListener("message", handleMessage);
    availableConsumersQueue.push(consumerId);
}

// Отправка буфера производителю для заполнения, если мы работаем и есть
// какие-то доступные буферы
function sendBufferToProducer() {
    if (running && availableBuffersQueue.length) {
        const bufferId = availableBuffersQueue.shift();
        const buffer = buffers[bufferId];
        producer.postMessage(
            {type: "fill", buffer, bufferId},
            [buffer.buffer] // Передача базового буфера `ArrayBuffer`
                            // производителю
        );
    }
}

// Отправка буфера потребителю для хэширования, если есть ожидающие буферы
// и доступные потребители
```

```
function sendBufferToConsumer() {
    if (pendingBuffersQueue.length && availableConsumersQueue.length) {
        const bufferId = pendingBuffersQueue.shift();
        const buffer = buffers[bufferId];
        const consumerId = availableConsumersQueue.shift();
        consumers[consumerId].postMessage(
            {type: "hash", buffer, bufferId},
            [buffer.buffer] // Передача базового буфера `ArrayBuffer`
                            // потребителю
        );
    }
}

// Начало выполнения работы
running = true;
while (availableBuffersQueue.length) {
    sendBufferToProducer();
}

// Прекращение производства новой работы через одну секунду.
setTimeout(() => {
    running = false;
    setLogging(true);
    const spinner = document.querySelector(".spinner-border");
    spinner.classList.remove("spinning");
    spinner.role = "presentation";
    document.getElementById("message").textContent = "Done";
}, 1000);

// Показать, что основной поток не заблокирован
let ticks = 0;
(function tick() {
    const ticker = document.getElementById("ticker");
    if (ticker) {
        ticker.textContent = ++ticks;
        setTimeout(tick, 10);
    }
})();
```

Листинг 16-12: Пример postMessage потока-производителя — pm-example-producer.js

```
// Это версия example-producer.js с `postMessage` + передаваемые элементы
import {log, setLogging} from "./example-misc.js";

// Идентификатор, используемый этим производителем при вызове `log`,
// задается с помощью `actions.init`
let logId = "producer";

// Обработка сообщений, осуществление соответствующих действий
const actions = {
    // Инициализировать производителя на основе данных в сообщении
    init(data) {
        const {fullspeed} = data;
        setLogging(!fullspeed);
        log(logId, "Running");
},
```

```
    // Заполнение буфера
    fill(data) {
        const {buffer, bufferId} = data;
        log(logId, `Filling buffer ${bufferId}`);
        for (let n = 0; n < buffer.length; ++n) {
            buffer[n] = Math.floor(Math.random() * 256);
        }
        self.postMessage(
            {type: "buffer", buffer, bufferId},
            [buffer.buffer] // Перенос базового буфера `ArrayBuffer` обратно
                            // в основной поток
        );
    }
}
self.addEventListener("message", ({data}) => {
    const action = data && data.type && actions[data.type];
    if (action) {
        action(data);
    }
});
```

Листинг 16-13: Пример postMessage потока-потребителя — pm-example-consumer.js

```
// Это версия example-consumer.js с `postMessage` + передаваемые элементы

import {log, setLogging} from "./example-misc.js";

// Идентификатор, используемый этим потребителем при вызове `log`,
// задается `init`
let logId;

// Идентификатор этого потребителя и флаг fullspeed
let consumerId = null;
let fullspeed;

// Массив, который мы используем для ожидания внутри `calcluateHash`,
//см. ниже
const a = new Int32Array(new SharedArrayBuffer(Int32Array.BYTES_PER_ELEMENT));

// Вычисляется хэш для данного буфера.
function calculateHash(buffer) {
    // Реальное вычисление хэша, такое как SHA-256 или даже MD5, заняло бы
    // гораздо больше времени, чем в коде ниже, поэтому после выполнения
    // базового хеша XOR (который не является надежным хешем, это просто
    // для простоты) этот код ожидает несколько миллисекунд, чтобы
    // предотвратить полную перегрузку основного потока сообщениями.
    // В реальном коде, вероятно, никто не стал бы этого делать, поскольку
    // смысл разгрузки работы на воркеры предназначен для перемещения
    // работы, занимающей довольно много времени вне основного потока.
    const hash = buffer.reduce((acc, val) => acc ^ val, 0);
    if (!fullspeed) {
        Atomics.wait(a, 0, 0, 10);
    }
    return hash;
}
```

```
// Обработка сообщений, осуществление соответствующих действий
const actions = {
    // Инициализация этого потребителя данными из сообщения
    init(data) {
        ({consumerId, fullspeed} = data);
        setLogging(!fullspeed);
        logId = `consumer${consumerId}`;
        log(logId, "Running");
    },
    // Хэширование заданного буфера
    hash(data) {
        const {buffer, bufferId} = data;
        log(logId, `Hashing buffer ${bufferId}`);
        const hash = calculateHash(buffer);
        self.postMessage(
            {type: "hash", hash, consumerId, buffer, bufferId},
            [buffer.buffer] // Перенос базового буфера `ArrayBuffer` обратно
                            // в основной поток
        );
    }
}
self.addEventListener("message", ({data}) => {
    const action = data && data.type && actions[data.type];
    if (action) {
        action(data);
    }
});
```

Поместите файлы из этих листингов на свой локальный сервер и запустите их через файл pm-example.html тем же способом, которым запускали файл **example.html** ранее.

Как и example.html, запуск файла **pm-example.html** не связывает пользовательский интерфейс браузера во время его работы. Он также вычисляет почти такое же количество хэшей (например, в зависимости от вашего оборудования, в общей сложности около 350 против около 380 для версии с совместно используемой памятью). Теперь попробуйте сделать это, добавив строку запроса ?fullspeed к URL-адресам (например, http://localhost/example.html?fullspeed и http://localhost/pm-example.html?fullspeed). Это отключает большую часть логирования и устраняет искусственную задержку при вычислении хэша потребителем. В этой полноскоростной версии вы, вероятно, увидите большую разницу (версия `postMessage` + передаваемые элементы выполняет примерно 10 тыс. хэшей по сравнению с версией с совместно используемой памятью с результатом примерно 18 тыс., опять же в зависимости от вашего оборудования). Так что в этом случае, да, версия с совместно используемой памятью работает быстрее, хотя обе версии, вероятно, можно было бы оптимизировать еще больше. Вывод таков: не используйте совместно используемую память, пока не будете уверены, что вам действительно нужно ее использовать, потому что многократно увеличивается возможность появления тонких ошибок. Однако, если вам это действительно нужно, надеюсь, эта глава дала вам несколько основных инструментов, чтобы справиться.

ОТ СТАРЫХ ПРИВЫЧЕК К НОВЫМ

Большая часть того, что вы узнали в этой главе, относится к новым возможностям, и вы не могли пользоваться ими раньше и, вероятно, не будете, за исключением очень специфических ситуаций. Но в таких ситуациях есть пара вещей, которые можно сделать.

Используйте совместно используемые блоки вместо многократного обмена большими блоками данных

Старая привычка: Отправка больших блоков данных туда и обратно между потоками.

Новая привычка: Используйте блок данных совместно между потоками с соответствующей синхронизацией/координацией, если вам действительно это нужно (то есть если передаваемых данных недостаточно).

Используйте Atomics.wait и Atomics.notify вместо разделения заданий воркеров для поддержки цикла событий (при необходимости)

Старая привычка: Искусственное разделение работы в воркере на задания, которые он может выполнить, чтобы он мог обработать следующее сообщение в очереди заданий.

Новая привычка: Там, где это уместно, рассмотрите возможность приостановки/возобновления выполнения работы с помощью `Atomics.wait` и `Atomics.notify`.

17

Различные аспекты

СОДЕРЖАНИЕ ГЛАВЫ

- Тип данных BigInt
- Двоичные целочисленные литералы
- Восьмеричные целочисленные литералы, попытка № 2
- Необязательные привязки `catch`
- Новые математические методы
- Оператор возведения в степень
- Оптимизация хвостового вызова
- Оператор нулевого слияния
- Опциональная цепочка
- Другие хитрости синтаксиса и дополнения к стандартной библиотеке
- Приложение Б: Возможности, доступные только для браузера

В этой главе вы узнаете о различных аспектах, которые на самом деле больше нигде в книге не упоминались: тип данных BigInt, новые формы числовых литералов; различные новые математические методы; новый оператор возведения в степень (и «загвоздка» в отношении его приоритета); оптимизация хвостовых вызовов (в том числе, почему нельзя полагаться на него, пока или, возможно, никогда); некоторые другие незначительные изменения; и, наконец, некоторые функции только для браузера, добавленные в Приложение Б, чтобы задокументировать то, что все равно есть.

ТИП ДАННЫХ BIGINT

BigInt[106] — это новый примитивный тип для работы с большими целыми числами в JavaScript, добавленный в ES2020.

Тип BigInt может содержать целое число любого размера, ограниченное только доступной памятью и/или разумным ограничением, налагаемым разработчиками движка

[106] `https://github.com/tc39/proposal-bigint`

JavaScript. (Ограничения разработчиков, вероятно, будут очень и очень высокими. В настоящее время V8 допускает до одного *миллиарда* бит.) Вы хотите сохранить число 1 234 567 890 123 456 789 012 345 678 (целое число с 28 значащими цифрами) в переменной? Обычный числовой тип JavaScript (Number) не позволяет задуматься об этом, во всяком случае, не совсем так; но BigInt покроет эту потребность.

Поскольку BigInt — это новый примитивный тип, `typeof` идентифицирует его как `"bigint"`.

У значений BigInt есть литеральная форма (цифры с суффиксом `n`; подробнее об этом уже скоро) и все обычные математические операторы (`+`, `*` и т. д.) для работы с ними. Например:

```
let a = 1000000000000000000n;
let b = a / 2n;
console.log(b); // 500000000000000000n
let c = b * 3n;
console.log(c); // 1500000000000000000n
```

Стандартные математические *методы* `Math`, такие как `Math.min`, *не* работают с этими значениями, поскольку такие методы определены для типа Number. (Вполне может поступить предложение, предоставляющее некоторые из этих методов для значений BigInt, такие как `min`, `max`, `sign` и т. п. Другие, подобные `sin` и `cos`, не имеют большого смысла для целочисленного типа.) Зачем же использовать тип данных BigInt? Существуют два основных варианта применения:

- Название как бы говорит само за себя: вы имеете дело с большими целыми числами, которые могут быть за пределами возможностей точного представления типа Number (то есть с числами больше 2^{53}).
- Вы имеете дело с финансовой информацией. Неточность типа чисел с плавающей точкой (знаменитая проблема `0.1 + 0.2 !== 0.3`) делает его использование неудачным выбором для финансовых задач — и все же, корзины покупок, работающие на наивном коде JavaScript, в любом случае часто используют такой тип данных. Вместо этого можно использовать BigInt: вы работаете с целыми числами, используя наименьшую единицу вашей валюты (или даже меньшую в некоторых случаях). Например, в США можно использовать 1 доллар = `100n`, то есть работать в центах, а не в долларах. (Хотя для некоторых целей может потребоваться $1 = `10000n` — сотых долей центов. Такое решение не было бы необычным, откладывая любое округление до центов до получения окончательного результата вычисления.) Учитывая сказанное, стоит отметить, существует десятичный тип[107], находящийся на этапе 1 («Предложение»), который может подойти для финансовых данных. Неясно, будет ли это предложение продвигаться, но, учитывая прецедент десятичных типов в базах данных и других языках, таких как C # и Java, я бы не ставил против него.

Давайте рассмотрим тип BigInt немного подробнее.

[107] `https://github.com/tc39-transfer/proposal-decimal`

Создание значения типа BigInt

Самый простой способ создать значение BigInt — использовать его буквенную нотацию, похожую на буквенную нотацию Number для целых чисел (почти) с буквой `n` в конце:

```
let i = 12345678901234567890123456789n;
console.log(i);          // 12345678901234567890123456789n
```

Он поддерживает десятичные, шестнадцатеричные и современные восьмеричные (не устаревшие!) системы счисления:

```
console.log(10n);   // 10n
console.log(0x10n); // 16n
console.log(0o10n); // 8n
```

Научное представление чисел (в частности, e-нотации, как `1e3` для `1000`) не поддерживается; нельзя написать `1e9n` вместо `1000000000n`. Это связано прежде всего с тем, что другие языки с аналогичной нотацией не поддерживают такой формат, хотя в этих языках также нет суффикса `n`, поэтому причины не поддерживать его могут быть не совсем применимы. Поддержка e-нотации может быть добавлена в последующем предложении, если это оправдывает варианты использования.

Вы также можете создать BigInt с помощью функции `BigInt`, передав ей либо строку, либо число:

```
let i;
// Вызов BigInt со строкой:
i = BigInt("12345678901234567890123456789");
console.log(i);          // 12345678901234567890123456789n
// Вызов BigInt с числом:
i = BigInt(9007199254740992);
console.log(i);          // 9007199254740992n
```

Возможно, вам интересно, почему я использовал 28-значное число в этом примере при вызове `BigInt` со строковым значением, но гораздо меньшее число при вызове `BigInt` с числовым значением. Можете ли вы придумать причину, по которой я это сделал?

Верно! Весь смысл BigInt заключается в том, что этот тип может надежно хранить гораздо большее число, чем тип Number. Хотя тип Number может содержать числа той же величины, что и внушительное число в примере, он не может обрабатывать их с какой-либо точностью. При попытке получить это число в виде числа фактически вы получите значение, превышающее 61 *миллиард*:

```
// Не делайте так
let i = BigInt(123456789012345678901234567 8);
console.log(i);          // 12345678901234568502454517 76n ?!??!!
```

Причина в том, что числовой литерал `123456789012345678901234567 8` определяет тип Number. К тому моменту, когда значение будет передано в функцию `BigInt`, оно уже утратит точность — оно *уже* станет числом 1 234 567 890 123 456 850 245 451 776, а не 1 234 567 890 123 456 789 012 345 678, еще до того, как функция `BigInt` увидит его. Вот для чего существует BigInt.

Явное и неявное преобразование

Ни BigInt, ни Number никогда не преобразуются друг в друга неявно. Их нельзя смешивать в качестве операндов с математическими операторами:

```
console.log(1n + 1);
// => TypeError: Нельзя смешивать BigInt и другие типы,
// используйте явные преобразования
```

Это происходит в первую очередь из-за потери точности: Тип Number не может обрабатывать большие целые числа, которые обрабатывает тип BigInt, а BigInt не может обрабатывать дробные значения, обрабатываемые Number. Таким образом, вместо того чтобы предоставлять программистам сложный набор правил о том, что происходит при смешивании типов в одном и том же вычислении, JavaScript просто не позволяет смешивать их. Это также должно оставить открытой дверь для добавления обобщенных *типов значений* в JavaScript в будущем.

Допускается явное преобразование. Преобразовать значение из типа Number в тип BigInt явно можно с использованием функции `BigInt`, как показано ранее. Если значение типа Number содержит дробную часть, `BigInt` выдает ошибку:

```
console.log(BigInt(1.7));
// => RangeError: Число 1.7 невозможно преобразовать в BigInt
//    потому что это не целое число
```

Преобразование типов BigInt в Number происходит при помощи функции `Number`. Если Number не может удерживать точное значение, потому что оно слишком велико, то выбирается ближайшее значение, которое может содержать этот тип данных, — что приводит к потере точности, как это часто бывает с типом Number. Если требуется преобразование без потери точности (вместо этого выбрасывающее ошибку или возвращающее значение `NaN`), вы можете написать для него служебную функцию:

```
function toNumber(b) {
    const n = Number(b);
    if (BigInt(n) !== b) {
        const msg = typeof b === "bigint"
            ? `Can't convert BigInt ${b}n to Number, loss of precision`
            : `toNumber expects a BigInt`;
        throw new Error(msg);
        // (или вернет NaN, в зависимости от ваших потребностей)
    }
    return n;
}
```

В отличие от строк, значения BigInt невозможно преобразовать в число с использованием унарного минуса или унарного плюса:

```
console.log(+"20");
// => 20
console.log(+20n);
// => TypeError: Не удается преобразовать значение BigInt в число
```

Это может показаться удивительным на первый взгляд, так до сих пор `+n` и `Number(n)` всегда были одинаковыми (при условии, что `Number` не была затенена, и кроме вызова функции), но это, оказывается, важно для asm.js[108] (asm.js — строгое подмножество JavaScript, разработанное, чтобы быть чрезвычайно оптимизируемым.). Хотя вы можете написать asm.js код вручную, его основная цель — быть результатом компиляторов, принимающих другие языки (C, C++, Java, Python и т. д.) в качестве входных данных. Не нарушая предположений, сделанных asm.js, это было важным конструктивным соображением для BigInt.

Значение BigInt может быть неявно преобразовано в строку обычными способами:

```
console.log("$" + 2n); // $2
```

Значения типа BigInt также поддерживают `toString` и `toLocaleString`.

Значения BigInt могут быть неявно преобразованы в логические значения: `0n` — лжеподобное значение, все остальные значения BigInt — истинноподобные.

Наконец, есть только один вид ноля с типом данных BigInt (не ноль и «отрицательный ноль», как у типа Number), значения BigInt всегда конечны (поэтому в них не бывает значений `Infinity` или `-Infinity`), и у них всегда есть числовое значение (не бывает значения `NaN`).

Производительность

Поскольку у значений типа BigInt нет фиксированного размера, как у 32-разрядного или 64-разрядного целочисленных типов (вместо этого он может быть настолько большим, насколько это необходимо), производительность BigInt не постоянна, как у Number. В общем, чем больше значение BigInt, тем больше времени потребуется для операций с ним, хотя, конечно, это зависит от особенностей реализации. Кроме того, как и большинство новых функций, производительность BigInt, несомненно, со временем улучшится, поскольку разработчики движка JavaScript собирают информацию о его использовании и нацеливают свои оптимизации на то, что обеспечит реальные преимущества.

Массивы BigInt64Array и BigUint64Array

Ряду приложений требуются целые числа, которые больше, чем может вместить тип Number, но которые могут храниться в 64-разрядном целочисленном типе. По этой причине предложение BigInt предоставляет два дополнительных типизированных массива — `BigInt64Array` и `BigUint64Array`. Это массивы 64-разрядных целых чисел, значения которых при извлечении в коде JavaScript получают тип BigInt (точно так же, как `Int32Array` и `Uint32Array` представляют собой массивы 32-разрядных целых чисел, значения которых при извлечении в коде JavaScript представляют собой тип Number).

Служебные функции

У функции `BigInt` есть два метода. Они предоставляют способ получить значение BigInt, обернутое в заданное количество битов, со знаком (`asIntN`) или без знака (`asUintN`):

[108] `http://asmjs.org/`

```
console.log(BigInt.asIntN(16, -20000n));
// => -20000n
console.log(BigInt.asUintN(16, 20000n));
// => 20000n
console.log(BigInt.asIntN(16, 100000n));
// => -31072n
console.log(BigInt.asUintN(16, 100000n));
   // => 34464n
```

Первый операнд — это количество битов, а второй — BigInt для (потенциально) оборачивания. Обратите внимание, что, поскольку 100 000 не может поместиться в 16-битное целое число, значение оборачивается обычным способом в дополнительном коде.

При оборачивании BigInt в значение со знаком BigInt обрабатывается так, как если бы он записывался в N-битовый тип в дополнительном коде. При оборачивании в значение без знака это похоже на операцию с остатком с 2^n, где n — количество битов.

НОВЫЕ ЦЕЛОЧИСЛЕННЫЕ ЛИТЕРАЛЫ

Двигаемся дальше от BigInt. В ES2015 добавлены две новые формы целочисленных литералов (числовые литералы, не имеющие дробной формы): двоичная и восьмеричная.

Двоичные целочисленные литералы

Двоичный (бинарный) целочисленный литерал — это числовой литерал, записанный в двоичном формате (основание 2; то есть с цифрами `0` и `1`). Такой литерал начинается с `0b` (ноль, за которым следует буква B, без учета регистра); за этим следуют двоичные цифры для обозначения числа:

```
console.log(0b100); // 4 (в десятичной системе)
```

Как и в случае с шестнадцатеричными литералами, в двоичном литерале нет десятичной точки. Это *целочисленные* литералы, а не *числовые*; вы можете использовать их только для записи целых чисел. Тем не менее, как и шестнадцатеричные литералы, результирующее число — стандартный числовой тип JavaScript, представленный числом с плавающей точкой.

Вы можете включить любое количество начальных нулей после `b`; они не имеют значения, но могут быть полезны для выравнивания кода или подчеркивания ширины поля битов, с которым вы работаете. Например, если бы вы определяли флаги, которые должны помещаться в восемь битов (возможно, для перехода в запись массива `Uint8Array`), вы могли бы сделать так:

```
const bitFlags = {
    something:        0b00000001,
    somethingElse:    0b00000010,
    anotherThing:     0b00000100,
    yetAnotherThing:  0b00001000
};
```

Но эти дополнительные нули после `0b` совершенно необязательны. Следующий код определяет точно такие же флаги, просто менее четко (с точки зрения стиля):

```
const bitFlags = {
    something:        0b1,
    somethingElse:    0b10,
    anotherThing:     0b100,
    yetAnotherThing:  0b1000
};
```

Восьмеричные целочисленные литералы, попытка № 2

В ES2015 добавили новую восьмеричную *целочисленную литеральную* форму (основание 8). Она использует префикс `0o` (ноль, за которым следует буква O, без учета регистра), далее следуют восьмеричные цифры (`0` по `7`):

```
console.log(0o10); // 8 (в десятичной системе)
```

Как и в случае с шестнадцатеричными и двоичными литералами, это целочисленные литералы, но они определяют стандартные числа с плавающей точкой.

Возможно, вы думаете: «Подождите, разве в JavaScript уже не было восьмеричной формы?» Вы правы! Раньше считалось, что начальный ноль, за которым следуют восьмеричные цифры, определял число в восьмеричном формате. Например, `06` определяет число шесть, а `011` определяет число девять. Этот формат был проблематичным, потому что такое выражение было легко спутать с десятичными. И еще потому, что при включении в выражение числа `8` или `9` движки JavaScript анализировали число как десятичное, что приводило к путанице, где два выражения, `011` и `09`, определяли число девять:

```
// Только в свободном режиме
console.log(011 === 09); // истина
```

Как запутанно! В спецификации третьего издания (1999) эта устаревшая восьмеричная форма была удалена как часть языка, но оставлена в разделе «совместимость» как нечто, что реализации могли выбирать, поддерживать или нет. Стандарт ES5 (2009) пошел дальше и сказал, что в строгом режиме реализации JavaScript больше не разрешается поддерживать устаревшие восьмеричные литералы, и переместил информацию о «совместимости» в Приложение Б только для браузера. (В ES2015 также запрещены восьмеричные десятичные литералы, такие как `09`.) Отказ от устаревшего формата — одна из многих причин использования строгого режима: `011` и `09` вызывают синтаксические ошибки в строгом режиме, что позволяет избежать путаницы.

Но это все уже история. Теперь, если требуется написать восьмеричное значение, используйте новую форму — `0o11`. Если необходимо записать десятичное значение, просто удалите бесполезные ноли в начале[109] (например, для числа «девять» используйте `9`, а не `09`).

[109] Начальный ноль, за которым следует десятичная точка, например, 0.1 — вполне рабочая форма. Это запрещено только тогда, когда за начальным нулем следует цифра (например, 03 или 01.1).

НОВЫЕ МАТЕМАТИЧЕСКИЕ МЕТОДЫ

В ES2015 добавили целый ряд новых функций к объекту `Math`. В целом они подпадают под следующие категории:

- Общие математические функции часто полезны в различных приложениях, в частности в области графики, геометрии и т. д.
- Функции для поддержки низкоуровневого кода, такие как цифровая обработка сигналов (DSP) и код, скомпилированный в JavaScript с других языков.

Конечно, между категориями есть некоторое совпадение, поскольку они немного произвольны. В этом разделе вы найдете большинство из них, которые перекрываются в категории поддержки низкого уровня.

Общие математические функции

В ES2015 добавлено множество общих математических функций, в первую очередь тригонометрических и логарифмических. Эти функции полезны для обработки графики, 3D-геометрии и т. п. Каждая из них возвращает «зависящее от реализации приближение» своей математической операции. Таблица 17-1, в которой они перечислены в алфавитном порядке с указанием операций, выполняемых каждой из них.

Таблица 17-1. Общие математические функции

Функция	Операция
`Math.acosh(x)`	Обратный гиперболический косинус `x`
`Math.asinh(x)`	Обратный гиперболический синус `x`
`Math.atanh(x)`	Обратный гиперболический тангенс `x`
`Math.cbrt(x)`	Кубический корень из `x`
`Math.cosh(x)`	Гиперболический косинус `x`
`Math.expm1(x)`	Вычитание 1 из экспоненциальной функции `x` (*e* возведено в степень `x`, где *e* — основание натуральных логарифмов)
`Math.hypot(v1, v1, ...)`	Квадратный корень из суммы квадратов его аргументов
`Math.log10(x)`	Логарифм `x` по основанию 10
`Math.log1p(x)`	Натуральный логарифм `1 + x`
`Math.log2(x)`	Логарифм `x` по основанию 2
`Math.sinh(x)`	Гиперболический синус `x`
`Math.tanh(x)`	Гиперболический тангенс `x`

Большинство из этих новых методов обеспечивают базовые тригонометрические и логарифмические операции.

Функции `Math.expm1` и `Math.log1p` на первый взгляд могут показаться странными, поскольку `Math.expm1(x)` логически — то же самое, что и `Math.exp(x) - 1`, а `Math.log1p(x)` логически то же, что и `Math.log(x + 1)`. Но в обоих случаях при значении `x`, стремящемуся к нулю, методы `expm1` и `log1p` могут предоставить более точный результат, чем эквивалентный код, использующий `exp` или `log`, из-за ограничений числового типа JavaScript. Реализация может выполнить вычисление `Math.expm1(x)` с большей точностью, чем поддерживает числовой тип JavaScript, а затем преобразовать конечный результат в число JavaScript (потеряв некоторую точность). Это делается вместо того, чтобы выполнять `Math.exp(x)`, преобразуя результат в число JavaScript (теряя некоторую точность), а затем вычитая из него 1 в пределах точности числового типа. Для этих операций, когда `x` стремится к нулю, это помогает отсрочить потерю точности до момента выполнения части `- 1` или `1 +`.

Поддержка низкоуровневых математических функций

В последние несколько лет наблюдается большой интерес и проведение работ над использованием JavaScript в качестве цели для кросс-компиляции кода с других языков, таких как C и C++. Обычно инструменты, которые делают это, компилируются в asm.js[110], высоко оптимизируемом подмножестве JavaScript. (Они также обычно поддерживают возможность выводить WebAssembly[111] вместо JavaScript или в дополнение к нему.) Например, два проекта, которые способны это реализовать, — это Emscripten[112] (который является серверной частью для компиляторов LLVM) и Cheerp[113] (ранее Duetto). Чтобы выполнять то, что они делают, нацеливаясь на asm.js, эти и другие инструменты должны реализовывать некоторые низкоуровневые операции способами, быстрыми и совместимыми с тем, как они выполняются в языках со встроенными 32-разрядными целыми числами и числами с плавающей точкой. Низкоуровневые операции часто также полезны при сжатии и цифровой обработке сигналов.

В ES2015 добавились некоторые функции для поддержки этих инструментов. Предоставляя для них определенные функции, среди прочего, эти инструменты обеспечивали цель оптимизации для движков JavaScript, иногда позволяя заменить вызов функции одной инструкцией процессора. Функции перечислены в алфавитном порядке в таблице 17-2 (далее).

Функции `Math`, как и все другие, в случае получения входных данных не числового типа, преобразуются в числовой тип перед выполнением операции.

ОПЕРАТОР ВОЗВЕДЕНИЯ В СТЕПЕНЬ (**)

Оператор возведения в степень (`**`) — операторный эквивалент функции `Math.pow`: он возводит число в степень другого числа. Взгляните на пример:

```
console.log(2**8); // 256
```

Будьте в курсе, что тут есть «загвоздка» — использование `x**y` в сочетании с унарным оператором до базы (`x`), например `-2**2`. В математике эквивалентное выражение -2^2 означает $-(2^2)$, что равно -4. Но в JavaScript все привыкли к тому, что унарный

[110] `http://asmjs.org/`
[111] `https://webassembly.org/`
[112] `https://github.com/kripken/emscripten`
[113] `https://leaningtech.com/cheerp/`

оператор – имеет очень высокий приоритет, и многие из нас (большинство?) прочитали бы `-2**2` как `(-2)**2` (что равно `4`), несмотря на его математический вид.

Таблица 17-2. Поддержка низкоуровневых математических функций

Функция	Описание

`Math.clz32(x)`	Конвертирует значение `x` в 32-битное целое число, затем подсчитывает количество нулей перед значением. Например, если вы вызовете `Math.clz32(0b1000)`, результат будет равен 28, потому что при конвертации `0b1000` в 32-битное целое число результат будет содержать 28 нулей перед 1:
	`0b00000000000000000000000000001000`
	`^^^^^^^^^^^^^^^^^^^^^^^^^^^^`
	Это подходит для DSP, сжатия, шифрования и т. д. Функция поддерживается многими языками со встроенным 32-битным целочисленным типом (как C и Java).

`Math.fround(x)`	Возвращает значение, ближайшее к `x`, которое помещается в 32-разрядное двоичное число с плавающей точкой. Напомним, что числа JS представлены двоичными числами с плавающей точкой («binary64» в спецификации IEEE-754, обычно называется «double»). Многие языки поддерживают 32-разрядную версию, «binary32» (обычно называемую «float»). Эта операция обеспечивает операцию `float(x)` для многих языков, преобразующих `double(x)` в ближайшее значение `float`. `fround` преобразует `x` в `binary32`, используя метод округления «округление до четного», затем преобразует обратно `binary64`, если только этот шаг не может быть оптимизирован на основе того, что делается с результатом. («Округление до четного» означает, что, если нет точного совпадения для значения `binary64`, то есть два значения `binary32`, одно численно меньше, а другое численно больше — выбирается четное число.)
`Math.imul(x, y)`	Умножает x и y, используя правила для умножения 32-разрядных целых чисел с дополнительным кодом с помощью оборачивания. Это эквивалентно операции `x * y`, где `x` и `y` — нативные 32-разрядные целые числа.
`Math.sign(x)`	Возвращает знак `x`. Это обычная операция во многих низкоуровневых алгоритмах. Поскольку `x` является числом JavaScript, у функции есть пять возможных результатов (хотя два из них почти всегда обрабатываются одинаково кодом, использующим результат): `NaN`, если `x` равно `Nan`; `-0`, если `x` равно `-0`; `-1`, если `x` отрицательно; `1`, если `x` положительно.
`Math.trunc(x)`	Усекает число до его целочисленной формы простым удалением (без округления) дробной части. Это будет эквивалентом `(int)x` в языке с 32-битным целым в дополнительном коде типом `(int)` — за исключением того, что, поскольку `x` и результат оба являются числами JavaScript, существуют некоторые сложности: `NaN`, положительная бесконечность, отрицательная бесконечность, `+0` и `-0` все приводят сами по себе. Число, большее `0`, но меньшее `1`, приводит к `+0`, а число, меньшее `0`, но большее `-1`, приводит к `-0`. В отличие от методов `Math.floor` и MAth.cail, результатом всегда является целая числовая часть исходного числа, тогда как (например) результат `Math.floor(-14.2)` равен `-15`, потому что метод `Math.floor` всегда округляется в меньшую сторону, даже с отрицательными числами. (Возможно, вы видели код типа: `x = x < 0 ? Math.ceil(x): Math.floor(x);` теперь это может быть просто выражение `n = Math.trunc(x)`.) Аналогично вы видели выражения `x = ~ ~ x` или `x = x | 0` или что-то подобное, но оба они предполагают преобразование `x` в 32-разрядное целое число. `Math.trunc` работает с числами, превышающими размер 32-разрядных целых чисел.

После продолжительных дискуссий и споров[114] комитет TC39 сделал выражение `-2**2` синтаксической ошибкой: следует писать `(-2)**2` или `-(2**2)`, чтобы не допустить неопределенность. Были веские аргументы в пользу использования математического соглашения и в пользу сохранения унарных операторов с более высоким приоритетом, чем оператор возведения в степень. В конце концов, вместо того чтобы выбрать бегуна в этой гонке, комитет TC39 решил: «Сделаем это синтаксической ошибкой: мы можем определить ее позже, если потребуется».

В разных реализациях в этой ситуации используются разные сообщения об ошибках с разной степенью полезности:

- «Unexpected token **» (Неожидаемый токен **);
- «Unparenthesized unary expression can't appear on the left-hand side of '**'» (Непарное унарное выражение не может отображаться в левой части '**');
- «Unary operator used immediately before exponentiation expression. Parenthesis (sic) must be used to disambiguate operator precedence» (Унарный оператор, используемый непосредственно перед выражением возведения в степень. Круглые скобки (так) должны использоваться для устранения неоднозначности приоритета оператора).

Поэтому, если вы обнаружите ошибку, указывающую на то, что на первый взгляд кажется вполне разумным использованием оператора `**`, проверьте, есть ли перед оператором унарный минус или плюс.

ИЗМЕНЕНИЯ В `DATE.PROTOTYPE.TOSTRING`

В ES2018 выражение `Date.prototype.toString` было впервые стандартизировано. Вплоть до ES2017 возвращаемая строка представляла собой «…строковое значение, зависящее от реализации, которое представляет (дату) как дату и время в текущем часовом поясе, используя комфортную, удобочитаемую форму». Но все значимые движки JavaScript в конечном счете были согласованы друг с другом, поэтому TC39 решил задокументировать согласованность. Согласно спецификации, теперь метод надежно отображает:

- трехбуквенная аббревиатура дня недели на английском языке (например, «Fri»);
- трехбуквенная аббревиатура месяца на английском языке (например, «Jan»);
- день, при необходимости дополненный нулем (например, «05»);
- год (например, «2019»);
- время в 24-часовом формате (например, «19:27:11»);
- часовой пояс в форме GMT, за которым следует +/– и смещение;
- опционально — «определенная реализацией» строка в круглых скобках, указывающая название часового пояса (например, «Pacific Standard Time»).

Каждый из параметров отделен от своего предшественника одним пробелом. Например:

```
console.log(new Date(2018, 11, 25, 8, 30, 15).toString());
// => Tue Dec 252018 08:30:15 GMT-0800 (Pacific Standard Time)
```

[114] https://esdiscuss.org/topic/exponentiation-operator-precedence

```
// или
// => Tue Dec 252018 08:30:15 GMT-0800
```

ИЗМЕНЕНИЯ В FUNCTION.PROTOTYPE.TOSTRING

Последние спецификации JavaScript стандартизировали выражение `Function.prototype.toString`. ES2019 продолжает этот процесс[115], повышая точность возвращаемого результата, используя фактический исходный текст, создавший функцию, при доступности, но не требуя, чтобы хост сохранял исходный текст (который он может выбросить после синтаксического анализа из-за потребления памяти). Связанные функции и прочие, предоставляемые движком или средой JavaScript, возвращают определение функции в следующей форме «нативной функции»:

```
function name(parameters) {[native code]}
```

Функции, определенные непосредственно исходным кодом JavaScript, возвращают либо фактический исходный текст, который их определил (если таковой имеется), либо форму «нативной функции», как показано в предыдущем примере. Это изменение по сравнению с ES2018, в котором говорилось, что движок JavaScript будет предоставлять «…строку, определенную реализацией…», соответствующую определенным критериям.

(Существует также предложение, находящееся на этапе 2[116], которое предоставляет авторам возможность подписаться на «подвергнутую цензуре» форму, где исходный текст функции никогда не сохраняется, и поэтому метод `toString` всегда будет возвращать форму «нативной функции».)

ДОПОЛНЕНИЯ КОНСТРУКТОРА NUMBER

В ES2015 добавили некоторые новые свойства и методы в конструктор `Number`.

«Безопасные» целые числа

Вероятно, вам известно, что числовой тип JavaScript[117] не является абсолютно точным: он не может быть и по-прежнему способен обрабатывать свой огромный диапазон значений, включая дробные значения, используя только 64 бита. Часто вместо точного числа числовой тип содержит очень близкое приближение числа. Например, числовой тип не может идеально хранить 0.1; вместо этого он содержит число, *очень-очень близкое* к 0.1. То же самое касается значений 0.2 и 0.3, ведущих к знаменитому примеру 0.1 + 0.2 != 0.3. Мы склонны думать об этой неточности только в отношении дробных чисел, но, если у вас достаточно большое число, неточность возникает даже в целых числах. Например:

```
console.log(33333333333333333333); // 33333333333333330000
```

---

[115] `https://github.com/tc39/Function-prototype-toString-revision`

[116] `https://github.com/domenic/proposal-function-prototype-tostring-censorship`

[117] Который является реализацией двоичного стандарта с плавающей точкой двойной точности IEEE-754.

Число, которое создается путем синтаксического анализа числового литерала `3333333333333333333` в числовой тип JavaScript, заметно меньше, чем было бы, если бы числа были точными с такой величиной. Но числовой тип не может точно обработать это число. Ему пришлось округлить число до ближайшего числа, которое он может точно обработать, что оказалось на несколько тысяч меньше.

Чтобы объяснить это, в JavaScript есть концепция *безопасного целого числа*. Число становится безопасным целым числом, если это целое число, значение которого больше -2^{53} и меньше 2^{53}. (Значение 53 в этом случае появилось из-за того факта, что числовой тип фактически содержит 53 бита двоичной точности; остальные биты — показатель степени.) В пределах этого диапазона вы можете быть уверены, что:

- Целое число представлено точно в числовом типе.
- Гарантируется, что целое число не будет результатом округления до него другого целого числа благодаря неточности формата с плавающей точкой.

Второе правило очень важно. Например, 2^{53} представлено точно в числовом типе, но числовой тип не может представлять $2^{53} + 1$. Если вы попытаетесь это реализовать, результат будет 2^{53}:

```
const a = 2**53;
console.log(a);   // 9007199254740992 (2**53)
const b = a + 1;
console.log(b);   // 9007199254740992 (2**53) (снова)
```

Поскольку это может быть результатом округления, 2^{53} — не «безопасное» число. Но $2^{53}-1$ *относится* к безопасным, потому что числовой тип не будет округлять до него любое неточное целое число:

```
const a = 2**53 - 1;
console.log(a);   // 9007199254740991
const b = a + 1;
console.log(b);   // 9007199254740992
const c = a + 2;
console.log(c);   // 9007199254740992
const d = a + 3;
console.log(d);   // 9007199254740994
```

Чтобы помочь вам избежать необходимости писать такие непонятные магические числа, как `2**53` и `-(2**53)` в коде, конструктор `Number` получил два свойства и метод. Давайте рассмотрим их.

Константы `Number.MAX_SAFE_INTEGER` и `Number.MIN_SAFE_INTEGER`

`Number.MAX_SAFE_INTEGER` представляет собой число $2^{53} - 1$ — максимальное безопасное целочисленное значение. `Number.MIN_SAFE_INTEGER` представляет собой число $-2^{53} + 1$ — минимальное безопасное целочисленное значение.

Метод Number.isSafeInteger

Number.isSafeInteger — это статический метод, принимающий аргумент и возвращающий значение true, если аргумент имеет относится к числовому типу, является целым числом и находится в диапазоне безопасных целых чисел. Если это не так, метод isSafeInteger возвращает значение false:

```
console.log(Number.isSafeInteger(42));           // истина
console.log(Number.isSafeInteger(2**53-1));      // истина
console.log(Number.isSafeInteger(-(2**53) + 1)); // истина
console.log(Number.isSafeInteger(2**53));        // ложь (не безопасное)
console.log(Number.isSafeInteger(-(2**53)));     // ложь (не безопасное)
console.log(Number.isSafeInteger(13.4));         // ложь (это не целое число)
console.log(Number.isSafeInteger("1"));          // ложь (строка, в не число)
```

При работе с целыми числами такого масштаба, способными раздвигать границы безопасного целых чисел, два удобных правила заключаются в том, что при выполнении операции c = a + b или c = a - b вы можете быть уверены в результате в c, если Number.isSafeInteger(a), Number.isSafeInteger(b) и Number.isSafeInteger(c) *все* возвращают значение true; в противном случае результат может быть неправильным. Недостаточно протестировать только значение c.

Метод Number.isInteger

Number.isInteger — это статический метод, принимающий аргумент и возвращающий значение true, если (как вы уже догадались!) аргумент представлен числовым типом, а также целым числом. Он не пытается привести свой аргумент к числовому типу, поэтому Number.isInteger("1") равно false.

Методы Number.isFinite и Number.isNaN

Методы Number.isFinite и Number.isNaN точно такие же, как их глобальные аналоги isFinite и isNaN — за исключением того, что они не приводят свой аргумент к числовому типу перед выполнением проверки. Вместо этого, если вы передадите им не-число, они вернут значение false.

Метод Number.isFinite определяет, является ли его аргумент числом — и если да, то является ли это число конечным. Он возвращает результат false для значения NaN:

```
const s = "42";
console.log(Number.isFinite(s));       // ложь: это строка, а не число
console.log(isFinite(s));              // истина: глобальная функция
приводит к типу
console.log(Number.isFinite(42));      // истина
console.log(Number.isFinite(Infinity)); // ложь
console.log(Number.isFinite(1 / 0));   // ложь: в JavaScript x / 0 =
                                       // Бесконечность
```

Метод Number.isNaN определяет, является ли его аргумент числом, и если да, то является ли он одним из значений NaN:

```
const s = "foo";
console.log(Number.isNaN(s));   // ложь: это строка, а не число
console.log(isNaN(s));          // истина: глобальная функция приводит к типу
const n1 = 42;
console.log(Number.isNaN(n1));  // ложь
console.log(isNaN(n1));         // ложь
const n2 = NaN;
console.log(Number.isNaN(n2));  // истина
console.log(isNaN(n2));         // истина
```

Методы `Number.parseInt` и `Number.parseFloat`

Это те же функции, что и глобальные `parseInt` и `parseFloat`. (Буквально те же: `Number.parseInt === parseInt` равно `true`.) Они относятся к части продолжающегося движения к снижению зависимости от глобальных стандартов по умолчанию.

Свойство `Number.EPSILON`

`Number.EPSILON` — это свойство данных, значение которого представляет собой разницу между 1 и наименьшим значением, большим 1, которое может быть представлено в виде числового значения JavaScript (это примерно $2.220446049250313080847263336181 6 \times 10^{-16}$). Термин происходит от *машинного эпсилона* — измерения ошибки округления с плавающей точкой в числовом анализе, которая обозначается греческой буквой эпсилон (ε) или жирной римской u.

СИМВОЛ `SYMBOL.ISCONCATSPREADABLE`

Возможно, вы знаете, что метод `concat` для массивов принимает произвольное количество аргументов и создает новый массив с записями из исходного массива плюс аргументы, которые вы ему даете. Если какой-либо из этих аргументов представлен массивом, он «выравнивает» записи массива (один их уровень) в результирующий массив:

```
const a = ["one", "two"];
const b = ["four", "five"];
console.log(a.concat("three", b));
// => ["one", "two", "three", "four", "five"]
```

Первоначально метод `concat` только расширял записи стандартных массивов таким образом. Он не делал этого с псевдомассивом `arguments` или другими массивоподобными объектами, такими как `NodeList` DOM.

Начиная с ES2015 метод `concat` был обновлен таким образом, что он теперь расширяет любой аргумент, относящийся к стандартным массивам (согласно `Array.isArray`), или будет содержать свойство `Symbol.isConcatSpreadable` с истинным значением.

Например, в следующем примере `obj` — это массивоподобный объект, но он не расширяется в результат с помощью метода `concat`; вместо этого объект помещается в результирующий массив:

```
const a = ["one", "two"];
const obj = {
    0: "four",
```

```
    1: "five",
    length: 2
};
console.log(a.concat("three", obj));
// => ["one", "two", "three", {"0": "four", "1": "five", length: 2}]
```

Но если вы добавите к объекту свойство `Symbol.isConcatSpreadable` с истинным значением, `concat` также расширит его записи:

```
const a = ["one", "two"];
const obj = {
    0: "four",
    1: "five",
    length: 2,
    [Symbol.isConcatSpreadable]: true
};
console.log(a.concat("three", obj));
// => ["one", "two", "three", "four", "five"]
```

Наличие его может быть удобно для прототипа массивоподобного класса, не наследующего от `Array`:

```
class Example {
    constructor(...entries) {
        this.length = 0;
        this.add(...entries);
    }
    add(...entries) {
        for (const entry of entries) {
            this[this.length++] = entry;
        }
    }
}
Example.prototype[Symbol.isConcatSpreadable] = true;

const a = ["one", "two"];
const e = new Example("four", "five");
console.log(a.concat("three", e));
// => ["one", "two", "three", "four", "five"]
```

Когда определялся ES2015, ходили разговоры о том, что в объект `NodeList` в DOM и т. п., возможно, добавят это свойство, чтобы они могли стать расширяемым методом `concat`, но в конечном счете этого не произошло (пока).

Ни один объект в стандартной библиотеке JavaScript не получил свойства `Symbol.isConcatSpreadable` по умолчанию. (Даже не массивы, а как описано, метод `concat` проверяет массивы явно.)

РАЗЛИЧНЫЕ ХИТРОСТИ СИНТАКСИСА

В этом разделе вы узнаете о некоторых последних хитростях синтаксиса, некоторые из которых *действительно* полезны.

Оператор нулевого слияния

При работе с необязательными свойствами объекта программисты часто используют удивительно мощный логический оператор ИЛИ (`||`), чтобы задать значение по умолчанию для возможно отсутствующего свойства:

```
const delay = this.settings.delay || 300;
```

Как вы, вероятно, знаете, оператор `||` вычисляет свой левый операнд и, если это значение является истинным, принимает это значение в качестве своего результата; в противном случае он вычисляет правый операнд и принимает это значение в качестве результата. Из-за этого в примере константе `delay` присваивается значение выражения `this.settings.delay`, если это истинное значение, или значение `300`, если `this.settings.delay` содержит ложное значение. Но есть проблема: программист, вероятно, хотел использовать значение `300`, только если свойство не существует или было значение `undefined`, но этот код также будет использовать `300`, если `this.settings.delay` равно `0`, потому что `0` является ложным.

Новый оператор в ES2020[118] исправляет это — «оператор нулевого слияния» (`??`):

```
const delay = this.settings.delay ?? 300;
```

В этом примере значение `delay` устанавливается в значение выражения `this.settings.delay`, если значение выражения не `null` или `undefined`, или `300`, если `this.settings.delay` равно `null` или `undefined`. Утверждение, использующее `??`, похоже на это, использующее условный оператор:

```
// Напомним, что `== null` (свободное равенство) проверяет как `null`,
// так и `undefined`
const delay = this.settings.delay == null ? 300: this.settings.delay;
```

за исключением того, что `this.settings.delay` вычисляется только один раз в выражении «`??`».

Поскольку оператор нулевого слияния работает по короткой схеме так же, как `||`, правый операнд не вычисляется, если левый операнд не `null` или `undefined`. Это означает, что любые побочные эффекты в правом операнде не выполняются, если он не используется. Например:

```
obj.id = obj.id ?? nextId++;
```

В этом коде `nextId` увеличивается только в том случае, если `obj.id` получает значение `null` или `undefined`.

Опциональная цепочка

Приходилось ли вам когда-нибудь писать подобный код?

```
x = some && some.deeply && some.deeply.nested && some.deeply.nested.value;
y = some && some[key] && some[key].prop;
```

[118] https://github.com/tc39/proposal-nullish-coalescing

Или такой?

```
if (x.callback) {
    x.callback();
}
```

С помощью оператора опциональной цепочки[119] в ES2020 можно написать так:

```
x = some?.deeply?.nested?.value;
y = some?.[key]?.prop;
```

и так:

```
x.callback?.();
```

Оператор «`?.`» вычисляет левый операнд и, если это значение оказывает `null` или `undefined`, он выдает результат `undefined`, выполняет остаток цепочки по короткой схеме. В противном случае он дает доступ к свойству и позволяет выполняться цепочке дальше. Во второй форме (`x.callback?.()`), если левый операнд не `null` или `undefined`, вызов продолжает выполнение.

Это удобно, когда свойства могут существовать или не существовать у объекта (или могут существовать со значением `undefined` или `null`), или когда вы получаете объект из API-функции, которая может вернуть `null`, а вы хотите получить свойство из объекта:

```
const x = document.getElementById("#optional")?.value;
```

В этом коде, если элемент не существует, метод `getElementById` возвращает значение `null`, поэтому результатом оператора опциональной цепочки будет значение `undefined`, которое хранится в `x`. Но, если элемент существует, метод `getElementById` возвращает элемент, и `x` присваивается значение свойства `value` элемента.

Аналогично если элемент может существовать или не существовать, и вы хотите вызвать для него метод:

```
document.getElementById("optional")?.addEventListener("click", function() {
    // ...
});
```

вызов функции не выполняется, если элемент не существует. Вот еще несколько примеров:

```
const some = {
    deeply: {
        nested: {
            value: 42,
            func() {
                return "example";
            }
```

[119] https://github.com/tc39/proposal-optional-chaining

```
        },
        nullAtEnd: null
    },
    nullish1: null,
    nullish2: undefined
};
console.log(some?.deeply?.nested?.value);     // 42
console.log(some?.missing?.value);            // undefined, не ошибка
console.log(some?.nullish1?.value);           // undefined, не ошибка, не null
console.log(some?.nullish2?.value);           // undefined, не ошибка
let k = "nested";
console.log(some?.deeply?.[k]?.value);        // 42
k = "nullish1";
console.log(some?.deeply?.[k]?.value);        // undefined, не ошибка, не null
k = "nullish2";
console.log(some?.deeply?.[k]?.value);        // undefined, не ошибка
k = "oops";
console.log(some?.deeply?.[k]?.value);        // undefined, не ошибка
console.log(some?.deeply?.nested?.func?.()); // "example"
console.log(some?.missing?.stuff?.func?.()); // undefined, не ошибка
console.log(some?.deeply?.nullAtEnd?.());     // undefined, не ошибка
console.log(some?.nullish1?.func?.());        // undefined, не ошибка
k = "nullish2";
console.log(some?.[k]?.func?.());             // undefined, не ошибка
```

Обратите внимание, что даже если проверяемое свойство равно `null`, результатом оператора опциональной цепочки будет `undefined`:

```
console.log(some?.nullish1?.value);           // undefined, не ошибка, не null
```

Также обратите внимание, что использование оператора опциональной цепочки только один раз в начале не приводит к переносу «опциональности» на последующие акцессоры к свойствам или вызовы. Например:

```
const obj = {
    foo: {
        val: 42
}
};
console.log(obj?.bar.val); // TypeError: Не удается считать свойство 'val'
                           // для undefined
```

Этот код выведет значение `undefined`, если у `obj` было значение `null` или `undefined`, поскольку оператор опциональной цепочки использовался для защиты от ситуации, когда `obj` принимает значение `null` или `undefined` при помощи `obj?.bar`. Но `obj` — это объект, поэтому вычисляется `obj.bar`. Поскольку свойства `bar` у `obj` нет, результатом становится значение `undefined`.

Поскольку опционная цепочка используется для защиты от этого при осуществлении доступа к `val`, код пытается получить `val` из `undefined` и терпит неудачу, получая ошибку.

Чтобы избежать этой ошибки, стоит использовать опциональную цепочку при доступе к `val`:

```
console.log(obj?.bar?.val);
```

Аналогично с тем же объектом это приведет к сбою:

```
obj?.bar(); // TypeError: obj.bar не является функцией
```

Поскольку после `bar` нет оператора «`?.`», ничто не проверяет его перед попыткой выполнить вызов. Правило таково: оператор «`?`» находится сразу после того, что может стать нулевым.

Необязательные привязки `catch`

Иногда неважно, какая именно ошибка произошла — вам просто нужно знать, сработало что-то или не сработало. В такой ситуации обычно можно увидеть такой код:

```
try {
    theOperation();
} catch (e) {
    doSomethingElse();
}
```

Обратите внимание, что код ни для чего не использует ошибку (`e`).

Начиная с ES2019 можно просто полностью убрать круглые скобки и привязку (`e`):

```
try {
    theOperation();
} catch {
    doSomethingElse();
}
```

Такой код почти повсеместно поддерживается в современных браузерах, только Edge Legacy его не поддерживает (но версия Edge на Chromium поддерживает). Устаревшие браузеры, такие как Internet Explorer, конечно, не поддерживают такой синтаксис.

Разрывы строк Юникода в JSON

Это вряд ли окажет влияние на вашу повседневную работу программиста.

Условно, JSON — это строгое подмножество JavaScript, но фактически это было неверно, пока в ES2019 не было внесено изменение, позволяющее двум символам появляться без экранирования в строковых литералах, хотя ранее должны были быть экранированы: символы Юникода «разделитель строк» (U+2028) и «разделитель абзацев» (U+2029). JSON позволял им быть неэкранированными в строках, но JavaScript этого не делал, что приводило к ненужному усложнению спецификации.

Начиная с ES2019[120] они оба допустимы в строковых литералах JavaScript без необходимости экранирования.

Правильно сформированный JSON из метода JSON.stringify

Этот метод тоже вряд ли появится на вашем радаре, но технически, в некоторых крайних случаях, метод `JSON.stringify` выдает недопустимый текст JSON.

[120] `https://github.com/tc39/proposal-json-superset`

Возможно, вы помните из главы 10, что строка JavaScript — это серия единиц кода UTF-16, которая допускает недопустимые (непарные) суррогаты. Если строковое значение, преобразованное в строку, содержит непарные суррогаты, результирующий JSON будет содержать непарные суррогаты в виде буквенных символов, что делает его недопустимым в формате JSON.

Изменение (в ES2019) заключается просто в том, чтобы выводить непарные суррогаты в виде экранирования Юникода[121]. Так, например, вместо того, чтобы выводить непарный суррогат U+DEAD в качестве буквального символа, движок выводит экранирующую последовательность Юникода `\uDEAD`.

РАЗЛИЧНЫЕ СТАНДАРТНЫЕ БИБЛИОТЕКИ/ГЛОБАЛЬНЫЕ ДОПОЛНЕНИЯ

В этом разделе вы узнаете о различных новых методах, доступных в стандартной библиотеке JavaScript, а также о некоторых изменениях в существующих методах.

Метод Symbol.hasInstance

Вы можете использовать хорошо известный символ `Symbol.hasInstance`, чтобы настроить поведение оператора `instanceof` для данной функции. Как правило, выражение `x instanceof F` проверяет, есть ли прототип `F.prototype` в любом месте цепочки `prototype` элемента `x`. Но если значение `F[Symbol.hasInstance]` представлено функцией, вместо этого вызывается эта функция — и все, что она возвращает, преобразуется в логическое значение и принимается как результат оператора `instanceof`:

```
function FakeDate() {}
Object.defineProperty(FakeDate, Symbol.hasInstance, {
    value(value) {return value instanceof Date;}
});
console.log(new Date() instanceof FakeDate); // истина
```

Это полезно для некоторых объектов, предоставляемых хостом, и для кодовых баз, где прототипы и функции конструктора в основном не используются в пользу функций конструктора, назначающих каждому объекту все его свойства и методы напрямую. Они могут использовать метод `Symbol.hasInstance` для возврата значения `true` для объектов с соответствующими свойствами/методами, даже если у них нет соответствующего прототипа.

Свойство Symbol.unscopables

Этот символ предназначен для поддержки устаревшего кода; вам вряд ли понадобится использовать его в своем коде, если только вы не пишете библиотеку, которая: А) может быть добавлена на сайт с устаревшим кодом и Б) добавляет методы к встроенным прототипам.

[121] https://github.com/gibson042/ecma262-proposal-well-formed-stringify

`Symbol.unscopables` позволяет указать, какие свойства объекта следует не учитывать, когда этот объект используется с помощью инструкции `with`. Современный JavaScript-код, как правило, не использует инструкцию `with` и она запрещена в строгом режиме, но вы знаете, что `with` добавляет *все* свойства объекта в цепочку области видимости внутри блока `with`, даже не перечисляемые и/или унаследованные свойства (например, метод `toString`):

```
const obj = {
    a: 1,
    b: 2
};
with (obj) {
    console.log(a, b, typeof toString); // 1 2 "function"
}
```

Можно указать инструкции `with` пропустить одно или несколько свойств, указав их с истинным значением в объекте в свойстве `Symbol.unscopables`:

```
const obj = {
    a: 1,
    b: 2,
    [Symbol.unscopables]: {
        b: true     // Делает `b` недоступным для просмотра, оставляя ее вне
                    // блоков `with`.
}
};
with (obj) {
    console.log(a, b, typeof toString); // ReferenceError: b не определено
}
```

Зачем такое может понадобиться? Возможно, по той причине, по которой это понадобилось TC39: им потребовалось такое решение при добавлении методов в `Array.prototype` — новые методы вступали в конфликт с существующим кодом, использующим `with`. Например, предположим, что в каком-то устаревшем коде есть такая функция:

```
function findKeys(arrayLikes) {
    var keys = [];
    with (Array.prototype) {
        forEach.call(arrayLikes, function(arrayLike) {
            push.apply(keys, filter.call(arrayLike, function(value) {
                return rexIsKey.test(value);
            }));
        });
    }
    return keys;
}
```

Код использует инструкцию `with`, чтобы поместить `forEach`, `push` и `filter` в область видимости внутри блока: это позволит легко использовать их для массивоподобных объектов, передаваемых в блок. Конструкция сломалась бы, когда в ES2015 был добавлен метод `keys`, потому что идентификатор `keys` внутри блока `with` был бы преобразован в свойство вместо используемой функцией переменной `keys`.

Благодаря `Symbol.unscopables` этот код не был поврежден методом `keys`, потому что свойство `Symbol.unscopables` из `Array.prototype` записывает все методы со строковыми именами, добавленные в `Array.prototype` в ES2015 и далее, включая `keys`; таким образом `keys` пропускается в блоке `with`.

Объект `globalThis`

В JavaScript появился новый глобальный элемент по умолчанию — `globalThis`[122]. Объект `globalThis` содержит то же значение, что `this` в глобальной области видимости. Оно обычно представлено ссылкой на глобальный объект, хотя хост (например, браузер), может определить любое значение в качестве `this` в глобальной области видимости, и, следовательно, в качестве `globalThis`. Если вы привыкли писать код в браузерах, то знаете, что в браузерах уже есть глобальная переменная для этой цели — `window`. (На самом деле их по крайней мере три, но это к делу не относится.) Но до появления `globalThis` не было стандартного глобального элемента за пределами браузеров, и было довольно неудобно обращаться к глобальному объекту в (относительно редких) ситуациях, когда вам нужно было использовать кросс-среду:

- Node.js предоставляет объект `global` (который, как оказалось, не может быть стандартизирован в браузерах, поскольку это нарушило бы некоторый существующий в Интернете код).
- В глобальной области видимости можно использовать `this` (именно так `globalThis` получил свое название), но большая часть кода не выполняется в глобальной области видимости (например, код в модулях, как описано в главе 13).

Объект `globalThis` позволяет легко получить доступ к тому же значению, содержащемуся в `this` в глобальной области видимости, которое почти во всех средах представлено ссылкой на глобальный объект.

Свойство `description` символа

В главе 5 рассказывается, что у символов могут быть описания:

```
const s = Symbol("example");
console.log(s); // Symbol(example)
```

Первоначально была возможность получить доступ к этому описанию только через `toString`. Начиная с ES2019, у символов есть свойство `description` (описание), его можно получить так[123]:

```
const s = Symbol("example");
console.log(s.description); // example
```

[122] `https://github.com/tc39/proposal-global`

[123] `https://github.com/tc39/proposal-Symbol-description`

Метод `String.prototype.matchAll`

Часто, если есть регулярное выражение с глобальным или липким флагом, требуется обработать все совпадения в строке. Можно сделать это с помощью функции `exec` объекта RegExp и цикла:

```
const s = "Testing 1 2 3";
const rex = /\d/g;
let m;
while ((m = rex.exec(s)) !== null) {
    console.log(`"${m[0]}" at ${m.index}, rex.lastIndex: ${rex.lastIndex}`);
}
// => "1" at 8, rex.lastIndex: 9
// => "2" at 10, rex.lastIndex: 11
// => "3" at 12, rex.lastIndex: 13
```

Однако это достаточно многословно и требует изменения свойства `lastIndex` объекта RegExp (хорошо документированная проблема с объектом RegExp). Вместо этого в ES2020 был добавлен метод `String.prototype.matchAll`[124], возвращающий итератор для совпадений и оставляющий свойства объекта RegExp нетронутыми:

```
const rex = /\d/g;
for (const m of "Testing 1 2 3".matchAll(rex)) {
    console.log(`"${m[0]}" at ${m.index}, rex.lastIndex: ${rex.lastIndex}`);
   }
// => "1" at 8, rex.lastIndex: 0
// => "2" at 10, rex.lastIndex: 0
// => "3" at 12, rex.lastIndex: 0
```

В дополнение к тому, что он не содержит свойства `lastIndex`, он более компактен и предоставляет удобный итератор. Это особенно эффективно при деструктуризации, особенно с именованными группами захвата:

```
const s = "Testing a-1, b-2, c-3";
for (const {index, groups: {type, num}} of s.matchAll(/(?<type>\w)-(?<num>\d)/g)) {
    console.log(`"${type}": "${num}" at ${index}`);
}
// => "a": "1" at 8
// => "b": "2" at 13
// => "c": "3" at 18
```

ПРИЛОЖЕНИЕ Б: ВОЗМОЖНОСТИ, ДОСТУПНЫЕ ТОЛЬКО ДЛЯ БРАУЗЕРА

Прежде чем перейти к сути этого раздела, небольшое примечание: вы не можете использовать некоторые из описанных здесь конструкций в строгом режиме, и, вероятно, вам вообще не стоит использовать ничего из этого[125]. Приложение Б (Annex B) посвящено

[124] `https://github.com/tc39/proposal-string-matchall`

[125] За исключением HTML-подобных комментариев на настоящей, правильно обслуживаемой странице XHTML, если скрипт действительно должен быть встроенным. Настоящие, правильно обслуживаемые страницы XHTML встречаются редко.

определению устаревшего поведения для движков JavaScript, работающих в веб-браузерах. В новом коде не стоит полагаться на устаревшее поведение.

Учитывая вышесказанное, вам могут прийти в голову два вопроса:

- В чем смысл Приложения Б?
- Зачем включать функции Приложения Б, которые по определению устарели, в книгу о новых возможностях?

Приложение Б включено в спецификацию как для документирования, так и для ограничения того, что делают движки JavaScript в веб-браузерах, часто выходящие за рамки стандартного языка. Но их должен знать (и делать) любой, кто создает движок JavaScript, если он хочет использовать этот движок в веб-браузере, работающем с реальным кодом. Устаревшие функции, недавно задокументированные в ES2015+, включены в эту книгу (минимально) для полноты.

Тогда что же находится в Приложении Б? Некоторые функции JavaScript были достаточно проблематичными (например, методы даты `getYear` и `setYear`, в отличие от методов `getFullYear` и `setFullYear`), и в ES2 их отлично документировали, но с примечанием, что они «не входят в эту спецификацию». ES3 отнесла эти вещи плюс некоторые другие (например, устаревшие восьмеричные литералы, которые легко спутать с десятичными) в «информационный» раздел совместимости в конце — Приложение Б. Также почти с момента появления JavaScript браузеры расширяли JavaScript за пределы описанного в спецификации, и некоторые второстепенные функции, определенные браузерами (`escape`, `unescape` и т. д.), были включены в Приложение Б как средство документирования и мягкого (в то время) определения общего имеющегося поведения. В ES5 добавили в приложение еще один метод, но в остальном мало что изменилось.

Именно ES2015 настойчиво документировал общее поведение (где это возможно) большого количества расширений, общих для веб-браузеров, а также впервые *потребовал*, чтобы функции в Приложении Б предоставлялись размещенными в браузере движками JavaScript. Теоретически движки JavaScript вне веб-браузеров не реализуют функции Приложения Б, хотя на практике движок, созданный для использования как внутри, так и вне браузеров, вряд ли увеличит сложность, отключив их в не-браузерных средах.

В этом разделе рассматриваются дополнения к Приложению Б в ES2015+: HTML-подобные комментарии, дополнения/изменения в регулярных выражениях, дополнительные свойства и методы `Object.prototype`, `String.prototype` и `RegExp.prototype`, в дополнение к синтаксису инициализатора объекта, различные биты свободного или неясного синтаксиса для обратной совместимости, а также особенное поведение `document.all`.

Но серьезно: не используйте их в новом коде.

HTML-ПОДОБНЫЕ КОММЕНТАРИИ

Движки JavaScript уже давно терпимо относятся к HTML-комментариям, обернутым вокруг кода скрипта. Это необходимо прежде всего для поддержки практики «комментирования» встроенного кода в элементе `script`, либо для *очень* старых браузеров, которые совсем не справятся с элементом `script` (и отобразят его содержимое

в документе), или как часть обертки CDATA для XHTML браузеров. Это выглядит так (или один из многих вариантов, или много неправильных попыток сделать это):

```
<script type="text/javascript"><!--//--><![CDATA[//><!--
// ...код...
//--><!]]></script>
```

Обратите внимание: первое выражение внутри, которое должно быть кодом JavaScript, — это HTML-подобный комментарий. Приложение Б было обновлено в ES2015, чтобы указать поведение такого кода.

Хитрости регулярного выражения

Приложение Б изменяет регулярное выражение малым количеством способов: добавляет очень небольшое расширение для управления экранированием в классах символов, допускает несопоставимые закрывающие квадратные скобки (`]`) и недопустимые группы кванторов (`{}`, `{foo}`), добавляет метод `compile`.

Расширение управляющего символа экранирования (\cX)

Приложение Б допускает использование цифр (в дополнение к обычным буквам) в управляющем экранировании (`\cX`), но только в том случае, если управляющее экранирование относится к классу символов (`[]`). Та же формула, которая применяется к буквам, определяет соответствующий управляющий символ — кодовую точку `X% 32`. Например, вот допустимый элемент управления экранированием:

```
console.log(/\cC/.test("\u0003")); // истина
```

Выражение истинно, потому что `\cC` определяет управляющий символ #3 («control c»), поскольку кодовая точка `C` — это `67`, а `67 % 32` равно `3`.

В стандартных регулярных выражениях (не относящихся к Приложению Б) за `\c` может следовать только английская буква (A-Z, без учета регистра), как в предыдущем примере. А Приложение Б также допускает использование цифр, *но* только если они относятся к классу символов. Например:

```
console.log(/[\c4]/.test("\u0014")); // истина
console.log(/\c4/.test("\u0014"));   // ложь
console.log(/\c4/.test("\\c4"));     // истина
```

В первом примере обнаружено совпадение `\u0014`, поскольку `\c4` относится к классу символов. Кодовая точка `4` — это `52`, а `52 % 32` равно `20`, то есть `0x14` в шестнадцатеричной системе. Второй пример не обнаруживает совпадения, потому что значение не относится к классу символов. Поэтому `\c4` не становится управляющим экранированием, и код возвращается к определению соответствия для `\ c4` как отдельных символов, как показано в третьем примере.

Допуск недопустимых последовательностей

В Приложении Б допускается пара недопустимых последовательностей, не разрешенных грамматикой основной спецификации например, неэкранированная закрывающая квадратная скобка (`]`) без открывающей (с открывающей: это сформирует класс символов, `[...]`) и неэкранированная открывающая фигурная скобка, которая не определяет квантор. Следующий код работает в движках, применяющих синтаксис Приложения Б, но может привести к ошибкам в движках, не поддерживающих его:

```
const mismatchedBracket = /]/;   // Должно быть /\]/
console.log(mismatchedBracket.test("testing] one two three")); // истина
console.log(mismatchedBracket.test("no brackets here"));       // ложь
const invalidQuanitfier = /{}/;  // Должно быть /\{\}/
console.log(invalidQuanitfier.test("use curly braces ({})"));   // истина
console.log(invalidQuanitfier.test("nothing curly here"));      // ложь
```

Метод `RegExp.prototype.compile`

Приложение Б вводит метод `compile` для объектов `RegExp`. Он полностью повторно инициализирует объект, для которого вызывается, изменяя его на совершенно другое регулярное выражение:

```
const rex = /^test/;
console.log(rex.test("testing")); // истина
rex.compile(/^ing/);
console.log(rex.test("testing")); // ложь
rex.compile("ing$");
console.log(rex.test("testing")); // истина
```

Метод `compile` принимает, по сути, те же параметры, что и конструктор `RegExp`, включая экземпляр `RegExp` в качестве первого параметра (в таком случае применяются шаблон и флаги экземпляра), или строку в качестве первого параметра и флаги в качестве необязательного второго параметра. Одно небольшое отличие заключается в том, что конструктор `RegExp` позволяет передавать объект регулярного выражения в качестве первого аргумента и новые флаги в виде строки во втором аргументе. Метод `RegExp.compile` этого не делает: вы не можете указать новые флаги, если в качестве первого аргумента указан экземпляр RegExp.

В Приложении Б отмечается, что использование метода `compile` *может* применяться в качестве сигнала движку JavaScript о том, что регулярное выражение предназначено для повторного использования и может быть хорошим кандидатом для оптимизации. Однако движки JavaScript обычно оптимизируют на основе измерения производительности кода в реальном времени и агрессивной оптимизации часто используемого кода, так что подсказка может не представлять никакой ценности для данного движка.

Дополнительные встроенные свойства

Встроенные объекты обладают некоторыми дополнительными свойствами, определенными в Приложении Б.

Дополнительные свойства объекта

Вы узнали об одном из дополнений к объектам из Приложения Б в главе 5: `__proto__`, как о свойстве-акцессоре, определенном в `Object.prototype`, так и о дополнительном синтаксисе инициализатора объекта для него.

Приложение Б также определяет некоторые другие свойства `Object.prototype` для обратной совместимости с кодом, который опирался на расширения SpiderMonkey (движок JavaScript в Mozilla), добавленные для свойства-акцессора до появления их в стандарте: `__defineGetter__`, `__defineSetter__`, `__lookupGetter__`, `__lookupSetter__`. Они делают именно то, что записано в их названиях — определяют и ищут функции геттеров и сеттеров для свойств:

```
"use strict";
const obj = {x: 10};
obj.__defineGetter__("xDoubled", function() {
    return this.x * 2;
});
console.log(obj.x);
// => 10
console.log(obj.xDoubled);
// => 20
try {
    obj.xDoubled = 27;
} catch (e) {
    console.error(e.message);
    // => Невозможно установить свойство xDoubled для #<Object>,
    // у которого есть только геттер
}
obj.__defineSetter__("xDoubled", function(value) {
    this.x = value / 2;
});
obj.xDoubled = 84;
console.log(obj.x);        // 42

console.log(obj.__lookupGetter__("xDoubled").toString());
// =>
// "function() {
//     return this.x * 2;
//}"
console.log(obj.__lookupSetter__("xDoubled").toString());
// =>
// "function(value) {
//     this.x = value / 2;
//}"
```

Следует отметить одну неочевидную вещь: методы «define» определяют акцессор к объекту, к которому вы их вызываете, но методы «lookup» будут следовать цепочке прототипов, если у объекта, к которому вы их вызываете, нет соответствующего акцессора. (Это похоже на свойства: установка свойства присваивает его объекту напрямую, но получение свойства идет вверх по цепочке прототипов.) Если метод «lookup2» возвращает функцию, она может быть определена для самого объекта, его прототипа или прототипа его прототипа и т. д.

Не стоит использовать эти методы в новом коде. Вместо этого необходимо применять общепринятую нотацию свойства-акцессора (`get foo()`, `set foo()`) при первом создании объекта — или, если вы хотите добавить акцессор постфактум, используйте `Object.defineProperty` или `Object.defineProperties`. Чтобы получить функцию для акцессора, надо использовать `Object.getOwnPropertyDescriptor` (используйте `Object.getPrototypeOf`, если хотите следовать цепочке прототипов, как это делают методы «lookup»).

Дополнительные строковые методы

В Приложении Б добавлены `substr` и куча методов, которые обертывают строку в HTML-теги.

Как и `substring`, метод `substr` создает подстроку из строки. Но он принимает разные аргументы: начальный индекс и *длину* подстроки, вместо начального и конечного индексов для `substring`. Это также позволяет задавать отрицательный начальный индекс, и в этом случае смещение считается от конца строки:

```
const str = "testing one two three";
console.log(str.substr(8, 3)); // "one"
console.log(str.substr(-5));   // "three"
```

Методы HTML: `anchor`, `big`, `blink`, `bold`, `fixed`, `fontcolor`, `fontsize`, `italics`, `link`, `small`, `strike`, `sub` и `sup`. Они буквально просто помещают начальные и конечные теги в строку:

```
console.log("testing".sub()); "<sub> testing </sub>"
```

Не используйте их в новом коде (не в последнюю очередь потому, что большинство этих HTML-тегов в любом случае устарели).

Различные фрагменты свободного или неясного синтаксиса

Приложение Б допускает некоторый синтаксис в свободном режиме, которого нет в основной спецификации. Ни одно из приведенных ниже действий не допустимо в строгом режиме — все они приводят к синтаксическим ошибкам. Поскольку я настоятельно рекомендую использовать строгий режим во всем новом коде (и он используется по умолчанию в модулях и конструкциях `class`), это, опять же, просто информация о конструкциях, которые вы можете найти в старом коде.

С помощью синтаксиса Приложения Б можно поместить метку перед объявлением функции (при условии, что это не функция-генератор):

```
label: function example() {}
```

Кажется, в метке нет никакого смысла, но (как и в случае с большинством «функциональных возможностей» Приложения Б), по-видимому, это было разрешено в движке в свое время. Поэтому движки должны допустить применение этого выражения, чтобы поддерживать устаревший код.

Аналогично вы можете прикрепить объявление функции к оператору `if` (без блока) или его оператору `else` (опять же без блока), например:

```
if (Math.random() < 0.5)
    function example() {
        console.log("1");
    }
else
    function example() {
        console.log("2");
    }
example();
```

Даже сейчас, когда объявления функций разрешены в блоках (описано в главе 3), этот пример был бы недействительным без семантики Приложения Б, потому что вокруг объявлений функций нет блоков. Для совместимости со старым кодом они разрешены только в свободном режиме.

Свободный синтаксис также позволяет переопределить `catch` связанный с `var` в блоке `catch`:

```
try {
    // ...
} catch (e) {
    var e;
    // ...
}
```

(Директива `var` также может находиться в инициализаторе цикла `for` или `for-in`, но не в инициализаторе цикла `for-of`.) Без синтаксиса Приложения Б это было бы синтаксической ошибкой. При помощи синтаксиса объявляется переменная в области функции (или в глобальной области), в которой отображается код, — но `e` в блоке `catch` по-прежнему содержит привязку к `catch`, и присвоение происходит к привязке к `catch`, а не к переменной:

```
"use strict";
function example() {
    e = 1;
    console.log(e);         // 1
    try {
        throw new Error("blah");
    } catch (e) {
        var e;              // Было бы ошибкой SyntaxError, если бы
                            // не синтаксис Приложения Б
        e = 42;
        console.log(e);     // 42
    }
    console.log(e);         // 1
}
example();
```

Не делайте этого в новом коде. Но в Интернете может быть устаревший код 1996 года, в котором такие выражения все еще есть…

Наконец (и это любопытно), знаете ли вы, что можно поместить инициализатор в переменную в цикле `for-in`? Это можно реализовать с использованием синтаксиса Приложения Б:

```
const obj = {
    a: 1,
    b: 2
};
for (var name = 42 in obj) {
    console.log(name);
}
// =>
// "a"
// "b"
```

Такое решение работает только с директивой `var`: применить `let` или `const` было бы невозможно, и этот код не будет работать в строгом режиме. Инициализатор выполняется и присваивается переменной, а затем немедленно перезаписывается первым именем свойства в объекте, поэтому вы никогда не увидите его в цикле. Но инициализатор с побочными эффектами можно наблюдать:

```
for (var name = console.log("hi") in {}) {// "hi"
}
let n = 1;
for (var name2 = n = 42 in {}) {
}
console.log(n);                                  // => 42
```

Не стоит так писать, но, если вы видите такой фрагмент в устаревшем коде, вы будете знать, почему код не вызывает ошибки и что он делает (например, чтобы вы могли безопасно исправить это).

Когда же `document.all` есть... или нет?

Это достаточно забавно, и тут можно рассказать следующую историю.

На заре Интернета `document.all` был функциональной возможностью в Microsoft в Internet Explorer (IE) — коллекция *всех* элементов в документе (с различными свойствами для вложенных коллекций подмножеств элементов), которую вы можете перебирать или искать элементы по их `id` или `name`. Напротив, Netscape Navigator (другой крупный браузер того времени) использовал новый стандарт DOM (`getElementById` и т. п.). Код, написанный для работы с обоими, должен был выбрать, какой из них использовать. Иногда программисты делали это примерно так:

```
if (document.getElementById) {// или различные варианты
    // ...использовать getElementById или другие возможности DOM...
} else {
    // ...использовать document.all или другие возможности Microsoft...
}
```

В других случаях они делали это наоборот, и в этом проблема:

```
if (document.all) {
    // ...использовать document.all или другие Microsoft-isms...
} else {
    // ...использовать getElementById или другие DOM-isms...
}
```

На месте `if (document.all)` они могут использовать `if (typeof document.all !== "undefined")` или `if (document.all != undefined)`, но цель была та же.

Конечно, технологии развивались, и Microsoft внедрила стандарты DOM, но не все веб-страницы обновили свой скрипт. Более новые (на тот момент) браузеры, такие как Chrome, стремившиеся поддерживать страницы, даже если они были написаны для IE, включали поддержку возможностей Microsoft, таких как `document.all` (и глобальная переменная `event` и т. п.).

Перенесемся на несколько лет вперед, и любая страница, проверенная на наличие `document.all` вместо `getElementById`, будет работать так же хорошо, как при движении по пути DOM, а не по пути Microsoft, и создатели браузеров хотели, чтобы так и было. Одним из способов, конечно, было бы отказаться от `document.all`. Но производители браузеров не хотели отказываться от его поддержки *полностью*, потому что необходимо было продолжать поддерживать страницы, которые использовали его без проверки. Что делать?

Решение было одновременно гениальным и ужасающим: они сделали `document.all` «ложноподобным» объектом. Более конкретно, они наделили его некоторыми интересными возможностями:

- Он приводит результат к значению `false` вместо `true`, когда вы приводите его к логическому значению.
- Когда вы применяете к нему `typeof`, результатом будет `"undefined"`.
- При использовании оператора ==, чтобы сравнить его с `null` или `undefined`, в результате будет значение `true` (и конечно, сравнения «!=» дадут в результате значение `false`).

Таким образом, `if (document.all)` и аналогичные проверки действуют так, как если бы `document.all` там не было. Но код, который вообще не выполняет проверку, все еще работает, потому что `document.all` все еще (на данный момент) работает.

Приложение Б определяет эту функциональность для объекта под названием [[IsHTMLDDA]] с внутренним слотом. `document.all` — единственный такой объект, по крайней мере, на данный момент.

ОПТИМИЗАЦИЯ ХВОСТОВОГО ВЫЗОВА

В теории ES2015+ требует, чтобы движок JavaScript реализовал *оптимизацию хвостовых вызовов* (TCO, Tail call optimization), которая в первую очередь полезна в ситуациях, связанных с рекурсией. На самом деле только один движок JavaScript поддерживает TCO, и это вряд ли изменится в ближайшее время. Подробнее об этом чуть позже.

Сначала краткое напоминание о стеке и фреймах стека. Рассмотрим этот код:

```
"use strict";
function first(n) {
    n = n - 1;
    return second(n);
}
function second(m) {
    return m * 2;
}
const result = first(22);
console.log(result); // 42
```

Без TCO, когда движок JavaScript обрабатывает вызов функции `first`, он вводит *фрейм стека* в стек с адресом возврата, аргументами, передаваемыми `first` (`22`), и, возможно, некоторыми другими бухгалтерскими данными, а затем переходит к началу кода функции `first`. Когда `first` вызывает функцию `second`, движок помещает в стек другой фрейм стека с обратным адресом и т. д. Когда `second` возвращает значение, ее фрейм извлекается из стека. Когда `first` возвращает значение, ее фрейм извлекается из стека (рисунок 17-1).

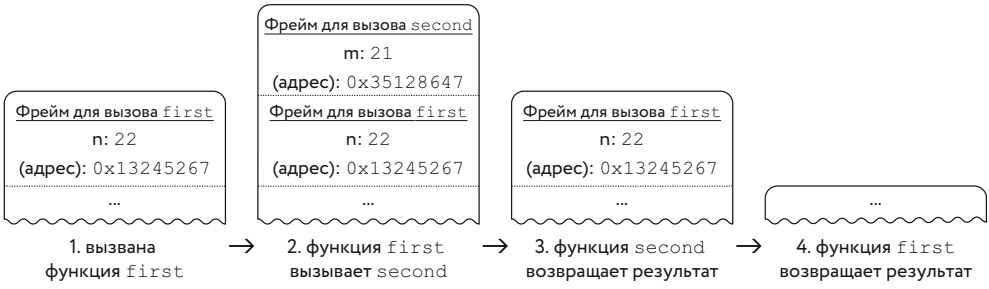


РИСУНОК 17-1

Поскольку стек не бесконечен, он может содержать столько фреймов, на сколько хватает места. Допустим, вы случайно написали код, вызывающий сам себя бесконечно (прямо или косвенно), и увидели ошибку переполнения стека, подобную этой:

```
RangeError: Maximum call stack size exceeded
```

Хотя, если посмотреть на `first`, можно заметить, что вызов `second` осуществляется перед возвратом значения в самом конце функции `first`, и функция `first` возвращает результат вызова функции `second`. Это означает, что вызов `second` — это *хвостовой вызов* (вызов в *хвостовой позиции*).

С TCO движок может извлечь фрейм стека для `first` из стека перед вызовом `second`; он просто должен дать фрейму стека функции `second` обратный адрес, в который возвращается значение функции `first`. Кроме этого обратного адреса, фрейм стека функции `first` не имеет особого смысла, поэтому TCO может его исключить (рисунок 17-2).

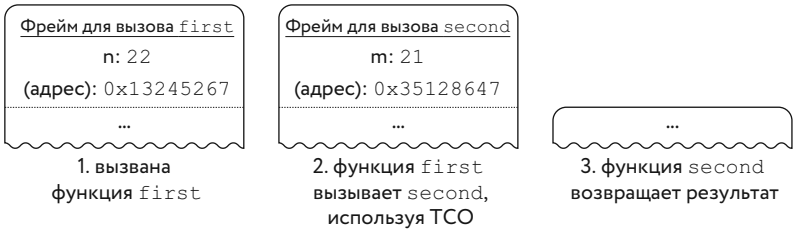


РИСУНОК 17-2

В этом конкретном примере избавление от фрейма стека функции `first` перед вызовом `second` на самом деле не имеет большого значения. Но (особенно когда вы имеете дело с рекурсией) отказ от наличия этих фреймов в стеке может иметь большое значение. Например, рассмотрим эту классическую факториальную функцию:

```
function fact(v) {
    if (v <= 1n) {
        return v;
    }
    return v * fact(v - 1n);
}
console.log(fact(5n)); // 120
```

(Я использую тип данных BigInt в этом коде, чтобы убрать ограничение емкости числового типа.) Размер стека ограничивает количество факториалов, которые может вычислить `fact`, потому что, если ряд начинается с достаточно большого числа (скажем, 100000), все эти фреймы стека из рекурсивных вызовов переполняют стек:

```
console.log(fact(100000n));
// => RangeError: Превышен максимальный размер стека вызовов
```

Однако, к сожалению, вызов, осуществляемый функцией `fact` самой себе, не находится в хвостовой позиции. Очень близко к этому, но после вызова `fact` умножает результат на `v` и возвращает результат. Но легко настроить функцию `fact` так, чтобы она могла использовать преимущества TCO, добавив к ней второй параметр и переместив умножение:

```
function fact(v, current = 1n) {
    if (v <= 1n) {
        return current;
    }
    return fact(v - 1n, v * current);
}
```

Теперь вызов функции `fact` к самой себе находится в хвостовом положении. С TCO при каждом вызове функцией `fact` самой себя фрейм стека для нового вызова заменяет фрейм стека для предыдущего. Точно так же, как фрейм функции `second` заменял фрейм функции `first` ранее, вместо того чтобы все помещать в стек до тех пор, пока не будет найден конечный результат, а затем все фреймы будут извлечены. Таким образом, стек больше не ограничивает размер факториалов, которые может вычислить функция `fact`.

Но, как я упоминал в начале этого раздела, только один крупный движок JavaScript в настоящее время реализует TCO: JavaScript Core в Safari (и других браузерах для iOS[126]). V8, SpiderMonkey и Chakra не поддерживают TCO — и, по крайней мере, команды V8 и SpiderMonkey в настоящее время не планируют этого делать. (Некоторое время у V8 была частичная поддержка, но она была удалена по мере развития движка.) Основной камень преткновения — влияние на трассировки стека. Вспомните пример `first`/`second` в начале этого раздела. Предположим, что функция `second` выдала ошибку. Поскольку фрейм стека функции `first` был заменен на фрейм `second` благодаря TCO, трассировка стека будет выглядеть так, как если бы функция `second` была бы вызвана на месте вызова функции `first`, а не внутри функции `first`. Существуют различные способы решения этой проблемы, и некоторые предлагаемые альтернативы, включая отказ от (а не автоматический) TCO, но на данный момент нет единого мнения о том, как двигаться дальше. Может быть, когда-нибудь.

ОТ СТАРЫХ ПРИВЫЧЕК К НОВЫМ

Все эти различные дополнения предоставляют при желании возможность изменения нескольких привычек.

Используйте двоичные литералы

Старая привычка: Использование шестнадцатеричных чисел для битовых флагов и т. п., где двоичный код может быть более понятным:

```
const flags = {
    something:       0x01,
    somethingElse:   0x02,
    anotherThing:    0x04,
    yetAnotherThing: 0x08
};
```

Новая привычка: Там, где это имеет смысл, используйте новый двоичный целочисленный литерал:

```
const flags = {
    something:       0b00000001,
    somethingElse:   0b00000010,
    anotherThing:    0b00000100,
    yetAnotherThing: 0b00001000
};
```

[126] Такие браузеры для iOS, как Chrome и Firefox, не могут использовать свои обычные движки JavaScript, потому что не-Apple приложения, не могут выделять исполняемую память, требующуюся для JIT-компиляции, которую выполняют V8 и SpiderMonkey. Однако V8 недавно представила версию V8 «только для интерпретатора», так что Chrome может начать использовать ее на iOS.

Используйте новые математические функции вместо различных математических обходных путей

Старая привычка: Используя различные математические обходные пути, например, для эмуляции 32-разрядной целочисленной математики или усечения, выполняя `value = value < 0 ? Math.ceil(value) : Math.floor(value)`.

Новая привычка: Где это уместно, используйте некоторые из новых функций `Math`, такие как `Math.imul` или `Math.trunc`.

Используйте оператор нулевого слияния для значений по умолчанию

Старая привычка: Использование логического оператора ИЛИ (`||`) или явной проверки `null`/`undefined` при предоставлении значений по умолчанию:

```
const delay = this.settings.delay || 300;
// или
const delay = this.settings.delay == null ? 300: this.settings.delay;
```

Новая привычка: Используйте оператор нулевого слияния, где это уместно, чтобы не все ложные значения (например, `0`) вызывали значение по умолчанию:

```
const delay = this.settings.delay ?? 300;
```

Используйте опциональную цепочку вместо проверок &&

Старая привычка: Использование логического оператора И (`&&`) или аналогичного при осуществлении доступа к вложенным свойствам объектов, которые могут быть или не быть там:

```
const element = document.getElementById("optional");
if (element) {
    element.addEventListener("click", function() {
        // ...
    });
}
```

Новая привычка: Используйте опциональную цепочку, где это уместно:

```
document.getElementById("optional") ?.addEventListener("click", function() {
    // ...
});
```

Уберите привязку ошибки (e) из “catch (e)”, если она не используется

Старая привычка: Написание `catch (e)`, когда не используется `e` (потому что у вас не было выбора):

```
try {
    theOperation();
} catch (e) {
    doSomethingElse();
}
```

Новая привычка: Не записывайте часть `(e)` при использовании современного синтаксиса (транспилирование или при ориентировании только на среды, которые поддерживают необязательную привязку `catch`):

```
try {
    theOperation();
} catch {
    doSomethingElse();
}
```

Используйте оператор возведения в степень (**) вместо метода Math.pow

Старая привычка: Использовать метод `Math.pow` для возведения в степень, например:

```
x = Math.pow(y, 32);
```

Новая привычка: Попробуйте вместо этого использовать оператор возведения в степень, поскольку метод `Math.pow` может быть перезаписан и использование операции возведения в степень не требует поиска идентификатора для `Math` или поиска свойства для `pow`:

```
x = y**32;
```

18

Грядущие функциональные возможности класса

СОДЕРЖАНИЕ ГЛАВЫ

- Публичные поля классов
- Приватные поля класса, методы экземпляра и акцессоры
- Статические поля класса и приватные статические методы

В этой главе вы узнаете о грядущих функциональных возможностях `class`, которые почти наверняка появятся в ES2021 и которые достаточно стабильны (или почти стабильны) для использования с транспиляцией сегодня (или даже без нее, в некоторых случаях, в современных средах): поля класса; частные поля и методы/акцессоры; а также статические поля и частные статические методы/акцессоры.

ПУБЛИЧНЫЕ И ПРИВАТНЫЕ ПОЛЯ КЛАССА, МЕТОДЫ И АКЦЕССОРЫ

Синтаксис класса ES2015 был намеренно просто отправной точкой. Многочисленные предложения, которые, вероятно, будут приняты в ES2021, расширяют его дополнительными полезными возможностями:

- определения публичного поля (свойства);
- приватные поля;
- приватные методы и акцессоры экземпляра;
- публичные статические поля;
- приватные статические поля;
- приватные статические методы.

Они делают уже полезный[127] синтаксис `class` еще более полезным, принимая задачи, которые раньше требовали сокрытия вещей в замыканиях, присвоения постфактум, закулисное использование `WeakMap`, неудобный синтаксис и т. д. — и делают эти решения простым синтаксисом, а также потенциально улучшают способы оптимизации результатов движками JavaScript.

Усовершенствования распределены по нескольким предложениям, которые могут продвигаться разными темпами, но все они находились на этапе 3 в начале 2020 года:

- *Объявления полей классов для JavaScript (часто называемые «предложением полей классов»:*
  `https://github.com/tc39/proposal-class-fields`
- *Приватные методы и геттеры/сеттеры для классов JavaScript:*
  `https://github.com/tc39/proposal-private-methods`
- *Статические возможности класса:*
  `https://github.com/tc39/proposal-static-class-features/`

В этом разделе рассматриваются различные дополнения. Сейчас они поддерживаются транспиляторами. Находясь на этапе 3, эти функции изначально реализуются движками JavaScript (например, приватные и публичные поля доступны в V8, поставляются без флага в Chrome 74 и выше).

Определения публичного поля (свойства)

С синтаксисом `class` в ES2015 *декларативно* определялись только конструкторы, методы и свойства-акцессоры; свойства данных создавались ad hoc[128] с помощью присваивания, часто (но не всегда) в конструкторе:

```
class Example {
    constructor() {
        this.answer = 42;
    }
}
```

Предложение полей классов добавляет в язык *определения публичных полей* (по сути, *определения* свойств). Следующее определяет точно такой же класс, как и в предыдущем примере:

```
class Example {
    answer = 42;
}
```

Определение — это просто имя свойства, за которым необязательно следует знак равенства (=) и выражение инициализатора, а затем завершается точкой с запятой (`;`). (Подробнее об инициализаторах чуть позже.) Обратите внимание, что здесь нет выражения `this.` перед именем свойства в определении.

---

[127] Имеется в виду — полезно, если вы используете функции конструктора; JavaScript также поддерживает парадигмы программирования, которые их не используют и, следовательно, не используют синтаксис `class`.

[128] «Специально для этого» (*лат.*) — *Прим. ред.*

«ПУБЛИЧНОЕ ПОЛЕ» ИЛИ «СВОЙСТВО»

Поскольку предложение полей классов добавляет в язык приватные поля, а они не относятся к свойствам (об этом вы узнаете в следующем разделе), все чаще свойства и приватные поля называют просто *полями*, и в дальнейшем свойства — *публичными полями*. Публичные поля по-прежнему относятся к свойствам; это просто другое название для них.

Если у вас есть несколько требующих определения свойств, они должны быть определены отдельно. Вы не можете сделать цепочку определений, как в случае с `var`, `let` и `const`:

```
class Example {
    answer = 42, question = "...";
    //         ^-- SyntaxError: Неожиданный токен, ожидаемый ";"
}
```

Вместо этого запишите каждое определение отдельно:

```
class Example {
    answer = 42;
    question = "...";
}
```

Для свойств, начальные значения которых не зависят от значений параметров конструктора, новый синтаксис более лаконичен. Если начальное значение действительно *зависит* от параметра конструктора, то добавление определения будет немного более подробным (по крайней мере, на данный момент):

```
class Example {
    answer;

    constructor(answer) {
        this.answer = answer;
    }
}
```

В этом примере определение публичного поля в начале класса избыточно с точки зрения свойств, которые `new Example` создает для вновь созданного объекта; объект в любом случае получит свойство `answer`. Но определения публичного свойства были добавлены в формулировку по нескольким причинам:

- Указание движку JavaScript заранее, какими свойствами будет обладать объект, уменьшает количество изменений формы, через которые проходит объект (изменения в его наборе свойств и т. п.), что улучшает способность движка быстро оптимизировать объект (см. врезку).

- Предварительное определение формы объекта также полезно для пользователей, читающих ваш класс, и обеспечивает удобное расположение комментариев к документации, описывающих свойства. Помните, что вы *читаете* код гораздо чаще, чем *пишете* его, поэтому иногда полезно приложить дополнительные усилия, чтобы помочь читателям.
- Наличие такого синтаксиса для публичных свойств обеспечивает равенство с определениями приватных полей, о которых вы узнаете в следующем разделе.
- Он предоставляет место для применения *декораторов* к свойству, если/когда предложение декораторов[129] будет реализовано.

Это будет зависеть от вашего стиля и стиля вашей команды, когда будете (и будете ли) вы определять публичные свойства декларативно, а не с помощью присваивания.

«ФОРМА» ОБЪЕКТА И ЕЕ ИЗМЕНЕНИЯ

Форма объекта JavaScript — это набор имеющихся у него полей и свойств, а также его прототип. Современные движки JavaScript агрессивно оптимизируют объекты; форма объекта — важная часть этого процесса. Предотвращение изменения формы объекта с течением времени помогает движку выполнять свою работу более эффективно. Например, посмотрите на следующий класс:

```
class Example {
    constructor(a) {
        this.a = a;
    }
    addB(b) {
        this.b = b;
    }
    addC(c) {
        this.c = c;
    }
}
```

Недавно созданный экземпляр класса Example содержит только одно свойство — `a`. (У него вообще ничего нет до присвоения свойства `a`, но во многих случаях движок JavaScript может избежать повторной оптимизации, если присвоение находится в начале конструктора и является безусловным.) Но позже экземпляр может получить свойство `b`, свойство с или их оба. Если это произойдет, движок JavaScript должен каждый раз корректировать свою оптимизацию, чтобы учесть изменения. Но если вы заранее сообщите движку, какие свойства получит объект (присвоив им значения в безусловной ранней части конструктора или определив их с помощью синтаксиса определения свойств), он может учитывать эти свойства только один раз в начале.

[129] `https://github.com/tc39/proposal-decorators`

Если в определении свойства есть инициализатор, этот код инициализатора выполняется точно так же, как если бы он находился внутри конструктора (за исключением того, что у него нет доступа к параметрам конструктора, если таковые имеются). Среди прочего это означает, что при использовании `this` в инициализаторе он получает то же значение, что и в конструкторе — ссылку на инициализируемый объект. Свойства экземпляра, созданные таким образом, настраиваются, доступны для записи и перечисляются точно так же, как если бы они были созданы с помощью присваивания в конструкторе.

Если у свойства нет инициализатора, оно создается со значением по умолчанию `undefined`:

```
class Example {
    field;
}
const e = new Example();
console.log("field" in e);   // истина
console.log(typeof e.field); // "undefined"
```

Хотя предложение полей классов находится только на этапе 3, его основные функциональные возможности, описанные в этом разделе, уже давно используются с помощью транспиляции. Пока движки JavaScript не будут поставляться с этой функцией (есть версии V8 и SpiderMonkey, скоро появятся и другие), вам нужно будет выполнить транспилирование, чтобы использовать ее — и, конечно, в зависимости от вашей целевой среды вам может потребоваться транспилирование для поддержки старых движков на некоторое время.

СТРЕЛОЧНЫЕ ФУНКЦИИ В ИНИЦИАЛИЗАТОРАХ

Поскольку инициализатор выполняется так, как если бы он находился внутри конструктора, некоторые программисты стали использовать свойство, ссылающееся на стрелочную функцию, как удобный способ создания функции, привязанной к экземпляру. Например:

```
class Example {
    handler = event => {
        event.currentTarget.textContent = this.text;
};

    constructor(text) {
        this.text = text;
}

    attachTo(element) {
        element.addEventListener("click", this.handler);
}
}
```

Это работает, потому что стрелочная функция закрывается над `this`, а `this` в месте определения стрелочной функции ссылается

на инициализируемый экземпляр (точно так же, как если бы инициализатор находился внутри конструктора).

Хотя на первый взгляд это кажется удобным, есть некоторые аргументы против. При таком решении обработчик `handler` помещается в сам экземпляр, а не `Example.prototype`, что означает:

- Это трудно имитировать для тестирования. Часто фреймворки тестирования работают на уровне прототипа.
- Это мешает наследованию. Предположим, у вас есть `class BetterExample extends Example` и определен новый/лучший обработчик `handler`? У него не было бы доступа к версии `Example` при помощи `super`.

Альтернативой будет размещение метода в прототипе, где его можно повторно использовать и (возможно) имитировать для тестирования, и привязать его к экземпляру, когда/где это необходимо. В этом случае вы могли бы сделать это в конструкторе, в методе `attachTo` или... в инициализаторе свойства в определении свойства! Например, так:

```
class Example {
    handler = this.handler.bind(this);

    constructor(text) {
        this.text = text;
}

    handler(event) {
        event.currentTarget.textContent = this.text;
}

    attachTo(element) {
        element.addEventListener("click", this.handler);
}
}
```

Этот код берет обработчик `handler` прототипа, привязывает его к `this` и присваивает результат в качестве свойства экземпляра. Поскольку инициализатор запускается до создания свойства, `this.handler` в инициализаторе ссылается на обработчик, который находится в прототипе.

Этот шаблон достаточно распространен, так как он представляет собой один из примеров использования декораторов (его часто называют декоратором @bound).

Имя публичного поля может быть вычислено; для этого используются скобки вокруг выражения, определяющего ключ свойства, как это делается в литерале объекта. Это особенно полезно, когда ключ свойства представлен символом:

```
const sharedUsefulProperty = Symbol.for("usefulProperty");
class Example {
    [sharedUsefulProperty] = "example";

    show() {
        console.log(this[sharedUsefulProperty]);
    }
}

const ex = new Example();
ex.show(); // "example"
```

Ранее вы узнали, что, если у определения публичного поля есть инициализатор, он выполняется точно так же, как если бы он был в конструкторе. В частности, инициализация выполняется в порядке исходного кода, как если бы она была написана в самом начале конструктора (в базовом классе) или сразу после вызова `super()` (помните, что в подклассе новый экземпляр создается вызовом `super()`, так что значение `this` до этого недоступно). Порядок означает, что более позднее свойство может полагаться на более раннее, поскольку более раннее свойство будет создано и инициализировано первым. В Листинге 18-1 приведен пример, показывающий как порядок, так и тот факт, что инициализация выполняется сразу после вызова функции `super()`.

Листинг 18-1: Порядок определения публичных свойств — property-definition-order.js

```
function logAndReturn(str) {
    console.log(str);
    return str;
}
class BaseExample {
    baseProp = logAndReturn("baseProp");
    constructor() {
        console.log("BaseExample");
    }
}
class SubExample extends BaseExample {
    subProp1 = logAndReturn("example");
    subProp2 = logAndReturn(this.subProp1.toUpperCase());
    constructor() {
        console.log("SubExample before super()");
    super();
        console.log("SubExample after super()");
        console.log(`this.subProp1 = ${this.subProp1}`);
        console.log(`this.subProp2 = ${this.subProp2}`);
    }
}
new SubExample();
```

После запуска Листинг 18-1 выведет:

```
SubExample before super()
baseProp
BaseExample
example
```

```
EXAMPLE
SubExample after super()
this.subProp1 = example
this.subProp2 = EXAMPLE
```

Приватные поля

Предложение полей класса также добавляет *приватные поля* в синтаксис `class`. Приватные поля отличаются от свойств объекта (публичных полей) несколькими аспектами. Ключевое отличие в том, что, как следует из названия, только код внутри класса может получить доступ к приватному полю класса. Пример приведен в Листинге 18-2. Помните, что для запуска примера вам может понадобиться транспилятор (и соответствующий плагин). Но, возможно, вы читаете эту книгу, когда движок JavaScript вашего браузера уже изначально поддерживает приватные поля. (Плагин для Babel v7 — это `@babel/plugin-proposal-class-properties`.)

Листинг 18-2: Простой пример приватных полей — private-fields.js

```
class Counter {
    #value;

    constructor(start = 0) {
        this.#value = start;
    }

    increment() {
        return ++this.#value;
    }

    get value() {
        return this.#value;
    }
}
const c = new Counter();
console.log(c.value); // 0
c.increment();
console.log(c.value); // 1
// console.log(c.#value); // Будет ошибкой SyntaxError
```

В Листинге 18-2 определен класс счетчика `Counter` с приватным полем экземпляра, называемым `#value`. Класс приращает `#value`, когда его метод `increment` вызывается и возвращает значение `#value` из своего свойства-акцессора `value`, но не позволяет внешнему коду непосредственно просматривать или изменять поле `#value`.

В этом примере следует отметить следующие моменты:

- Приватное поле определяется так же, как и публичное. Оно помечается как частное простым присвоением ему имени, начинающееся со знака хэша (`#`). Это (часть `#value`) называется *приватным идентификатором*.
- Чтобы получить доступ к полю, используется его личный идентификатор (`#value`) в выражении акцессора — `this.#value`.

- Код вне определения класса `Counter` не может получить доступ к полю, и это синтаксическая ошибка, а не ошибка *выполнения* (в большинстве случаев; подробнее об этом чуть позже). То есть это хорошая упреждающая ошибка, возникающая во время синтаксического анализа кода, а не позже, когда код выполняется, так что она будет обнаружена на ранней стадии.

У приватных полей есть некоторые ключевые отличия от свойств объекта:

- Приватные поля можно создать только при помощи определения приватных полей, их нельзя создать ad hoc через присвоение или `defineProperty`/`defineProperties`. Попытка использовать неопределенное приватное поле — это синтаксическая ошибка.
- Приватный идентификатор, который вы вводите в коде (`#value` в примере), — не фактическое имя поля (хотя это имя вы используете в своем коде). Вместо него «под капотом» движок JavaScript присваивает полю глобально уникальное имя (называемое *приватным именем*), которое вы, программист, никогда не увидите. Приватный идентификатор фактически представлен константой в области видимости определения `class`. Значение константы получает приватное имя в качестве значения. При обращении к полю (например, `this.#value`) значение приватного идентификатора `#value` ищется в текущей области, и полученное приватное имя используется для поиска поля в объекте. (Подробнее об этом чуть позже.) В отличие от этого, в случае с именем свойства имя, которое вы вводите в код, — это фактическое имя свойства.
- Приватные поля хранятся отдельно от свойств объекта — во внутреннем слоте объекта с именем [[PrivateFieldValues]], который (в спецификации) представляет собой список пар имя/значение (хотя, конечно, движки JavaScript могут свободно оптимизировать эти данные, поскольку вы никогда не видите этот список напрямую). Имя — это приватное имя, а не приватный идентификатор. Причина для такого положения станет ясна чуть позже.
- Приватные поля невозможно удалить из объекта. Попытка применить оператор `delete` к приватному полю вызовет синтаксическую ошибку (ведь оператор `delete` удаляет свойства, а приватные поля не свойства), и нет никакого эквивалента `delete` для приватных полей. В сочетании с тем фактом, что необходимо определять приватные поля с помощью определения поля, это означает, что набор приватных полей, доступных в классе, будет фиксированным; он никогда не меняется (это делает его высокооптимизируемым).
- Приватные поля недоступны через рефлексию. Они — не свойства, поэтому ни один ориентированный на свойства метод, такой как `Object.getOwnPropertyNames` или `Reflect.ownKeys`, не применим к приватным полям. У `Object` или `Reflect` тоже нет никаких новых методов для применения к приватным полям.
- К приватным полям нельзя получить доступ с помощью обозначения в скобках (например, `this["#value"]` не работает). На самом деле динамического механизма доступа к ним вообще нет. Имена должны быть записаны с использованием буквенных обозначений. (Последующие предложения могут изменить это.)

- Приватные поля недоступны в подклассах. Если класс `A` определяет приватное поле, только код в классе `A` может получить к нему доступ. Если `B` становится подклассом A (`class B extends A`), код в подклассе `B` не сможет получить доступ к приватным полям класса `A`. Это естественное следствие того, что приватный идентификатор является частью области видимости `class`: он не входит в область видимости подкласса, поэтому его нельзя использовать. (Здесь есть нюанс, о котором вы узнаете позже, касающийся *вложенных* классов.)
- Исходя из этого, будет отлично, если у класса `A` и у подкласса `B` будет приватное поле с одним приватным идентификатором (`#value` или любым другим). Эти поля полностью разделены, поскольку каждое из них получит свое собственное приватное имя, а приватный идентификатор преобразуется в приватное имя с использованием области видимости класса. Идентификатор `#value` в коде класса `A` относится к приватному полю класса `A`, а идентификатор `#value` класса `B` относится к приватному полю класса `B`.

Давайте посмотрим, как приватные идентификаторы преобразуются в приватные имена. Вспомните *объект среды* из главы 2, в котором хранятся переменные (и константы). Приватные идентификаторы хранятся в аналогичном объекте, называемом *средой приватных имен* (PNE, Private Name Environment). На рисунке 18–1 изображено, как приватный идентификатор, среда приватного имени, приватное имя и экземпляр класса связаны друг с другом.

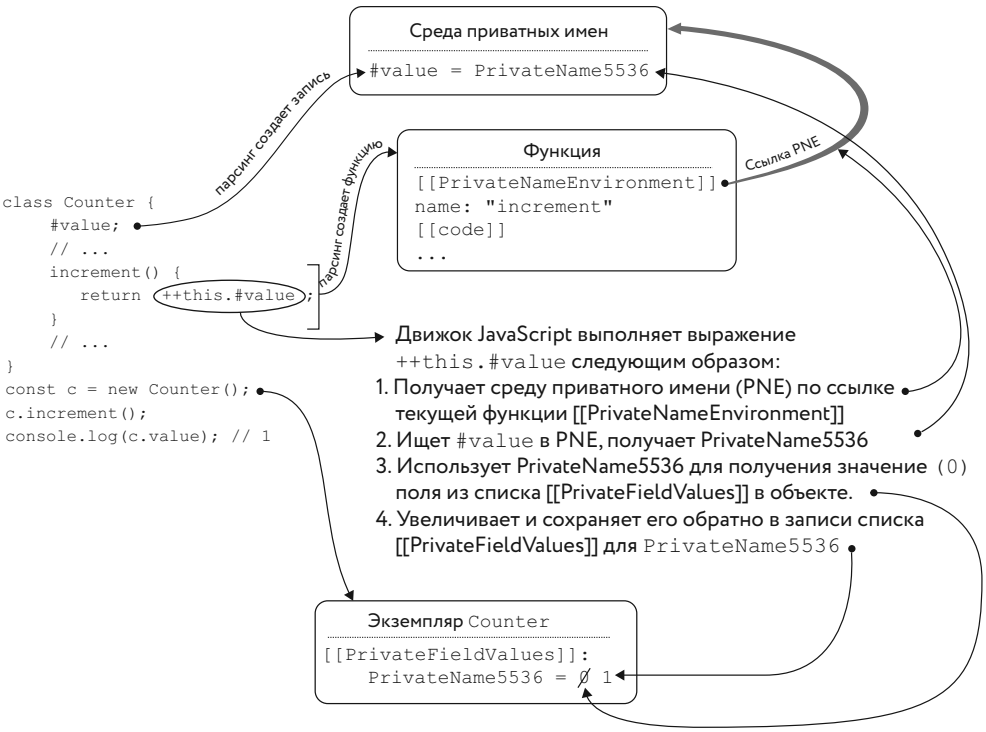


РИСУНОК 18-1

Поскольку у приватных идентификаторов *лексическая область видимости* (разрешение идентификатора на его приватное имя зависят от области действия), если у вас есть вложенный класс (класс внутри класса), внутренний класс может получить доступ к приватным полям внешнего класса (Листинг 18-3).

Листинг 18-3: Приватные поля во вложенных классах — private-fields-in-nested-classes.js

```
class Outer {
    #outerField;
    constructor(value) {
        this.#outerField = value;
}
    static createInner() {
        return class Inner {
            #innerField;
            constructor(value) {
                this.#innerField = value;
            }
            access(o) {
                console.log(`this.#innerField = ${this.#innerField}`);
                // Работает, потому что #outerField находится в области
                // видимости:
                console.log(`o.#outerField = ${o.#outerField}`);
            }
        };
    }
}
const Inner = Outer.createInner();
const o = new Outer(1);
const i = new Inner(2);
i.access(o);
// =>
// this.#innerField = 2
// o.#outerField = 1
```

Код в классе `Inner` можете видеть с приватные поля в классе `Outer`, поскольку класс `Inner` определен *внутри* класса `Outer`. То есть к приватному идентификатору применяются обычные правила определения области видимости (`#outerField`).

Это разрешение приватных идентификаторов на основе области видимости гарантирует, что они тесно связаны с классом, в котором определены. Даже если вычисляется одно и то же определение `class` более одного раза, что приводит к созданию двух копий класса, приватные поля в одной копии *недоступны* для другой копии, поскольку приватный идентификатор в области видимости одного класса получает другое значение приватного имени, чем тот же идентификатор в другой области видимости класса. Это похоже на то, когда у вас есть функция, возвращающая функцию, закрывающуюся над переменной или константой:

```
let nextId = 0;
function outer() {
    const id = ++nextId;
    return function inner() {
        return id;
```

```
    };
}
const f1 = outer();
const f2 = outer();
```

Обе функции, созданные в классе `inner` (`f1` and `f2`), закрываются над константой под названием `id`, но эти константы отделены друг от друга и содержат разные значения. То же самое относится и к приватным идентификаторам в классах.

Один из способов, которым можно выполнить определение `class` более одного раза, — это загрузить один и тот же класс в нескольких базах realm (например, как в главном окне, так и в `iframe`), а затем обмениваться данными между ними. Но также можно получить несколько копий класса в пределах одной базы realm, просто выполнив определение `class` более одного раза (точно так же, как в предыдущем примере несколько раз выполнялось определение функции `inner`), см. Листинг 18-4.

Листинг 18-4: Приватные поля в копиях классов — private-fields-in-class-copies.js

```
function makeCounterClass() {
    return class Counter {
        #value;

        constructor(start = 0) {
            this.#value = start;
        }

        increment() {
            return ++this.#value;
        }

        get value() {
            return this.#value;
        }

        static show(counter) {
            console.log(`counter.#value = ${counter.#value}`);
        }
    };
}

const Counter1 = makeCounterClass();
const Counter2 = makeCounterClass();

const c = new Counter1();
c.increment();
Counter1.show(c);   // "counter.#value = 1"
Counter2.show(c);   // TypeError: Не удается считать приватный член #value
                    // из объект, класс которого не объявлял его
```

(Конкретная ошибка зависит от движка JavaScript или используемого вами транспилятора. Другая версия — «TypeError: попытка получить приватное поле не в экземпляре».)

`Counter1` и `Counter2` — это две копии одного класса. Следовательно, идентификатор `#value` в `Counter1` — это не тот же самый идентификатор `#value`, что

и в `Counter2`. Они содержат разные значения приватных имен, поэтому код в `Counter1` не может получить доступ к полю `#value` в экземпляре, созданном `Counter2`, и наоборот. Это означает, что код может не получить доступ к приватному полю двумя различными способами:

1. Если приватный идентификатор не определен в области, в которой он используется. Это хорошая ранняя синтаксическая ошибка.
2. Если приватный идентификатор определен в области, в которой он используется, но его значение приватного имени используется для объекта, у которого нет приватного поля с этим приватным именем.

`Counter1` и `Counter2` попадают во вторую ситуацию. Закомментированный код в Листинге 18-2 несколько страниц назад, который пытался использовать `#value`, когда он вообще не входил в область видимости, был в первой ситуации.

Это все, что вам нужно знать, чтобы использовать приватные поля в своих классах. Если вы хотите глубже погрузиться в двухэтапный механизм (преобразование приватного идентификатора в приватное имя, затем использование приватного имени для поиска поля), см. Листинг 18-5: в нем показан код из Листинга 18-4, но *эмулирующий* приватные поля с использованием эквивалентов абстрактных операций, описанных в спецификации. (Эта эмуляция предназначена исключительно для того, чтобы помочь вам понять, как работают приватные поля; это не полифилирование или что-то близкое к нему.)

Листинг 18-5: Приватные поля в копиях классов с эмуляцией — private-fields-in-class-copiesemulated.js

```
// ==== Начало: Код для эмуляции операций спецификации ====

// Имена операций (NewPrivateName, PrivateFieldFind,
// PrivateFieldGet и PrivateFieldFind) и их параметры
// (description, P, O, value) взяты из спецификации.

// Создается новое приватное имя с заданным описанием.
// Приватные имена не описываются как объекты в спецификации, но
// использовать объект в этой эмуляции удобно.
function NewPrivateName(description) {
    return {description};
}

// Поиск заданного приватного поля в заданном объекте.
// P = приватное имя, O = объект.
function PrivateFieldFind(P, O) {
    const privateFieldValues = O["[[PrivateFieldValues]]"];
    const field = privateFieldValues.find(entry =>
                  entry["[[PrivateName]]"] === P);
    return field;
}

// Добавляется новое приватное поле к объекту (возможно только при
// первоначальном создании).
// P = приватное имя, O = объект, value = значение.
function PrivateFieldAdd(P, O, value) {
```

```
    if (PrivateFieldFind(P, O)) {
        throw new TypeError(`Field ${P.description} already defined for object`);
    }
    const field = {
        "[[PrivateName]]": P,
        "[[PrivateFieldValue]]": value
    };
    O["[[PrivateFieldValues]]"].push(field);
    return value;
}

// Возвращает значение заданного приватного поля данного объекта.
// P = приватное имя, O = объект.
function PrivateFieldGet(P, O) {
    const field = PrivateFieldFind(P, O);
    if (!field) {
        throw new TypeError(
            `Cannot read private member ${P.description} from an object ` +
            `whose class did not declare it`
        );
    }
    return field["[[PrivateFieldValue]]"];
}

// Задается значение заданного приватного поля данного объекта.
// P = приватное имя, O = объект, value = значение.
function PrivateFieldSet(P, O, value) {
    const field = PrivateFieldFind(P, O);
    if (!field) {
        throw new TypeError(
            `Cannot write private member ${P.description} to an object ` +
            `whose class did not declare it`
        );
    }
    field["[[PrivateFieldValue]]"] = value;
    return value;
}

// ==== Окончание: Код для эмуляции операций спецификации ====


// Это код из файла private-fields-in-class-copies.js, обновлен для
// использования приведенного выше кода "spec" для эмуляции приватных полей,
// чтобы примерно показать, как они работают "за кадром".
function makeCounterClass() {
    // Следующие две строки эмулируют то, что делает движок JavaScript,
    // когда он начинает обработку определения `class`:
    // 1. Создание среды приватного имени и привязка ее к определению
    //    `class` (в этом коде "ссылка" - это то, что код в классе
    //    закрывается над константой `privateNameEnvironment`).
    // 2. Выполнение части определения `#value;` один раз для каждого
    //    класса: Создание приватного имени для идентификатора приватного
    //    имени и сохранение его в среде приватного имени.
    const privateNameEnvironment = new Map();
    privateNameEnvironment.set("#value", NewPrivateName("#value"));
    return class Counter {
        // Исходный код: #value;
```

```
        constructor(start = 0) {
            // Эмулирует ту часть конструкции объекта, которая создает
            // Внутренний слот [[PrivateFieldValues]].
            this["[[PrivateFieldValues]]"] = [];

            // Эмулирует часть определения `#value;` для каждого объекта
            PrivateFieldAdd(privateNameEnvironment.get("#value"), this,
                undefined);

            // Исходный код: this.#value = start;
            PrivateFieldSet(privateNameEnvironment.get("#value"), this, start);
        }

        increment() {
            // Исходный код: return ++this.#value;
            const privateName = privateNameEnvironment.get("#value");
            const temp = PrivateFieldGet(privateName, this);
            return PrivateFieldSet(privateName, this, temp + 1);
        }

        get value() {
            // Исходный код: return this.#value;
            return PrivateFieldGet(privateNameEnvironment.get("#value"), this);
        }

        static show(counter) {
            // Исходный код: console.log(`counter.#value = ${counter.#value}`);
            const value =
                PrivateFieldGet(privateNameEnvironment.get("#value"), counter);
            console.log(`counter.#value = ${value}`);
        }
    };
}

const Counter1 = makeCounterClass();
const Counter2 = makeCounterClass();

const c = new Counter1();
c.increment();
Counter1.show(c);   // "counter.#value = 1"
Counter2.show(c);   // TypeError: попытка получить приватное поле
                    // для не-экземпляра
```

Приватные методы и акцессоры экземпляра

Предложение «Приватные методы и геттеры/сеттеры для классов JavaScript» основано на приватном механизме предложения полей классов для улучшения синтаксиса `class` с помощью приватных методов и акцессоров.

Приватные методы

Приватный метод можно определить простой постановкой символа # перед именем метода, см. Листинг 18-6.

Листинг 18-6: Приватные методы — private-methods.js

```
class Example {
    #value;
    #x;
    #y;

    constructor(x, y) {
        this.#x = x;
        this.#y = y;
        this.#calculate();
    }

    #calculate() {
        // Здесь представьте затратную операцию...
        this.#value = this.#x * this.#y;
    }

    get x() {
        return this.#x;
    }
    set x(x) {
        this.#x = x;
        this.#calculate();
    }

    get y() {
        return this.#y;
    }
    set y(y) {
        this.#y = y;
        this.#calculate();
    }

    get value() {
        return this.#value;
    }
}

const ex = new Example(21, 2);
console.log(`ex.value = ${ex.value}`); // 42
// Это станет синтаксической ошибкой:
// ex.#calculate();
```

Как и приватные поля, приватные методы должны быть определены в конструкции `class`. Их нельзя добавить в класс впоследствии, как методы прототипов (например, при помощи `MyClass.prototype.newMethod = function()` …). Так же, как и приватные поля, идентификатор метода (`#calculate`) находится в лексической области видимости.

Но, *в отличие* от приватных полей, приватные методы не хранятся в самом экземпляре (объекте), а используются совместно объектами (экземплярами класса). Это означает, что `#calculate` в предыдущем примере — это *не* то же самое, что определение приватного поля и назначение ему функции. Это создало бы новую функцию для каждого экземпляра вместо совместного использования экземплярами функции.

Можно предположить, что приватные методы будут помещены в объект-прототип для класса (`MyClass.prototype`), но это не то, как они используются экземплярами совместно.

Точный механизм совместного использования приватных методов экземплярами может незначительно измениться в период между написанием этой книги и принятием предложения, поэтому не будем слишком углубляться в эту тему; если вам интересно, посмотрите врезку. Ключевой вывод заключается в том, что они привязаны к экземплярам класса, но являются общими для всех экземпляров.

КАК ПРИВАТНЫЕ МЕТОДЫ СВЯЗАНЫ С ОБЪЕКТАМИ

Как и в случае с приватными полями, приватный идентификатор приватного метода имеет лексическую область видимости; его значением будет приватное имя. Но приватное имя — это не просто ключ в списке, как в случае с приватными полями. Приватное имя непосредственно содержит метод. Таким образом, движок JavaScript ищет приватный идентификатор в среде приватного имени, чтобы получить приватное имя, а затем получает метод непосредственно из этого приватного имени. Если бы это было все, что делал движок, вы могли бы получить приватный метод «из» любого объекта, выполнив `x.#privateMethod`, где x — это все, что угодно, даже если это не экземпляр класса, поскольку метод — это часть приватного имени, которое разрешает #privateMethod.

Это, мягко говоря, сбивало бы с толку и ограничивало дальнейшие усовершенствования языка, поэтому в спецификации есть явная проверка. Она позволяет убедиться, что экземпляр, «из» которого вы получаете метод, на самом деле предназначен для этого метода. У каждого приватного метода есть внутреннее поле (называемое [[Brand]]), представляющее бренд для класса, частью которого они являются, и у экземпляров есть список [[PrivateBrands]] для классов, к которым они принадлежат. Прежде чем разрешить коду получать приватный метод из экземпляра, движок JavaScript гарантирует, что [[Brand]] метода находится в списке [[PrivateBrands]] экземпляра. Конечно, это механизм спецификации; движки могут свободно оптимизировать при условии, что внешний результат соответствует спецификации.

Этот механизм может измениться до принятия предложения, поскольку существует несколько способов получения одного и того же конечного результата. Но тот факт, что приватные методы оказываются общими для всех экземпляров, не изменится.

Приватные методы могут быть вызваны с любым значением `this`, как и любой другой метод. `this` не обязательно ссылаться на экземпляр класса, к которому принадлежит метод. Например, можно использовать приватный метод в качестве обработчика событий DOM:

```
class Example {
    constructor(element) {
        element.addEventListener("click", this.#clickHandler);
}

    #clickHandler(event) {
        // Из-за того, как он был подключен, `this` здесь является элементом
        // DOM, к которому этот метод был присоединен в качестве обработчика
        // событий
        console.log(`Clicked, element's contents are: ${this.textContent}`);
}
}
```

Тем не менее обычно требуется привязать метод к экземпляру класса, чтобы иметь возможность получать доступ к свойствам, полям и другим методам экземпляра. (В противном случае можно определить приватный метод `static`, о котором вы узнаете позже в этой главе.) В случае с публичными методами вы, возможно, использовали шаблон, показанный ранее, для этих целей:

```
class Example {
    clickHandler = this.clickHandler.bind(this);

    constructor(element) {
        element.addEventListener("click", this.clickHandler);
    }

    clickHandler(event) {
        // ...
    }
}
```

Или сделали это в конструкторе:

```
class Example {
    constructor(element) {
        this.clickHandler = this.clickHandler.bind(this);
        element.addEventListener("click", this.clickHandler);
    }

    clickHandler(event) {
        // ...
    }
}
```

В обоих случаях они считывают `clickHandler` из экземпляра (который получает его из прототипа экземпляра, поскольку у экземпляра еще нет собственного свойства для него), создают для него связанную функцию и присваивают ее обратно как собственное свойство экземпляра.

Невозможно сделать это, используя одно и то же имя с приватными методами (у вас не может быть приватного поля и приватного метода с одинаковым именем; см. врезку на следующей странице; и нельзя назначить приватный метод, как если бы он был полем или свойством), но можно определить отдельное приватное поле для назначения связанной функции и использовать это:

```
class Example {
    #boundClickHandler = this.#clickHandler.bind(this);

    constructor(element) {
        element.addEventListener("click", this.#boundClickHandler);
    }

    #clickHandler(event) {
        // ...
    }
}
```

можно сделать это в конструкторе, хотя вам все равно нужно определение поля, поэтому наличие инициализатора для него, как в последнем примере, будет разумным:

```
class Example {
    #boundClickHandler;

    constructor(element) {
        this.#boundClickHandler = this.#clickHandler.bind(this);
        element.addEventListener("click", this. #boundClickHandler);
    }

    #clickHandler(event) {
        // ...
    }
}
```

Нет необходимости делать привязки в определении (или конструкторе), если вы собираетесь использовать связанную функцию только один раз, как в этом примере (просто привязка в вызове `addEventListener`). Но если вы собираетесь использовать эту связанную функцию более одного раза, привязка один раз и повторное использование связанной функции будет разумным решением.

ПРИВАТНЫЕ ИДЕНТИФИКАТОРЫ ДОЛЖНЫ БЫТЬ УНИКАЛЬНЫМИ В КЛАССЕ

Приватный метод и приватное поле не могут получить один и тот же идентификатор. То есть это недопустимо:

```
// Неверно
class Example {
    #a;

    #a() {// SyntaxError: Дублирующий приватный элемент
        // ...
    }
}
```

Хотя некоторые языки позволяют это делать. Применение такого подхода сбивает с толку. JavaScript этого не позволяет. Фактически все

приватные функциональные возможности, о которых вы узнаете в этой главе, следуют одному и тому же правилу: не может существовать двух разных элементов класса (например, приватного поля и приватного метода) с одним и тем же приватным идентификатором в одном и том же классе. Это происходит потому, что все приватные идентификаторы используют одну и ту же среду приватных имен «под капотом».

Приватные акцессоры

Приватные акцессоры работают точно так же, как приватные методы. Они вызываются при помощи приватного идентификатора, и только код внутри класса может получить к ним доступ:

```
class Example {
    #cappedValue;

    constructor(value) {
        // Сохранение значения с помощью сеттера из акцессора
        this.#value = value;
    }

    get #value() {
        return this.#cappedValue;
    }
    set #value(value) {
        this.#cappedValue = value.toUpperCase();
    }

    show() {
        console.log(`this.#value = ${this.#value}`);
    }

    update(value) {
        this.#value = value;
    }
}

const ex = new Example("a");
ex.show(); // "this.#value = A"
ex.update("b")
ex.show(); // "this.#value = B"
// ex.#value = "c"; // Будет синтаксической ошибкой, `#value` является
                    // приватным
```

Приватные акцессоры могут быть полезны для отладки, мониторинга изменений и т. д. Точно так же, как приватные методы (и с помощью того же механизма), функции-акцессоры — общие для всех экземпляров.

Публичные статические поля, приватные статические поля и приватные статические методы

Синтаксис `class` ES2015 содержит только одну функциональную возможность `static` — публичные методы `static` (добавленные в конструктор класса вместо данного экземплярам прототипа). Предложение «Статические возможности класса» основывается на предложениях о полях класса и приватных методах, чтобы завершить сетку объектов с помощью полей `static`, приватных полей `static` и приватных методов `static`.

Публичные статические поля

Публичные поля `static` создают свойства в конструкторе класса. Решение этой задачи — не редкость, но с синтаксисом `class` из ES2015 это приходилось делать постфактум. Например, если написан класс и у него есть несколько «стандартных» экземпляров, которые коду могло бы потребоваться повторно использовать, вы могли бы сделать их доступными в качестве свойств класса, назначив эти свойства после конструкции `class`:

```
class Thingy {
    constructor(label) {
        this.label = label;
    }
}
Thingy.standardThingy = new Thingy("A standard Thingy");
Thingy.anotherStandardThingy = new Thingy("Another standard Thingy");

console.log(Thingy.standardThingy.label);        // "A standard Thingy"
console.log(Thingy.anotherStandardThingy.label); // "Another standard
                                                 // Thingy"
```

С новым синтаксисом из предложения «Статические возможности класса» можно сделать это декларативно, используя ключевое слово `static`:

```
class Thingy {
    static standardThingy = new Thingy("A standard Thingy");
    static anotherStandardThingy = new Thingy("Another standard Thingy");

    constructor(label) {
        this.label = label;
    }
}

console.log(Thingy.standardThingy.label);        // "A standard Thingy"
console.log(Thingy.anotherStandardThingy.label); // "Another standard
                                                 // Thingy"
```

Публичные поля инициализируются в порядке исходного кода, поэтому инициализатор более позднего поля может ссылаться на более раннее поле:

```
class Thingy {
    static standardThingy = new Thingy("A standard Thingy");
    static anotherStandardThingy = new Thingy(
```

```
        Thingy.standardThingy.label.replace("A", "Another")
);

    constructor(label) {
        this.label = label;
    }
}

console.log(Thingy.standardThingy.label);        // "A standard Thingy"
console.log(Thingy.anotherStandardThingy.label); // "Another standard
                                                 // Thingy"
```

Однако инициализатор более *раннего* поля не может ссылаться на более *позднее* поле. Он получит значение `undefined`, поскольку свойство не существует (пока).

Благодаря новому синтаксису инициализатор статического поля получает доступ к приватным функциональным возможностям класса. Если бы вы добавили его в конструктор класса постфактум, у него не было бы такого доступа, потому что код для него не входил бы в область действия класса, к которому привязана среда приватного имени.

Приватные статические поля

Способ определения приватного статического поля, как ни странно, заключается в том, чтобы определить его как публичное статическое поле, но присвоить ему имя с помощью приватного идентификатора. Например, если вы писали класс, использующий экземпляры повторно на основе значения параметра, получаемого конструктором, вы могли бы хранить кэш известных экземпляров в приватном статическом поле:

```
class Example {
    static #cache = new WeakMap();

    constructor(thingy) {
        const cache = Example.#cache;
        const previous = cache.get(thingy);
        if (previous) {
            return previous;
        }
        cache.set(thingy, this);
    }
}

const obj1 = {};
const e1 = new Example(obj1);
const e1again = new Example(obj1);
console.log(e1 === e1again); // истина, был возвращен тот же экземпляр
const obj2 = {};
const e2 = new Example(obj2);
console.log(e1 === e2);      // ложь, был создан новый экземпляр
```

Приватные статические методы

Наконец, можно также определить приватные методы `static`, связанные с конструктором класса, а не с экземпляром, для приватных служебных методов, которым не нужно работать с `this`:

```
class Example {
    static #log(...msgs) {
        console.log(`${new Date().toISOString()}:`, ...msgs);
    }
    constructor(a) {
        Example.#log("Initializing instance, a =", a);
    }
}

const e = new Example("one");
// => "2018-12-20T14:03:12.302Z: Initializing instance, a = one"
```

Как и в случае с публичными статическими методами ES2015, вам необходимо получить к ним доступ через конструктор (то есть `Example.#log`, а не просто `#log`), поскольку они связаны с конструктором класса.

(Может появиться последующее предложение, дополняющее приватные методы автономными служебными функциями с областью видимости `class`[130], которые будут вызываться напрямую.)

ОТ СТАРЫХ ПРИВЫЧЕК К НОВЫМ

Перед вами несколько старых привычек, которые, возможно, вам захочется обновить, опираясь на новые функциональные возможности.

Используйте определения свойств вместо создания свойств в конструкторе (где это уместно)

Старая привычка: Всегда создавать свои свойства в конструкторе, потому что выбора не было:

```
class Example {
    constructor() {
        this.created = new Date();
    }
}
```

Новая привычка: Определите свойства класса, чтобы свести к минимуму изменения формы вашего класса, для краткости — и чтобы иметь возможность применять к ним декораторы по мере необходимости:

```
class Example {
    created = new Date();
}
```

[130] https://github.com/tc39/proposal-static-class-features/blob/ master/FOLLOWONS.md#lexical-declarations-in-class-bodies

Используйте приватные поля вместо префиксов (где это уместно)

Старая привычка: Использовать префиксы для псевдоприватных полей:

```
let nextExampleId = 1;
class Example {
    constructor() {
        this._id = ++nextExampleId;
    }
    get id() {
        return this._id;
    }
}
```

Новая привычка: Используйте истинные приватные поля:

```
let nextExampleId = 1;
class Example {
    #id = ++nextExampleId;
    get id() {
        return this.#id;
    }
}
```

Используйте приватные методы вместо функций вне класса для приватных операций

Старая привычка: Использовать функции, определенные вне класса, для обеспечения конфиденциальности, даже если требуется передать в них экземпляр:

```
// (Функция-оболочка не нужна, если это код модуля;
// обычно достаточно конфиденциальности модуля)
const Example = (() => {

    // (Представьте, что код использует много материала из Example,
    // а не только два элемента)
    function expensivePrivateCalculation(ex) {
        return ex._a + ex._b;
    }
    return class Example {
        constructor(a, b) {
            this._a = a;
            this._b = b;
            this._c = null;
        }
        get a() {
            return this._a;
        }
        set a(value) {
            this._a = value;
            this._c = null;
        }
```

```
        get b() {
            return this._b;
        }
        set b(value) {
            this._b = value;
            this._c = null;
        }
        get c() {
            if (this._c === null) {
                this._c = expensivePrivateCalculation(this);
            }
            return this._c;
        }
    };
})();
const ex = new Example(1, 2);
console.log(ex.c); // 3
```

Новая привычка: Используйте приватные методы (в этом примере также могут использоваться приватные поля, но я не использовал их, чтобы избежать смешения понятий):

```
class Example {
    constructor(a, b) {
        this._a = a;
        this._b = b;
        this._c = null;
    }
    // (Представьте, что код использует много материала из `this`,
    // а не только два элемента)
    #expensivePrivateCalculation() {
        return this._a + this._b;
    }
    get a() {
        return this._a;
    }
    set a(value) {
        this._a = value;
        this._c = null;
    }
    get b() {
        return this._b;
    }
    set b(value) {
        this._b = value;
        this._c = null;
    }
    get c() {
        if (this._c === null) {
            this._c = this.#expensivePrivateCalculation();
        }
        return this._c;
    }
}
const ex = new Example(1, 2);
console.log(ex.c); // 3
```

Тем не менее вы, вероятно, тоже захотите пойти дальше и использовать приватные поля:

```
class Example {
    #a;
    #b;
    #c = null;

    constructor(a, b) {
        this.#a = a;
        this.#b = b;
    }
    // (Представьте, что код использует много материала из `this`,
    // а не только два элемента)
    #expensivePrivateCalculation() {
        return this.#a + this.#b;
    }
    get a() {
        return this.#a;
    }
    set a(value) {
        this.#a = value;
        this.#c = null;
    }
    get b() {
        return this.#b;
    }
    set b(value) {
        this.#b = value;
        this.#c = null;
    }
    get c() {
        if (this.#c === null) {
            this.#c = this.#expensivePrivateCalculation();
        }
        return this.#c;
    }
}
const ex = new Example(1, 2);
console.log(ex.c); // 3
```

19

Взгляд в будущее…

СОДЕРЖАНИЕ ГЛАВЫ

- Оператор `await` верхнего уровня
- Слабые ссылки и обратные вызовы очистки
- Индексы соответствия RegExp
- Метод String.prototype.replaceAll
- Выражение `Atomics.asyncWait`
- Различные изменения синтаксиса
- Осуждаемые устаревшие возможности RegExp
- Благодарности!

Эта заключительная глава следует за предварительным рассмотрением предстоящих функций класса в главе 18 с предварительным просмотром того, что еще будет дальше (функции на этапе 3 на момент написания этой книги). Но то, что будет дальше, постоянно меняется. Будьте в курсе изменений, используя ресурсы, описанные в главе 1, в том числе, конечно же, сайт этой книги — `https://thenewtoys.dev`.

Изменения и дополнения охватывают широкий спектр от относительно незначительных (но удобных) настроек синтаксиса, таких как числовые разделители, до значительных дополнений, таких как `await` верхнего уровня.

Здесь рассматриваются большинство текущих функций, хотя некоторые из них уже рассматривались:

- Несколько новых служебных функциональных возможностей промисов были рассмотрены в главе 8.
- Объект `import.meta` рассматривался в главе 13.
- Вы только что узнали о публичных и приватных полях классов, а также о приватных методах и акцессорах (включая статические) в главе 18.

Некоторые читатели могут быть удивлены, не увидев в этой главе описания декораторов[131]. Они уже некоторое время находятся в процессе подготовки предложений, и та или иная версия предложения широко используется с помощью транспиляторов и TypeScript. Но предложение все еще находится на этапе 2 и претерпело три серьезных изменения, при этом первоначальная версия третьего («Статические декораторы»), вероятно, претерпит заметные изменения перед продвижением на следующие этапы. Так что, к сожалению, это слишком подвижная цель, чтобы рассматривать ее в этой главе.

> **ПРЕДЛОЖЕНИЯ НА ЭТАПЕ 3 МОГУТ ПОЛУЧИТЬ ИЗМЕНЕНИЯ**
>
> Помните, что функции на этапе 3 могут измениться до завершения процесса внедрения, а в редких случаях могут и вовсе не быть приняты. В этой главе вы узнаете о предложениях в том виде, в каком они были в начале 2020 года, но впоследствии они могут измениться. Обратитесь к репозиторию каждого предложения на GitHub (и https://thenewtoys.dev) для получения актуальной информации.

ОПЕРАТОР `AWAIT` ВЕРХНЕГО УРОВНЯ

Предложение await верхнего уровня[132], находящееся на этапе 3 в начале 2020 года, позволит использовать `await` на верхнем уровне модуля. Давайте взглянем.

Обзор и примеры использования

Возможно, вы помните из главы 9, что в функции `async` можно использовать метод `await`, чтобы дождаться выполнения промиса, прежде чем продолжить выполнение логики функции. Предложение await верхнего уровня переносит это на модули, позволяя коду модуля верхнего уровня ожидать выполнения промиса, прежде чем логика модуля верхнего уровня продолжит выполняться.

Давайте вкратце расскажем, как работает `await` в функции `async`. Рассмотрим функцию `fetchJSON`:

```
function fetchJSON(url, options) {
    const response = await fetch(url, options);
    if (!response.ok) {
        throw new Error("HTTP error " + response.status);
}
    return response.json();
}
```

Когда вы вызываете функцию `fetchJSON`, код в ней выполняется синхронно вплоть до вызова метода `fetch` включительно, который возвращает промис. Функция `fetchJSON` «ожидает» промис, приостановивший выполнение `fetchJSON`, возвращая новый

[131] https://github.com/tc39/proposal-decorators

[132] https://github.com/tc39/proposal-top-level-await

промис. Как только промис из `fetch` выполнен, логика в функции `fetchJSON` продолжается, в итоге выполняя возвращенный промис.

По сути, это работает так же, как и с `await` верхнего уровня в модулях: выполнение тела происходит до тех пор, пока `await` не станет промисом. Затем его выполнение приостанавливается, возобновляясь позже, когда промис будет выполнен. Грубо говоря, выполнение модуля возвращает промис своего экспорта, а не возвращает его напрямую — точно так же как функция `async` возвращает промис возвращаемого значения, а не возвращает его значение напрямую.

При использовании функции `fetchJSON` любой код, который хочет использовать возвращаемые им данные, должен дождаться выполнения промиса от `fetchJSON`. Концептуально это именно то, что происходит и с методом `await` верхнего уровня в модулях: все, что хочет использовать экспорт модуля, должно дождаться выполнения промисов модуля. В случае модулей то, что ожидает промис модуля, — это не ваш код (во всяком случае, не напрямую), а загрузчик модуля в вашей хост-среде (например, браузер или Node.js). Загрузчик модуля ожидает выполнения промиса модуля, прежде чем завершить загрузку любых зависящих от него модулей.

Когда вам может понадобиться `await` верхнего уровня?

Обобщенный ответ таков: в любое время экспорт вашего модуля бесполезен до тех пор, пока не будет доступно что-то, что модуль загружает асинхронно. Но давайте рассмотрим несколько особенностей.

Если модуль импортирует из динамически загружаемого модуля (модуль, который вы загружаете, «вызывая» метод `import()`, о котором вы узнали в главе 13), и если вам нужны импортируемые из него элементы, прежде чем можно будет использовать экспорт, можно применить `await` для ожидания промиса от `import()`. Предложение дает хороший пример этого: загрузка информации о локализации на основе текущего языка браузера. Вот этот пример (слегка измененный):

```
const strings = await import(`./i18n/${navigator.language}.js`);
```

Для этого примера предположим, что это появляется в модуле, экспортирующем функцию преобразования, которая, как ожидается, будет выполнять свою работу синхронно с данными в `strings`. Эту функцию невозможно использовать до тех пор, пока она не получит строки, поэтому есть смысл запретить полную загрузку модуля до тех пор, пока он их не получит.

Промис, ожидаемый модулем, не обязательно должен быть от `import()`; это может быть любой промис. Например, если строки получены из базы данных в Node.js модуль:

```
const strings = await getStringsFromDatabase(languageToUse);
```

Другим вариантом будет использование метода `import()` с инструкцией `await` для использования резервного модуля, если основной модуль недоступен. Продолжая наш предыдущий пример, модуль, импортирующий строки, может вернуться к языку по умолчанию, если файл локализации для `navigator.language` недоступен:

```
const strings = await
    import(`./i18n/${navigator.language}.js`)
.catch(() => import(`./i18n/${defaultLanguage}.js`));
```

или

```
let strings;
try {
    strings = await import(`./i18n/${navigator.language}.js`);
} catch {
    strings = await import(`./i18n/${defaultLanguage}.js`);
}
```

Использование `await` на верхнем уровне модуля приведет к двум важным последствиям:

- Это задерживает выполнение любых модулей, зависящих от выполнения инструкции, до тех пор пока ожидаемый промис не будет выполнен.
- Если ожидаемый промис отклоняется, и код не обрабатывает это отклонение, это все равно что получить не перехваченную синхронную ошибку в коде верхнего уровня вашего модуля — загрузка модуля завершается сбоем, а также загрузка модуля любого модуля, зависящего от него.

По этим причинам важно использовать `await` верхнего уровня только в тех случаях, когда экспорт модуля невозможно использовать до тех пор, пока не будет выполнен ожидаемый промис.

Пример

Давайте рассмотрим пример `await` верхнего уровня. В Листинге 19-1 есть модуль точки входа (`main.js`), который импортирует из `mod1` (`mod1.js`, Листинг 19-2), `mod2` (`mod2.js`, Листинг 19-3) и `mod3` (`mod3.js`, Листинг 19-4). Модуль `mod2` использует `await` верхнего уровня.

Листинг 19-1: Пример await верхнего уровня — main.js

```
import {one} from "./mod1.js";
import {two} from "./mod2.js";
import {three} from "./mod3.js";

console.log("main evaluation - begin");

console.log(one, two, three);

console.log("main evaluation - end");
```

Листинг 19-2: Пример await верхнего уровня — mod1.js

```
console.log("mod1 evaluation - begin");
export const one = "one";
console.log("mod1 evaluation - end");
```

Листинг 19-3: Пример await верхнего уровня — mod2.js

```
console.log("mod2 evaluation - begin");

// Функция искусственной задержки
function delay(ms, value) {
    return new Promise(resolve => setTimeout(() => {
        resolve(value);
    }, ms));
}

// export const two = "two";                // Не используется `await`
                                            // верхнего уровня
export const two = await delay(10, "two"); // Используется `await` верхнего
                                            // уровня
console.log("mod2 evaluation - end");
```

Листинг 19-4: Пример await верхнего уровня — mod3.js

```
console.log("mod3 evaluation - begin");
export const three = "three";
console.log("mod3 evaluation - end");
```

Поначалу, когда я писал эту главу, ни в Node.js, ни в одном браузере не было поддержки `await` верхнего уровня, и для запуска этого примера вам пришлось бы самостоятельно установить движок V8 (см. врезку). Но поскольку мы собирались поднажать, Node.js версии 14 вышла с поддержкой флага `--harmony-top-level-await`, так что можно использовать ее (возможно, к тому времени, когда вы читаете это, инструкция уже даже не будет за флагом). Однако знание того, как установить и использовать V8 напрямую, иногда все же полезно, поэтому, если вам интересно, ознакомьтесь со вставкой.

УСТАНОВКА V8

Если вы не можете найти браузер или версию Node.js, содержащие ультрасовременные функции, которые вы хотите опробовать и которые, как вы знаете, недавно были поддержаны в V8, вы можете установить V8 и использовать ее непосредственно. Один из простых способов установить V8 — это использовать инструмент «JavaScript (engine) Version Updater» (jsvu)[133]. Чтобы установить jsvu, обратитесь к инструкциям на странице проекта. По состоянию на начало 2020 года инструкции были здесь:

1. Откройте оболочку (командная строка/окно терминала).
2. Используйте команду `npm install jsvu -g` для установки `jsvu`.

[133] `https://github.com/GoogleChromeLabs/jsvu`

3. Обновите `PATH`, чтобы включить директорию, куда `jsvu` будет помещать исполняемые файлы для движков JavaScript:
   - В Windows: каталог — это `%USERPROFILE%\.jsvu`. Используйте пользовательский интерфейс Windows для обновления `PATH`, чтобы включить этот каталог. В Windows 10 это можно сделать, щелкнув значок **Search** (Поиск), набрав слово **environment** (Среда) и выбрав пункт **Edit** (Изменить) системные переменные среды в результатах локального поиска. В появившемся диалоговом окне **System Properties** (Свойства системы) нажмите кнопку **Environment Variables…** (Переменные среды…). В поле **System variables** (Системные переменные) дважды щелкните мышью на **PATH** (Путь). Нажмите кнопку **New** (Создать), чтобы добавить новую запись в список: `%USERPROFILE%\.jsvu`
   - В *nix: каталог — это `${HOME}/.jsvu`. Добавьте его в свой путь, добавив следующую строку в скрипт инициализации вашей оболочки (например, `~/.bashrc`): `export PATH="${HOME}/.jsvu:${PATH}"`
4. Если у вас открыта оболочка, закройте ее и откройте новую, чтобы она использовала новый `PATH`.

Теперь вы можете установить V8 через jsvu:

1. В оболочке выполните команду `jsvu --engines=v8`.
2. Поскольку вы запускаете его в первый раз, он попросит подтвердить вашу операционную систему; просто нажмите клавишу **Enter** (при условии, что он автоматически определил вашу систему правильно).
3. Если вы используете Windows: После установки V8 должен появиться файл **v8.cmd** в директории.jsvu. Введите `dir%USERPROFILE%\.jsvu\v8.cmd` для проверки, команда должна отобразить **v8.cmd**. Но с начала 2020 года существует специфичная для Windows проблема с jsvu, не позволяющая создать файл v8.cmd. Если его нет, скопируйте его из раздела загрузок главы или создайте его со следующим содержимым:

```
    @echo off
    "%USERPROFILE%\.jsvu\engines\v8\v8"
--snapshot_blob="%USERPROFILE%\.jsvu\engines\v8\snapshot_
blob.bin"%*
```

4. Обратите внимание, что здесь только две строки. Вторая строка, начинающаяся с `"%USERPROFILE%`, довольно длинная, и слова переносятся в тексте этой вставки.

Запустите файл **main.js** из примера. Если вы используете V8 непосредственно (см. вставку), запустите

```
v8 --module --harmony-top-level-await main.js
```

Если вы используете Node.js версии 14 или более позднюю, убедитесь, что у вас есть `package.json` с `"type": "module"`, как показано в главе 13 (файл есть в главе загрузки), и запустите

```
node main.js
```

или если вашей версии нужен флаг:

```
node --harmony-top-level-await main.js
```

Если вы используете браузер с поддержкой `await` верхнего уровня (возможно, за флагом, а возможно, и нет), включите файл **main.js** с помощью тега `script type="module"` в HTML-файле и запустите его.

Так будет выглядеть результат:

```
mod1 evaluation - begin
mod1 evaluation - end
mod2 evaluation - begin
mod3 evaluation - begin
mod3 evaluation - end
mod2 evaluation - end
main evaluation - begin
one two three
main evaluation - end
```

Загрузчик модулей в среде хоста извлекает и анализирует модули, создает их экземпляры (см. главу 13) и строит дерево зависимостей, а затем начинает их выполнять. Если ни один из модулей не использовал инструкцию `await`, поскольку дерево — это просто `main.js`, импортированный из трех модулей, каждый из них будет выполняться в том порядке, в котором они отображаются в импортах `main.js`. Но, глядя на вывод результата, можно заметить, что `mod2` *начинает* выполняться, а потом выполняется `mod3` прежде, чем закончится выполнение `mod2`. Вот что происходит:

- Загрузчик модуля оценивает код верхнего уровня в `mod1` (из которого импортирует первый модуль `main.js`); поскольку этот модуль не использует `await`, он проходит весь код до конца:

```
mod1 evaluation - begin
mod1 evaluation - end
```

- Загрузчик начинает выполнять код верхнего уровня для `mod2`:

```
mod2 evaluation - begin
```

- Когда код достигает строки `await`, он приостанавливается, ожидая выполнения промиса.
- В то же время, поскольку `mod3` не зависит от `mod2`, загрузчик выполняет его. В модуле `mod3` нет инструкций `await`, следовательно он выполняется полностью:

```
mod3 evaluation - begin
mod3 evaluation - end
```

- Когда промис `mod2` дожидается разрешения, выполнение `mod2` продолжается. Поскольку это была единственная инструкция `await`, код выполняется до конца модуля:

```
mod2 evaluation - end
```

- Поскольку все его зависимости теперь выполняются, загрузчик выполняет `main.js`, использующий то, что он импортировал из модулей:

```
main evaluation - begin
one two three
main evaluation - end
```

Давайте сравним это с тем, что произошло бы, если бы `mod2` не использовал инструкцию `await` верхнего уровня. Откройте файл **mod2.js** в редакторе и найдите следующие две строки:

```
// export const two = "two";                // Не используется `await`
                                            // верхнего уровня
export const two = await delay(10, "two"); // Используется `await` верхнего
                                            // уровня
```

Переместите маркер комментария из начала первой строки в начало второй, чтобы у вас получилось:

```
export const two = "two";                   // Не используется `await`
                                            // верхнего уровня
// export const two = await delay(10, "two"); // Используется `await`
                                            // верхнего уровня
```

Теперь модуль не использует `await` верхнего уровня. Снова запустите `main.js`. На этот раз вывод на экран будет следующим:

```
mod1 evaluation - begin
mod1 evaluation - end
mod2 evaluation - begin
mod2 evaluation - end
mod3 evaluation - begin
mod3 evaluation - end
main evaluation - begin
one two three
main evaluation - end
```

Обратите внимание, что модуль `mod2` был выполнен до конца, прежде чем началось выполнение модуля `mod3`, потому что `mod2` больше не ожидает выполнения промиса.

В исходном коде обратите внимание, что `mod1` мог экспортировать переменную `const`, которая получает свое значение из чего-то ожидаемого:

```
export const two = await delay(10, "two"); // Используется `await` верхнего
                                            // уровня
```

В этом примере он делает все это в одном операторе с объявлением `export`, но это не обязательно должно быть совместно. Например, при использовании динамического `import()` промис выполняется с помощью объекта пространства имен модуля загруженного модуля. Предположим, вашему модулю необходимо локально использовать функцию `example` из динамического модуля, а также необходимо повторно экспортировать экспорт `value` динамического модуля. Это можно было реализовать примерно так:

```
const {example, value} = await import("./dynamic.js");
export {value};
example("some", "arguments");
```

Или, конечно, можно просто сохранить ссылку на объект пространства имен, хотя часто лучше выбрать части, необходимые для включения встряхивания дерева (глава 13):

```
const dynamic = await import("./dynamic.js");
export const {value} = dynamic;
dynamic.example("some", "arguments");
```

Обработка ошибок

Ранее в разделе «Обзор и примеры использования» вы, возможно, задавались вопросом о том факте, что код не обрабатывал отклонение всех промисов. Например, одним из «резервных» примеров был такой:

```
let strings;
try {
    strings = await import(`./i18n/${navigator.language}.js`);
} catch {
    strings = await import(`./i18n/${defaultLanguage}.js`);
}
```

В этом примере есть код для обработки отклонения промиса от первого вызова `import()`, но ничего для обработки отклонения промиса от второго. Это может показаться тревожным сигналом, если вы помните фундаментальное правило промисов из главы 8:

> *Либо обработать ошибки, либо распространить цепочку промисов на своего вызывающего абонента.*

Нарушает ли пример кода это правило?

Нет, но было разумно задаться вопросом, так ли это. Это не так по той же причине, по которой аналогичный код в функции `async` не нарушает правило: промис модуля отклоняется, если ожидаемый промис отклоняется, и промис модуля возвращается «вызывающему» (в данном случае, загрузчику модуля); таким образом, правило «распространить цепочку промисов на вызывающего абонента» выполняется неявно. Точно так же, как функция `async` всегда возвращает промис, и любые не перехваченные ошибки в функцию, отклоняющую этот промис, асинхронные модули всегда возвращают промис (загрузчику модуля), и любые не перехваченные ошибки отклоняют этот промис. Загрузчик обрабатывает это, сообщая об ошибке через любой существующий для этих целей механизм, определенный хостом (часто какое-либо консольное сообщение) и помечая модуль как сбой (что приводит к сбою загрузки всех модулей, зависящих от него).

Что делать в предыдущем примере, если не загружается ни один модуль для `navigator.language` и `DefaultLanguage`? Загрузчику модуля не удается загрузить модуль, в котором находится код (и любые модули, зависящие от него), сообщая об этом в консоли (или аналогичным образом).

Так же, как и в случае с функциями `async`, потребуется явно передавать цепочку вызывающей стороне. Это будет неявно.

СЛАБЫЕ ССЫЛКИ И ОБРАТНЫЕ ВЫЗОВЫ ОЧИСТКИ

В этом разделе вы узнаете о предложении слабых ссылок — WeakRefs[134], которое привносит слабые ссылки и обратные вызовы очистки (также называемые *финализаторами*) в JavaScript. Предложение находится на этапе 3 по состоянию на начало 2020 года и активно добавляется в движки JavaScript.

(Краткое примечание: если прочитать документацию разработчика из предложения или другую документацию, основанную на нем, можно заметить сильное сходство между примерами и фрагментами кода в этом разделе и примерами в этой документации. Это потому, что я был слегка вовлечен в разработку этого предложения: я написал для него документацию разработчика.)

СЛАБЫЕ ССЫЛКИ И ОБРАТНЫЕ ВЫЗОВЫ ОЧИСТКИ — ЭТО РАСШИРЕННЫЕ ФУНКЦИИ

Как говорится в предложении, правильное использование слабых ссылок и обратных вызовов очистки требует тщательного обдумывания, и их лучше избегать, если это возможно. Сборка мусора сложна и будто происходит в странное время. Используйте их с осторожностью!

[134] `https://github.com/tc39/proposal-weakrefs`

Слабые ссылки

Обычно, когда есть ссылка на объект, этот объект будет сохраняться в памяти до тех пор, пока вы не удалите свою ссылку на него (если других нет). Это верно независимо от того, есть ли у вас прямая ссылка или косвенная ссылка через какой-то промежуточный объект или объекты. Это обычная ссылка на объект, также называемая *сильной ссылкой* (strong reference).

Использование WeakRef позволяет получить *слабую ссылку* на объект. Слабая ссылка не препятствует тому, чтобы объект был *удален* при *сборке мусора* (или же *утилизован*), если сборщик мусора движка JavaScript решит освободить память объекта.

Слабая ссылка создается с помощью конструктора `WeakRef`, передавая объект, на который вы хотите оставить слабую ссылку (его *цель*, также известную как *референт*):

```
const ref = new WeakRef({"some": "object"});
```

Если вам нужно использовать объект, вы можете получить сильную ссылку из WeakRef, используя его метод `deref` («разыменование»):

```
let obj = ref.deref();
if (obj) {
    // ...использование `obj`...
}
```

(А потом в какой-то момент вы позволите `obj` выйти из области видимости или назначите ему значение `undefined` или `null` и т. д., так что сильная ссылка будет удалена.)

Метод `deref` возвращает целевой объект, удерживаемый WeakRef, или значение `undefined`, если целевой объект был утилизован при сборке мусора.

Зачем вам нужна ссылка на объект, который может исчезнуть, если сборщик мусора решит восстановить состояние памяти? Один из основных вариантов использования таких конструкций — кэширование. Предположим, что у вашей страницы/приложения есть некоторые ресурсы данных, которые затратно извлекать из хранилища (и они не будут кэшироваться вашей хост-средой), а конкретные необходимые ресурсы варьируются в зависимости от срока службы страницы/приложения. При получении ресурса в первый раз можно сохранить его в кэше, используя слабую ссылку, чтобы, если он понадобится странице позже, она могла избежать затратного извлечения, если ресурс не был утилизован за это время. (Кэш в реальном мире, вероятно, будет надежно хранить наиболее часто или недавно используемые записи и использовать слабые ссылки для остальных.)

Другой вариант использования — обнаружение утечек ресурсов с помощью комбинации слабых ссылок и обратных вызовов очистки; вы узнаете об этом в следующем разделе.

Давайте рассмотрим базовый пример использования WeakRef; см. Листинг 19-5. Здесь создается буфер ArrayBuffer, занимающий 100 миллионов байт, и сохраняется слабая ссылка на него. Затем периодически выделяются другие экземпляры ArrayBuffer, сохраняются сильные ссылки на них, и осуществляется проверка, был ли востребован *слабо удерживаемый* ArrayBuffer (тот, который удерживается через WeakRef). В конце концов, сборщик мусора движка JavaScript, скорее всего, решит вернуть большой начальный буфер ArrayBuffer, удерживаемый лишь слабо.

Листинг 19-5: Пример WeakRef — weakref-example.js

```
const firstSize = 100000000;
console.log(`main:  Allocating ${firstSize} bytes of data to hold weakly...`);
let data = new ArrayBuffer(firstSize);
let ref = new WeakRef(data);
data = null; // Удаляется сильная ссылка, остается только слабая
let moreData = [];
let counter = 0;
let size = 50000;

setTimeout(tick, 10);

function tick() {
    ++counter;
    if (size < 100000000) {
        size *= 10;
    }
    console.log();
    console.log(`tick(${counter}): Allocating ${size} bytes more data...`);
    moreData.push(new ArrayBuffer(size));
    console.log(`tick(${counter}): Getting the weakly held data...`);
    const data = ref.deref();
    if (data) {
        console.log(`tick(${counter}): weakly held data still in memory.`);
        // Этот оператор `if` - просто проверка на разумность,
        // чтобы избежать бесконечного зацикливания,
        // если слабо удерживаемые данные не будут утилизованы.
        if (counter < 100) {
            setTimeout(tick, 10);
        } else {
            console.log(`tick(${counter}): Giving up`);
        }
    } else {
        console.log(`tick(${counter}): weakly held data was garbage
                    collected.`);
    }
}
```

В Node.js v14 есть поддержка слабых ссылок за флагом (к тому времени, когда вы прочитаете это, они могут быть и не за флагом), и они поддерживаются в Firefox версии Nightly с начала 2020 года. Чтобы запустить пример в Node.js v14:

```
node --harmony-weak-refs weakref-example.js
```

Или, если у вас установлена отдельная версия V8 (см. вставку «Установка версии V8» в разделе «Оператор await верхнего уровня» ранее в этой главе), можете запустить ее в версии V8:

```
v8 --harmony-weak-refs weakref-example.js
```

Выходные данные будут варьироваться в зависимости от вашей системы, но выглядеть они будут примерно так:

```
main:    Allocating 100000000 bytes of data to hold weakly...

tick(1): Allocating 500000 bytes more data...
tick(1): Getting the weakly held data...
tick(1): weakly held data still in memory.

tick(2): Allocating 5000000 bytes more data...
tick(2): Getting the weakly held data...
tick(2): weakly held data still in memory.

tick(3): Allocating 50000000 bytes more data...
tick(3): Getting the weakly held data...
tick(3): weakly held data still in memory.

tick(4): Allocating 500000000 bytes more data...
tick(4): Getting the weakly held data...
tick(4): weakly held data still in memory.

tick(5): Allocating 500000000 bytes more data...
tick(5): Getting the weakly held data...
tick(5): weakly held data was garbage collected.
```

В этом выводе можно заметить, что между четвертым и пятым вызовами `tick` V8 утилизовал слабо удерживаемый буфер. (Он может работать намного дольше, чем в этом примере; выполнение кода довольно сильно варьируется.) Вот несколько заключительных замечаний по слабым ссылкам:

- Если ваш код только что создал слабую ссылку для целевого объекта или получил целевой объект из метода `deref` WeakRef, этот целевой объект не будет утилизован до конца текущего задания JavaScript[135] (включая любые задания реакции на промисы, которые выполняются в конце задания скрипта). То есть вы можете только «видеть», как объект утилизуется между витками цикла событий. Это делается в первую очередь для того, чтобы избежать очевидного поведения сборщика мусора любого движка JavaScript в коде, потому что, если бы это было так, люди писали бы код, полагающийся на это поведение, а оно потерпело бы неудачу при изменении поведения сборщика мусора. (Сбор мусора — сложная проблема. Разработчики движка JavaScript постоянно совершенствуют и совершенствуют его работу.) Если вам было интересно, использование в предыдущем примере функции `setTimeout` связано с этим аспектом WeakRefs. Если бы в примере просто вызывался `tick` в цикле, слабо удерживаемый `ArrayBuffer` никогда не был бы утилизован.
- Если у нескольких слабых ссылок одна и та же цель, они согласуются друг с другом. Результат вызова метода `deref` для одной из этих ссылок будет

[135] Возможно, вы помните из главы 8, что задание — это единица работы, которую поток будет выполнять от начала до конца, не выполняя ничего другого, и что существуют задания скриптов (например, начальное выполнение скрипта, обратный вызов таймера, обратный вызов события) и задания промисов (обратные вызовы выполнения промисов, отклонение и обработчики).

совпадать с результатом вызова `deref` для другой (в том же задании) — вы не получите от одной целевой объект, а от другой значение `undefined`.
- Если цель слабой ссылки также находится в объекте `FinalizationRegistry` (о котором вы узнаете немного позже), цель слабой ссылки очищается *до* или одновременно с вызовом любого обратного вызова очистки, связанного с реестром.
- Цель слабой ссылки невозможно изменить. Это всегда будет только исходный целевой объект или значение `undefined`, когда этот целевой объект был утилизован.
- Слабая ссылка может никогда не возвращать значение `undefined` из метода `deref`, даже если у цели нет сильной привязки, потому что сборщик мусора может не принять решение об утилизации объекта.

Обратные вызовы очистки

Еще предложение слабых ссылок предоставляет *обратные вызовы очистки*, также известные как *финализаторы*.

Обратный вызов очистки — это функция, которую сборщик мусора может вызвать, когда объект был утилизирован. В отличие от финализаторов, деструкторов и подобных им инструментов в некоторых других языках, в JavaScript обратный вызов очистки не получает ссылку на объект, который утилизируется или был утилизирован. На самом деле, насколько можно судить по коду, объект *уже* был утилизирован до того, как был вызван обратный вызов очистки (и вполне вероятно, что так оно и было на самом деле, но это тонкость реализации сборщика мусора). Проектирование его таким образом позволяет избежать проблемы, с которой сталкивались некоторые другие среды, когда объект, который был недоступен (ваш код не имеет возможности получить к нему доступ), снова становится доступным (потому что финализатор получает ссылку на объект и сохраняет его где-то). Подход JavaScript (не предоставляющий объект для обратных вызовов очистки) упрощает рассуждения об обратных вызовах очистки и делает эти процессы более гибкими для разработчиков движка.

Зачем нужен обратный вызов очистки?

Один из вариантов использования — освобождение других объектов, связанных с восстановленным объектом. Например, если у вас есть кэш слабых ссылок, слабо удерживающих объекты, и один из этих объектов утилизирован, в кэше все еще есть запись для этого объекта в памяти: ключ, слабая ссылка и, возможно, какая-то другая информация. Вы можете использовать обратный вызов очистки, чтобы освободить эту запись кэша, когда связанный с ней целевой объект будет утилизирован. Однако этот вариант использования не ограничивается слабыми ссылками. У вас могут быть объекты, которые больше не нужны, если другой объект будет утилизирован. Например, объект Wasm, который вы могли бы освободить, если соответствующий объект JavaScript удален при сборке мусора, или кросс-воркер прокси, где есть прокси в одном потоке (обычно основном) для объекта в потоке воркера. Если прокси утилизован при сборке мусора, объект воркера может быть освобожден.

Другой вариант использования — обнаружение утечек ресурсов и сообщение об этом. Предположим, у вас есть класс, представляющий открытый файл или подключение к базе данных или что-то подобное. Разработчики, использующие класс, должны вызывать его

метод `close` при завершении работы с ним. Этот метод освобождает дескриптор файла или закрывает соединение с базой данных и т. д. Если пользователь не вызывает метод `close`, а просто освобождает объект: ваш класс не освободит дескриптор файла или основное соединение с базой данных. В долго работающей программе это может в итоге вызвать проблему, когда в процессе заканчиваются файловые дескрипторы или база данных достигает предела одновременных подключений от одного и того же клиента. Вы можете использовать обратный вызов очистки, чтобы отправить разработчику предупреждающее сообщение о том, что вызвать `close` не удалось. Обратный вызов очистки также может освободить дескриптор файла или соединение с базой данных, но его основной целью будет предупреждение разработчика об ошибке, чтобы он мог исправить свой код, указав ему вызвать метод `close`.

Мы скоро вернемся к обоим этим вариантам использования, в том числе к тому, почему не стоит просто использовать обратный вызов очистки для освобождения внешних ресурсов во втором примере. А пока давайте посмотрим, как на самом деле используются обратные вызовы очистки.

Чтобы запросить обратные вызовы очистки, создается *реестр финализации* с помощью конструктора `FinalizationRegistry`, и ему передается функция, которую вы хотите вызвать:

```
const registry = new FinalizationRegistry(heldValue => {
    // ... здесь реализуйте очистку...
});
```

Затем объекты регистрируются в реестре, используя метод `register`, чтобы получать для них обратные вызовы. Для каждого целевого объекта передается объект и *сохраненное значение* для него (`"some value"` в этом примере), например:

```
registry.register(theObject, "some value");
```

Метод `register` принимает три аргумента:

- `target`: объект, для которого вы хотите выполнить обратный вызов финализации. Реестр не содержит сильной ссылки на объект, так как это предотвратило бы сборку мусора.
- `heldValue`: значение, которое будет храниться в реестре, чтобы предоставить его обратному вызову очистки, если целевой объект будет утилизирован. Это может быть примитив или объект. Если вы ничего не указываете, используется значение `undefined`.
- `unregisterToken`: (не показано в предыдущем примере). Необязательный объект, который можно использовать позже для отмены регистрации целевого объекта, если обратный вызов очистки для него больше не требуется. Подробнее об этом чуть позже.

После того как целевой объект зарегистрирован, если он будет утилизирован, ваш обратный вызов очистки может быть вызван в какой-то момент в будущем с сохраненным значением, которое вы для него указали. Именно там выполняется очистка — возможно, с использованием информации из сохраненного значения. Сохраненное значение (также называемое «памяткой») может быть любым желаемым значением — примитивом

или объектом, даже `undefined`. Если сохраненное значение представлено объектом, реестр сохраняет на него *сильную* ссылку (чтобы позже передать ее в ваш обратный вызов очистки), что означает, что сохраненное значение не будет утилизировано, если целевой объект не будет утилизирован (что удаляет запись для него из реестра) или вы отмените регистрацию целевого объекта.

Если потребуется отменить регистрацию объекта позже, нужно передать третий аргумент в метод `register` — маркер отмены регистрации, упомянутый ранее. Обычно в качестве маркера отмены регистрации используется сам объект, и это нормально. Когда обратный вызов очистки для объекта больше не нужен, вызовите метод `unregister` с токеном отмены регистрации. Вот пример использования самого целевого объекта:

```
registry.register(theObject, "some value", theObject);
// ...некоторое время спустя, если `theObject` больше не нужен...
registry.unregister(theObject);
```

Однако это не обязательно должен быть целевой объект — может быть и другой:

```
registry.register(theObject, "some value", tokenObject);
// ...некоторое время спустя, если `theObject` больше не нужен...
registry.unregister(tokenObject);
```

Реестр хранит только слабую ссылку на маркер отмены регистрации — не в последнюю очередь потому, что это может быть сам целевой объект.

Давайте рассмотрим пример из предложения (слегка измененный) и гипотетический класс `FileStream`:

```
class FileStream {
    static #cleanUp(fileName) {
        console.error(`File leaked: ${fileName}!`);
    }

    static #finalizationGroup = new FinalizationRegistry(FileStream.#cleanUp);

    #file;

    constructor(fileName) {
        const file = this.#file = File.open(fileName);
        FileStream.#finalizationGroup.register(this, fileName, this);
        // ...неотложно запускает асинхронное чтение содержимого файла...
    }

    close() {
        FileStream.#finalizationGroup.unregister(this);
        File.close(this.#file);
        // ...другая очистка...
    }

    async *[Symbol.iterator]() {
        // ...извлекаются данные из файла...
    }
}
```

Здесь можно увидеть все части, выполняющие задачи: создание `FinalizationRegistry`, добавление в него объектов, ответы на обратные вызовы очистки и отмена регистрации объектов из реестра, когда они закрыты явно.

Это пример второго варианта использования, о котором рассказывалось ранее. Если пользователь класса `FileStream` не вызывает метод `close`, базовый объект `File` не закрывается, что может привести к утечке дескриптора файла. Таким образом, обратный вызов очистки регистрирует имя файла, чтобы предупредить разработчика, что он не был закрыт, чтобы разработчик мог исправить свой код.

Глядя на это, вы можете подумать: «Почему бы просто не использовать обратный вызов очистки для освобождения этих ресурсов всегда? Зачем вообще нужен метод `close`?»

Ответ таков: потому что вызов на очистку может не поступить никогда, а если и поступит, то намного позже, чем ожидается. Ранее я сказал: «…это функция, которую сборщик мусора *может* вызвать, когда объект был утилизирован…», а не «… это функция, которую сборщик мусора *будет* вызывать, когда объект был утилизирован…». Нет никакой гарантии, что сборщик мусора вызовет обратный вызов очистки в любой реализации. Из предложения:

> *Предлагаемая спецификация позволяет соответствующим реализациям пропускать обратные вызовы завершения по любой причине или без причины.*

Это означает, что нельзя использовать обратные вызовы очистки для управления внешними ресурсами. С точки зрения этого второго варианта использования (и примера `FileStream`), вашему файлу или базам данных API нужен метод `close`, и разработчикам необходимо вызывать его. Причина наличия обратного вызова очистки в этом API заключается в том, что если разработчик работает на платформе с обратными вызовами очистки и выпускает объект из API без вызова `close`, то API может предупредить разработчика, что он не вызывал метод `close`. Этот код может быть запущен позже на платформе, где обратные вызовы очистки не вызываются.

Есть хорошие признаки того, что основные движки JavaScript будут вызывать обратные вызовы очистки в обычных обстоятельствах. Однако есть пара обстоятельств, при которых они вряд ли это сделают, даже если движки начнут делать это в обычных условиях:

- Если среда JavaScript резко прерывает работу (например, при закрытии окна или вкладки в браузере). В большинстве случаев это сделало бы любую выполненную кодом очистку излишней.
- Если объект `FinalizationRegistry`, с которым связан обратный вызов, освобожден вашим кодом (у вас больше нет ссылки на него). После освобождения ссылки на реестр нет особого смысла хранить ее в памяти и выполнять обратные вызовы. Если они вам нужны, не освобождайте реестр.

Но если обратные вызовы очистки могут не произойти, как насчет кэшей, использующих слабые ссылки? Если вы не можете полагаться на обратные вызовы очистки для удаления записей утилизированных объектов, следует ли вам выполнить какое-то инкрементное сканирование записей, для которых метод `deref` возвращает значение `undefined`?

Руководство от людей, стоящих за этим предложением, гласит: «Нет, не делайте этого». Это излишне усложняет код, при относительно небольшом выигрыше (освобождая

немного больше памяти), а для активного кэша вы, скорее всего, обнаружите и замените эти записи органично в любом случае (когда ресурс для них запрашивается снова).

Давайте посмотрим на обратные вызовы очистки в действии, см. Листинг 19-6.

Листинг 19-6: Пример обратного вызова очистки — cleanup-callback-example.js

```
let stop = false;
const registry = new FinalizationRegistry(heldValue => {
    console.log(`Object for '${heldValue}' has been reclaimed`);
    stop = true;
});
const firstSize = 100000000;
console.log(`main:    Allocating ${firstSize} bytes of data to hold weakly...`);
let data = new ArrayBuffer(firstSize);
registry.register(data, "data", data);
data = null; // Высвобождение ссылки
let moreData = [];
let counter = 0;
let size = 50000;

setTimeout(tick, 10);

function tick() {
    if (stop) {
        return;
    }
    ++counter;
    if (size < 100000000) {
        size *= 10;
    }
    console.log();
    console.log(`tick(${counter}): Allocating ${size} bytes more data...`);
    moreData.push(new ArrayBuffer(size));
    // Этот оператор `if` - просто проверка на разумность,
    // чтобы избежать бесконечного зацикливания,
    // если слабо удерживаемые данные никогда не восстанавливаются
    // или хост никогда не вызывает обратный вызов очистки.
    if (counter < 100) {
        setTimeout(tick, 10);
    }
}
```

Node.js V14 и V8 сами по себе содержат слабые ссылки за флагом, но в зависимости от вашей версии они могут содержать более старую семантику, а не новую. Предложение изменилось немного позже в процессе. Вы можете запустить пример следующим образом:

```
# V8:
v8 --harmony-weak-refs cleanup-callback-example.js
# Node:
node --harmony-weak-refs cleanup-callback-example.js
```

Если выдается сообщение об ошибке, в котором говорится, что объект `FinalizationRegistry` не определен, или если вы видите сообщение «Object for '[object FinalizationRegistry Cleanup Iterator]' has been reclaimed» вместо «Object for 'data' has

been reclaimed» при вызове обратного вызова, запустите вместо этого код из файла **cleanup-callback-example-oldersemantics.js**.

При запуске код выводит что-то похожее на:

```
main:    Allocating 100000000 bytes of data to hold weakly...

tick(1): Allocating 500000 bytes more data...

tick(2): Allocating 5000000 bytes more data...

tick(3): Allocating 50000000 bytes more data...

tick(4): Allocating 500000000 bytes more data...
Finalizer called
Object for 'data' has been reclaimed
```

Этот вывод показывает, что объект был удален при сборке мусора после четвертого вызова таймера.

Наконец, у объектов `FinalizationRegistry` есть необязательный метод `cleanupSome`. Его можно вызвать для запуска обратных вызовов для выбранного реализацией количества объектов в реестре, которые были утилизированы, но чьи обратные вызовы еще не были вызваны:

```
registry.cleanupSome?.();
```

Обычно эта функция не вызывается. Предоставьте сборщику мусора движка JavaScript выполнять очистку по мере необходимости. Эта функция существует в первую очередь для поддержки долго работающего кода, который не поддается циклу событий, что куда чаще встречается в WebAssembly, чем в обычном коде JavaScript.

Вы можете предоставить методу `cleanupSome` обратный вызов, отличный от зарегистрированного в объекте реестра, чтобы переопределить его только для этих обратных вызовов очистки:

```
registry.cleanupSome?.(heldValue => {
    // ...
});
```

Даже если есть ожидающие обратные вызовы, номер, инициируемый вызовом метода `cleanupSome`, определяется реализацией. Реализация может не выполнять ни один из них, может выполнять все ожидающие выполнения вызовы или что-то среднее.

Обратите внимание, что в предыдущих примерах используется синтаксис опциональной цепочки, о котором вы узнали в главе 17. Так что, если реализация не определяет `cleanupSome`, вызов пропускается без ошибок.

ИНДЕКСЫ СООТВЕТСТВИЯ REGEXP

Массив совпадений, возвращаемый выражением `RegExp.prototype.exec`, содержит в себе свойство `index`. Оно указывает индекс в строке, в которой произошло совпадение, но не указывает, где возникли группы захвата. Индексы соответствия регулярных

выражений[136] изменяют это положение, добавляя свойство `indices`. Свойство `indices` содержит массив массивов `[start, end]`. Первая запись предназначена для общего совпадения, последующие — для групп захвата.

Взгляните на пример:

```
const rex = /(\w+) (\d+)/;
const str = "==> Testing 123";
const match = rex.exec(str);
for (const [start, end] of match.indices) {
    console.log(`[${start}, ${end}]: "${str.substring(start, end)}"`);
}
```

Это регулярное выражение ищет серию символов или «слово», за которыми следуют пробел и серия цифр. Захватываются как слово, так и цифры. В реализации с индексами соответствия результат этого примера выглядит следующим образом:

```
[4, 15]: "Testing 123"
[4, 11]: "Testing"
[12, 15]: "123"
```

Новая функция также поддерживает именованные группы захвата. Возможно, вы помните из главы 15, что, если группы захвата поименованы в выражении, массив `match` получит свойство объекта с именем `groups` с содержимым именованных групп. Это предложение делает то же самое, предоставляя индексы содержимого группы с выражением `match.indices.groups`:

```
const rex = /(?<word>\w+) (?<num>\d+)/;
const str = "==> Testing 123";
const match = rex.exec(str);
for (const key of Object.keys(match.groups)) {
    const [start, end] = match.indices.groups[key];
    console.log(
        `Group "${key}" - [${start}, ${end}]: "${str.substring(start, end)}"`
    );
}
```

В этом примере даны имена двум группам захвата из предыдущего примера и используются выражения `match.groups` и `match.indexes.groups` для получения информации об этих именованных группах. Так будет выглядеть результат:

```
Group "word" - [4, 11]: "Testing"
Group "num" - [12, 15]: "123"
```

Массив совпадений стал довольно богатым объектом с появлением именованных групп захвата и массивов индексов. Вот полный массив `match` (включая его свойства, отличные от массива) для предыдущего примера в формате псевдо-JSON («псевдо», потому что в нем используются квадратные скобки, указывающие на массивы, но также есть пары имя: значение для дополнительных свойств массива, отличных от массива):

[136] `https://github.com/tc39/proposal-regexp-match-indices`

```
[
    "Testing 123",
    "Testing",
    "123",
    index: 4,
    input: "==> Testing 123",
    "groups": {
        word: "Testing",
        num: "123"
    },
    "indices": [
        [4, 15],
        [4, 11],
        [12, 15],
        "groups": {
            "word": [4, 11],
            "num": [12, 15]
        }
    ]
]
```

МЕТОД STRING.PROTOTYPE.REPLACEALL

В течение многих лет люди, не знакомые с методом JavaScript `String.prototype.replace`, совершали одну и ту же ошибку — не понимали, что он заменяет *первое* совпадение только в том случае, если вы передаете ему строку или не глобальное регулярное выражение. Например:

```
console.log("a a a".replace("a", "b")); // "b a a", not "b b b"
```

Чтобы заменить все совпадения, необходимо передать регулярное выражение с глобальным флагом:

```
console.log("a a a".replace(/a/g, "b")); // "b b b"
```

Это не так уж плохо, когда текст, который вы хотите изменить, — это текст, который вы вводите буквально, но, если он вводится пользователем или поступает подобным образом, вам придется экранировать символы, представляющие особое значение в регулярных выражениях (и для этого нет встроенной функции, несмотря на попытки добавить ее несколько лет назад).

Предложение `replaceAll`[137] значительно упрощает задачу, заменяя все совпадения:

```
console.log("a a a".replaceAll("a", "b")); // "b b b"
```

Метод `replaceAll` («заменить все») ведет себя идентично методу `replace` («заменить»), за исключением двух аспектов:

- Если вы передадите ему строку в качестве аргумента поиска, он заменит все совпадения, а не только первое — в этом весь смысл `replaceAll`!

[137] `https://github.com/tc39/proposal-string-replaceall`

- Если вы передаете регулярное выражение *без* глобального флага, оно выдает ошибку. Это делается для того, чтобы избежать путаницы. Означает ли отсутствие глобального флага «*не* заменять все»? Или флаг просто игнорируется? Ответ этого предложения заключается в том, чтобы выбросить ошибку.

Методы `replaceAll` и `replace` делают одно и то же, если вы передаете им регулярное выражение с глобальным флагом.

ВЫРАЖЕНИЕ `ATOMICS.ASYNCWAIT`

В главе 16 вы узнали о совместно используемой памяти и объекте `Atomics`, включая его метод `wait`. Кратко резюмируя, можно использовать метод `Atomics.wait` для синхронного ожидания местоположения в совместно используемой памяти до тех пор, пока не придет «уведомление» (через `Atomics.notify`) от другого потока. Можно использовать `Atomics.wait` в потоках воркеров, но обычно не в главном потоке (например, в главных потоках браузера или Node.js), поскольку они должны быть неблокирующимися.

Предложение `Atomics.asyncWait` на этапе 3[138] позволяет ожидать уведомления в ячейке общей памяти без блокировки, используя промис. При вызове `Atomics.asyncWait`, вместо получения возвращаемого строкового значения, как в случае `Atomics.wait`, вы получаете промис, который будет выполнен с помощью строки. Возможные выполняемые строки совпадают со строками, возвращаемыми `Atomics.wait`:

- `"ok"`, если поток был «приостановлен», а затем возобновлен (вместо ожидания тайм-аута);
- `"timed-out"`, если поток был «приостановлен» и возобновлен по истечении тайм-аута;
- `"not-equal"`, если поток не был остановлен, поскольку значение в массиве не совпало с заданным.

Слово «приостановлен» в этом списке заключено в кавычки, потому что основной поток никогда не приостанавливается буквально, как потоки воркеров, с помощью `Atomics.wait` — просто промис остается невыполненным в течение некоторого периода времени. Промис от `asyncWait` никогда не отклоняется: он всегда выполняется с одной из трех перечисленных строк.

Например, если есть буфер `SharedArrayBuffer` под названием `sharedArray`, основной поток мог бы ждать уведомления по индексу `index`, примерно так:

```
Atomics.asyncWait(sharedArray, index, 42, 30000)
.then(result => {
    // ...здесь результат `result` будет "ok", "timed-out" или "not-equal"...
});
```

Если значение в `sharedMemory[index]` не равно `42` на момент проверки движком JavaScript, обратный вызов выполнения увидит строку `"not-equal"`. В противном

[138] `https://github.com/tc39/proposal-atomics-wait-async`

случае движок будет ждать уведомления, прежде чем выполнить промис. Если до истечения тайм-аута (в данном примере 30000 миллисекунд) не появится уведомления, промис будет выполнен со строкой `"timed-out"`. Если уведомление поступит, промис выполняется со строкой `"ok"`.

Как обычно в случае с совместно используемой памятью и потоками, здесь водятся драконы. Подробности об опасностях и ловушках см. главу 16.

РАЗЛИЧНЫЕ ХИТРОСТИ СИНТАКСИСА

В этом разделе вы узнаете о некоторых незначительных изменениях синтаксиса, которые вносятся различными предложениями.

Числовые разделители

Иногда числовой литерал может быть немного трудным для чтения:

```
const x = 10000000;
```

Быстро — это 100 тысяч? Миллион? 10 миллионов? 100 миллионов?

При написании чисел для чтения людьми мы склонны каким-то образом группировать цифры. Например, в некоторых культурах они помещаются в группы по три с запятыми между ними: 10,000,000.

Предложение о числовых разделителях[139] (в настоящее время находится на этапе 3) позволяет использовать символы подчеркивания в качестве разделителей в числовых литералах:

```
const x = 10_000_000;
```

Это значительно облегчает понимание того, что это число составляет 10 миллионов.

Вы можете использовать числовые разделители в десятичных литералах (как в целочисленной части, так и в дробной части), двоичных литералах, шестнадцатеричных литералах и современных восьмеричных литералах.

Разделители разрешено размещать где угодно, кроме:

- Сразу после базового префикса числа:
  Недопустимо: `0x_AAAA_BBBB`
  Допустимо: `0xAAAA_BBBB`
- Непосредственно рядом с десятичной точкой:
  Недопустимо: `12_345_.733_444`
  Недопустимо: `12_345._733_444`
  Допустимо: `12_345.733_444`
- В конце числа, после последней цифры:
  Недопустимо: `12_345_`
  Допустимо: `12_345`

[139] `https://github.com/tc39/proposal-numeric-separator`

- Рядом с другим разделителем:
  Недопустимо: `10__123_456`
  Допустимо: `10_123_456`

Вы также не можете *начинать* число с подчеркивания, например: `_1234`. Это идентификатор, а не число.

Поддержка Hashbang

Разрешать файлу JavaScript начинаться с «hashbang» — обычное явление для хостов с интерфейсом в виде командной строки (вроде Node.js), например так:

```
#!/home/tjc/bin/node/bin/node
console.log("Hello from Node.js command line script");
```

Технически это был недопустимый синтаксис в соответствии со спецификацией. Поэтому спецификация обновлена[140], чтобы разрешить это (в настоящее время на этапе 3).

ОСУЖДАЕМЫЕ УСТАРЕВШИЕ ВОЗМОЖНОСТИ REGEXP

Предложение на этапе 3[141] стандартизирует, но официально осуждает некоторые давно поддерживаемые «функциональные возможности» регулярных выражений, которых нет в спецификации. Например, группы захвата доступны не только в объекте сопоставления, который вы получаете из `exec`, но также и в качестве свойств самой функции `RegExp` (не экземпляра RegExp, а *функции*), в виде `$1` для первой группы захвата, `$2` для второй и т. д. до `$9`:

```
const s = "Testing a-1 b-2 c-3";
const rex = /(\w)-(\d)/g;
while (rex.exec(s)) {
    // Не рекомендуется!
    console.log(`"${RegExp.$1}": "${RegExp.$2}"`);
}
// => "a": "1"
// => "b": "2"
// => "c": "3"
```

Предложение может кодифицировать и другую унаследованную функцию, но по состоянию на начало 2020 года в этом не было уверенности.

Не используйте эти статические свойства в `RegExp`. Причина, по которой они стандартизированы, заключается в обеспечении согласованного поведения во всех реализациях, в частности, в отношении возможности их удаления: предложение позволяет удалить эти свойства с помощью инструкции `delete`. Учтите, что если ваш код использует группы захвата, любой код в любом месте кодовой базы может видеть результаты последнего сопоставления, выполненного с помощью этих статических свойств

[140] https://github.com/tc39/proposal-hashbang

[141] https://github.com/tc39/proposal-regexp-legacy-features

`RegExp.$1` и т. д. Предложение переопределяет их, чтобы код, заботящийся о безопасности, мог удалить их после выполнения сопоставления.

Предложение также гарантирует, что эти устаревшие функции не предоставляются подклассами `RegExp`, поскольку последние могут корректировать предоставляемую ими информацию без учета влияния на устаревшие функции.

СПАСИБО, ЧТО ПРОЧИТАЛИ!

Большинство предложений, рассмотренных в этой главе, — это то, что нельзя было делать раньше. Так что на пути к новым привычкам можно обозначить не так уж много возможностей, кроме одного: используйте эти новые функциональные возможности там, где это кажется вам целесообразным, принимая во внимание предупреждения.

Так что вместо «От старых привычек к новым» я скажу: спасибо, что прочитали! Я надеюсь, что эта книга станет полезной для вас. Не забывайте следить за происходящим в будущем. Вы узнали, как быть в курсе событий в главе 1 и на веб-сайте книги по адресу: `https://thenewtoys.dev`. Счастливого кодирования!

ПРИЛОЖЕНИЕ

Фантастические возможности и где они обитают

(Приношу извинения Джоан Роулинг.)

ФУНКЦИОНАЛЬНЫЕ ВОЗМОЖНОСТИ В АЛФАВИТНОМ ПОРЯДКЕ

#: см. приватные поля (классы), приватные методы и акцессоры (классы:), приватные статические методы и акцессоры (классы)
***: см. оператор возведения в степень*
...: см. синтаксис итеративного расширения, синтаксис свойства расширения, остаточные параметры, синтаксис Rest, синтаксис Rest (деструктуризация), синтаксис расширения
`__defineGetter__` (Приложение Б): глава 17
`__defineSetter__` (Приложение Б): глава 17
`__lookupGetter__` (Приложение Б): глава 17
`__lookupSetter__` (Приложение Б): глава 17
`__proto__` (Приложение Б): глава 5
«Безопасные» целые числа (тип number): глава 17 (см. также тип данных BigInt в главе 17)
`Array.from`: глава 11
`Array.from`: глава 11
`Array.prototype.copyWithin`: глава 11
`Array.prototype.entries`: глава 11
`Array.prototype.fill`: глава 11
`Array.prototype.find`: глава 11
`Array.prototype.findIndex`: глава 11
`Array.prototype.flat`: глава 11
`Array.prototype.flatMap`: глава 11
`Array.prototype.includes`: глава 11

`Array.prototype.keys`: глава 11
`Array.prototype.values`: глава 11
`ArrayBuffer`: глава 11
`async`: глава 9
`Atomics.asyncWait`: глава 19
`await` верхнего уровня: глава 19
`await`: глава 9
`class`: глава 4
`const`: глава 2
`DataView`: глава 11
`export`: глава 13
`extends`: глава 4
`FinalizationRegistry`: глава 19
`for-await-of`: глава 9
`for-of`: глава 6
`globalThis`: глава 17
HTML-подобные комментарии (Приложение Б): глава 17
`import, import()`: глава 13
`import.meta`: глава 13
`JSON`: разрывы строк Юникода: глава 17
`JSON`: экранирование для непарных суррогатов в Юникоде: глава 17
`let`: глава 2
`Math.acosh`: глава 17
`Math.asinh`: глава 17
`Math.atanh`: глава 17
`Math.cbrt`: глава 17
`Math.clz32`: глава 17
`Math.cosh`: глава 17
`Math.expm1`: глава 17
`Math.fround`: глава 17
`Math.hypot`: глава 17
`Math.imul`: глава 17
`Math.log10`: глава 17
`Math.log1p`: глава 17
`Math.log2`: глава 17
`Math.sign`: глава 17
`Math.sinh`: глава 17
`Math.tanh`: глава 17
`Math.trunc`: глава 17
`new.target`: глава 4
`Number.EPSILON`: глава 17

Number.isFinite: глава 17
Number.isInteger: глава 17
Number.isNaN: глава 17
Number.isSafeInteger: глава 17
Number.MAX_SAFE_INTEGER: глава 17
Number.MIN_SAFE_INTEGER: глава 17
Number.parseFloat: глава 17
Number.parseInt: глава 17
Object.assign: глава 5
Object.entries: глава 5
Object.fromEntries: глава 5
Object.getOwnPropertyDescriptors: глава 5
Object.getOwnPropertySymbols: глава 5
Object.is: глава 5
Object.setPrototypeOf: глава 5
Object.values: глава 5
SharedArrayBuffer: глава 16
String.fromCodePoint: глава 10
String.prototype.codePointAt: глава 10
String.prototype.endsWith: глава 10
String.prototype.includes: глава 10
String.prototype.match обновление: глава 10
String.prototype.matchAll: глава 17
String.prototype.normalize: глава 10
String.prototype.padEnd: глава 10
String.prototype.padStart: глава 10
String.prototype.repeat: глава 10
String.prototype.replace обновление: глава 10
String.prototype.replaceAll: глава 19
String.prototype.search обновление: глава 10
String.prototype.split обновление: глава 10
String.prototype.startsWith: глава 10
String.prototype.substr (Приложение Б): глава 17
String.prototype.trimEnd: глава 10
String.prototype.trimLeft (Приложение Б): глава 10
String.prototype.trimRight (Приложение Б): глава 10
String.prototype.trimStart: глава 10
String.raw: глава 10
super вне классов: глава 5
super: глава 4
Symbol, тип данных: глава 5

`Symbol.asyncIterator`: глава 9
`Symbol.hasInstance`: глава 17
`Symbol.isConcatSpreadable`: глава 17
`Symbol.iterator`: глава 6
`Symbol.match`: глава 10
`Symbol.replace`: глава 10
`Symbol.search`: глава 10
`Symbol.species`: глава 4
`Symbol.split`: глава 10
`Symbol.toPrimitive`: глава 5
`Symbol.toStringTag`: глава 5
`Symbol.unscopables`: глава 17
`TDZ` (временная мертвая зона): глава 2
`yield`: глава 6
асинхронные генераторы: глава 9
асинхронные итераторы: глава 9
асинхронные функции: глава 9
блочная область видимости: глава 2
висящие запятые в списках параметров: глава 3
возобновление потоков: глава 16
воркеры в качестве модулей: глава 13
восьмеричные целочисленные литералы: глава 17
временная мертвая зона (TDZ): глава 2
вычисляемые имена методов: глава 4, глава 5
вычисляемые имена свойств: глава 5
генераторы: глава 6
двоичные целочисленные литералы: глава 17
деструктуризация массива: глава 7
деструктуризация объекта: глава 7
деструктуризация параметров: глава 7
деструктуризация: глава 7
динамический импорт: глава 13
значения деструктуризации по умолчанию: глава 7
значения параметров по умолчанию: глава 3
изменения в `Date.prototype.toString`: глава 17
изменения в `Function.prototype.toString`: глава 17
именованные группы захвата (RegExp): глава 15
именованные группы захвата RegExp: глава 15
именованный импорт: глава 13
именованный экспорт: глава 13
импорт по умолчанию: глава 13

индексы соответствия RegExp: глава 19
индексы соответствия RegExp: глава 19
итераторы: глава 6
итерация массива: глава 6
итерация строк: глава 10
итерируемая деструктуризация: глава 7
итерируемые: глава 6
карты: глава 12
лексическое this: глава 3
множество: глава 12
модули: глава 13
настройка прототипа объекта: глава 5
необязательные привязки `catch`: глава 17
непомеченные шаблонные литералы: глава 10
новый тип — глобальные переменные: глава 2
обратные вызовы очистки: глава 19
объект Atomics: глава 16
объявления свойств (классы): глава 18
объявления функций внутри блоков: глава 3
оператор await верхнего уровня: глава 19
оператор возведения в степень: глава 17
оператор нулевого слияния: глава 17
оптимизация хвостового вызова: глава 17
опциональная цепочка: глава 17
остаточные параметры: глава 3
остаточный синтаксис деструктуризации: глава 7
осуждаемые устаревшие возможности RegExp: глава 19
отключаемые прокси: глава 14
поведение `document.all` (Приложение Б): глава 17
поддержка hashbang: глава 19
поля (классы): глава 18
помеченные шаблонные функции/помеченные шаблонные литералы: глава 10
порядок свойств: глава 5
приватные методы и акцессоры (классы): глава 18
приватные поля (классы): глава 18
приватные статические методы и акцессоры (классы): глава 18
приостановка потоков: глава 16
прокси: глава 14
промисы: глава 8
публичные объявления свойств («поля») (классы): глава 18
рефлексия: глава 14

свойство description символа: глава 17
свойство flags RegExp: глава 15
свойство имени функции: глава 3
синтаксис Rest (деструктуризация): глава 7
синтаксис Rest: глава 7
синтаксис итеративного расширения: глава 6
синтаксис метода: глава 5
синтаксис расширения свойств: глава 5
синтаксис расширения: глава 5, глава 6
слабые карты: глава 12
слабые множества: глава 12
слабые ссылки: глава 19
совместно используемая память: глава 16
создание подклассов для встроенных компонентов: глава 4
стабильная сортировка массива: глава 11
статические методы: глава 4
статические свойства («поля»), методы и акцессоры (классы): глава 18
стенография свойств: глава 5
стрелочные функции: глава 3
строковые методы HTML (Приложение Б): глава 17
тип BigInt: глава 17
типизированные массивы: глава 11
улучшенная поддержка строк Юникода: глава 10
утверждения ретроспективной проверки (RegExp): глава 15
утверждения ретроспективной проверки RegExp: глава 15
финализаторы: см. обратные вызовы очистки, глава 19
флаг `s` (RegExp): глава 15
флаг s RegExp: глава 15
флаг u (RegExp): глава 15
флаг u RegExp: глава 15
флаг y (RegExp): глава 15
флаг y RegExp: глава 15
функции-генераторы: глава 6
хитрости RegExp (Приложение Б): глава 17
хорошо известные символы: глава 5
целочисленные литералы — восьмеричные: глава 17
целочисленные литералы — двоичные: глава 17
числовые разделители: глава 19
шаблонные литералы: глава 10
экранирование кодовой точки Юникода (RegExp): глава 15
экранирование кодовой точки Юникода RegExp: глава 15

экранирование свойства Юникода (RegExp): глава 15
экранирование свойства Юникода RegExp: глава 15
экспорт по умолчанию: глава 13

НОВЫЕ ПОЛОЖЕНИЯ

#: см. приватные поля (классы), приватные методы и акцессоры (классы:), приватные статические методы и акцессоры (классы)
`Atomics.asyncWait:` глава 19
`await` верхнего уровня: глава 19
`const:` глава 2
`DataView:` глава 11
`FinalizationRegistry:` глава 19
`let:` глава 2
`Object.setPrototypeOf:` глава 5
`Symbol,` тип данных: глава 5
`TDZ` (временная мертвая зона): глава 2
асинхронные генераторы: глава 9
асинхронные итераторы: глава 9
асинхронные функции: глава 9
блочная область видимости: глава 2
возобновление потоков: глава 16
временная мертвая зона (TDZ): глава 2
генераторы: глава 6
динамический импорт: глава 13
значения параметров по умолчанию: глава 3
именованный импорт: глава 13
именованный экспорт: глава 13
импорт по умолчанию: глава 13
итераторы: глава 6
итерируемые: глава 6
лексическое this: глава 3
модули: глава 13
настройка прототипа объекта: глава 5
непомеченные шаблонные литералы: глава 10
новый тип — глобальные переменные: глава 2
обратные вызовы очистки: глава 19
объект Atomics: глава 16
объявления свойств (классы): глава 18
оператор await верхнего уровня: глава 19
отключаемые прокси: глава 14
поля (классы): глава 18

помеченные шаблонные функции/помеченные шаблонные литералы: глава 10
порядок свойств: глава 5
приватные методы и акцессоры (классы): глава 18
приватные поля (классы): глава 18
приватные статические методы и акцессоры (классы): глава 18
приостановка потоков: глава 16
прокси: глава 14
промисы: глава 8
публичные объявления свойств («поля») (классы): глава 18
рефлексия: глава 14
слабые карты: глава 12
слабые множества: глава 12
слабые ссылки: глава 19
совместно используемая память: глава 16
создание подклассов для встроенных компонентов: глава 4
статические методы: глава 4
статические свойства («поля»), методы и акцессоры (классы): глава 18
стрелочные функции: глава 3
тип BigInt: глава 17
типизированные массивы: глава 11
финализаторы: см. обратные вызовы очистки, глава 19
функции-генераторы: глава 6
хорошо известные символы: глава 5
шаблонные литералы: глава 10
экспорт по умолчанию: глава 13

НОВЫЙ СИНТАКСИС, КЛЮЧЕВЫЕ СЛОВА, ОПЕРАТОРЫ, ЦИКЛЫ И Т. П.

#: см. приватные поля (классы), приватные методы и акцессоры (классы:), приватные статические методы и акцессоры (классы)
***: см. оператор возведения в степень*
...: см. синтаксис итеративного расширения, синтаксис свойства расширения, остаточные параметры, синтаксис Rest, синтаксис Rest (деструктуризация), синтаксис расширения
`async`: глава 9
`await` верхнего уровня: глава 19
`await`: глава 9
`class`: глава 4
`const`: глава 2
`export`: глава 13
`extends`: глава 4

`for-await-of`: глава 9
`for-of`: глава 6
HTML-подобные комментарии (Приложение Б): глава 17
`import`, import(): глава 13
`import.meta`: глава 13
`let`: глава 2
`new.target`: глава 4
`super` вне классов: глава 5
`super`: глава 4
`yield`: глава 6
висящие запятые в списках параметров: глава 3
вычисляемые имена методов: глава 4, глава 5
вычисляемые имена свойств: глава 5
генераторы: глава 6
деструктуризация массива: глава 7
деструктуризация объекта: глава 7
деструктуризация параметров: глава 7
деструктуризация: глава 7
динамический импорт: глава 13
значения деструктуризации по умолчанию: глава 7
значения параметров по умолчанию: глава 3
итерируемая деструктуризация: глава 7
необязательные привязки catch: глава 17
непомеченные шаблонные литералы: глава 10
объявления свойств (классы): глава 18
оператор await верхнего уровня: глава 19
оператор возведения в степень: глава 17
оператор нулевого слияния: глава 17
опциональная цепочка: глава 17
остаточные параметры: глава 3
остаточный синтаксис деструктуризации: глава 7
поля (классы): глава 18
помеченные шаблонные функции/помеченные шаблонные литералы: глава 10
приватные методы и акцессоры (классы): глава 18
приватные поля (классы): глава 18
приватные статические методы и акцессоры (классы): глава 18
публичные объявления свойств («поля») (классы): глава 18
синтаксис Rest (деструктуризация): глава 7
синтаксис Rest: глава 7
синтаксис итеративного расширения: глава 6
синтаксис метода: глава 5

синтаксис расширения свойств: глава 5
синтаксис расширения: глава 5, глава 6
статические методы: глава 4
статические свойства («поля»), методы и акцессоры (классы): глава 18
стенография свойств: глава 5
улучшенная поддержка строк Юникода: глава 10
функции-генераторы: глава 6
числовые разделители: глава 19
шаблонные литералы: глава 10

НОВЫЕ ЛИТЕРАЛЬНЫЕ ФОРМЫ

восьмеричные целочисленные литералы: глава 17
двоичные целочисленные литералы: глава 17
тип BigInt: глава 17
числовые разделители: глава 19
шаблонные литералы: глава 10

ДОПОЛНЕНИЯ И ИЗМЕНЕНИЯ СТАНДАРТНОЙ БИБЛИОТЕКИ

«Безопасные» целые числа (тип number): глава 17 (см. также тип данных BigInt в главе 17)
`Array.from`: глава 11
`Array.from`: глава 11
`Array.prototype.copyWithin`: глава 11
`Array.prototype.entries`: глава 11
`Array.prototype.fill`: глава 11
`Array.prototype.find`: глава 11
`Array.prototype.findIndex`: глава 11
`Array.prototype.flat`: глава 11
`Array.prototype.flatMap`: глава 11
`Array.prototype.includes`: глава 11
`Array.prototype.keys`: глава 11
`Array.prototype.values`: глава 11
`ArrayBuffer`: глава 11
`Atomics.asyncWait`: глава 19
`DataView`: глава 11
`FinalizationRegistry`: глава 19
`globalThis` глава 17
`Math.acosh`: глава 17
`Math.asinh`: глава 17
`Math.atanh`: глава 17
`Math.cbrt`: глава 17

`Math.clz32`: глава 17
`Math.cosh`: глава 17
`Math.expm1`: глава 17
`Math.fround`: глава 17
`Math.hypot`: глава 17
`Math.imul`: глава 17
`Math.log10`: глава 17
`Math.log1p`: глава 17
`Math.log2`: глава 17
`Math.sign`: глава 17
`Math.sinh`: глава 17
`Math.tanh`: глава 17
`Math.trunc`: глава 17
`Number.EPSILON`: глава 17
`Number.isFinite`: глава 17
`Number.isInteger`: глава 17
`Number.isNaN`: глава 17
`Number.isSafeInteger`: глава 17
`Number.MAX_SAFE_INTEGER`: глава 17
`Number.MIN_SAFE_INTEGER`: глава 17
`Number.parseFloat`: глава 17
`Number.parseInt`: глава 17
`Object.assign`: глава 5
`Object.entries`: глава 5
`Object.fromEntries`: глава 5
`Object.getOwnPropertyDescriptors`: глава 5
`Object.getOwnPropertySymbols`: глава 5
`Object.is`: глава 5
`Object.setPrototypeOf`: глава 5
`Object.values`: глава 5
`SharedArrayBuffer`: глава 16
`String.fromCodePoint`: глава 10
`String.prototype.codePointAt`: глава 10
`String.prototype.endsWith`: глава 10
`String.prototype.includes`: глава 10
`String.prototype.match` обновление: глава 10
`String.prototype.matchAll`: глава 17
`String.prototype.normalize`: глава 10
`String.prototype.padEnd`: глава 10
`String.prototype.padStart`: глава 10
`String.prototype.repeat`: глава 10

`String.prototype.replace` обновление: глава 10
`String.prototype.replaceAll`: глава 19
`String.prototype.search` обновление: глава 10
`String.prototype.split` обновление: глава 10
`String.prototype.startsWith`: глава 10
`String.prototype.substr` (Приложение Б): глава 17
`String.prototype.trimEnd`: глава 10
`String.prototype.trimLeft` (Приложение Б): глава 10
`String.prototype.trimRight` (Приложение Б): глава 10
`String.prototype.trimStart`: глава 10
`String.raw`: глава 10
`Symbol`, тип данных: глава 5
`Symbol.asyncIterator`: глава 9
`Symbol.hasInstance`: глава 17
`Symbol.isConcatSpreadable`: глава 17
`Symbol.iterator`: глава 6
`Symbol.match`: глава 10
`Symbol.replace`: глава 10
`Symbol.search`: глава 10
`Symbol.species`: глава 4
`Symbol.split`: глава 10
`Symbol.toPrimitive`: глава 5
`Symbol.toStringTag`: глава 5
`Symbol.unscopables`: глава 17
изменения в `Date.prototype.toString`: глава 17
изменения в Function.prototype.toString: глава 17
именованные группы захвата (RegExp): глава 15
именованные группы захвата RegExp: глава 15
индексы соответствия RegExp: глава 19
индексы соответствия RegExp: глава 19
карты: глава 12
множество: глава 12
настройка прототипа объекта: глава 5
объект Atomics: глава 16
осуждаемые устаревшие возможности RegExp: глава 19
прокси: глава 14
промисы: глава 8
рефлексия: глава 14
свойство description символа: глава 17
свойство `flags` (RegExp): глава 15
свойство flags RegExp: глава 15

свойство имени функции: глава 3
слабые карты: глава 12
слабые множества: глава 12
слабые ссылки: глава 19
стабильная сортировка массива: глава 11
строковые методы HTML (Приложение Б): глава 17
типизированные массивы: глава 11
утверждения ретроспективной проверки (RegExp): глава 15
утверждения ретроспективной проверки RegExp: глава 15
флаг s (RegExp): глава 15
флаг s RegExp: глава 15
флаг u (RegExp): глава 15
флаг u RegExp: глава 15
флаг y (RegExp): глава 15
флаг y RegExp: глава 15
хорошо известные символы: глава 5
экранирование кодовой точки Юникода (RegExp): глава 15
экранирование кодовой точки Юникода RegExp: глава 15
экранирование свойства Юникода (RegExp): глава 15
экранирование свойства Юникода RegExp: глава 15

ПРОЧЕЕ

`__defineGetter__` (Приложение Б): глава 17
`__defineSetter__` (Приложение Б): глава 17
`__lookupGetter__` (Приложение Б): глава 17
`__lookupSetter__` (Приложение Б): глава 17
`__proto__` (Приложение Б): глава 5
HTML-подобные комментарии (приложение Б): глава 17
JSON: разрывы строк Юникода: глава 17
JSON: экранирование для непарных суррогатов в Юникоде: глава 17
воркеры в качестве модулей: глава 13
итерация массива: глава 6
итерация строк: глава 10
объявления функций внутри блоков: глава 3
оптимизация хвостового вызова: глава 17
поведение document.all (Приложение Б): глава 17
поддержка hashbang: глава 19
хитрости RegExp (Приложение Б): глава 17

АЛФАВИТНЫЙ УКАЗАТЕЛЬ

Apple, компания 30
ArrayBuffer, объект 333
  необработанный доступ к данным в 337
  порядок байтов 336
  совместное использование массивами 339
Atomics, объект 492
  использование для приостановки и возобновления потоков 497
  низкоуровневые возможности 495
await, оператор
  верхнего уровня 592
  использование промиса 271
Babel, инструмент 38
BigInt, тип данных 32, 527
  варианты применения 528
  производительность 531
  создание значения 529
Chromium, браузер 30
const, директива
  вместо var 66
Date.prototype.toString, выражение 537
document.all, объект 557
DOM, интерфейс 182
DOM, коллекция 203
do-while, цикл 62
ECMA-262 28
ECMA-402 32
Ecma International, ассоциация 28
ECMAScript 28
  версии 28
Edge v79, браузер 30
Firefox, браузер 30
for each, идиома 167
for-in, цикл 65
for-of, цикл 168, 182
for, цикл 62
Function.prototype.toString, выражение 538
Google
  Chrome, браузер 30
  компания 420
Harmony 27
Hashbang, последовательность 614
Internet Explorer, браузер 30
JavaScript, язык
  библиотека 547
  движок 58
  изменения 27
  модульная система 400
  новые возможности 69
  отслеживание изменений 36
  поддержка функций разными версиями 37
  создатель 27
JSON, формат 546
let, директива
  вместо var 66
Map, коллекция 32, 347
Meltdown, уязвимость 481
Microsoft
  Edge v44, браузер 30
  Edge v79, браузер 30
  Office JavaScript, надстройка 30
  WebView, компонент 30
  компания 30
Mozilla, компания 28, 30
new.target
  синтаксис 131

Node.js, платформа 37
Number.EPSILON, свойство 541
Opera, браузер 30
Proxy, объект 431
Realm, база данных 150
  один поток на одну 268
Reflect, объект
  функции 431
RegExp
  индексы соответствия 609
  устаревшие возможности 614
Safari, браузер 30
SameValue, операция 156
Set
  коллекция 355
  объект 32
Spectre, уязвимость 481
Spread, оператор 31
super, ключевое слово 113
  в статических методах 120
  использование super внутри метода 145
Symbol.unscopables, свойство 547
Symbol, тип данных 146
TC39, комитет 28
  процесс 33
  состав 32
  участие в процессе 35
Traceur, инструмент 38
var, директива 46
WeakMap, коллекция 32
WeakRef, коллекция 32
WeakSet, коллекция 32
while, цикл 62
Абстрактный класс 133
Акцессор 105
  приватный 584
Анонимная группа захвата 470
Аргумент 70
Арность функции 81
Асинхронное программирование 31
Асинхронные
  генераторы 283
  итераторы 279
  операции 227
Атомарная операция 493
Атомарное сравнение и обмен 495
Байненс, Матиас 420
Бандлинг 420
Безопасные целые числа 538
Библиотека 32
Биндинг 57, 395
Блок 338
Блочная область видимости 31, 46
  вместо встроенных анонимных
    функций 66
  в циклах 55
Браузер 29
  возможности, доступные только для 550
Буквальное указание имени свойства 143
Виртуальная машина 29
Висящая запятая 95
Внешняя среда 58
Временная мертвая зона 49
  обзор 406
Встряхивание дерева 390, 418
Выравнивание памяти 488
Гарантированно-уникальные значения 147
Генератор 184
  асинхронный 283
  в качестве метода 188
  завершение работы 195
  остановка 197
  просто производящий значения 185
  создание итераторов с помощью 186
Глобальный объект
  доступ к 51
Движок JavaScript 29
  JavaScriptCore 30
  JScript 30
  Microsoft Chakra engine 30
  SpiderMonkey 30
  V8 30
Деление по модулю 332
Дерево модулей 376
Деструктуризация 31, 205
  массива 209
  объекта 206
  параметров 217
  синтаксис 207
  синтаксис Rest в шаблонах 213
  цель 207
  шаблон 206

Динамическое метапрограммирование 32
Домашний объект 144
Живой код 418
Задание 246
Заменяющие токены 467
Замыкание в циклах 56
Значения по умолчанию 211
  арность функции 81
  выражения 78
  область видимости 79
Именованная группа захвата 462
  функциональность 462
Импорт
  в режиме реального времени 397
  динамический 411
  записей 394
  метаданные 420
  модуля из-за побочных эффектов 394
  объекта пространства имен модуля 392
  переименование 391
  разновидности 410
Имя свойства 146
Истинноподобное значение 212
Итератор 168
  асинхронный 279
  интерфейс 168
  итерируемый 179
  неявное использование 168
  создание с помощью генератора 186
  строковый 310
  явное использование 170
Итерация 303
  обобщенная 167
  остановка на ранней стадии 171
Итерируемый итератор 179
Итерируемый объект 31, 168, 279
  действия с 175
Карта 347
  варианты использования слабой 360
  итерация содержимого 352
  ключ 348
  основные операции с 348
  производительность 355
  равенство ключей 350
  слабая 360
  создание из итерируемых 351
  создание подклассов для 354
Класс 98
  абстрактный 133
  акцессоры 579
  выражение 136
  использование при создании функций конструктора 137
  конструктор 100
  метод прототипа 102
  объявление 135
  окончательный 133
  приватные методы 579
  приватные поля 572
  свойство-акцессор 105
  создание подклассов 110
  статический метод 104
Ключ 146, 347, 360
  значения, ссылающиеся на 364
Ключевое слово
  чувствительное к контексту 186
Кодовая точка
  экранирующей последовательности 471
Комментарии 551
Компиляция 38
Константа 53
  значение 54
  объявление 46
  попытка присвоить новое значение 53
Контекст 481
  безопасный 481
  группа 481
Критическая секция 483
  блокировка 483
Лжеподобное значение 212
Линтер 47
Литерал
  двоичный (бинарный) целочисленный 532
  объектный 31
  шаблонный 31, 287
Ловушка 431
  apply 443
  construct 443
  defineProperty 443
  deleteProperty 445
  get 446

getOwnPropertyDescriptor 447
getPrototypeOf 448
has 449
isExtensible 449
ownKeys 449
preventExtensions 450
set 451
setPrototypeOf 451
обработчик 432
прокси 442
Магическое число 53
Массив 181, 215
BigInt64Array 531
BigUint64Array 531
базовая (и итеративная) деструктуризация 209
выровненный 324
диапазонов 314
индекс 161, 327
методы типизированного 341
миф о 161
необработанного текста 293
подкласс 312
подклассы типизированного 341
порядок байтов 336
преобразование значений 331
псевдо 82
расширение 182
совместно используемый 485
стандартные методы 341
типизированный 31, 320
хранилище для типизированного 333
элемент 169
Массивоподобный объект 322
Математические функции
возведение в степень 535
Машинный эпсилон 541
Мертвый код 418
Метка 74
Метод 127
Array.from 312
Array.of 311
Atomics.wait 612
catch 235
concat 541
copyWithin 318
endsWith 305
entries 317
fill 322
finally 237
find 321
findIndex 322
flat 324
flatMap 326
includes 324
keys 315
match 308
Number.isFinite 540
Number.isInteger 540
Number.isNaN 540
Number.isSafeInteger 540
Object.assign 155, 165
Object.entries 157
Object.fromEntries 158
Object.getOwnPropertyDescriptors 158
Object.getOwnPropertySymbols 158
Object.getPrototypeOf 166
Object.setPrototypeOf 141, 166
padEnd 306
padStart 306
parseFloat 541
parseInt 541
Promise.all 252
Promise.allSettled 254
Promise.any 255
Promise.race 254
Promise.reject 251
Promise.resolve 250
Reflect.apply 427
Reflect.construct 427
Reflect.get 429
Reflect.ownKeys 429
Reflect.set 429
RegExp.prototype.compile 553
replace 308
return 171, 195
search 308
set 343
split 308
startsWith 305
String.prototype.codePointAt 300
String.prototype.includes 306

String.prototype.matchAll 550
String.prototype.normalize 301
String.prototype.repeat 304
String.raw 295
subarray 343
Symbol.hasInstance 547
Symbol.toPrimitive 159
then 229
then с двумя аргументами 243
throw 195
throw в обработчиках then, catch и finally 241
trimEnd 307
trimStart 307
values 316
вычисляемое имя 107
генератор в качестве 188
геттера 105
приватный 579
приватный статический 586
прототипа 102
сеттера 105
синтаксис 103, 144, 166
статический 104
Миксины 130
Множество 355
итерация содержимого 357
основные операции с 356
порядок значений в 355
производительность 359
слабое 369
создание из итерируемых 357
создание подклассов для 359
Модуль
CJS 388
ESM 376, 388
nomodule, атрибут 384
дерево 376
динамический импорт 411
загрузка 400
загрузка веб-воркера в качестве 421
запись 402
именованный экспорт 376, 379
использование в Node.js 386
карта 404
объект пространства имен 392, 398
объект среды 398
синтаксический анализ кода 404
скрипты 384
спецификатор 376, 411
спецификаторы в Node.js 388
экземпляры 400
Мьютекс 483
Намек 160
Наследование 98
свойств и методов прототипа суперкласса 115
статических методов 119
Необработанное отклонение 236
Необходимое инвариантное поведение 442
Нормализация 301
Нулевого слияния, оператор 543
Обещание 223
Область видимости
лексическая 575
Обработка ошибок 599
Обратная ссылка 466
Обратный вызов 70, 224
очистки 604
сравнение с промисом 234
Объект
ArrayBuffer 333
atomics 492
DataView 337
document.all 557
Reflect 425
вызываемый 427
деструктуризация 206
дополнительные свойства 554
импорт 376
использование в качестве значения заполнения 323
итерируемый 168
результирующий 170
среды 58, 574
форма 568
целевой 156
Окончательный класс 133
Оптимизация
преждевременная 63
Опциональная цепочка 543
Османи, Адди 420

Отсрочка 223
Ошибка типа 160
Памятка 605
Параметр 70
  rest 214
  деструктуризации 156
  деструктуризация 217
  значения по умолчанию 76, 95
  остаточный 81, 95
  список 80
Переменная
  глобальная 51
  использование до объявления 48
  локальная 48
  объявление 46
  узкая область видимости 66
  условная 483
Повторные объявления 47
Подкласс 133
  написание конструкторов 114
Поднятие 48
Поле 567
  приватное 572
  приватное статическое 586
  публичное 567
  публичное статическое 585
Полифилирование 38
Порядок свойств 161
Поток 479
  воркера 480
  отправляющий 480
  потребителель 483
  производитель 483
Приватные
  акцессоры 584
  идентификаторы 572
  имена 573
  методы 579
  свойства 374
  статические методы 586
  статические поля 586
Привязка
  в объекте среды 398
  идентификатора 57
  изменяемая 57
  косвенная 378, 397
  неизменяемая 57
  необязательная 546
Производительность 63
Прокси-объект 431
Промис 223
  возвращение 255
  задание 246
  исключение 273
  использование существующего 229
  и элементы thenable 228
  как средство наблюдения 223
  конструктор Promise 247
  микрозадача 246
  нативный 271
  неправильное разветвление цепочки 263
  обработчик бездействия 262
  оператор await использует 271
  отклонение 273
  параллельные 258
  подклассы 264
  разрешение 273
  связывание в цепочки 230
  серии 256
  служебные методы 252
  создание 246
  создание асинхронными функциями 270
  состояния 224
  сравнение с обратным вызовом 234
  сторонний 271
Прототип итератора объекта 172
Прототипические языки 98
Прототипическое наследование 97
Прототип объекта
  получение и настройка 141
Публичные статические поля 585
Регулярное выражение 31, 295, 552
Реестр финализации 605
Результирующий объект 170
Рекурсия 558
Ретроспективная проверка
  негативная 467, 469
  позитивная 467, 468
  утверждения 467
Референт 601

Сборка мусора 601, 603
Свойство
  __proto__, буквальное указание имени в браузерах 143
  акцессор 105, 429
  вычисляемое имя 164
  имя 146
  несуществующее 449
  порядок 161
  расширения 156
  синтаксис расширения 163, 165
  сокращенный синтаксис 165
  стенография 140
Символ 32, 122, 146
  глобальный 150
  глобальный реестр 153
  избегание коллизии имен 165
  используемый для итерации 171
  не для конфиденциальности 149
  описание 149
  создание и использование 148
  хорошо известные 154
Синтаксис 555
  class 31
  Rest 31
  итеративного расширения 181
  оператора Spread 31
  улучшения 31
Скрипт
  задание 246
  макрозадача 246
Совместно используемая память 31, 479
  блокировка 483
  выравнивание памяти 488
  критическая секция 483
  но не объекты 489
  очередь 483
  пример 498
  синхронизация данных 480
  создание 484
Создание строк из шаблонов 292
Среда приватных имен 574
Ссылка
  сильная 601
  слабая 601
Стенография свойств 140
Стрелочная функция 31, 38, 70
  в инициализаторах 569
  вместо различных обходных путей 92
  для обратных вызовов 93
  и оператор \ 72
  как конструктор 75
  синтаксис 70
Суперкласс 113
Транспилирование 38
Улучшения параметров функций 31
Утверждение ретроспективной проверки 467
Финализатор 604
Флаги
  липкие 460
Форма объекта 568
Функция
  анонимное выражение 85
  аргумент 70
  арность 81
  генератор 172, 184
  исполнитель 226, 248
  использование асинхронной функции в неожиданном месте 278
  обработчик 233
  параллельные операции в асинхронной 276
  параметр 70
  помеченная 290
  свойство имени 85
  стрелочная 31
Фьючерс 223
Хансен, Ларс Т. 492
Хвостовой вызов 559
Хэш 517
Целочисленный индекс 162
Цепочка наследования конструктора 120
Числовые разделители 613
Шаблонный литерал 287
  вместо конкатенации строк 309
  и автоматическая вставка точки с запятой 297
  непомеченный 288
  повторное использование 297

Эйх, Брендан 27
Экранирование
  управляющее 552
Экспорт
  записей 395
  именованный 379
  косвенный 390
  объекта пространства имен другого модуля 393
  объявления анонимной функции или класса 382
  переименование 389
  по умолчанию 381
  разновидности 408
Юникод 31
  два типа нормализации 302
  двоичные свойства 472
  каноническая эквивалентность 302
  кодовая единица 299
  кодовая точка 299
  перечисляемые свойства 472
  псевдонимы свойств в двоичном 473
  разрывы строк в JSON 546
  свойства 472
  строка JavaScript 298
  улучшенная поддержка 297
  эквивалентность совместимости 302
  экранирование кодовой точки 471
  экранирование свойства 472
  экранированные последовательности кодовой точки 300
  экранирующая последовательность 294

Все права защищены. Книга или любая ее часть не может быть скопирована, воспроизведена в электронной или механической форме, в виде фотокопии, записи в память ЭВМ, репродукции или каким-либо иным способом, а также использована в любой информационной системе без получения разрешения от издателя. Копирование, воспроизведение и иное использование книги или ее части без согласия издателя является незаконным и влечет уголовную, административную и гражданскую ответственность.

Научно-популярное издание

МИРОВОЙ КОМПЬЮТЕРНЫЙ БЕСТСЕЛЛЕР

Краудер Ти Джей

НОВЫЕ ВОЗМОЖНОСТИ JAVASCRIPT
КАК НАПИСАТЬ ЧИСТЫЙ КОД ПО ВСЕМ ПРАВИЛАМ СОВРЕМЕННОГО ЯЗЫКА

Главный редактор *Р. Фасхутдинов*
Руководитель направления *В. Обручев*
Научный редактор *Д. Пустошилов*
Литературный редактор *Е. Пригородова*
Ответственный редактор *Д. Калачева*
Младший редактор *Д. Данилова*
Художественный редактор *Е. Пуговкина*
Компьютерная верстка *Э. Брегис*
Корректоры *А. Баскакова, Л. Макарова*

Страна происхождения: Российская Федерация
Шығарылған елі: Ресей Федерациясы

ООО «Издательство «Эксмо»
123308, Россия, город Москва, улица Зорге, дом 1, строение 1, этаж 20, каб. 2013.
Тел.: 8 (495) 411-68-86.
Home page: www.eksmo.ru E-mail: info@eksmo.ru
Өндіруші: «ЭКСМО» АҚБ Баспасы,
123308, Ресей, қала Мәскеу, Зорге көшесі, 1 үй, 1 ғимарат, 20 қабат, офис 2013 ж.
Тел.: 8 (495) 411-68-86.
Home page: www.eksmo.ru E-mail: info@eksmo.ru.
Тауар белгісі: «Эксмо»
Интернет-магазин : www.book24.ru

Интернет-магазин : www.book24.kz
Интернет-дукен : www.book24.kz
Импортёр в Республику Казахстан ТОО «РДЦ-Алматы».
Қазақстан Республикасындағы импорттаушы «РДЦ-Алматы» ЖШС.
Дистрибьютор и представитель по приему претензий на продукцию,
в Республике Казахстан: ТОО «РДЦ-Алматы»
Қазақстан Республикасында дистрибьютор және өнім бойынша арыз-талаптарды
қабылдаушының өкілі «РДЦ-Алматы» ЖШС,
Алматы қ., Домбровский көш., 3«а», литер Б, офис 1.
Тел.: 8 (727) 251-59-90/91/92; E-mail: RDC-Almaty@eksmo.kz
Өнімнің жарамдылық мерзімі шектелмеген.
Сертификация туралы ақпарат сайтта: www.eksmo.ru/certification

Сведения о подтверждении соответствия издания согласно законодательству РФ
о техническом регулировании можно получить на сайте Издательства «Эксмо»:
www.eksmo.ru/certification
Өндірген мемлекет: Ресей. Сертификация қарастырылмаған

Дата изготовления / Подписано в печать 28.02.2023.
Формат 70x100 1/16. Печать офсетная. Усл. печ. л. 51,85.
Тираж экз. Заказ

БОМБОРА – лидер на рынке полезных и вдохновляющих книг.
Мы любим книги и создаем их, чтобы вы могли творить, открывать
мир, пробовать новое, расти. Быть счастливыми. Быть на волне.

bombora.ru bomborabooks bombora

ISBN 978-5-04-159515-9

12+

ЧИТАЙ ГОРОД

В электронном виде книги издательства вы можете купить на **www.litres.ru**

ЛитРес: один клик до книг

Москва. ООО «Торговый Дом «Эксмо»
Адрес: 123308, г. Москва, ул. Зорге, д.1, строение 1.
Телефон: +7 (495) 411-50-74. **E-mail:** reception@eksmo-sale.ru

По вопросам приобретения книг «Эксмо» зарубежными оптовыми покупателями обращаться в отдел зарубежных продаж ТД «Эксмо»
E-mail: **international@eksmo-sale.ru**

International Sales: International wholesale customers should contact Foreign Sales Department of Trading House «Eksmo» for their orders.
international@eksmo-sale.ru

По вопросам заказа книг корпоративным клиентам, в том числе в специальном оформлении, обращаться по тел.: +7 (495) 411-68-59, доб. 2261.
E-mail: **ivanova.ey@eksmo.ru**

Оптовая торговля бумажно-беловыми и канцелярскими товарами для школы и офиса «Канц-Эксмо»:
Компания «Канц-Эксмо»: 142702, Московская обл., Ленинский р-н, г. Видное-2, Белокаменное ш., д. 1, а/я 5. Тел./факс: +7 (495) 745-28-87 (многоканальный).
e-mail: **kanc@eksmo-sale.ru**, сайт: www.**kanc-eksmo.ru**

Филиал «Торгового Дома «Эксмо» в Нижнем Новгороде
Адрес: 603094, г. Нижний Новгород, улица Карпинского, д. 29, бизнес-парк «Грин Плаза»
Телефон: +7 (831) 216-15-91 (92, 93, 94). **E-mail**: reception@eksmonn.ru

Филиал ООО «Издательство «Эксмо» в г. Санкт-Петербурге
Адрес: 192029, г. Санкт-Петербург, пр. Обуховской обороны, д. 84, лит. «Е»
Телефон: +7 (812) 365-46-03 / 04. **E-mail**: server@szko.ru

Филиал ООО «Издательство «Эксмо» в г. Екатеринбурге
Адрес: 620024, г. Екатеринбург, ул. Новинская, д. 2щ
Телефон: +7 (343) 272-72-01 (02/03/04/05/06/08)

Филиал ООО «Издательство «Эксмо» в г. Самаре
Адрес: 443052, г. Самара, пр-т Кирова, д. 75/1, лит. «Е»
Телефон: +7 (846) 207-55-50. **E-mail**: RDC-samara@mail.ru

Филиал ООО «Издательство «Эксмо» в г. Ростове-на-Дону
Адрес: 344023, г. Ростов-на-Дону, ул. Страны Советов, 44А
Телефон: +7(863) 303-62-10. **E-mail**: info@rnd.eksmo.ru

Филиал ООО «Издательство «Эксмо» в г. Новосибирске
Адрес: 630015, г. Новосибирск, Комбинатский пер., д. 3
Телефон: +7(383) 289-91-42. E-mail: eksmo-nsk@yandex.ru

Обособленное подразделение в г. Хабаровске
Фактический адрес: 680000, г. Хабаровск, ул. Фрунзе, 22, оф. 703
Почтовый адрес: 680020, г. Хабаровск, А/Я 1006
Телефон: (4212) 910-120, 910-211. **E-mail**: eksmo-khv@mail.ru

Республика Беларусь: ООО «ЭКСМО АСТ Си энд Си»
Центр оптово-розничных продаж Cash&Carry в г. Минске
Адрес: 220014, Республика Беларусь, г. Минск, проспект Жукова, 44, пом. 1-17, ТЦ «Outleto»
Телефон: +375 17 251-40-23; +375 44 581-81-92
Режим работы: с 10.00 до 22.00. **E-mail:** exmoast@yandex.by

Казахстан: «РДЦ Алматы»
Адрес: 050039, г. Алматы, ул. Домбровского, 3А
Телефон: +7 (727) 251-58-12, 251-59-90 (91,92,99). E-mail: RDC-Almaty@eksmo.kz

Полный ассортимент продукции ООО «Издательство «Эксмо» можно приобрести в книжных магазинах «Читай-город» и заказать в интернет-магазине: www.chitai-gorod.ru.
Телефон единой справочной службы: 8 (800) 444-8-444. Звонок по России бесплатный.

Интернет-магазин ООО «Издательство «Эксмо»
www.book24.ru
Розничная продажа книг с доставкой по всему миру.
Тел.: +7 (495) 745-89-14. E-mail: **imarket@eksmo-sale.ru**

Хочешь стать автором «Эксмо»?

eksmo.ru
Официальный интернет-магазин издательства «Эксмо»

ЛУЧШИЕ КНИГИ О БИЗНЕСЕ С ЛОГОТИПОМ ВАШЕЙ КОМПАНИИ? ЛЕГКО!

Удивить своих клиентов, бизнес-партнеров, сделать памятный подарок сотрудникам и рассказать о своей компании читателям бизнес-литературы? Приглашаем стать партнерами выпуска актуальных и популярных книг. О вашей компании узнает наиболее активная аудитория.

ПАРТНЕРСКИЕ ОПЦИИ:

- Специальный тираж уже существующих книг с логотипом вашей компании.
- Размещение логотипа на супер-обложке для малых тиражей (от 30 штук).
- Поддержка выхода новинки, которая ранее не была доступна читателям (50 книг в подарок).

ПАРТНЕРСКИЕ ВОЗМОЖНОСТИ:

- Рекламная полоса о вашей компании внутри книги.
- Вступительное слово в книге от первых лиц компании-партнера.
- Обращение первых лиц на суперобложке.
- Отзыв на обороте обложки вложение информационных материалов о вашей компании (закладки, листовки, мини-буклеты).

У вас есть возможность обсудить свои пожелания с менеджерами корпоративных продаж. Как?

Звоните:
+7 495 411 68 59, доб. 2261

Заходите на сайт:
eksmo.ru/b2b

Руководство по JavaScript для тех, кто хочет кодить быстро и эффективно

«Новые возможности JavaScript» — это сборник правил написания кода на современном языке JavaScript. На наглядных примерах автор объясняет, как работают последние версии JS, какие приемы в нем можно использовать, чтобы сделать код коротким и чистым, а каких ошибок лучше избегать, чтобы не было багов.

Книга будет полезна всем, кто имеет по крайней мере базовое представление о JavaScript и хочет изучить новые возможности языка, появившиеся в последние годы.

Благодаря ей вы узнаете:

- ЧЕМ ПОСЛЕДНИЕ ВЕРСИИ JS ОТЛИЧАЮТСЯ ДРУГ ОТ ДРУГА
- КАК РАСШИРИЛСЯ ФУНКЦИОНАЛ ОБЪЕКТОВ В КОДЕ
- ЧТО НОВОГО МОГУТ ПРЕДЛОЖИТЬ ФУНКЦИИ
- КАК ИСПОЛЬЗОВАТЬ СОВРЕМЕННЫЙ СИНТАКСИС ЯЗЫКА
- КАКИМИ БУДУТ ДАЛЬНЕЙШИЕ УЛУЧШЕНИЯ JS

Это не академическая книга для экспертов в программировании. Это практическая книга для обычных разработчиков на JavaScript, желающих развивать свои навыки программирования и идти в ногу со временем.

Ти Джей Краудер – инженер-программист с 30-летним стажем, руководитель британской компании Farsight Software, которая занимается консалтингом и разработкой программного обеспечения.

БОМБОРА
ИЗДАТЕЛЬСТВО

БОМБОРА – лидер на рынке полезных и вдохновляющих книг.
Мы любим книги и создаем их, чтобы вы могли творить, открывать мир, пробовать новое, расти. Быть счастливыми. Быть на волне.

bombora.ru bomborabooks bombora

ISBN 978-5-04-159515-9