

O'REILLY®

Рецепты TypeScript

Программирование
на уровне типов
для реальных задач



SPRINT
book

Стефан
Баумгартнер

TypeScript Cookbook

Real World Type-Level Programming

Stefan Baumgartner

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Рецепты TypeScript

Программирование на уровне типов для реальных задач

Стефан Баумгартнер

ББК 32.988.02-018
УДК 004.738.5
Б29

Баумгартнер Стефан

Б29 Рецепты TypeScript. — Астана: Спринт Бук, 2025. — 432 с.: ил.
ISBN 978-601-08-4355-4

TypeScript — один из важнейших инструментов для JavaScript-разработчиков. Тем не менее даже опытные специалисты получают множество сообщений об ошибках от компилятора TypeScript, неприятно удивляясь этому. Откройте «Рецепты TypeScript». В этом практическом руководстве его автор Стефан Баумгартнер описывает способы решения наиболее распространенных задач на TypeScript.

Те, кто уже знаком с основами JavaScript и TypeScript, найдут в книге действенные рецепты, которые помогут справиться с широким спектром вопросов, от настройки проектов со сложной структурой до определения расширенных вспомогательных типов. Каждый такой рецепт поможет разобраться с конкретной проблемой и объяснит, почему и как это работает.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.988.02-018
УДК 004.738.5

Права на издание получены по соглашению с O'Reilly. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги. Издательство не несет ответственности за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на интернет-ресурсы были действующими. В книге возможны упоминания организаций, деятельность которых запрещена на территории Российской Федерации, таких как Meta Platforms Inc., Facebook, Instagram и др.

ISBN 978-1098136659 англ.

Authorized Russian translation of the English edition of TypeScript Cookbook
ISBN 978-1098136659 © 2023 Stefan Baumgartner.

This translation is published and sold by permission of O'Reilly Media, Inc.,
which owns or controls all rights to publish and sell the same.

ISBN 978-601-08-4355-4

© Перевод на русский язык ТОО «Спринт Бук», 2024

© Издание на русском языке, оформление ТОО «Спринт Бук», 2024

Оглавление

https://t.me/it_boooks/2

Предисловие	11
Введение	13
Кому стоит прочесть эту книгу	14
Цели написания книги	14
Структура издания	15
Условные обозначения	17
Использование примеров кода	18
От издательства	19
Благодарности	20
Глава 1. Настройка проекта	22
1.1. Проверка типов JavaScript	23
1.2. Установка TypeScript	27
1.3. Поддержка типов «в стороне»	30
1.4. Миграция проекта на TypeScript	32
1.5. Загрузка типов из репозитория Definitely Typed	35
1.6. Настройка полнофункционального проекта	38
1.7. Настройка тестов	43
1.8. Типизация модулей ECMAScript из URL	46
1.9. Загрузка различных типов модулей в Node	50
1.10. Работа с Deno и зависимостями	53
1.11. Использование предопределенных конфигураций	56
Глава 2. Основные типы	58
2.1. Эффективное аннотирование	58
2.2. Работа с any и unknown	62
2.3. Выбор правильного типа объекта	65
2.4. Работа с типами кортежей	68
2.5. Различия между интерфейсами и псевдонимами типов	71
2.6. Определение перегрузок функций	74

2.7. Определение типов параметра <code>this</code>	77
2.8. Работа с <code>Symbol</code>	80
2.9. Понимание пространств имен значений и типов	83
Глава 3. Система типов	87
3.1. Моделирование данных с помощью типов объединений и пересечений	87
3.2. Явное определение моделей с размеченными типами объединения.....	92
3.3. Проверка полноты с помощью приема <code>assertNever</code>	95
3.4. Закрепление типов с помощью <code>Const Context</code>	99
3.5. Сужение типов с помощью предикатов типа	102
3.6. Понимание <code>void</code>	105
3.7. Работа с типами ошибок в предложениях <code>catch</code>	108
3.8. Создание моделей исключающего ИЛИ с помощью приема <code>optional never</code>	111
3.9. Эффективное использование утверждений типа	114
3.10. Использование сигнатур индекса.....	117
3.11. Различение отсутствующих свойств и значений <code>undefined</code>	122
3.12. Работа с перечислениями	124
3.13. Определение номинальных типов в системе структурных типов.....	129
3.14. Включение свободного автозаполнения для подмножеств строк.....	133
Глава 4. Обобщенные типы.....	136
4.1. Обобщение сигнатур функций.....	136
4.2. Создание связанных аргументов функций	139
4.3. Избавление от <code>any</code> и <code>unknown</code>	143
4.4. Создание экземпляров обобщенного типа	146
4.5. Создание новых типов объектов	151
4.6. Изменение объектов с помощью сигнатур утверждений.....	156
4.7. Сопоставление типов с помощью карт типов.....	159
4.8. Использование <code>ThisType</code> для определения <code>this</code> в объектах.....	164
4.9. Добавление контекста <code>const</code> к параметрам обобщенного типа	168
Глава 5. Условные типы	173
5.1. Управление сигнатурами сложных функций.....	173
5.2. Фильтрация с помощью типа <code>never</code>	178
5.3. Группировка элементов по типу.....	182
5.4. Удаление определенных свойств объекта	188
5.5. Выведение типов в условных выражениях	191

Глава 6. Типы литералов шаблонов строк	197
6.1. Определение пользовательской системы событий	198
6.2. Создание обратных вызовов событий с помощью типов манипуляций со строками и переназначения ключей	200
6.3. Написание функции форматирования	204
6.4. Извлечение типов параметров формата	208
6.5. Работа с ограничениями рекурсии	211
6.6. Использование литералов шаблонов строк в качестве дискриминантов	215
Глава 7. Вариативные типы кортежей	219
7.1. Типизация функции <code>concat</code>	219
7.2. Типизация функции <code>promisify</code>	223
7.3. Типизация функции <code>curry</code>	227
7.4. Типизация гибкой функции <code>curry</code>	230
7.5. Типизация простейшей функции <code>curry</code>	234
7.6. Создание перечисления из кортежа	237
7.7. Разделение всех элементов сигнатуры функции	241
Глава 8. Вспомогательные типы	243
8.1. Установка определенных необязательных свойств	244
8.2. Изменение вложенных объектов	247
8.3. Переназначение типов	251
8.4. Получение всех необходимых ключей	254
8.5. Разрешение хотя бы одного свойства	257
8.6. Разрешение ровно одного свойства, всех свойств или ни одного	262
8.7. Преобразование типов объединения в типы пересечения	267
8.8. Использование библиотеки <code>type-fest</code>	274
Глава 9. Стандартная библиотека и определения внешних типов	279
9.1. Перебор объектов с помощью <code>Object.keys</code>	280
9.2. Явное выделение небезопасных операций с помощью утверждений типов и <code>unknown</code>	284
9.3. Работа с <code>defineProperty</code>	286
9.4. Расширение типов для <code>Array.prototype.includes</code>	292
9.5. Фильтрация нулевых значений	296
9.6. Расширение модулей	298
9.7. Расширение глобальных переменных	300
9.8. Добавление модулей, которые отличаются от JS, в граф модулей	304

Глава 10. TypeScript и React	308
10.1. Написание прокси-компонентов.....	309
10.2. Написание контролируемых компонентов	312
10.3. Типизация пользовательских хуков	315
10.4. Типизация обобщенных компонентов forwardRef.....	317
10.5. Предоставление типов для контекстного API.....	322
10.6. Типизация компонентов высшего порядка	327
10.7. Типизация обратных вызовов в синтетической системе событий React ...	330
10.8. Типизация полиморфных компонентов	333
Глава 11. Классы	338
11.1. Выбор правильного модификатора видимости	339
11.2. Явное переопределение методов.....	344
11.3. Описание конструкторов и прототипов	348
11.4. Использование обобщенных типов в классах	352
11.5. Решение об использовании классов или пространства имен.....	355
11.6. Написание статических классов	360
11.7. Работа со строгой инициализацией свойств	364
11.8. Работа с типами this в классах.....	369
11.9. Декораторы	373
Глава 12. Стратегии разработки типов	380
12.1. Написание простых в сопровождении типов.....	381
12.2. Поэтапное уточнение типов	384
12.3. Проверка контрактов с помощью satisfies	395
12.4. Тестирование сложных типов	399
12.5. Проверка типов данных во время выполнения с помощью Zod	402
12.6. Обход ограничений доступа к индексу	407
12.7. Принятие решения о том, использовать ли перегрузку функций или условные типы	410
12.8. Именованное обобщенных типов	417
12.9. Прототипирование в TypeScript Playground.....	419
12.10. Предоставление нескольких версий библиотеки	424
12.11. Знать, когда остановиться	428
Об авторе	430
Иллюстрация на обложке	431

Отзывы о книге

Стефан Баумгартнер подробно описывает все процессы, начиная с настройки проекта и заканчивая передовыми приемами типизации, предоставляя множество практических примеров и ценную информацию, которая сделает вас экспертом по TypeScript, готовым к решению сложных задач.

*Адди Османи, руководитель отдела
разработки Chrome, Google*

Эта книга — важный ресурс для разработчиков, которые хотят научиться эффективно использовать TypeScript. Стефан собрал четкие и краткие рецепты решения реальных задач во всеобъемлющее руководство, которое поможет вам пройти путь от новичка до эксперта.

*Симона Котин, менеджер
по разработке Angular, Google*

В книге показано, как решать всевозможные задачи с помощью расширенных типов. Более того, она научит вас использовать возможности TypeScript для самостоятельного создания новых типов.

*Натан Шивли-Сандерс,
инженер-программист в команде TypeScript*

Эта книга — чрезвычайно ценный справочник для всех, кто работает с TypeScript. Она содержит множество ценной информации в таком формате, который вы можете легко использовать.

Мэтт Покок, автор книги Total TypeScript

Иногда TypeScript может замедлять работу программистов, но эта книга — идеальный помощник! Комплексные приемы, предлагаемые для решения распространенных задач TypeScript, делают ее незаменимым инструментом повышения производительности.

*Ванесса Бёнер, ведущий
фронтенд-разработчик, Zaycu*

В этой прекрасной книге содержится ценнейшая информация. Мне очень понравились лаконичные вопросы и ответы, предваряющие подробное обсуждение нюансов. Из каждой главы я почерпнул массу полезных приемов и новых интересных паттернов. Любому разработчику TypeScript будет полезно изучить приемы и техники, представленные в этой книге. Настоятельно рекомендую!

*Джош Голдберг,
автор книги Learning TypeScript*

Я осознал очень много проблем, с которыми сталкивался при работе в TypeScript. Советы Стефана относительно их ясны, точны и содержательны. Это руководство помогает мне чувствовать себя более уверенно при использовании TypeScript.

Фил Нэш, Developer Advocate, Sonar

Предисловие

Мне всегда интересно наблюдать за развитием языков программирования и их влиянием на разработку программного обеспечения (ПО). TypeScript, расширенная версия JavaScript, — не исключение. Этот язык быстро превратился в один из самых распространенных, заняв уникальное место в мире веб-разработки. TypeScript заслуженно получил широкое признание и высокую оценку, так что вполне уместно привести в этой книге его подробное описание.

Как активный пользователь TypeScript, я должен сказать, что точность и надежность, которые он привнес в JavaScript, не только вдохновляют, но и поражают. Одна из ключевых причин этого — безопасность типов, которая является ответом на давнюю критику JavaScript. Позволяя разработчикам определять строгие типы для переменных, TypeScript упростил обнаружение ошибок в процессе компиляции, что значительно улучшило качество и удобство сопровождения кода.

Эта книга — очень нужное руководство. Во вступлении отмечается, что популярность TypeScript стремительно растет. Однако интерес к языку позволяет выявлять проблемы, с которыми сталкиваются разработчики при его внедрении. Данное издание призвано улучшить ситуацию в этой области.

Книга содержит много практических рекомендаций и предназначена для решения реальных задач, с которыми сталкиваются пользователи TypeScript. В ней собрано более ста рецептов, посвященных самым разным концепциям: от базовых до передовых. Разработчикам часто приходится бороться с системой проверки типов, и решить эту задачу поможет данное издание. Благодаря подробным объяснениям вы научитесь эффективно работать с TypeScript.

Одно из многих достоинств издания — описанный здесь подход к быстрому развитию TypeScript. Новые версии языка выходят регулярно, поэтому оставаться в курсе актуальных событий — сложная задача. В этой книге акцент ставится на долгосрочных аспектах TypeScript, что позволяет вам быть уверенными в актуальности вашего обучения, несмотря на постоянно меняющиеся обстоятельства.

Помимо описания множества рецептов, книга предлагает понять сложную связь между JavaScript и TypeScript. Уяснив суть симбиотических отношений между этими двумя языками, вы сможете раскрыть истинный потенциал TypeScript. В книге рассказано, как выполнять утверждение типов, работать с обобщенными типами (дженериками, generics) и даже как интегрировать TypeScript с популярными библиотеками и фреймворками, такими как React.

Эта книга — одновременно и отличное руководство, и справочник. Используя ее как руководство, вы легко сможете пройти путь от новичка до эксперта. И вы можете пользоваться ею как справочником все время, пока работаете с TypeScript. Структура книги составлена так, что каждую главу можно читать отдельно, но при этом все они образуют целостную базу знаний.

Популярность TypeScript стремительно растет, поэтому книга станет незаменимым помощником для всех любителей данного языка. Представленные в ней реальные примеры и множество решений помогают ориентироваться в сложном мире TypeScript.

Книга поможет вам независимо от того, начинаете вы изучать этот язык программирования или хотите погрузиться в его глубины. Я от всей души поздравляю Стефана Баумгартнера с созданием этого шедевра!

Начните путешествие в TypeScript.

*Адди Османи, руководитель отдела
разработки Google Chrome*

Введение

Вы можете прочитать это предложение, лишь открыв данную книгу, бумажную или цифровую версию. Это говорит о том, что вы интересуетесь TypeScript, одним из самых популярных языков программирования за последние годы. Согласно опросу State of JavaScript, проведенному в 2022 году (<https://2022.stateofjs.com/>), почти 70 % всех его участников активно используют TypeScript. В опросе StackOverflow за 2022 год (<https://survey.stackoverflow.co/2022>) этот язык входит в пятерку самых популярных и занимает четвертое место по степени удовлетворенности пользователей. В начале 2023 года количество еженедельных скачиваний TypeScript на NPM (<https://oreil.ly/ZHWn8>) превысило 40 миллионов.

Без сомнения, TypeScript — это феномен!

Несмотря на популярность, язык по-прежнему доставляет многим разработчикам немало хлопот. Часто можно услышать фразу «*Борьба с системой проверки типов*»; еще в одной фразе звучит призыв *бросить туда пару ату, чтобы система замолчала*.

Некоторые люди чувствуют, что их работа замедляется, и пишут только для того, чтобы угодить компилятору, хотя знают, что их код *должен* работать. Однако единственная цель TypeScript — сделать работу разработчиков JavaScript более продуктивной. Действительно ли инструмент не справляется со своими задачами или это разработчики ожидают от него чего-то такого, на что он не рассчитан?

Ответ находится где-то посередине, и узнать его поможет эта книга. В ней вы найдете более ста рецептов, охватывающих широкий спектр тем: от сложных настроек проектов до передовых приемов типизации. Вы узнаете о нюансах и внутреннем устройстве системы типов, а также об исключениях и компромиссах, на которые необходимо пойти, чтобы не нарушить ее основу: JavaScript. Помимо этого, вы изучите методологии, паттерны проектирования и методы разработки, позволяющие создавать более качественный и надежный код на TypeScript. В конце концов, вы поймете, не только *как* сделать что-то, но и *почему*.

Моя цель — дать вам руководство, которое поможет вам пройти путь от новичка до эксперта, а также краткий справочник, который вы сможете использовать и после прочтения книги. Учитывая, что TypeScript выходит четыре раза в год, в одной книге невозможно перечислить все самые современные возможности. Именно поэтому акцент в этой книге ставится на долгосрочных аспектах языка программирования, чтобы вы могли подготовиться ко всем грядущим изменениям. Добро пожаловать!

Кому стоит прочесть эту книгу

Книга написана для разработчиков, инженеров и архитекторов, которые достаточно хорошо знают JavaScript и уже успели опробовать свои силы в TypeScript. Вы понимаете базовые концепции типов и способы их применения, а также имеете представление о том, каковы непосредственные преимущества статических типов. Вы находитесь на том этапе, когда процесс становится интересным: вам необходимо иметь более глубокие знания о системе типов и активно работать с TypeScript не только в целях создания надежного и масштабируемого приложения, но и для того, чтобы обеспечить совместную работу между вами и вашими коллегами.

Вы хотите узнать о том, как некие элементы ведут себя в TypeScript, а также понять причины такого поведения. Все это описано в данной книге. Вы узнаете о настройке проекта, особенностях и поведении системы типов, сложных типах и их использовании, а также о работе с фреймворками и применении методологии разработки типов. Эта книга призвана помочь вам пройти путь от новичка до подмастерья и в конечном счете до эксперта. Если вам нужно руководство для активного изучения сложных возможностей TypeScript, а также справочник, который вы сможете использовать на протяжении всей своей карьеры, то эта книга подойдет как нельзя лучше.

Цели написания книги

Главная цель заключалась в том, чтобы сосредоточиться на решении повседневных задач. TypeScript — замечательный язык программирования, а возможности системы типов настолько велики, что мы достигаем точки, когда люди бросают самим себе вызов, решая сложные головоломки TypeScript (<https://tsch.js.org/>). Они интересны, однако часто находятся вне контекста реального мира и поэтому не являются частью этой книги.

Я хочу убедиться, что материал книги отражает то, с чем вы столкнетесь в повседневной жизни как разработчик TypeScript, и имеет отношение к задачам, которые вытекают из реальных ситуаций, и комплексным решениям. Я научу вас приемам и методологиям, которые вы сможете использовать в различных ситуациях, а не только в одном рецепте. По ходу книги вы будете находить ссылки на предыдущие рецепты, показывающие, как конкретный пример может быть применен в новом контексте.

Примеры либо взяты непосредственно из исходного кода реальных проектов, либо сокращены до необходимого минимума, чтобы проиллюстрировать концепцию, не требуя при этом слишком глубоких знаний предметной области. Некоторые примеры довольно необычны, и вы также увидите множество объектов `Person` с именем `Stefan`.

Книга посвящена возможностям TypeScript, добавляемым поверх JavaScript; таким образом, чтобы полностью понимать примеры, вам необходимо иметь базовые знания JavaScript. Я не ожидаю, что вы будете гуру JavaScript, но умение читать базовый код JavaScript является обязательным условием. JavaScript и TypeScript тесно взаимосвязаны, поэтому в некоторых главах книги обсуждаются функции JavaScript и их поведение, но всегда через призму TypeScript.

Эта книга предназначена для того, чтобы дать вам быстрое решение задачи: рецепт. Каждый из них заканчивается *обсуждением*, что позволяет расширить контекст и смысл решения. В зависимости от стиля автора в книгах рецептов O'Reilly акцент делается либо на решении, либо на обсуждении. Данная книга — безусловно, для *обсуждения*.

За свою почти 20-летнюю карьеру человека, пишущего программное обеспечение, я ни разу не сталкивался с ситуациями, когда одно решение подходит для всех задач. Именно поэтому я хочу подробно рассказать вам о том, как я пришел к своим выводам, их смыслу и компромиссам. В конечном счете эта книга должна стать руководством для подобных дискуссий. Зачем делать обоснованные предположения, если у вас есть веские аргументы в пользу своих решений?

Структура издания

Книга поможет вам изучить TypeScript от начала и до конца. Мы начнем с настройки проекта, познакомимся с основными типами и внутренним устройством системы типов, а затем перейдем к более сложным темам, таким как условные и вспомогательные типы. Далее мы перейдем к главам, в которых описаны характерные особенности языка, такие как двойственность классов и поддержка React, и закончим информацией о том, какой способ разработки типов является наилучшим.

Несмотря на то что в книге есть определенный сюжет, каждую главу и каждый рецепт можно читать и использовать по отдельности. Уроки разработаны так, чтобы показать связь с предыдущими (или следующими!) рецептами, но в конечном счете каждая глава является самостоятельной. Вы можете прочесть книгу от начала до конца последовательно или выбирать конкретный рецепт в зависимости от ваших обстоятельств.

Вот краткий обзор содержания книги.

TypeScript необходимо работать со всеми разновидностями JavaScript, а их очень много. В главе 1 «Настройка проекта» вы узнаете о возможностях конфигурации для различных сред выполнения языка, систем модулей и целевых платформ.

В главе 2 «Основные типы» мы поговорим об иерархии типов и разнице между `any` и `unknown`. Вы узнаете, какой код используется в том или ином пространстве имен, и получите ответ на извечный вопрос о том, следует ли выбирать псевдоним типа или интерфейс для описания ваших типов объектов.

Одна из самых длинных глав книги — глава 3 «Система типов». В ней вы узнаете о типах объединения (*union types*) и пересечения (*intersection types*), о том, как определять размеченные типы объединения и использовать приемы *assert never* и *optional never*. Кроме того, мы поговорим о способах, позволяющих сузить и расширить типы в зависимости от вашего сценария использования. Прочитав эту главу, вы поймете, почему в TypeScript есть утверждения типов и нет приведения типов, почему к перечислениям обычно относятся неодобрительно и как найти номинальные элементы в структурной системе типов.

В TypeScript есть система обобщенных типов, которую мы подробно рассмотрим в главе 4 «Обобщенные типы». Эти типы не только делают ваш код более пригодным для повторного использования, но и открывают доступ к расширенным возможностям TypeScript. В этой главе мы переходим от основ TypeScript к более сложным областям системы типов, что является достойным завершением первой части.

В главе 5 «Условные типы» объясняется, почему система типов TypeScript является собственным языком метапрограммирования. Благодаря возможности выбора типов на основе определенных условий люди изобрели выдающиеся вещи, такие как полноценный синтаксический анализатор SQL или словарь в системе типов. Мы используем условные типы как инструмент, позволяющий системе статических типов быть более гибкой в динамических ситуациях.

В главе 6 «Типы литералов шаблонов строк» вы увидите, как TypeScript интегрирует синтаксический анализатор строк в систему типов. Извлечение имен из строк формата, определение динамической системы событий на основе ввода строк и динамическое создание идентификаторов — все это возможно!

В главе 7 «Вариативные типы кортежей» вы познакомитесь с функциональным программированием. *Кортеж* имеет особое значение в TypeScript и помогает описывать параметры функций и объектоподобные массивы, а также создавать гибкие вспомогательные функции.

Еще больше приемов метапрограммирования описано в главе 8 «Вспомогательные типы». В TypeScript есть несколько встроенных вспомогательных типов (*helper types*), которые упрощают вывод типов из других типов. В этой главе вы узнаете не только о том, как их использовать, но и о том, как создавать собственные. Кроме того, данная глава является следующей точкой отсчета, поскольку к этому моменту вы изучите все основные компоненты языка и системы типов, которые сможете применить в следующей части книги.

Восемь предыдущих глав были посвящены изучению тонкостей системы типов. В главе 9 «Стандартная библиотека и определения внешних типов» вы сможете объединить ваши знания с определениями типов, сделанными другими людьми. Вы увидите ситуации, которые работают не так, как ожидалось, и узнаете, как можно использовать встроенные определения типов по своему усмотрению.

В главе 10 «TypeScript и React» вы узнаете о том, как один из самых популярных JavaScript-фреймворков интегрирован в TypeScript, о функциях, которые делают возможным расширение синтаксиса *JSX*, и как это вписывается в общую концепцию TypeScript. К тому же вы узнаете, как создавать надежные типы для компонентов и хуков и работать с файлом определения типа, который был добавлен в реальную библиотеку после ее создания.

Классы — основной элемент объектно-ориентированного программирования, который был доступен в TypeScript задолго до появления их аналогов в JavaScript. Это приводит к интересной двойственности возможностей, которую мы подробно рассмотрим в главе 11 «Классы».

Книга заканчивается главой 12 «Стратегии разработки типов». Акцент в ней сделан на том, чтобы дать вам навыки, позволяющие самостоятельно создавать расширенные типы, принимать правильные решения о том, как продвигать свой проект, и работать с библиотеками, которые проверяют типы за вас. Вдобавок мы поговорим о специальных обходных путях и скрытых возможностях, а также о том, какие имена присваивать обобщенным типам и не являются ли расширенные типы слишком сложными. Эта глава особенно интересна, поскольку после долгого пути от новичка до подмастерья вы достигнете уровня эксперта.

Все примеры доступны в виде интерактивной среды TypeScript Playground или проекта CodeSandbox на сайте книги (<https://typescript-cookbook.com/>). В частности, эти площадки предлагают промежуточное состояние кода, поэтому вы можете самостоятельно пробовать разные варианты его поведения. Я всегда говорю, что невозможно выучить язык программирования, просто читая о нем. Вам нужно активно писать код, чтобы понять, как он работает. Воспринимайте это как приглашение экспериментировать с типами программирования.

Условные обозначения

TypeScript позволяет использовать множество стилей программирования и вариантов форматирования. Чтобы избежать путаницы, я решил автоматически форматировать все примеры с помощью Prettier (<https://prettier.io/>). Если вы привыкли к другому стилю форматирования (например, предпочитаете запятые вместо точек с запятой после каждого объявления свойств ваших типов), то можете продолжать использовать свой стиль.

В книге содержится немало примеров и описано множество функций. Существует немало способов написания функций, и я предпочитаю писать в основном *объявления функций*, а не *функциональные выражения*, за исключением тех случаев, когда необходимо объяснить различия между этими обозначениями. Во всех остальных случаях это скорее дело вкуса, чем технических причин.

Все примеры проверены на TypeScript 5.0, самой последней версии на момент написания данной книги. Язык постоянно меняется, и правила тоже. В книге мы сосредоточимся на долговечных вещах, которым можно доверять в разных версиях. Если ожидается дальнейшее развитие или фундаментальное изменение чего-либо, то в тексте будут даны соответствующие предупреждения и примечания.

Здесь используются следующие условные обозначения.

Курсив

Курсивом выделены новые термины и важные понятия.

Моноширинный шрифт

Используется для листингов программ, а также внутри абзацев для обозначения таких элементов, как переменные и функции, базы данных, типы данных, переменные среды, операторы и ключевые слова, имена файлов и их расширений.

Шрифт без засечек

Используется для обозначения каталогов, путей, URL и адресов электронной почты.



Этот элемент указывает на совет или предложение.



Такой элемент указывает на общее примечание.



Этот элемент указывает на предупреждение.

Использование примеров кода

Дополнительные материалы (примеры кода, упражнения и т. д.) доступны для скачивания на сайте <https://typescript-cookbook.com>.

Эта книга призвана помочь вам выполнять свою работу. В общем случае все примеры кода вы можете использовать в своих программах и документации. Вам не нужно обращаться в издательство O'Reilly за разрешением, если вы не собираетесь воспроизводить существенные части кода, или если вы разрабатываете программу

и используете в ней несколько фрагментов кода из книги, или если вы будете цитировать эту книгу либо примеры из нее, отвечая на вопросы заинтересованных лиц. Вам потребуется разрешение издательства O'Reilly, если вы хотите продавать или распространять примеры из книги либо добавить существенные объемы представленного в ней программного кода в документацию вашего продукта.

Мы рекомендуем, но не требуем добавлять ссылку на первоисточник при цитировании. Под ссылкой на первоисточник мы подразумеваем указание авторов, издательства и ISBN.

За получением разрешения на использование значительных объемов программного кода из книги обращайтесь по адресу permissions@oreilly.com.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу

comp@sprintbook.kz

(издательство «SprintBook», компьютерная редакция).

Мы будем рады узнать ваше мнение!

Благодарности

Александр Роземанн, Себастьян Гирлингер, Доминик Ангерер и Георг Котмайер — первые, к кому я обращаюсь, если мне нужно создать что-то новое. Наши регулярные встречи и общение не только интересны, но и дают мне обратную связь, необходимую для оценки всех моих решений. Они первые, кто услышал о книге, а также первые, кто дал отзывы.

Общение с Мэттом Пококом, Джо Превайтом, Дэном Вандеркамом, Натаном Шивли-Сандерсом и Джошем Голдбергом в социальных сетях принесло множество новых идей. Их подход к TypeScript может отличаться от моего, но в конечном счете они расширили мой кругозор и позаботились о том, чтобы я не стал слишком самоуверенным.

Фил Нэш, Симона Котин и Ванесса Бёнер были не только первыми рецензентами окончательного варианта рукописи, но и давними соратниками и друзьями, которые всегда готовы проверить мои идеи на здравомыслие. Адди Османи вдохновлял меня на протяжении всей моей карьеры, и я очень горжусь тем, что он согласился открыть мою новую книгу.

Лена Мачеко, Александра Рапяну и Майк Кусс без колебаний засыпали меня техническими задачами и вопросами, основанными на их реальном опыте. Там, где мне не хватало наглядного примера, они давали мне отличный исходный материал для анализа.

Я бы потерял счет всем разработкам TypeScript, если бы не Питер Крёнер, который постоянно дает мне знать о выходе новой версии TypeScript. Наши совместные выпуски подкастов, посвященные релизам TypeScript, стали легендарными, и к тому же они все больше и больше не о TypeScript.

Мои технические редакторы Марк Халпин, Фабиан Фридл и Бернхард Майр предоставили технические отзывы, о которых я только мог мечтать. Они подвергли сомнению каждое предположение, проверяли каждый пример кода и убедились, что все мои рассуждения имеют смысл и что я ничего не пропустил. Их любовь к деталям и умение вести дискуссию на высоком уровне сделали эту книгу не просто очередной коллекцией интересных фактов, но руководством и справочником, имеющим прочную основу.

Этой книги не было бы, если бы не Аманда Куинн. Написав книгу *TypeScript in 50 Lessons* в 2020 году, я думал, что сказал об этом языке все, что хотел. Именно Аманда навела меня на мысль о создании книги рецептов и хотела посмотреть, какие мои идеи не попали в мою первую книгу. Через три часа у меня было гото-

вое предложение и оглавление с более чем сотней пунктов. Аманда была права: мне было что сказать, и я бесконечно благодарен ей за поддержку и руководство.

Если Аманда помогала на ранних этапах, то Шира Эванс следила за тем, чтобы проект успешно продвигался и не сорвался. Ее отзывы были бесценны, а ее прагматичный и практичный подход сделал совместную работу очень приятной.

Элизабет Фаерм и Тереза Джонс взяли на себя ответственность за производство. Они чрезвычайно внимательны к деталям и позаботились о том, чтобы этап производства был захватывающим и доставлял массу удовольствия! Конечный результат — это прекрасные впечатления, которыми я не могу насытиться.

Во время написания мне очень помогли Porcupine Tree, Beck, Nobuo Uematsu, Camel, The Beta Band и многие другие.

Самый большой вклад в создание этой книги внесла моя семья. Дорис, Клеменс и Аарон — это все, о чем я когда-либо мечтал, и без их бесконечной любви и поддержки я не смог бы реализовать свои амбиции. Спасибо вам за все.

ГЛАВА 1

Настройка проекта

https://t.me/it_books/2

Итак, вы хотите начать работу с TypeScript! Главный вопрос: с чего начать? Вы можете интегрировать язык в свои проекты разными способами, и все они немного различаются в зависимости от потребностей вашего проекта. JavaScript работает во многих средах исполнения. Точно так же существует множество способов настроить и TypeScript, чтобы он соответствовал вашим потребностям.

В этой главе рассматриваются все возможности внедрения в ваш проект языка TypeScript как дополнения к JavaScript, которое делает возможными базовое автозавершение и индикацию ошибок, а также настройки для полнофункциональных приложений на Node.js и в браузере.

Инструментарий JavaScript — область с безграничными возможностями (некоторые говорят, что новая цепочка сборки JavaScript выпускается каждую неделю, почти так же часто, как и новые фреймворки). Поэтому в данной главе больше внимания уделяется тому, что можно сделать с помощью одного только компилятора TypeScript, без каких-либо дополнительных инструментов.

TypeScript предлагает все необходимое для транспиляции, за исключением возможности создавать минимизированные и оптимизированные пакеты для распространения в Интернете. С этой задачей справляются такие сборщики, как ESBuild (<https://esbuild.github.io/>) или Webpack (<https://webpack.js.org/>). Кроме того, существуют конфигурации, содержащие другие транспиляторы, например Babel.js (<https://babeljs.io/>), которые могут прекрасно работать с TypeScript.

Сборщики и другие транспиляторы не будут рассматриваться в этой главе. Обратитесь к их документации по добавлению TypeScript и используйте информацию из данной главы для правильной настройки конфигурации.

Будучи проектом с более чем десятилетней историей, TypeScript содержит некоторые старые фрагменты, от которых не может просто избавиться ради совместимости. Поэтому в данной главе мы рассмотрим современный синтаксис JavaScript и последние разработки в области веб-стандартов.

Если вам все еще нужно использовать Internet Explorer 8 или Node.js 10, то, во-первых, извините, разрабатывать что-то для этих платформ очень сложно. А во-вторых, вы сможете собрать воедино компоненты для старых платформ, используя информацию из этой главы и официальной документации TypeScript (<https://typescriptlang.org/>).

1.1. Проверка типов JavaScript

Задача

Вы хотите выполнить базовую проверку типов для JavaScript, затратив наименьшее количество усилий.

Решение

Добавьте однострочный комментарий с `@ts-check` в начало каждого файла JavaScript, который хотите проверить. В правильных редакторах вы уже увидите красные волнистые линии, когда TypeScript сталкивается с чем-то не совсем подходящим.

Обсуждение

TypeScript был разработан как надмножество JavaScript, и каждый допустимый файл JavaScript также является допустимым файлом TypeScript. Это означает, что TypeScript очень хорошо справляется с определением потенциальных ошибок в обычном коде JavaScript.

Вы можете использовать его, если вам не нужна полноценная настройка TypeScript, но требуются базовые подсказки и проверки типов, чтобы упростить рабочий процесс разработки.

Наличие редактора или IDE — хорошее предварительное условие, если вы хотите только проверять типы в JavaScript.

Редактор, который отлично сочетается с TypeScript, — это Visual Studio Code (<https://code.visualstudio.com/>). VSCode был первым крупным проектом, в котором использовался TypeScript еще до выпуска этого языка.

Многие рекомендуют VSCode, если вы хотите писать на JavaScript или TypeScript. На самом деле любой редактор хорош, если в нем есть поддержка TypeScript. А в настоящее время большинство из них поддерживают его.

В Visual Studio Code при проверке типа JavaScript появляются красные волнистые линии, когда что-то не сходится (рис. 1.1). Это самый низкий порог входа в TypeScript. Система типов TypeScript имеет разные уровни строгости при работе с кодовой базой.

Сначала система типов будет пытаться *вывести* (infer) типы из кода JavaScript путем использования. Если в вашем коде есть строка наподобие этой:

```
let a_number = 1000;
```

то TypeScript правильно выведет `number` как тип `a_number`.

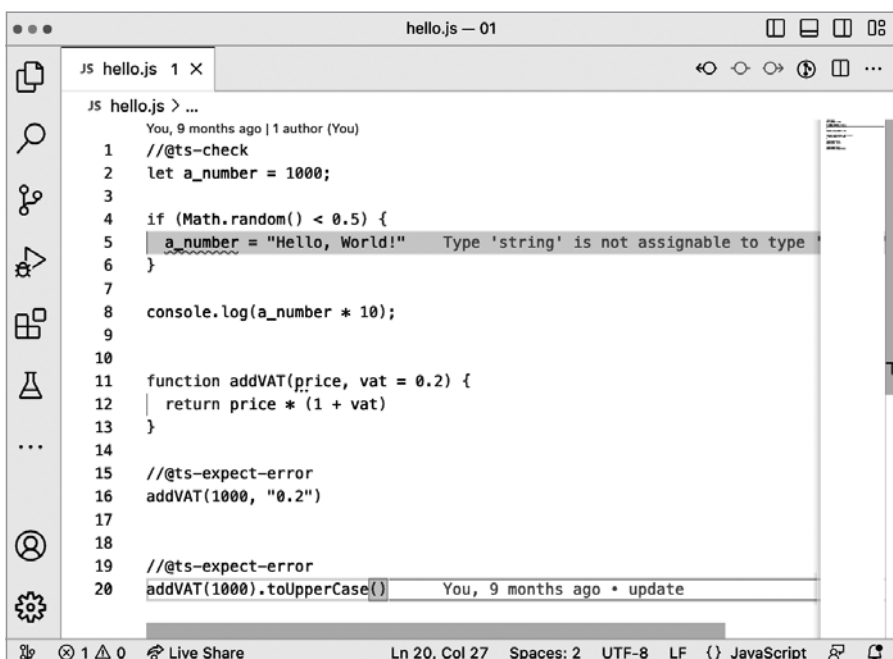


Рис. 1.1. Красные волнистые линии в редакторах кода: обратная связь первого уровня, если что-то в вашем коде не сходится

Одна из трудностей JavaScript состоит в том, что типы являются динамическими. Привязки через `let`, `var` или `const` могут менять тип в зависимости от использования¹.

Рассмотрим такой пример:

```

let a_number = 1000;

if (Math.random() < 0.5) {
  a_number = "Hello, World!";
}

console.log(a_number * 10);
  
```

Мы присваиваем число `a_number` и меняем привязку на `string`, если условие в следующей строке оценивается как `true`. Это не было бы большой проблемой, если бы мы не пытались умножить `a_number` в последней строке. Примерно в 50 % случаев данный пример приведет к нежелательному поведению.

¹ Объекты, имеющие привязку `const`, все еще могут изменять значения и свойства, а значит, и свои типы.

Здесь может помочь TypeScript. Добавив однострочный комментарий с `@ts-check` в самом верху нашего файла JavaScript, TypeScript активирует следующий уровень строгости: проверка типов в файлах JavaScript на основе информации о типе, доступной в файле JavaScript.

В нашем примере TypeScript обнаружит, что мы пытались присвоить привязке `string`, которую TypeScript вывел как число. В редакторе мы получим сообщение об ошибке:

```
// @ts-check
let a_number = 1000;

if (Math.random() < 0.5) {
  a_number = "Hello, World!";
// ^-- Type 'string' is not assignable to type 'number'.ts(2322)
}

console.log(a_number * 10);
```

Теперь мы можем приступить к исправлению кода, а TypeScript будет направлять нас.

Выведение типов для JavaScript имеет большое значение. В следующем примере TypeScript выводит типы, рассматривая такие операции, как умножение и сложение, а также значения по умолчанию:

```
function addVAT(price, vat = 0.2) {
  return price * (1 + vat);
}
```

Функция `addVat` принимает два аргумента. Второй является необязательным, так как для него установлено значение по умолчанию `0.2`. TypeScript выдаст предупреждение, если вы попытаетесь передать значение, которое не подходит:

```
addVAT(1000, "a string");
//      ^-- Argument of type 'string' is not assignable
//      to parameter of type 'number'.ts(2345)
```

Кроме того, поскольку мы используем операции умножения и сложения в теле функции, TypeScript понимает, что мы вернем число из этой функции:

```
addVAT(1000).toUpperCase();
//      ^-- Property 'toUpperCase' does not
//      exist on type 'number'.ts(2339)
```

В некоторых ситуациях требуется нечто большее, чем просто выведение типа. В файлах JavaScript вы можете аннотировать аргументы функций и привязки, используя аннотации типов JSDoc (<https://jsdoc.app/>). JSDoc — это соглашение о комментариях, которое позволяет вам описывать переменные и интерфейсы функций так, чтобы они были читабельны для людей и при этом их могли интерпретировать

машины. TypeScript возьмет ваши аннотации и будет использовать их в качестве типов для системы типов:

```
/** @type {number} */
let amount;

amount = '12';
//      ^-- Argument of type 'string' is not assignable
//      to parameter of type 'number'.ts(2345)

/**
 * Adds VAT to a price
 *
 * @param {number} price The price without VAT
 * @param {number} vat The VAT [0-1]
 *
 * @returns {number}
 */
function addVAT(price, vat = 0.2) {
  return price * (1 + vat);
}
```

JSDoc также позволяет определять новые сложные типы объектов:

```
/**
 * @typedef {Object} Article
 * @property {number} price
 * @property {number} vat
 * @property {string} string
 * @property {boolean=} sold
 */

/**
 * Now we can use Article as a proper type
 * @param {[Article]} articles
 */
function totalAmount(articles) {
  return articles.reduce((total, article) => {
    return total + addVAT(article);
  }, 0);
}
```

Однако синтаксис может показаться немного неуклюжим. В рецепте 1.3 мы рассмотрим более удачные способы аннотирования объектов.

Если у вас есть кодовая база JavaScript, которая хорошо документирована с помощью JSDoc, то добавление одной строки поверх ваших файлов действительно поможет вам, если в коде что-то пойдет не так.

1.2. Установка TypeScript

Задача

Красных волнистых линий в редакторе недостаточно: вам нужны обратная связь с командной строкой, коды состояния, конфигурация и параметры для проверки типа JavaScript и компиляции TypeScript.

Решение

Установите TypeScript через основной реестр пакетов Node: NPM (<https://npmjs.com/>).

Обсуждение

TypeScript написан на языке TypeScript¹, скомпилирован в JavaScript и использует среду выполнения JavaScript Node.js (<https://nodejs.org/>) в качестве основной среды выполнения². Даже если вы не пишете приложения на Node.js, инструментарий для ваших приложений JavaScript будет работать на Node. Поэтому убедитесь, что скачали Node.js с официального сайта (<https://nodejs.org/>), и ознакомьтесь с его инструментами командной строки.

Приступая к новому проекту, обязательно инициализируйте папку проекта с помощью нового файла `package.json`. Он содержит всю информацию для Node и его менеджера пакетов NPM, которая позволяет определить содержимое вашего проекта. Создайте новый файл `package.json` с содержимым по умолчанию в папке вашего проекта, используя инструмент командной строки NPM:

```
$ npm init -y
```



В книге вы будете знакомиться с командами, которые должны выполняться в вашем терминале. Для удобства эти команды будут приводиться в том виде, в каком они представлены в BASH или аналогичных оболочках, доступных для Linux, macOS или подсистемы Windows для Linux. Начальный знак \$ — условное обозначение команды, но вам не нужно его вводить. Обратите внимание, что все команды работают и в обычном интерфейсе командной строки Windows, а также в PowerShell.

¹ В программировании можно создать новый язык на основе существующего. А потом транслировать программы, написанные на новом языке, в код на старом языке. — *Примеч. пер.*

² TypeScript работает и в других средах исполнения JavaScript, таких как Deno и браузер, но они не являются главными целями.

NPM — это менеджер пакетов Node. Он поставляется с интерфейсом командной строки, реестром и другими инструментами, позволяющими устанавливать зависимости. После инициализации файла `package.json` установите TypeScript из NPM. В данном случае он устанавливается как зависимость разработки, то есть TypeScript не будет добавлен, если вы собираетесь опубликовать свой проект как библиотеку в самом NPM:

```
$ npm install -D typescript
```

Вы можете установить TypeScript глобально, чтобы компилятор TypeScript был доступен везде, но я настоятельно рекомендую устанавливать TypeScript отдельно для каждого проекта. В зависимости от того, насколько часто вы взаимодействуете со своими проектами, у вас будут разные версии TypeScript, которые будут синхронизированы с кодом вашего проекта. Глобальная установка (и обновление) TypeScript может привести к повреждению проектов, с которыми вы давно не работали.



Если вы устанавливаете зависимости фронтенда через NPM, то вам понадобится дополнительный инструмент, позволяющий убедиться, что ваш код работает и в браузере, — сборщик. TypeScript не содержит сборщик, работающий с поддерживаемыми системами модулей, поэтому вам необходимо настроить соответствующий инструмент. Часто используются такие инструменты, как Webpack (<https://webpack.js.org/>) и ESBuild (<https://esbuild.github.io/>). Все инструменты предназначены для выполнения TypeScript. Или же вы можете перейти на полную нативную версию (см. рецепт 1.8).

Теперь, когда TypeScript установлен, инициализируйте новый проект TypeScript. Используйте для этого NPX: он позволяет запускать утилиту командной строки, установленную для вашего проекта.

С помощью команды:

```
$ npx tsc --init
```

можно запустить локальную версию компилятора TypeScript вашего проекта и передать флаг `init` для создания нового файла `tsconfig.json`.

Файл `tsconfig.json` — это основной файл конфигурации для вашего проекта TypeScript. Он содержит все необходимые настройки, позволяющие TypeScript понимать, как интерпретировать ваш код, как сделать типы доступными для зависимостей и нужно ли включить или выключить определенные функции.

По умолчанию TypeScript устанавливает эти параметры для вас:

```
{
  "compilerOptions": {
    "target": "es2016",
    "module": "commonjs",
    "esModuleInterop": true,
```

```

    "forceConsistentCasingInFileNames": true,
    "strict": true,
    "skipLibCheck": true
  }
}

```

Рассмотрим их подробнее.

Значение `es2016` параметра `target` означает, что если вы запустите компилятор TypeScript, то он скомпилирует ваши файлы TypeScript в синтаксис, совместимый с ECMAScript 2016. В зависимости от поддерживаемых браузеров или среды вы можете установить либо более свежую версию (версии ECMAScript называются по году выпуска), либо что-то более старое, например `es5`, для тех, кто вынужден поддерживать очень старые версии Internet Explorer. Конечно, я надеюсь, что вам не придется этого делать.

Значение `commonjs` параметра `module` позволит вам писать, используя синтаксис модуля ECMAScript. Но вместо того, чтобы переносить этот синтаксис в вывод, TypeScript скомпилирует его в формат CommonJS. Это означает, что

```
import { name } from "./my-module";
```

```
console.log(name);
//...
```

становится

```
const my_module_1 = require("./my-module");
console.log(my_module_1.name);
```

после компиляции. CommonJS была системой модулей для Node.js и стала очень распространенной из-за популярности Node. С тех пор в Node.js появились и модули ECMAScript, которые мы рассмотрим в рецепте 1.9.

Параметр `esModuleInterop`, равный `true`, гарантирует, что модули, не являющиеся модулями ECMAScript, будут приведены в соответствие со стандартом после импортирования. Параметр `forceConsistentCasingInFileNames` поддерживает использование файловых систем, чувствительных к регистру, помогая взаимодействовать с теми, кто использует файловые системы, нечувствительные к регистру. А параметр `skipLibCheck` предполагает, что установленные вами файлы определения типов (подробнее об этом позже) не содержат ошибок. Поэтому ваш компилятор не станет проверять их и будет работать немного быстрее.

Одной из самых интересных особенностей является параметр `strict`, устанавливающий строгий режим TypeScript. Если задать для него значение `true`, то TypeScript будет вести себя по-другому в некоторых областях. Это позволяет команде разработки TypeScript определять, как должна вести себя система типов.

Если в TypeScript будут внесены критические изменения из-за меняющегося представления о системе типов, то они будут добавлены в строгий режим. Это означает,

что ваш код может быть поврежден, если вы обновите TypeScript и всегда будете работать в строгом режиме.

Чтобы дать вам время адаптироваться к изменениям, TypeScript позволяет включать или выключать определенные функции строгого режима по отдельности.

В дополнение к настройкам по умолчанию я настоятельно рекомендую использовать еще две:

```
{
  "compilerOptions": {
    //...
    "rootDir": "./src",
    "outDir": "./dist"
  }
}
```

Благодаря этому коду TypeScript получает указание забирать исходные файлы из папки `src` и помещать скомпилированные файлы в папку `dist`. Такая настройка позволяет отделить сгенерированные файлы от созданных вами. Конечно, вам придется создать папку `src`; папка `dist` будет создана после компиляции.

А вот и компиляция. Выполнив настройку проекта, создайте файл `index.ts` в папке `src`:

```
console.log("Hello World");
```

Расширение `.ts` указывает на то, что это файл TypeScript. Теперь запустите его, введя в командной строке эту команду:

```
$ npx tsc
```

и посмотрите, как работает компилятор.

1.3. Поддержка типов «в стороне»

Задача

Вы хотите писать обычный JavaScript без дополнительных этапов сборки, но при этом получить некую поддержку редактора и необходимую информацию о типах для ваших функций. Однако вы не хотите определять сложные типы объектов с помощью JSDoc, как было показано в рецепте 1.1.

Решение

Сохраните файлы с определениями типов «в стороне» и запустите компилятор TypeScript в режиме «проверка JavaScript».

Обсуждение

Постепенное внедрение всегда было целью TypeScript. С помощью инструмента, который я назвал типами «в стороне», вы можете описать синтаксис TypeScript для объектных типов и расширенных возможностей, таких как обобщенные и условные типы (см. главу 5), вместо неуклюжих комментариев JSDoc, но по-прежнему использовать JavaScript для написания кода вашего реального приложения.

Где-нибудь в вашем проекте, возможно в папке `@types`, создайте файл определения типа. Его расширение — `.d.ts`, и в отличие от обычных файлов `.ts`, он предназначен для хранения объявлений, но не реального кода.

Здесь вы можете писать свои интерфейсы, псевдонимы типов и сложные типы:

```
// @types/person.d.ts

// Интерфейс для объектов этой структуры
export interface Person {
  name: string;
  age: number;
}

// Интерфейс, расширяющий исходный.
// Это сложно написать только с помощью комментариев JSDoc.
export interface Student extends Person {
  semester: number;
}
```

Обратите внимание, что вы экспортируете интерфейсы из файлов объявлений. Это делается для того, чтобы вы могли импортировать их в файлы JavaScript:

```
// index.js
/** @typedef { import("../@types/person").Person } Person */
```

Комментарий в первой строке дает TypeScript указание импортировать тип `Person` из файла `@types/person` и сделать его доступным под именем `Person`.

Теперь вы можете использовать этот идентификатор для аннотирования параметров функции или объектов точно так же, как это делается с примитивными типами наподобие `string`:

```
// index.js, continued

/**
 * @param {Person} person
 */
function printPerson(person) {
  console.log(person.name);
}
```

Чтобы убедиться в получении обратной связи от редактора, вам все равно нужно добавить однострочный комментарий с `@ts-check` в начале файлов JavaScript (см. рецепт 1.1). Или вы можете настроить свой проект так, чтобы он всегда проверял JavaScript.

Откройте файл `tsconfig.json` и установите флаг `checkJs` в значение `true`. Это позволит извлекать все файлы JavaScript из папки `src` и даст вам постоянную обратную связь об ошибках типа в вашем редакторе. Вы также можете запустить компилятор командой `npm run tsc`, чтобы посмотреть в командной строке, есть ли у вас ошибки. Если вы не хотите, чтобы TypeScript транслировал ваши файлы JavaScript в старые версии JavaScript, то установите значение `noEmit` в `true`:

```
{
  "compilerOptions": {
    "checkJs": true,
    "noEmit": true,
  }
}
```

При этом TypeScript будет просматривать ваши исходные файлы и предоставлять вам всю необходимую информацию о типах, но не затронет ваш код.

Этот прием также известен как масштабируемый. Библиотеки JavaScript, такие как `React` (<https://preactjs.org/>), работают именно так и предоставляют фантастические инструменты как своим пользователям, так и тем, кто вносит свой вклад в разработку.

1.4. Миграция проекта на TypeScript

Задача

Вы хотите получить все преимущества TypeScript для своего проекта, но вам необходимо перенести (`migrate`) всю кодовую базу.

Решение

Переименуйте модули, изменяя расширение каждого файла с `.js` на `.ts`. Используйте несколько опций и функций компилятора, которые помогут вам устранить ошибки.

Обсуждение

Преимущество использования файлов TypeScript вместо файлов JavaScript с типами состоит в том, что ваши типы и реализации находятся в одном файле.

Это улучшает поддержку редактора и дает доступ к большему количеству функций TypeScript, а также повышает совместимость с другими инструментами.

Однако простое переименование всех файлов из `.js` в `.ts`, скорее всего, приведет ко множеству ошибок. Именно поэтому следует переносить файл за файлом и постепенно повышать безопасность типов по мере продвижения.

Самая большая проблема при миграции — то, что вы вдруг имеете дело с проектом на TypeScript, а не на JavaScript. Тем не менее многие ваши модули будут на JavaScript и, не имея информации о типе, не пройдут этап проверки типов.

Упростите себе и TypeScript задачу, отключив проверку типов для JavaScript, но разрешив модулям TypeScript загружать файлы JavaScript и ссылаться на них:

```
{
  "compilerOptions": {
    "checkJs": false,
    "allowJs": true
  }
}
```

Если вы запустите компиляцию командой `npx tsc` теперь, то увидите, что TypeScript извлекает все файлы JavaScript и TypeScript в вашей исходной папке и создает соответствующие файлы JavaScript в вашей целевой папке. TypeScript также преобразует ваш код, чтобы сделать его совместимым с указанной целевой версией.

Если вы работаете с зависимостями, то увидите, что некоторые из них не содержат информации о типе. Это также приведет к ошибкам TypeScript:

```
import _ from "lodash";
//           ^- Could not find a declaration
//           file for module 'lodash'.
```

Установите сторонние определения типов, чтобы избавиться от этой ошибки. Как это сделать, описано в рецепте 1.5.

Выполняя миграцию файлов, вы можете понять, что у вас не выйдет получить все типы для одного файла за один раз. Существуют зависимости, и вы быстро столкнетесь с необходимостью корректировать слишком большое количество файлов, прежде чем сможете найти тот, который вам действительно нужен.

Вы всегда можете решить просто смириться с ошибкой. По умолчанию TypeScript устанавливает параметр компилятора `noEmitOnError` в `false`:

```
{
  "compilerOptions": {
    "noEmitOnError": false
  }
}
```

Это означает, что независимо от количества ошибок в вашем проекте TypeScript будет генерировать файлы результатов, стараясь не блокировать вас. Возможно, вы захотите включить эту настройку после завершения миграции.

В строгом режиме для функционального флага TypeScript `noImplicitAny` установлено значение `true`. Этот флаг гарантирует, что вы не забудете присвоить тип переменной, константе или параметру функции. Даже если это просто `any`:

```
function printPerson(person: any) {
  // Не имеет смысла, но подходит для any
  console.log(person.gobbledegook);
}

// Тоже не имеет смысла, но any это допускает
printPerson(123);
```

Тип `any` — универсальный в TypeScript. С ним совместимо любое значение; кроме того, `any` позволяет получить доступ к любому свойству или вызвать любой метод. Этот тип фактически отключает проверку типов, давая вам немного свободы во время процесса миграции.

В качестве альтернативы можно аннотировать параметры с помощью значения `unknown`. Кроме того, это даст вам возможность передавать в функцию что угодно, но не позволит вам ничего с этим делать, пока вы не узнаете больше информации о типе.

Вы также можете решить игнорировать ошибки, добавив комментарий `@ts-ignore` перед строкой, которую вы хотите исключить из проверки типов. Комментарий `@ts-nocheck` в начале вашего файла полностью отключает проверку типов для этого конкретного модуля.

Директива комментария, которая идеально подходит для миграции, — это `@ts-expect-error`. Она работает так же, как `@ts-ignore`, поскольку поглощает ошибки, возникающие при проверке типов, но выдает красные волнистые линии, если ошибка типа не обнаружена.

При миграции это поможет вам найти места, которые вы успешно перенесли на TypeScript. Когда директив `@ts-expect-error` не останется, миграция завершена:

```
function printPerson(person: Person) {
  console.log(person.name);
}

// Эта ошибка будет пропущена
// @ts-expect-error
printPerson(123);
```

```
function printNumber(nr: number) {  
  console.log(nr);  
}  
  
// v- Unused '@ts-expect-error' directive.ts(2578)  
// @ts-expect-error  
printNumber(123);
```

Самое замечательное в этом приеме то, что вы переключаете внимание на другие обязанности. Обычно нужно убедиться, что вы передаете функции правильные значения; теперь вы можете удостовериться, что функция способна обрабатывать правильные входные данные.

Все возможности устранения ошибок в процессе миграции имеют одну общую черту: они являются явными. Вам нужно явно установить комментарии `@ts-expect-error`, аннотировать параметры функций как `any` или полностью игнорировать файлы при проверке типов. Таким образом, вы всегда сможете найти эти аварийные выходы в процессе миграции и убедиться, что со временем избавились от них.

1.5. Загрузка типов из репозитория Definitely Typed

Задача

Вы используете зависимость, которая не была написана на TypeScript и поэтому не имеет объявлений типов.

Решение

Установите определения типов из репозитория Definitely Typed (<https://oreil.ly/nZ4xZ>), поддерживаемого сообществом.

Обсуждение

Definitely Typed — один из крупнейших и наиболее активных репозиториях на GitHub, в котором собраны высококачественные определения типов TypeScript, разработанные и поддерживаемые сообществом.

Количество поддерживаемых определений типов приближается к 10 000, и редко какая библиотека JavaScript недоступна.

Все определения типов анализируются, проверяются и развертываются в реестре пакетов Node.js NPM в пространстве имен `@types`. На информационном сайте каждого пакета NPM есть индикатор, который показывает, доступны ли определения типов Definitely Typed (рис. 1.2).

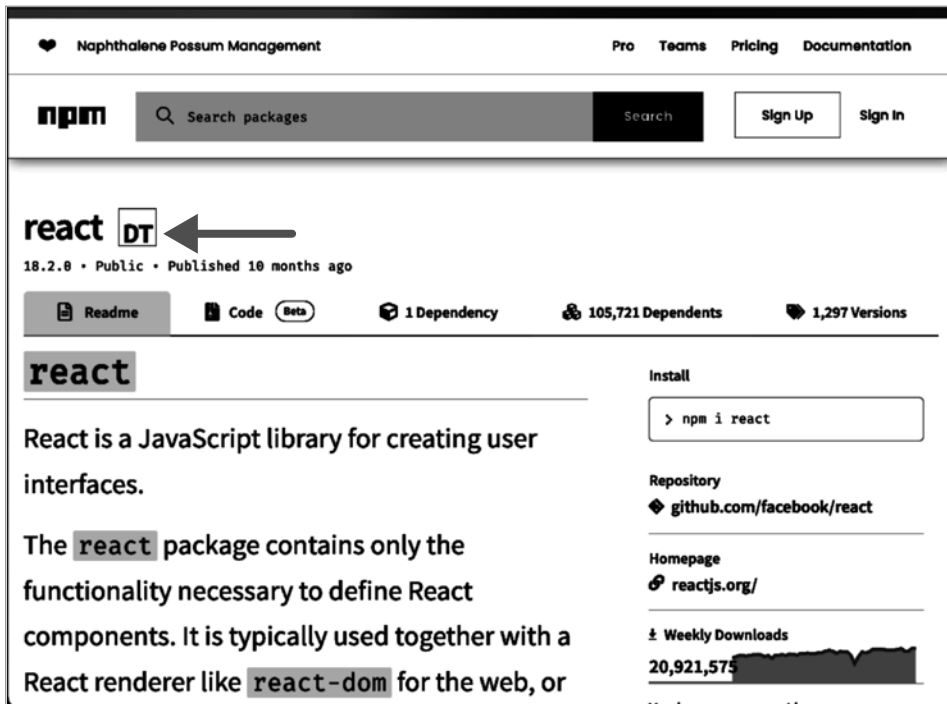


Рис. 1.2. На сайте NPM для React рядом с названием пакета отображается логотип DT. Он указывает на доступные определения типов из Definitely Typed

Щелкнув на логотипе, вы перейдете на сайт с определениями типов. Если в пакете уже есть определения типов от производителя, то рядом с названием пакета отображается небольшой логотип TS (рис. 1.3).

Чтобы установить, например, определения типов для React — популярного фреймворка JavaScript, вы устанавливаете пакет `@types/react` в свои локальные зависимости:

```
# Установка React
$ npm install --save react

# Установка определений типов
$ npm install --save-dev @types/react
```



В этом примере мы устанавливаем типы в зависимости разработки, поскольку используем их в процессе разработки приложения, а в скомпилированном результате типы все равно не используются.

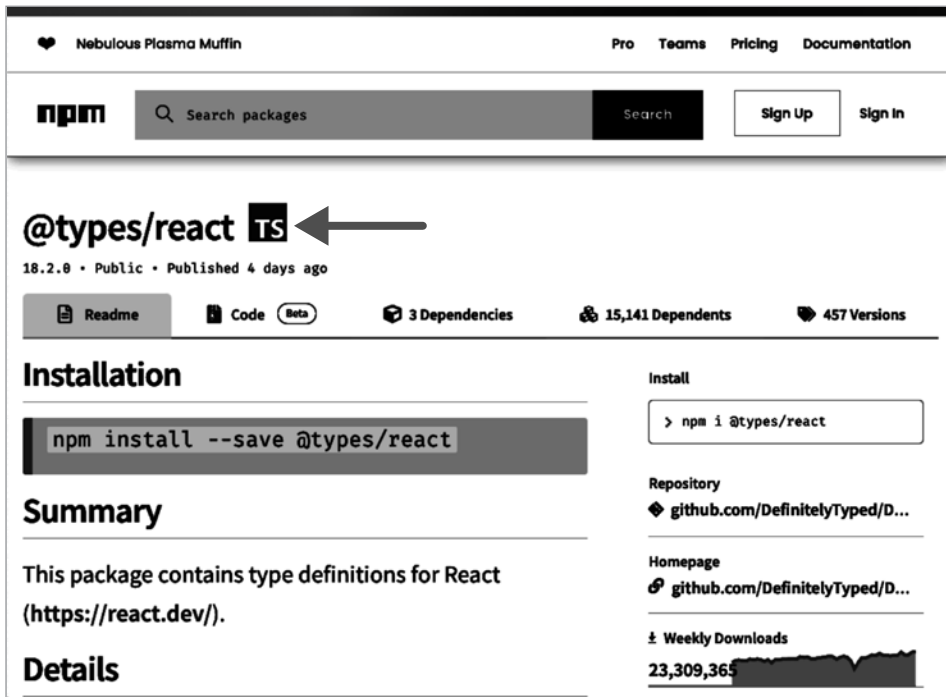


Рис. 1.3. Определения типов для React на Definitely Typed

По умолчанию TypeScript подбирает определения типов, которые может найти в видимых папках `@types` внутри корневой папки вашего проекта. Он также получит все определения типов из `node_modules/@types`; обратите внимание, что именно туда NPM устанавливает, например, `@types/react`.

Мы делаем это потому, что параметр компилятора `typeRoots` в `tsconfig.json` имеет значение `@types` и `./node_modules/@types`. Если вам нужно переопределить этот параметр, то не забудьте добавить исходные папки, если хотите получить определения типов из Definitely Typed:

```
{
  "compilerOptions": {
    "typeRoots": ["./typings", "./node_modules/@types"]
  }
}
```

Обратите внимание, что, просто установив определения типов в `node_modules/@types`, TypeScript будет загружать их во время компиляции. Это означает, что если некоторые типы объявляют глобальные значения, то TypeScript примет их.

Возможно, вы захотите явно указать, каким пакетам разрешено вносить вклад в глобальную область видимости, указав их в параметре `types` в параметрах вашего компилятора:

```
{
  "compilerOptions": {
    "types": ["node", "jest"]
  }
}
```

Обратите внимание, что этот параметр повлияет только на вклад в глобальную область видимости. Если вы загружаете модули узла с помощью инструкций `import`, то TypeScript все равно будет подбирать правильные типы из `@types`:

```
// Если `@types/lodash` установлен, мы получаем определения
// типов свойств для этого пакета NPM
import _ from "lodash"

const result = _.flattenDeep([1, [2, [3, [4]], 5]]);
```

Мы вернемся к этой настройке в рецепте 1.7.

1.6. Настройка полнофункционального проекта

Задача

Вы хотите написать полнофункциональное приложение, ориентированное на Node.js и браузер, с общими зависимостями.

Решение

Создайте два файла `tsconfig` для каждого интерфейса и серверной части и загрузите общие зависимости как композиты.

Обсуждение

И Node.js, и браузер используют JavaScript, но у них совершенно разное понимание того, что разработчики должны делать с этой средой. Node.js предназначен для серверов, инструментов командной строки и всего, что работает без пользовательского интерфейса, — *headless*. У него есть собственный набор API и стандартная библиотека. Этот небольшой сценарий запускает HTTP-сервер:

```
const http = require('http'); ❶

const hostname = '127.0.0.1';
```

```
const port = process.env.PORT || 3000; ❷
const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World');
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`); ❸
});
```

Хотя это, несомненно, JavaScript, некоторые вещи свойственны только Node.js.

- ❶ "http" — встроенный модуль Node.js для всего, что связано с HTTP. Он загружается через `require`, который является индикатором системы модулей Node, называемой *CommonJS*. Существуют и другие способы загрузки модулей в Node.js, как мы увидим в рецепте 1.9, но в последнее время CommonJS является наиболее распространенным.
- ❷ Объект `process` — глобальный объект, содержащий информацию о переменных среды и текущем процессе Node.js в целом. Он тоже уникален для Node.js.
- ❸ Объект `console` и его функции доступны практически во всех средах выполнения JavaScript, но то, что он делает в Node, отличается от того, что он делает в браузере. В Node этот объект выдает результат на `STDOUT`, а в браузере — выводит строку в инструментах разработки.

Конечно, существует еще много уникальных API для Node.js. Но то же самое относится и к JavaScript в браузере:

```
import { msg } из `./msg.js`; ❶

document.querySelector('button')?.addEventListener("click", () => { ❷
  console.log(msg); ❸
});
```

- ❶ Раньше модули нельзя было загружать. После многих лет отсутствия такой возможности модули ECMAScript нашли свое применение в JavaScript и браузерах. Эта строка загружает объект из другого модуля JavaScript. Модуль работает в браузере нативно и является второй системой модулей для Node.js (см. рецепт 1.9).
- ❷ JavaScript в браузере предназначен для взаимодействия с событиями пользовательского интерфейса. Объект `document` и идея `querySelector`, указывающего на элементы в *объектной модели документа* (Document Object Model, DOM), уникальны для браузера, как и добавление прослушвателя событий и прослушивание событий `click`. В Node.js этого нет.
- ❸ И снова объект `console`. Он имеет тот же API, что и в Node.js, но результат немного другой.

Различия настолько велики, что трудно создать один проект TypeScript, который будет работать и с Node.js, и с браузером. Если вы пишете полнофункциональное приложение, то вам нужно создать два конфигурационных файла TypeScript, которые будут работать с каждой частью вашего стека.

Сначала поработаем над серверной частью. Предположим, что вы хотите создать сервер Express.js в Node.js (Express — это популярный серверный фреймворк для Node). Сначала создайте новый проект NPM, как было показано в рецепте 1.1. Затем установите Express в качестве зависимости:

```
$ npm install --save express
```

И установите определения типов для Node.js и Express из Definitely Typed:

```
$ npm install -D @types/express @types/node
```

Создайте новую папку `server`. Именно в нее будет помещен ваш код Node.js. Вместо того чтобы создавать новый файл `tsconfig.json` с помощью `tsc`, создайте новый файл `tsconfig.json` в папке `server` вашего проекта. Вот его содержимое:

```
// server/tsconfig.json
{
  "compilerOptions": {
    "target": "ESNext",
    "lib": ["ESNext"],
    "module": "commonjs",
    "rootDir": "./",
    "moduleResolution": "node",
    "types": ["node"],
    "outDir": "../dist/server",
    "esModuleInterop": true,
    "forceConsistentCasingInFileNames": true,
    "strict": true,
    "skipLibCheck": true
  }
}
```

Многое из этого вам уже известно, но несколько моментов все же стоит отметить.

- Свойству `module` присвоено значение `commonjs`, соответствующее исходной системе модулей Node.js. Все операторы `import` и `export` будут транспилированы в их аналог CommonJS.
- Свойству `types` присвоено значение `["node"]`. Это свойство содержит все библиотеки, которые вы хотите сделать глобально доступными. Если `"node"` находится в глобальной области видимости, то вы получите информацию о типах для `require`, `process` и других специфических библиотек Node.js, которые находятся в глобальном пространстве.

Чтобы скомпилировать ваш серверный код, выполните команду:

```
$ npx tsc -p server/tsconfig.json
```

Теперь для клиента:

```
// client/tsconfig.json
{
  "compilerOptions": {
    "target": "ESNext",
    "lib": ["DOM", "ESNext"],
    "module": "ESNext",
    "rootDir": "./",
    "moduleResolution": "node",
    "types": [],
    "outDir": "../../dist/client",
    "esModuleInterop": true,
    "forceConsistentCasingInFileNames": true,
    "strict": true,
    "skipLibCheck": true
  }
}
```

Есть некоторое сходство, но, опять же, имеется и несколько отличительных черт.

- Вы добавляете `DOM` в свойство `lib`. Это позволит вам определять типы для всего, что связано с браузером. Если раньше вам нужно было устанавливать типы Node.js с помощью Definitely Typed, то TypeScript предоставляет самые последние определения типов для браузера вместе с компилятором.
- Массив `types` пуст. Это *удалит* `"node"` из наших глобальных типов. Вы можете устанавливать определения типов только в каждом `package.json`, поэтому определения типов `"node"`, которые мы задали ранее, будут доступны во всей кодовой базе. Однако для клиентской части вы хотите избавиться от них.

Чтобы скомпилировать код внешнего интерфейса, выполните команду:

```
$ npx tsc -p client/tsconfig.json
```

Пожалуйста, обратите внимание, что были настроены два разных файла `tsconfig.json`. Такие редакторы, как Visual Studio Code, получают информацию о конфигурации только для файлов `tsconfig.json` в каждой папке. Вы можете назвать их `tsconfig.server.json` и `tsconfig.client.json` и поместить их в корневую папку проекта (и настроить все свойства каталога). `tsc` будет использовать правильные конфигурации и выдавать ошибки, если обнаружит их, но редактор в основном будет хранить молчание или работать с конфигурацией по умолчанию.

Ситуация немного усложняется, если вы хотите иметь общие зависимости. Один из способов добиться этого — использовать ссылки на проекты и составные проекты. Это означает, что вы извлекаете общий код в отдельную папку, но сообщаете TypeScript, что данный проект должен быть зависимым от другого проекта.

Создайте папку `shared` на том же уровне, что и папки `client` и `server`. Создайте файл `tsconfig.json` в папке `shared` с таким содержимым:

```
// shared/tsconfig.json
{
  "compilerOptions": {
    "composite": true,
    "target": "ESNext",
    "module": "ESNext",
    "rootDir": "../shared/",
    "moduleResolution": "Node",
    "types": [],
    "declaration": true,
    "outDir": "../dist/shared",
    "esModuleInterop": true,
    "forceConsistentCasingInFileNames": true,
    "strict": true,
    "skipLibCheck": true
  },
}
```

Еще раз обращают на себя внимание два момента.

- Для флага `composite` установлено значение `true`. Это позволяет другим проектам ссылаться на данный проект.
- Флаг `declaration` также установлен в значение `true`. Это позволит генерировать файлы `d.ts` из вашего кода, чтобы другие проекты могли использовать информацию о типах.

Чтобы включить их в код клиента и сервера, добавьте следующую строку в файлы `client/tsconfig.json` и `server/tsconfig.json`:

```
// server/tsconfig.json
// client/tsconfig.json
{
  "compilerOptions": {
    // То же, что и раньше
  },
  "references": [
    { "path": "../shared/tsconfig.json" }
  ]
}
```

Теперь все готово. Вы можете написать общие зависимости и добавить их в свой клиентский и серверный код.

Однако здесь есть один нюанс. Схема прекрасно работает, если вы делитесь, например, только моделями и информацией о типах. Но как только вы поделитесь реальной функциональностью, то увидите, что две разные системы модулей (CommonJS в Node, модули ECMAScript в браузере) не могут быть объединены

в одном скомпилированном файле. Вы либо создаете модуль ESNext и не можете импортировать его в код CommonJS, либо создаете код CommonJS и не можете импортировать его в браузер.

Вы можете выполнить два этих действия:

- скомпилировать в CommonJS и позволить сборщику позаботиться о разрешении работы модулей в браузере;
- скомпилировать в модули ECMAScript и написать современные Node.js приложения, основанные на модулях ECMAScript. Дополнительная информация доступна в рецепте 1.9.

Вы начинаете с нуля, поэтому я настоятельно рекомендую второй вариант.

1.7. Настройка тестов

Задача

Вы хотите писать тесты, но глобальные переменные для фреймворков тестирования мешают работе вашего промышленного кода.

Решение

Создайте отдельный проект `tsconfig` для разработки и сборки и исключите из него все тестовые файлы.

Обсуждение

В экосистеме JavaScript и Node.js существует множество фреймворков для модульного тестирования и средств запуска тестов. Они различаются в деталях, имеют разные подходы или предназначены для определенных нужд. Некоторые из них могут быть просто красивее других.

Одни средства запуска тестов, такие как Ava (<https://oreil.ly/R6xFr>), используют импорт модулей, чтобы получить доступ к фреймворку. Другие средства предоставляют набор глобальных переменных. Возьмем, к примеру, Mocha (<https://mochajs.org/>):

```
import assert from "assert";
import { add } from "..";

describe("Adding numbers", () => {
  it("should add two numbers", () => {
    assert.equal(add(2, 3), 5);
  });
});
```

`assert` происходит из встроенной в Node.js библиотеки `assertion`, а `describe`, `it` и многие другие являются глобальными переменными, предоставляемыми Mocha. Кроме того, они существуют лишь во время работы интерфейса командной строки Mocha CLI.

Данный факт создает некоторые трудности при настройке типов, поскольку эти функции необходимы для написания тестов, но недоступны при выполнении реального приложения.

Решение состоит в том, чтобы создать два разных конфигурационных файла: обычный `tsconfig.json` для разработки, который может быть использован вашим редактором (вспомните рецепт 1.6), и отдельный `tsconfig.build.json`, который вы используете, когда захотите скомпилировать ваше приложение.

Первый файл содержит все необходимые глобальные переменные, в том числе типы для Mocha; второй гарантирует, что ни один тестовый файл не будет добавлен в вашу компиляцию.

Пройдем весь процесс поэтапно. В качестве примера мы рассматриваем Mocha, но другие программы запуска тестов, предоставляющие глобальные переменные, например Jest (<https://jestjs.io/>), работают точно так же. Сначала установите Mocha и ее типы:

```
$ npm install --save-dev mocha @types/mocha @types/node
```

Создайте новый файл `tsconfig.base.json`. Разработка и сборка различаются лишь набором добавляемых файлов и активируемых библиотек, поэтому вам нужно, чтобы все остальные настройки компилятора находились в одном файле, который можно повторно использовать для обоих вариантов. Пример файла для приложения Node.js выглядит так:

```
// tsconfig.base.json
{
  "compilerOptions": {
    "target": "esnext",
    "module": "commonjs",
    "esModuleInterop": true,
    "forceConsistentCasingInFileNames": true,
    "strict": true,
    "outDir": "./dist",
    "skipLibCheck": true
  }
}
```

Исходные файлы должны быть расположены в папке `src`, тестовые — в соседней папке `test`. Кроме того, настройка, которую вы создадите в этом рецепте, позволит вам создавать файлы, заканчивающиеся на `.test.ts`, в любом месте вашего проекта.

Создайте новый файл `tsconfig.json` с базовой конфигурацией разработки. Этот файл используется для обратной связи с редактором и запуска тестов с помощью Mocha. Вы расширяете базовые настройки из файла `tsconfig.base.json` и сообщаете TypeScript, какие папки выбрать для компиляции:

```
// tsconfig.json
{
  "extends": "./tsconfig.base.json",
  "compilerOptions": {
    "types": ["node", "mocha"],
    "rootDirs": ["test", "src"]
  }
}
```

Обратите внимание, что вы добавляете типы для Node и Mocha. Свойство `types` определяет, какие глобальные переменные доступны, и в настройках разработки у вас есть и те и другие.

Кроме того, вы можете заметить, что компиляция тестов перед их выполнением является обременительной задачей. Есть подходы, которые помогут сократить время компиляции тестов. Например, команда `ts-node` запускает вашу локальную установку Node.js и сначала выполняет компиляцию TypeScript в памяти:

```
$ npm install --save-dev ts-node
$ npx mocha -r ts-node/register tests/*.ts
```

Когда среда разработки настроена, можно переходить к среде сборки. Создайте файл `tsconfig.build.json`. Он выглядит так же, как `tsconfig.json`, но вы сразу заметите разницу:

```
// tsconfig.build.json
{
  "extends": "./tsconfig.base.json",
  "compilerOptions": {
    "types": ["node"],
    "rootDirs": ["src"]
  },
  "exclude": ["**/*.test.ts", "**/test/**"]
}
```

Помимо изменения типов и корневых каталогов, вы определяете, какие файлы следует исключить из проверки типов и компиляции. Вы используете паттерны подстановочных знаков, которые исключают все файлы, заканчивающиеся на `.test.ts`, расположенные в папках `test`. В зависимости от ваших предпочтений вы также можете добавить в этот массив папки `.spec.ts` или `spec`.

Скомпилируйте свой проект, обратившись к нужному JSON-файлу:

```
$ npx tsc -p tsconfig.build.json
```

В файлах результатов (расположенных в `dist`) вы не увидите ни одного тестового файла. Кроме того, вы по-прежнему можете обращаться к `describe` и `it` при редактировании исходных файлов, но при попытке компиляции получите ошибку:

```
$ npx tsc -p tsconfig.build.json

src/index.ts:5:1 - error TS2593: Cannot find name 'describe'.
Do you need to install type definitions for a test runner?
Try `npm i --save-dev @types/jest` or `npm i --save-dev @types/mocha`
and then add 'jest' or 'mocha' to the types field in your tsconfig.
5 describe("this does not work", () => {})
  ~~~~~
```

Found 1 error in src/index.ts:5

Если вам не нравится засорять свои глобальные переменные в режиме разработки, то вы можете выбрать настройку, аналогичную той, которая описана в рецепте 1.6, но она не позволит вам писать тесты рядом с вашими исходными файлами.

Наконец, вы всегда можете выбрать программу для тестирования, которая предпочитает модульную систему.

1.8. Типизация модулей ECMAScript из URL

Задача

Вы хотите работать без сборщиков и использовать возможности браузера по загрузке модулей для своего приложения, но при этом иметь всю информацию о типе.

Решение

Установите для `target` и `module` в параметрах компилятора `tsconfig` значение `esnext` и укажите на ваши модули с расширением `.js`. Кроме того, установите типы в зависимости через NPM и используйте свойство `paths` в `tsconfig`, чтобы указать TypeScript, где искать типы:

```
// tsconfig.json
{
  "compilerOptions": {
    "target": "esnext",
    "module": "esnext",
    "paths": {
      "https://esm.sh/lodash@4.17.21": [
        "node_modules/@types/lodash/index.d.ts"
      ]
    }
  }
}
```

Обсуждение

Современные браузеры поддерживают загрузку модулей «из коробки». Вместо того чтобы собирать приложение в более компактный набор файлов, вы можете напрямую использовать необработанные файлы JavaScript.

Сети доставки контента (Content Delivery Networks, CDN), такие как `esm.sh` (<https://esm.sh/>), `unpkg` (<https://unpkg.com/>) и другие, предназначены для распространения модулей узлов и зависимостей JavaScript в виде URL, которые можно использовать для загрузки нативных модулей ECMAScript.

Благодаря правильному кэшированию и современному HTTP модули ECMAScript становятся реальной альтернативой для приложений.

TypeScript не содержит современного сборщика, поэтому вам в любом случае придется устанавливать дополнительный инструмент. Но если вы решите сначала перейти на модуль, то при работе с TypeScript следует учитывать несколько моментов.

Чего вы хотите добиться, так это написать операторы `import` и `export` на TypeScript, но сохранить синтаксис загрузки модулей и позволить браузеру обрабатывать разрешение модулей:

```
// Файл module.ts
export const obj = {
  name: "Stefan",
};

// Файл index.ts
import { obj } from "./module";

console.log(obj.name);
```

Чтобы добиться этого, дайте TypeScript следующие указания.

1. Выполнить компиляцию в версию ECMAScript, которая понимает модули.
2. Использовать синтаксис модуля ECMAScript для генерации кода модуля.

Обновите два свойства в вашем файле `tsconfig.json`:

```
// tsconfig.json
{
  "compilerOptions": {
    "target": "esnext",
    "module": "esnext"
  }
}
```

Параметр `module` указывает TypeScript, как преобразовывать операторы `import` и `export`. По умолчанию загрузка модулей выполняется в CommonJS, как было показано в рецепте 1.2. При установке значения `module` в `esnext` будет использоваться загрузка модулей ECMAScript, и благодаря этому сохранится синтаксис.

Параметр `target` указывает TypeScript на версию ECMAScript, в которую вы хотите перенести свой код. Раз в год выходит новый релиз ECMAScript с новыми возможностями. Если установить `target` в `esnext`, то всегда будет использоваться последняя версия ECMAScript.

В зависимости от целей совместимости вы можете установить для этого свойства версию ECMAScript, совместимую с браузерами, которые хотите поддерживать. Обычно это версия с указанием года (например, `es2015`, `es2016`, `es2017` и т. д.). Модули ECMAScript работают со всеми версиями начиная с `es2015`. Если вы выберете более старую версию, то не сможете загружать модули ECMAScript в браузере.

Изменение этих параметров компилятора уже делает одну важную вещь: оставляет синтаксис нетронутым. Проблема возникает, когда вы хотите запустить свой код.

Обычно операторы `import` в TypeScript указывают на файлы без расширения. Вы пишете `import { obj } from './module'`, опустив расширение `.ts`. После компиляции это расширение по-прежнему отсутствует. Но браузеру необходимо расширение, которое фактически указывало бы на соответствующий файл JavaScript.

Решение: добавьте расширение `.js`, даже если при разработке указываете на файл `.ts`. TypeScript достаточно умен, чтобы понять это:

```
// index.ts

// Это по-прежнему загружает типы из 'module.ts', но сохраняет
// ссылку после компиляции нетронутой
import { obj } from './module.js';

console.log(obj.name);
```

Для модулей вашего проекта это все, что вам нужно!

Все становится намного интереснее, когда вы хотите использовать зависимости. Если вы работаете с нативной версией, то, возможно, захотите загружать модули из CDN, например `esm.sh` (<https://esm.sh/>):

```
import _ from "https://esm.sh/lodash@4.17.21"
//           ^- Error 2307

const result = _.flattenDeep([1, [2, [3, [4]], 5]]);

console.log(result);
```

TypeScript выдаст ошибку со следующим сообщением: `Cannot find module ... or its corresponding type declarations. (2307)` (Невозможно найти модуль... или соответствующие ему объявления типов. (2307))

Разрешение модуля TypeScript работает, когда файлы находятся на диске, а не загружаются с сервера по протоколу HTTP. Чтобы получить требуемую информацию, нужно предоставить TypeScript собственное разрешение.

Мы загружаем зависимости из URL, однако информация о типе для этих зависимостей хранится в NPM. Для `lodash` можно установить информацию о типе из Definitely Typed:

```
$ npm install -D @types/lodash
```

Для зависимостей, которые поставляются с собственными типами, вы можете установить их напрямую:

```
$ npm install -D preact
```

После установки типов используйте свойство `paths` в параметрах компилятора, чтобы указать TypeScript, как разрешить ваш URL:

```
// tsconfig.json
{
  "compilerOptions":
  {
    // ...
    "paths": {
      "https://esm.sh/lodash@4.17.21": [
        "node_modules/@types/lodash/index.d.ts"
      ]
    }
  }
}
```

Убедитесь, что указали на правильный файл!

Кроме того, есть запасной вариант, если вы не хотите использовать типы или просто не можете их найти. В TypeScript можно использовать тип `any`, чтобы намеренно отключить проверку типов. Для модулей можно сделать нечто очень похожее — игнорировать ошибку TypeScript:

```
// @ts-ignore
import _ from "https://esm.sh/lodash@4.17.21"
```

Параметр `ts-ignore` удаляет *следующую* строку из проверки типов и может быть использован везде, где вы хотите игнорировать ошибки типов (см. рецепт 1.4). Это фактически означает, что вы не получите никакой информации о типах для ваших зависимостей и можете столкнуться с ошибками, но данное решение может быть идеальным в случае старых зависимостей, которые вам просто необходимы, но вы не можете найти никаких типов для них.

1.9. Загрузка различных типов модулей в Node

Задача

Вы хотите использовать модули ECMAScript в Node.js и функцию взаимодействия CommonJS для библиотек.

Решение

Установите разрешение модуля TypeScript на "nodeNext" и дайте своим файлам имена с расширением `.mts` или `.cts`.

Обсуждение

С появлением Node.js система модулей CommonJS стала одной из самых популярных в экосистеме JavaScript.

Идея проста и эффективна: определить экспорт в одном модуле и потребовать его в другом:

```
// person.js
function printPerson(person) {
  console.log(person.name);
}

exports = {
  printPerson,
};

// index.js
const person = require("./person");
person.printPerson({ name: "Stefan", age: 40 });
```

Эта система оказала огромное влияние на модули ECMAScript, а также была использована по умолчанию для разрешения модулей и транспилятора TypeScript. Если вы посмотрите на синтаксис модулей ECMAScript в примере 1.1, то увидите, что ключевые слова допускают различные транспиляции. Это означает, что при настройке модуля `commonjs` операторы `import` и `export` преобразуются в `require` и `exports`.

Пример 1.1. Использование системы модулей ECMAScript

```
// person.ts
type Person =
{
  name: string;
```

```

    age: number;
  };

export function printPerson(person) {
  console.log(person.name);
}

// index.ts
import * as person from "./person";
person.printPerson({ name: "Stefan", age: 40 });

```

По мере стабилизации модулей ECMAScript Node.js тоже начал их внедрять. Основы обеих модульных систем кажутся очень похожими; тем не менее есть некоторые различия в деталях, такие как обработка экспорта по умолчанию или асинхронная загрузка модулей ECMAScript.

Нет возможности применять обе системы модулей одинаково, но с разным синтаксисом, поэтому разработчики Node.js решили предоставить возможность использовать обе системы и назначили разные расширения файлов, позволяющие указать предпочтительный тип модуля. В табл. 1.1 показаны различные расширения, как они называются в TypeScript, во что TypeScript их компилирует и что они могут импортировать. Благодаря функциональной совместимости CommonJS можно импортировать модули CommonJS из модулей ECMAScript, но не наоборот.

Таблица 1.1. Расширения модулей и что они импортируют

Расширения	TypeScript	Компилируется в	Может импортировать
.js	.ts	CommonJS	.js, .cjs
.cjs	.cts	CommonJS	.js, .cjs
.mjs	.mts	Модули ES	.js, .cjs, .mjs

Разработчики библиотек, которые публикуются на NPM, получают дополнительную информацию в своем файле `package.json`, позволяющую обозначить основной тип пакета (`module` или `commonjs`) и указать на список основных или резервных файлов, чтобы загрузчики модулей могли подобрать нужный файл:

```

// package.json
{
  "name": "dependency",
  "type": "module",
  "exports": {
    ".": {
      // Точка входа для `import "dependency"` в модулях ES
      "import": "./esm/index.js",

```

```

    // Точка входа для `require("dependency")` в CommonJS
    "require": "./commonjs/index.cjs",
  },
},
// Резервный вариант CommonJS
"main": "./commonjs/index.cjs"
}

```

В TypeScript вы пишете в основном в синтаксисе модулей ECMAScript и позволяете компилятору решать, какой формат модуля создать в итоге. Сейчас их, возможно, два: CommonJS и модули ECMAScript.

Чтобы сделать возможными оба варианта, вы можете установить разрешение модуля в `tsconfig.json` на `NodeNext`:

```

{
  "compilerOptions": {
    "module": "NodeNext"
    // ...
  }
}

```

С этим флагом TypeScript подберет нужные модули, описанные в файле `package.json`, распознает окончания `.mts` и `.cts` и будет следовать данным табл. 1.1 при импорте модулей.

Для вас как для разработчика существуют различия в импорте файлов. CommonJS не требовал расширений при импорте, поэтому TypeScript по-прежнему поддерживает импорт без расширений. Код из примера 1.1 (см. выше) по-прежнему работает, если вы используете только CommonJS.

Импорт с помощью расширений файлов, как и в рецепте 1.8, позволяет импортировать модули как в модулях ECMAScript, так и в модулях CommonJS:

```

// index.mts
import * as person from "./person.js"; // работает в обоих вариантах
person.printPerson({ name: "Stefan", age: 40});

```

Если совместимость с CommonJS не будет работать, то вы всегда можете воспользоваться оператором `require`. Добавьте `"node"` в качестве глобальных типов в параметры компилятора:

```

// tsconfig.json
{
  "compilerOptions": {
    "module": "NodeNext",
    "types": ["node"],
  }
}

```

Затем импортируйте с помощью этого специфического для TypeScript синтаксиса:

```
// index.mts
import person = require("./person.cjs");

person.printPerson({ name: "Stefan", age: 40 });
```

В модуле CommonJS это будет просто еще один вызов `require`; в модулях ECMAScript код будет содержать вспомогательные функции Node.js:

```
// скомпилированный index.mts
import { createRequire as _createRequire } from "module";
const __require = _createRequire(import.meta.url);
const person = __require("./person.cjs");
person.printPerson({ name: "Stefan", age: 40 });
```

Обратите внимание, что это снизит совместимость со средами выполнения, отличающимися от Node.js, такими как браузер, но в итоге может решить проблемы совместимости.

1.10. Работа с Deno и зависимостями

Задача

Вы хотите использовать TypeScript с Deno, современной средой выполнения JavaScript для приложений вне браузера.

Решение

Это просто: TypeScript уже встроен в эту среду.

Обсуждение

Deno — современная среда выполнения JavaScript, созданная теми же людьми, которые разрабатывали Node.js. Deno во многом похожа на Node.js, но имеет и существенные различия:

- использует стандарты веб-платформ для своих основных API; это значит, вам будет легче переносить код из браузера на сервер;
- разрешает доступ к файловой системе или сети, только если вы явно активируете его;
- обрабатывает зависимости не через централизованный реестр, а — опять же с помощью функций браузера — через URL.

А еще она поставляется со встроенными инструментами разработки и TypeScript!

Deno — это инструмент с самым низким порогом входа, если вы хотите попробовать TypeScript. Не нужно загружать никакой другой инструмент (компилятор `tsc` уже встроен) или настраивать TypeScript. Вы пишете файлы `.ts`, а Deno делает все остальное:

```
// main.ts
function sayHello(name: string) {
  console.log(`Hello ${name}`);
}

sayHello("Stefan");

$ deno run main.ts
```

TypeScript от Deno может делать все то же самое, что и `tsc`, и обновляется с каждым обновлением этой среды. Однако есть некоторые различия, когда вы хотите настроить его.

Во-первых, конфигурация по умолчанию отличается от конфигурации по умолчанию, выдаваемой командой `tsc --init`. Флаги функций строгого режима устанавливаются по-другому, и в конфигурацию входит поддержка React (на стороне сервера!).

Чтобы внести изменения в конфигурацию, необходимо создать файл `deno.json` в корневой папке. Deno автоматически обнаружит его, если вы не запретите ему это делать. Файл `deno.json` содержит несколько настроек для среды выполнения Deno, в том числе параметры компилятора TypeScript:

```
{
  "compilerOptions": {
    // Ваши параметры компилятора TSC
  },
  "fmt": {
    // Параметры автоформата
  },
  "lint": {
    // Параметры для линтера
  }
}
```

Вы можете ознакомиться с дополнительными возможностями на сайте Deno (<https://deno.com/>).

Во-вторых, библиотеки по умолчанию тоже различаются. Deno поддерживает стандарты веб-платформ и имеет совместимые с браузерами API, однако в нее внесены некоторые сокращения из-за отсутствия графического интерфейса пользователя. Поэтому отдельные типы библиотек — например, библиотека DOM — конфликтуют с тем, что предоставляет среда.

Некоторые библиотеки могут быть вам интересны:

- `deno.ns` — пространство имен Deno по умолчанию;
- `deno.window` — глобальный объект для Deno;
- `deno.worker` — эквивалент для Web Workers в среде выполнения Deno.

DOM и подмножества добавлены в Deno, но не включены по умолчанию. Если ваше приложение предназначено как для браузера, так и для Deno, то настройте среду на добавление всех библиотек браузера и Deno:

```
// deno.json
{
  "compilerOptions": {
    "target": "esnext",
    "lib": ["dom", "dom.iterable", "dom.asynciterable", "deno.ns"]
  }
}
```

Aleph.js (<https://alephjs.org/>) — это пример фреймворка, который предназначен как для Deno, так и для браузера.

Кроме того, в Deno другой способ распространения информации о типе зависимостей. Внешние зависимости загружаются в среду по URL из CDN. Сама среда размещает свою стандартную библиотеку по адресу <https://deno.land/std>.

Но вы можете использовать такие CDN, как `esm.sh` (<https://esm.sh/>) или `unpkg` (<https://unpkg.com/>) (см. рецепт 1.8). Они распространяют типы, отправляя заголовков `X-TypeScript-Types` вместе с HTTP-запросом, показывая Deno, что нужно загрузить объявления типов. Это относится и к зависимостям, которые не имеют сторонних объявлений типов, но используют Definitely Typed.

Таким образом, в тот момент, когда вы устанавливаете зависимость, Deno получит не только исходные файлы, но и всю информацию о типах.

Если вы не загружаете зависимость из CDN, а размещаете ее локально, то можете указать на файл объявления типов в момент импорта зависимости:

```
// @deno-types="./charting.d.ts"
import * as charting from "./charting.js";
```

или добавьте ссылку на типизацию в саму библиотеку:

```
// charting.js
///
```

Эта ссылка также называется *директивой тройной косой черты* и является функцией TypeScript, а не Deno. Существуют различные такие директивы, в основном используемые для систем зависимостей модулей, существовавших до появления

языка ECMAScript. В документации TypeScript (<https://oreil.ly/EvUWm>) дается очень хороший обзор. Но если вы применяете модули ECMAScript, то, скорее всего, не будете использовать директивы тройной косой черты.

1.11. Использование предопределенных конфигураций

Задача

Вы хотите использовать TypeScript для определенного фреймворка или платформы, но не знаете, с чего начать настройку.

Решение

Используйте предопределенную конфигурацию из `tsconfig/bases` (<https://oreil.ly/ljsVT>) и расширьте ее на основе этой конфигурации.

Обсуждение

Напомню, что на сайте Definitely Typed размещаются поддерживаемые сообществом определения типов для популярных библиотек. Точно так же на сайте `tsconfig/bases` (<https://oreil.ly/ljsVT>) размещается набор поддерживаемых сообществом рекомендаций по конфигурациям TypeScript, которые вы можете использовать в качестве отправной точки для собственного проекта. Речь идет о таких фреймворках, как Ember.js, Svelte или Next.js, и о средах выполнения JavaScript, таких как Node.js и Deno.

Конфигурационные файлы сведены к минимуму и содержат в основном рекомендуемые библиотеки, модули и целевые настройки, а также множество флагов строгого режима, которые имеют смысл для соответствующей среды.

Например, это рекомендуемая конфигурация для Node.js 18 с настройками строгого режима и модулями ECMAScript:

```
{
  "$schema": "https://json.schemastore.org/tsconfig",
  "display": "Node 18 + ESM + Strictest",
  "compilerOptions": {
    "lib": [
      "es2022"
    ],
    "module": "es2022",
    "target": "es2022",
    "strict": true,
  }
}
```



```
"esModuleInterop": true,  
"skipLibCheck": true,  
"forceConsistentCasingInFileNames": true,  
"moduleResolution": "node",  
"allowUnusedLabels": false,  
"allowUnreachableCode": false,  
"exactOptionalPropertyTypes": true,  
"noFallthroughCasesInSwitch": true,  
"noImplicitOverride": true,  
"noImplicitReturns": true,  
"noPropertyAccessFromIndexSignature": true,  
"noUncheckedIndexedAccess": true,  
"noUnusedLocals": true,  
"noUnusedParameters": true,  
"importsNotUsedAsValues": "error",  
"checkJs": true  
}  
}
```

Чтобы использовать эту конфигурацию, установите ее через NPM:

```
$ npm install --save-dev @tsconfig/node18strictest-esm
```

и подключите ее к собственной конфигурации TypeScript:

```
{  
  "extends": "@tsconfig/node18-strictest-esm/tsconfig.json",  
  "compilerOptions": {  
    // ...  
  }  
}
```

При этом все настройки будут взяты из predefined конфигурации. Теперь вы можете приступить к настройке собственных свойств, например корневого и внешнего каталогов.

ГЛАВА 2

Основные типы

https://t.me/it_books/2

Теперь, когда все настроено, пришло время написать немного кода на TypeScript! Начинать должно быть легко, но вскоре вы столкнетесь с ситуациями, когда не будете уверены в том, что поступаете правильно. Следует использовать интерфейсы или псевдонимы типов? Нужно делать аннотации или позволить выведению типов сделать свое дело? Безопасно ли использовать `any` и `unknown`? Некоторые пользователи в Интернете говорят, что эти типы никогда не следует использовать, так почему же они являются частью TypeScript?

В этой главе вы найдете ответы на все эти вопросы. Вы узнаете об основных типах, из которых состоит TypeScript, и о том, как их будет использовать опытный разработчик TypeScript. Эта глава служит фундаментом для следующих. Из нее вы получите представление о том, как компилятор TypeScript получает свои типы и интерпретирует ваши аннотации.

Речь идет о взаимодействии вашего кода, редактора и компилятора, а также о продвижении вверх и вниз по иерархии типов, как будет показано в рецепте 2.3. Кем бы вы ни были: опытным разработчиком TypeScript или новичком — в этой главе вы найдете полезную для вас информацию.

2.1. Эффективное аннотирование

Задача

Аннотирование типов — это трудоемкая и скучная задача.

Решение

Делайте аннотации только в тех случаях, когда хотите проверить свои типы.

Обсуждение

Аннотация типа позволяет явно указать, какие типы следует ожидать. Вы знаете, как это делается в других языках программирования, где длинное выражение `StringBuilder stringBuilder = new StringBuilder()` гарантирует, что вы действи-

тельно имеете дело со `StringBuilder`. Противоположностью является выведение типа, когда TypeScript пытается определить тип за вас:

```
// Выведение типа
let aNumber = 2;
// aNumber: number

// Аннотация типа
let anotherNumber: number = 3;
// anotherNumber: number
```

Кроме того, аннотации типов служат наиболее очевидным и заметным синтаксическим различием между TypeScript и JavaScript.

Когда вы начинаете изучать TypeScript, вам, возможно, захочется добавить аннотации везде, чтобы обозначить ожидаемые типы. Это может показаться очевидным, но вы также можете использовать аннотации экономно и позволить TypeScript определять типы за вас.

Аннотация типа позволяет указать места, где необходимо проверять контракты. Если вы добавляете аннотацию типа к объявлению переменной, то даете компилятору указание проверить, совпадают ли типы во время присваивания:

```
type Person = {
  name: string;
  age: number;
};

const me: Person = createPerson();
```

Если `createPerson` возвращает что-то, несовместимое с `Person`, то TypeScript выдаст ошибку. Добавляйте аннотацию типа к объявлению переменной, если действительно хотите быть уверены, что имеете дело с правильным типом.

Кроме того, с этого момента `me` имеет тип `Person`, и TypeScript будет рассматривать его как `Person`. Если у `me` есть другие свойства, например `profession`, то TypeScript не позволит вам получить к ним доступ. Они не определены в `Person`.

Если вы добавляете аннотацию типа к возвращаемому значению сигнатуры функции, то даете компилятору указание проверить совпадение типов в момент возврата этого значения:

```
function createPerson(): Person {
  return { имя: "Stefan", возраст: 39 };
}
```

Если вы возвращаете что-то, что не соответствует `Person`, то TypeScript выдаст ошибку. Добавляйте аннотацию типа к возвращаемому значению сигнатуры функции, если хотите быть полностью уверены, что возвращаете правильный тип. Это особенно удобно, если вы работаете с функциями, которые создают большие объекты из различных источников.

Если вы добавляете аннотацию типа к параметрам сигнатуры функции, то даете компилятору указание проверять соответствие типов в момент передачи аргументов:

```
function printPerson(person: Person) {
  console.log(person.name, person.age);
}

printPerson(me);
```

На мой взгляд, это самая важная и необходимая аннотация типа. Все остальное может быть выведено:

```
type Person = {
  name: string;
  age: number;
};

// Выведен!
// return type is { name: string, age: number }
function createPerson() {
  return { name: "Stefan", age: 39 };
}

// Выведен!
// me: { name: string, age: number}
const me = createPerson();

// Аннотировано! Вам нужно проверить, совместимы ли типы
function printPerson(person: Person) {
  console.log(person.name, person.age);
}

// Все работает
printPerson(me);
```

Вы можете задействовать выводимые типы объектов там, где ожидаете увидеть аннотацию, поскольку в TypeScript используется *система структурных типов*. В ней компилятор будет учитывать только члены (свойства) типа, а не фактическое имя.

Типы совместимы, если все члены проверяемого типа доступны в типе значения. Мы также говорим, что *форма* или *структура* типа должна совпадать:

```
type Person = {
  name: string;
  age: number;
};

type User = {
  name: string;
  age: number;
```

```
    id: number;
};

function printPerson(person: Person) {
    console.log(person.name, person.age);
}

const user: User = {
    name: "Stefan",
    age: 40,
    id: 815,
};

printPerson(user); // работает!
```

У типа `User` больше свойств, чем у `Person`, но все свойства, которые есть у `Person`, есть и у `User`, и они имеют один и тот же тип. Именно поэтому в `printPerson` можно передавать объекты `User`, даже если между типами нет явной связи.

Но если вы передадите литерал, то TypeScript сообщит, что в нем есть лишние свойства, которых там быть не должно:

```
printPerson({
    name: "Stefan",
    age: 40,
    id: 1000,
    // ^- Argument of type '{ name: string; age: number; id: number; }'
    // is not assignable to parameter of type 'Person'.
    // Object literal may only specify known properties,
    // and 'id' does not exist in type 'Person'.(2345)
});
```

Это позволяет убедиться в том, что вы не ожидали наличия свойств в данном типе, а потом задаться вопросом, почему их изменение не дало никакого эффекта.

С помощью системы структурных типов вы можете создавать интересные шаблоны с переменными-носителями, имеющими выведенный тип, а также повторно использовать одну и ту же переменную в разных частях вашего программного обеспечения, не связывая их друг с другом подобным образом:

```
type Person = {
    name: string;
    age: number;
};

type Studying = {
    semester: number;
};

type Student = {
    id: string;
```

```
    age: number;
    semester: number;
};

function createPerson() {
    return { name: "Stefan", age: 39, semester: 25, id: "XPA" };
}

function printPerson(person: Person) {
    console.log(person.name, person.age);
}

function studyForAnotherSemester(student: Studying) {
    student.semester++;
}

function isLongTimeStudent(student: Student) {
    return student.age - student.semester / 2 > 30 && student.semester > 20;
}

const me = createPerson();

// Все работает!
printPerson(me);
studyForAnotherSemester(me);
isLongTimeStudent(me);
```

Типы `Student`, `Person` и `Studying` частично совпадают, но не связаны друг с другом. `createPerson` возвращает значение, совместимое со всеми тремя типами. Если вы добавили слишком много аннотаций, то вам потребуется создать гораздо больше типов и провести куда больше проверок, чем нужно, без какой-либо пользы.

Поэтому добавляйте аннотации везде, где хотите, чтобы ваши типы проверялись, по крайней мере, для аргументов функций.

2.2. Работа с `any` и `unknown`

Задача

В TypeScript есть два высших типа (top type): `any` и `unknown`. Какой из них следует использовать?

Решение

Используйте `any`, если действительно хотите отключить проверку типа, и `unknown`, если вам нужно соблюдать осторожность.

Обсуждение

И `any`, и `unknown` являются высшими типами; это означает, что с ними совместимо каждое значение:

```
const name: any = "Stefan";
const person: any = { name: "Stefan", age: 40 };
const notAvailable: any = undefined;
```

Поскольку `any` — это тип, с которым совместимо любое значение, то вы можете получить доступ к свойству данного типа без ограничений:

```
const name: any = "Stefan";
// Допустимо для TypeScript, но приведет к сбою в JavaScript
console.log(name.profession.experience[0].level);
```

Добавок `any` совместим со всеми подтипами, кроме `never`. Это означает, что вы можете сузить набор возможных значений, назначив новый тип:

```
const me: any = "Stefan";
// Хорошо!
const name: string = me;
// Плохо, но подойдет для системы типов.
const age: number = me;
```

Будучи таким гибким, `any` может стать постоянным источником потенциальных ошибок и ловушек, поскольку вы фактически отключаете проверку типов.

Все, кажется, согласны с тем, что не стоит использовать `any` в своих кодовых базах, однако есть некоторые ситуации, когда этот тип действительно полезен.

Миграция

Когда вы переходите с JavaScript на TypeScript, скорее всего, у вас уже есть кодовая база с большим количеством неявной информации о том, как работают ваши структуры данных и объекты. Прописать все за один раз может оказаться утомительной задачей. Тип `any` может помочь вам постепенно перейти на более безопасную кодовую базу.

Нетипизированные сторонние зависимости

Возможно, у вас есть JavaScript-зависимость, которая все еще отказывается использовать TypeScript (или что-то подобное). Или, что еще хуже, для нее не существует актуальных типов. `Definitely Typed` — отличный ресурс, но его сопровождением занимаются добровольцы. Это формализация того, что существует в JavaScript, но не является прямым производным от него. Там могут быть ошибки (даже в таких популярных определениях типов, как у `React`), или они просто могут быть устаревшими!

Именно здесь `any` может вам помочь. Если вы знаете, как работает библиотека, если документация достаточно хороша для того, чтобы помочь вам приступить

к работе, и если вы используете ее нечасто, то можно использовать `any`, а не бороться с типами.

Прототипирование на JavaScript

TypeScript работает немного иначе, чем JavaScript, и ему приходится идти на множество компромиссов, чтобы избежать крайних случаев. Это также означает, что если вы напишете некий код, который будет работать в JavaScript, то в TypeScript получите ошибки:

```
type Person = {
  name: string;
  age: number;
};

function printPerson(person: Person) {
  for (let key in person) {
    console.log(`${key}: ${person[key]}`);
    // Элемент неявно имеет тип 'any' --^
    // поскольку выражение типа 'string'
    // не может быть использовано для индексирования типа 'Person'.
    // Не найдена сигнатура индекса с параметром типа 'string'
    // для типа 'Person'.(7053)
  }
}
```

О том, почему это ошибка, читайте в рецепте 9.1. В подобных случаях `any` может помочь вам на время отключить проверку типов, ведь вы знаете, что делаете. А поскольку вы можете переходить от любого типа к `any`, а также обратно к любому другому типу, то у вас есть небольшие явные небезопасные блоки по всему коду, в которых вы отвечаете за происходящее:

```
function printPerson(person: any) {
  for (let key in person) {
    console.log(`${key}: ${person[key]}`);
  }
}
```

Как только вы убедитесь, что эта часть кода работает, вы можете начать добавлять нужные типы, обходить ограничения TypeScript и вводить утверждения типов:

```
function printPerson(person: Person) {
  for (let key in person) {
    console.log(`${key}: ${person[key as keyof Person]}`);
  }
}
```

Всякий раз, когда вы используете тип `any`, обязательно активируйте флаг `noImplicitAny` в вашем файле `tsconfig.json`. По умолчанию он активирован в режиме `strict`. TypeScript требует, чтобы вы явно аннотировали `any`, когда тип

не определен с помощью выведения или аннотации. Это помогает в дальнейшем находить потенциально проблемные ситуации.

Альтернативой `any` является `unknown`. Он позволяет использовать те же значения, но действия, которые вы можете делать с его помощью, сильно различаются. Если `any` позволяет делать все, то `unknown` не позволяет делать ничего. Все, что вам доступно, — это передача значений. В тот момент, когда вы хотите вызвать функцию или сделать тип более конкретным, вам сначала нужно выполнить проверку типа:

```
const me: unknown = "Stefan";
const name: string = me;
//   ^- Тип 'unknown' не может быть присвоен типу 'string'.(2322)
const age: number = me;
//   ^- Тип 'unknown' не может быть присвоен типу 'number'.(2322)
```

Проверка типов и анализ потока управления помогут вам добиться большего с помощью `unknown`:

```
function doSomething(value: unknown) {
  if (typeof value === "string") {
    // значение: string
    console.log("It's a string ", value.toUpperCase());
  } else if (typeof value === "number") {
    // значение: number
    console.log("it's a number ", value * 2);
  }
}
```

Если ваши приложения работают с большим количеством различных типов, то `unknown` отлично помогает убедиться в том, что вы можете переносить значения по всему коду, но при этом не столкнетесь с проблемами безопасности, вызванными гибкими возможностями `any`.

2.3. Выбор правильного типа объекта

Задача

Вы хотите разрешить использование значений, которые являются объектами JavaScript, но есть три различных типа объектов: `object`, `Object` и `{}`. Какой из них следует использовать?

Решение

Используйте `object` для составных типов, таких как объекты, функции и массивы. Используйте `{}` для всего, что имеет значение.

Обсуждение

TypeScript делит свои типы на две ветви. В первую — *примитивные типы* — входят `number`, `boolean`, `string`, `symbol`, `bigint` и некоторые подтипы. Во вторую — *составные типы* — входит все, что является подтипом объекта и в итоге состоит из других составных типов или примитивных типов.

На рис. 2.1 представлен общий обзор.

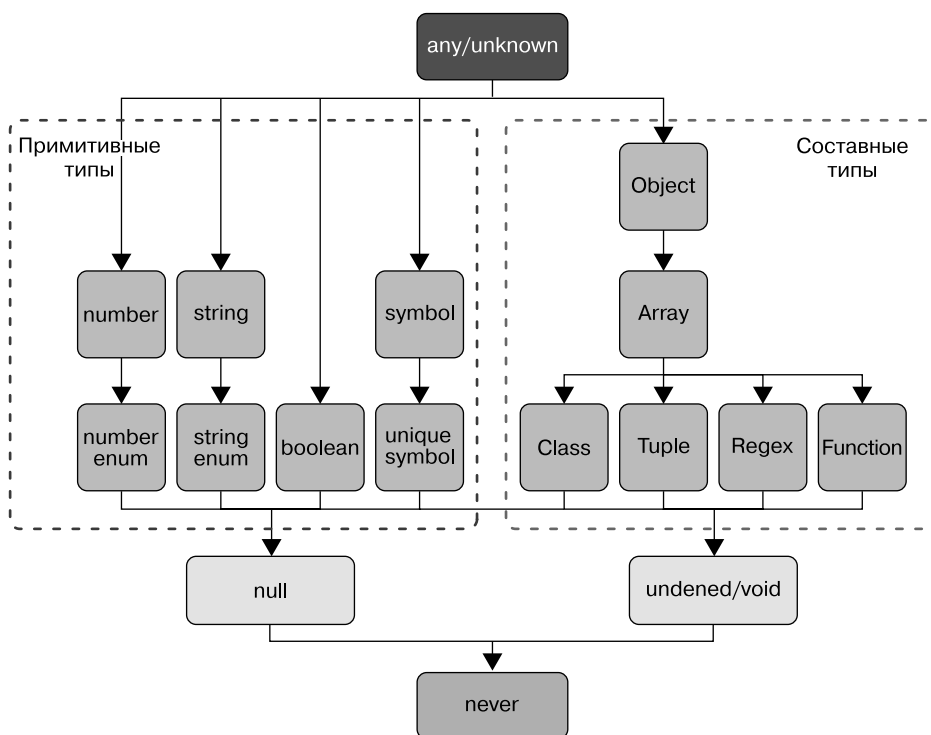


Рис. 2.1. Иерархия типов в TypeScript

В некоторых ситуациях вам нужны значения, которые являются составными типами, либо потому, что вы хотите изменить определенные свойства, либо потому, что просто хотите быть уверенными в том, что не передадите никаких примитивных значений. Например, `Object.create` создает новый объект и принимает его прототип в качестве первого аргумента. Это может быть только составной тип; в противном случае ваш код JavaScript во время выполнения завершит работу с ошибкой:

```

Object.create(2);
// Uncaught TypeError: Object prototype may only be an Object or null: 2
//   at Function.create (<anonymous>)
  
```

В TypeScript три типа — тип пустого объекта `{}`, интерфейс объекта `O` в верхнем регистре и тип объекта `o` в нижнем регистре — по-видимому, делают одно и то же. Какой из них вы используете для составных типов?

Типы `{}` и `Object` допускают примерно одинаковые значения, которые могут быть любыми, кроме `null` или `undefined` (при условии, что активирован режим `strict` или `strictNullChecks`):

```
let obj: {}; // Similar to Object
obj = 32;
obj = "Hello";
obj = true;
obj = () => { console.log("Hello") };
obj = undefined; // Ошибка
obj = null; // Ошибка
obj = { name: "Stefan", age: 40 };
obj = [];
obj = /.*/;
```

Интерфейс `Object` совместим со всеми значениями, имеющими прототип `Object`, то есть со всеми значениями каждого примитивного и составного типа.

Однако `Object` — это определенный интерфейс в TypeScript, и у него есть некие требования к конкретным функциям. Например, метод `toString`, который представляет собой `toString() => string` и является частью любого ненулевого значения, — часть прототипа `Object`. Если присвоить значение с помощью другого метода `toString`, то TypeScript выдаст ошибку:

```
let okObj: {} = {
  toString() {
    return false;
  }
}; // OK

let obj: Object = {
  toString() {
    return false;
  }
}
// ^- Type 'boolean' is not assignable to type 'string'.ts(2322)
```

Тип `Object` может вызвать некоторую путаницу из-за такого поведения, поэтому в большинстве случаев лучше использовать `{}`.

В TypeScript также есть тип `object`, написанный в *нижнем регистре*. Это, скорее всего, тип, который вам нужен, поскольку он позволяет использовать любые составные типы, но не примитивные:

```
let obj: object;
obj = 32; // Ошибка
obj = "Hello"; // Ошибка
```

```
obj = true; // Ошибка
obj = () => { console.log("Hello") };
obj = undefined; // Ошибка
obj = null; // Ошибка
obj = { name: "Stefan", age: 40 };
obj = [];
obj = /.*/;
```

Если вам нужен тип, исключающий функции, регулярные выражения, массивы и тому подобное, то обратитесь к главе 5, в которой мы создадим такой тип самостоятельно.

2.4. Работа с типами кортежей

Задача

Вы используете массивы JavaScript для организации своих данных. Порядок важен, как и типы в каждой позиции. Но выведение типов в TypeScript делает работу с ним очень трудоемкой.

Решение

Аннотируйте типы кортежей.

Обсуждение

Как и применение объектов, использование массивов JavaScript — популярный способ организации данных в сложном объекте. Вместо того чтобы писать типичный объект `Person`, как мы делали в других рецептах, можно хранить записи поэлементно:

```
const person = ["Stefan", 40]; // Имя и возраст
```

Преимущество использования массивов перед объектами состоит в том, что у элементов массива нет имен свойств. Когда вы присваиваете каждому элементу переменные с помощью деструктуризации, становится очень легко присваивать им собственные имена:

```
// objects.js
// Using objects
const person = {
  name: "Stefan",
  age: 40,
};

const { name, age } = person;
```

```

console.log(name); // Stefan
console.log(age); // 40

const { anotherName = name, anotherAge = age } = person;

console.log(anotherName); // Stefan
console.log(anotherAge); // 40

// arrays.js
// Using arrays
const person = ["Stefan", 40]; // Имя и возраст

const [name, age] = person;

console.log(name); // Stefan
console.log(age); // 40

const [anotherName, anotherAge] = person;

console.log(anotherName); // Stefan
console.log(anotherAge); // 40

```

В случае API, в которых вам нужно постоянно назначать новые имена, использовать массивы очень удобно, как объясняется в главе 10.

Однако при использовании TypeScript и выведении типов этот паттерн может вызвать некоторые проблемы. По умолчанию TypeScript выводит тип массива на основе присваивания. Массивы представляют собой бесконечные коллекции с одним и тем же элементом в каждой позиции:

```

const person = ["Stefan", 40];
// person: (string | number)[]

```

Таким образом, TypeScript считает, что `person` — это массив, каждый элемент которого может быть либо строкой, либо числом и допускает множество элементов после двух исходных. Это означает, что при деструктуризации каждый элемент будет иметь тип `string` или `number`:

```

const [name, age] = person;
// name: string | number
// age: string | number

```

Это делает паттерн, удобный в JavaScript, очень трудоемким в TypeScript. Вам придется проверять поток управления, чтобы сузить тип до фактического, когда из назначения должно быть ясно, что в этом нет необходимости.

Всякий раз, когда вы считаете, что вам нужно выполнить дополнительную работу в JavaScript только для того, чтобы удовлетворить требования TypeScript, обычно есть более хороший вариант. В этом случае вы можете использовать типы кортежей, чтобы уточнить, как нужно интерпретировать ваш массив.

Типы кортежей являются родственными типам массивов, которые работают с другой семантикой. Массивы могут быть потенциально бесконечного размера, и каждый элемент имеет один и тот же тип (независимо от того, насколько он широк), а типы кортежей имеют фиксированный размер, и у каждого элемента — определенный тип.

Чтобы получить типы кортежей, достаточно их явно аннотировать:

```
const person: [string, number] = ["Stefan", 40];
```

```
const [name, age] = person;  
// name: string  
// age: number
```

Фантастика! Типы кортежей имеют фиксированную длину. Это означает, что длина тоже кодируется в типе. Поэтому присваивания, выходящие за допустимые границы, невозможны; TypeScript выдаст ошибку:

```
person[1] = 41; // ОК!  
person[2] = false; // Ошибка  
//^- Type 'false' is not assignable to type 'undefined'.(2322)
```

Кроме того, TypeScript позволяет добавлять метки к типам кортежей. Это всего лишь метаинформация для редакторов и обратной связи с компилятором, но она позволяет более четко определить, чего ожидать от каждого элемента:

```
type Person = [name: string, age: number];
```

Это поможет вам и вашим коллегам понять, чего ожидать, как и в случае с типами объектов.

Типы кортежей можно использовать и для аннотирования аргументов функций. Эту функцию:

```
function hello(name: string, msg: string): void {  
  // ...  
}
```

можно написать и с помощью типов кортежей:

```
function hello(...args: [name: string, msg: string]) {  
  // ...  
}
```

И это дает очень гибкий подход к ее определению:

```
function h(a: string, b: string, c: string): void {  
  //...  
}  
// equal to  
function h(a: string, b: string, ...r: [string]): void {  
  //...  
}
```

```
// equal to
function h(a: string, ...r: [string, string]): void {
  //...
}
// equal to
function h(...r: [string, string, string]): void {
  //...
}
```

Эти элементы также известны как *остаточные* элементы, которые есть в JavaScript и позволяют определять функции с почти неограниченным списком аргументов. Когда остаточный элемент становится последним элементом, то поглощает все лишние аргументы. Когда вам нужно собрать аргументы в коде, вы можете использовать кортеж, прежде чем задействовать их в своей функции:

```
const person: [string, number] = ["Stefan", 40];

function hello(...args: [name: string, msg: string]): {
  // ...
}

hello(...person);
```

Типы кортежей полезны во многих ситуациях. Дополнительные сведения о типах кортежей приведены в главах 7 и 10.

2.5. Различия между интерфейсами и псевдонимами типов

Задача

В TypeScript типы объектов объявляются двумя способами: с помощью интерфейсов и псевдонимов типов. Какой из них следует использовать?

Решение

Используйте псевдонимы типов для типов, находящихся в пределах вашего проекта, а интерфейсы — для контрактов, предназначенных для применения другими пользователями.

Обсуждение

Оба подхода к определению типов объектов были предметом обсуждения во многих статьях в блогах на протяжении долгих лет. И все они со временем устарели. На момент написания этой книги разница между псевдонимами типов и интерфейсами

практически отсутствует. А все, что *отличалось*, постепенно было приведено в соответствие.

Синтаксически разница между интерфейсами и псевдонимами типов имеет нюансы:

```
type PersonAsType = {
  name: string;
  age: number;
  address: string[];
  greet(): string;
};

interface PersonAsInterface {
  name: string;
  age: number;
  address: string[];
  greet(): string;
}
```

Вы можете использовать интерфейсы и псевдонимы типов для одних и тех же целей, в одних и тех же ситуациях:

- в объявлении реализации для классов;
- в качестве аннотации типа для объектных литералов;
- для структур рекурсивного типа.

Однако есть одно важное различие, которое может привести к побочным эффектам, с которыми обычно не хочется иметь дело: интерфейсы допускают слияние объявлений, а псевдонимы типов — нет. Слияние объявлений позволяет добавлять свойства в интерфейс даже после того, как он был объявлен:

```
interface Person {
  name: string;
}

interface Person {
  age: number;
}

// Person сейчас { name: string; age: number; }
```

TypeScript часто использует этот прием в файлах `lib.d.ts`, что позволяет просто добавлять новые API JavaScript на основе версий ECMAScript. Это отличная возможность, если вы хотите расширить, например, `Window`, но она может иметь неприятные последствия в других ситуациях, допустим, таких:

```
// Некоторые данные мы собираем в веб-форме
interface FormData {
  name: string;
```



```
    age: number;
    address: string[];
}

// Функция, которая отправляет эти данные на сервер
function send(data: FormData) {
    console.log(data.entries()) // это компилируется!
    // Но происходит сбой во время выполнения
}
```

Итак, откуда взялся метод `entries()`? Это DOM API! `FormData` — один из интерфейсов, предоставляемых API браузера, а их множество. Они общедоступны, и ничто не мешает вам расширять эти интерфейсы. И вы не получите никаких уведомлений, если сделаете это.

Вы, конечно, можете спорить о правильности именования, но проблема сохраняется для всех интерфейсов, которые вы делаете доступными глобально, возможно, из какой-то зависимости, даже не подозревая, что она добавляет интерфейс в глобальное пространство.

Изменив этот интерфейс на псевдоним типа, вы сразу же узнаете об этой проблеме:

```
type FormData = {
//  ^-- Duplicate identifier 'FormData'.(2300)
    name: string;
    age: number;
    address: string[];
};
```

Слияние объявлений — отличная функция, если вы создаете библиотеку, которая будет использоваться в других частях вашего проекта, а может быть, и в сторонних проектах, полностью написанных другими командами. Она позволяет вам определить интерфейс, который описывает ваше приложение, но дает вашим пользователям возможность адаптировать его к своим требованиям.

Представьте систему плагинов, в которой загрузка новых модулей расширяет функциональность: слияние объявлений — это функция, которую вы не хотели бы потерять.

Однако в границах вашего модуля использование псевдонимов типов предотвращает случайное повторное применение или расширение уже объявленных типов. Псевдонимы типов — хороший выбор, если вы не ожидаете, что их будут применять другие пользователи.

Производительность

Использование псевдонимов типов вместо интерфейсов вызвало некоторую дискуссию, так как интерфейсы считались гораздо более производительными при оценке, чем псевдонимы типов, что даже привело к появлению рекомендации

по производительности в официальной Вики TypeScript (<https://oreil.ly/8Y0hP>). Эту рекомендацию не следует воспринимать буквально.

При создании простые псевдонимы типов могут работать быстрее, чем интерфейсы, поскольку последние никогда не закрываются и могут быть объединены с другими объявлениями. Но в иных ситуациях интерфейсы могут работать быстрее, так как заранее известно, что они относятся к типам объектов. Райан Канава из команды TypeScript ожидает, что разница в производительности будет заметна при большом количестве объявленных интерфейсов или псевдонимов типов: около пяти тысяч, согласно этому твиту (https://oreil.ly/Y_2oS).

Если ваша кодовая база TypeScript работает плохо, это не потому, что вы объявили слишком много псевдонимов типов вместо интерфейсов или наоборот.

2.6. Определение перегрузок функций

Задача

API вашей функции очень гибкий и позволяет использовать аргументы различных типов, где важен контекст. Это сложно определить в единственной сигнатуре функции.

Решение

Используйте перегрузки функций.

Обсуждение

JavaScript очень гибок, когда речь идет об аргументах функций. Вы можете передавать практически любые параметры любой длины. Если тело функции корректно обрабатывает входные данные, то все в порядке. Это позволяет создавать очень эргономичные API, но при этом довольно сложно определить тип.

Представьте концепцию механизма запуска задач. С помощью функции `task` вы определяете новые задачи по имени и передаете либо обратный вызов, либо список других задач, которые должны быть выполнены. Или и то и другое — список задач, которые должны быть выполнены *перед* запуском обратного вызова:

```
task("default", ["scripts", "styles"]);

task("scripts", ["lint"], () => {
  // ...
});
```

```
task("styles", () => {
  // ...
});
```

Если вы думаете: «Это очень похоже на Gulp шестилетней давности», то вы правы. Его гибкий API, в котором нельзя было сделать много ошибок, был одной из причин популярности Gulp.

Определение типа в подобных функциях может стать настоящим кошмаром.

Реализовать необязательные аргументы, разные типы в одной и той же позиции сложно, даже если вы используете типы объединения¹:

```
type CallbackFn = () => void;

function task(
  name: string, param2: string[] | CallbackFn, param3?: CallbackFn
): void {
  //...
}
```

Этот код позволяет отразить все варианты предыдущего примера, но является неправильным, поскольку допускает комбинации, которые не имеют никакого смысла:

```
task(
  "what",
  () => {
    console.log("Two callbacks?");
  },
  () => {
    console.log("That's not supported, but the types say yes!");
  }
);
```

К счастью, в TypeScript есть способ решить подобные проблемы: перегрузка функций. Его название намекает на схожие концепции из других языков программирования: то же определение, но с другим поведением. Самое большое отличие TypeScript от других языков программирования состоит в том, что перегрузки функций работают только на уровне системы типов и не влияют на фактическую реализацию.

Идея состоит в том, что вы определяете каждый возможный сценарий как отдельную сигнатуру функции. Последняя сигнатура — это фактическая реализация:

```
// Типы для системы типов
function task(name: string, de+pendencies: string[]): void;
function task(name: string, callback: CallbackFn): void
```

¹ Типы объединения позволяют объединить два разных типа в один (более подробную информацию см. в главе 3).

```
function task(name: string, dependencies: string[], callback: CallbackFn): void
// Актуальная реализация
function task(
  name: string, param2: string[] | CallbackFn, param3?: CallbackFn
): void {
  //...
}
```

Здесь важно отметить несколько моментов.

TypeScript воспринимает в качестве возможных типов только объявления перед фактической реализацией. Если сигнатура фактической реализации тоже имеет значение, то продублируйте ее.

Кроме того, сигнатура фактической функции реализации не может быть какой угодно. TypeScript проверяет, можно ли реализовать перегрузку с помощью сигнатуры реализации.

Если у вас разные типы возвращаемых данных, то вы несете ответственность за то, чтобы входные и выходные данные совпадали:

```
function fn(input: number): number
function fn(input: string): string
function fn(input: number | string): number | string {
  if(typeof input === "number") {
    return "this also works";
  } else {
    return 1337;
  }
}
```

```
const typeSaysNumberButItsAString = fn(12);
const typeSaysStringButItsANumber = fn("Hello world");
```

Сигнатура реализации обычно работает с очень широким типом. Это значит, вам придется проводить много проверок, которые в любом случае потребовалось бы выполнять в JavaScript. Это хорошо, так как побуждает вас быть особенно внимательными.

Если вам нужны перегруженные функции как отдельный тип, чтобы использовать их в аннотациях и назначать несколько реализаций, то вы всегда можете создать псевдоним типа:

```
type TaskFn = {
  (name: string, dependencies: string[]): void;
  (name: string, callback: CallbackFn): void;
  (name: string, dependencies: string[], callback: CallbackFn): void;
}
```

Как видите, вам нужны только перегрузки системы типов, а не само определение реализации.

2.7. Определение типов параметра `this`

Задача

При написании функции обратного вызова предполагается использование `this`, но вы не знаете, как определить этот тип параметра при написании отдельной функции.

Решение

Определите тип параметра `this` в начале сигнатуры функции.

Обсуждение

Одной из причин путаницы для начинающих разработчиков JavaScript является постоянно меняющийся характер указателя на объект `this`:

Иногда при написании JavaScript мне хочется крикнуть: «Это нелепо!»
Но я никогда не могу понять, к чему относится *это*.

Неизвестный разработчик JavaScript

Предыдущее утверждение справедливо, особенно если у вас есть опыт работы с объектно-ориентированным языком программирования, основанным на классах, где `this` всегда относится к экземпляру класса. В JavaScript `this` — нечто совершенно другое, но необязательно более сложное для понимания. Что еще важнее, TypeScript может значительно упростить использование `this`.

Тип параметра `this` находится в области видимости функции и указывает на объект или значение, привязанные к этой функции. В обычных объектах `this` довольно прост:

```
const author = {
  name: "Stefan",
  // функция сокращена
  hi() {
    console.log(this.name);
  },
};

author.hi(); // выводит 'Stefan'
```

Но функции в JavaScript являются значениями и могут быть привязаны к другому контексту, эффективно изменяя значение `this`:

```
const author = {
  name: "Stefan",
};
```

```
function hi() {
  console.log(this.name);
}

const pet = {
  name: "Finni",
  kind: "Cat",
};

hi.apply(pet); // выводит "Finni"
hi.call(author); // выводит "Stefan"

const boundHi = hi.bind(author);

boundHi(); // выводит "Stefan"
```

Не помогает и то, что семантика `this` снова меняется, если вы используете стрелочные функции вместо обычных:

```
class Person {
  constructor(name) {
    this.name = name;
  }

  hi() {
    console.log(this.name);
  }

  hi_timeout() {
    setTimeout(function() {
      console.log(this.name);
    }, 0);
  }

  hi_timeout_arrow() {
    setTimeout(() => {
      console.log(this.name);
    }, 0);
  }
}

const person = new Person("Stefan")
person.hi(); // выводит "Stefan"
person.hi_timeout(); // выводит "undefined"
person.hi_timeout_arrow(); // выводит "Stefan"
```

TypeScript позволяет получить больше информации о том, что такое `this` и, что более важно, каким оно должно быть, с помощью типов параметров `this`.

Взгляните на следующий пример. Мы получаем доступ к элементу `button` через DOM API и привязываем к нему прослушиватель событий. В функции обратного

вызова элемент `this` имеет тип `HTMLButtonElement`; это значит, вы можете получить доступ к таким свойствам, как `classList`:

```
const button = document.querySelector("button");
button?.addEventListener("click", function() {
  this.classList.toggle("clicked");
});
```

Информацию о `this` предоставляет функция `addEventListener`. Если вы извлекаете функцию на этапе рефакторинга, то сохраните функциональность, но TypeScript выдает ошибку, так как потеряет контекст для `this`:

```
const button = document.querySelector("button");
button.addEventListener("click", handleToggle);

function handleToggle() {
  this.classList.toggle("clicked");
// ^- 'this' implicitly has type 'any' because it does not have a type annotation
}
```

Хитрость состоит в том, чтобы указать TypeScript, что `this` должен иметь определенный тип. Это можно сделать, добавив в самой первой позиции сигнатуры вашей функции параметр с именем `this`:

```
const button = document.querySelector("button");
button?.addEventListener("click", handleToggle);

function handleToggle(this: HTMLButtonElement) {
  this.classList.toggle("clicked");
}
```

Этот аргумент удаляется после компиляции. Теперь у TypeScript есть вся необходимая информация, позволяющая убедиться, что `this` должен быть типа `HTMLButtonElement`. В свою очередь, это означает, что вы получите ошибки, если используете `handleToggle` в другом контексте:

```
handleToggle();
// ^- The 'this' context of type 'void' is not
// assignable to method's 'this' of type 'HTMLButtonElement'.
```

Вы можете сделать `handleToggle` еще более полезным, если определите `this` как `HTMLElement`, суперттип `HTMLButtonElement`:

```
const button = document.querySelector("button");
button?.addEventListener("click", handleToggle);

const input = document.querySelector("input");
input?.addEventListener("click", handleToggle);

function handleToggle(this: HTMLElement) {
  this.classList.toggle("clicked");
}
```

При работе с типом параметров `this` вам может потребоваться использовать два вспомогательных типа, которые будут либо извлекать, либо удалять параметры `this` из вашего типа функции:

```
function handleToggle(this: HTMLElement) {
  this.classList.toggle("clicked");
}

type ToggleFn = typeof handleToggle;
// (this: HTMLElement) => void

type IthoutThis = OmitThisParameter<ToggleFn>
// () => void

type ToggleFnThis = ThisParameterType<ToggleFn>
// HTMLElement
```

В классах и объектах есть другие вспомогательные типы для `this`. Более подробная информация доступна в рецептах 4.8 и 11.8.

2.8. Работа с Symbol

Задача

Вы видите символьный тип — `symbol`, появляющийся в некоторых сообщениях об ошибках, но не знаете, что означает `symbol` и как его использовать.

Решение

Используйте тип `symbol` для свойств объектов, которые должны быть уникальными и неитерлируемыми. Он отлично подходит для хранения конфиденциальной информации и доступа к ней.

Обсуждение

Тип `symbol` — примитивный тип данных в JavaScript и TypeScript, который, помимо прочего, может использоваться для свойств объектов. Он обладает некоторыми уникальными свойствами, которыми отличается от типов `number` и `string`.

Символы можно создавать с помощью фабричной функции `Symbol()`:

```
const TITLE = Symbol('title')
```

Тип `symbol` не имеет функции конструктора. Параметром является необязательное описание. При вызове фабричной функции `TITLE` присваивается уникальное значение только что созданного символа. Теперь этот символ уникален, отличим от

всех других символов и не будет конфликтовать с другими символами, имеющими такое же описание:

```
const ACADEMIC_TITLE = Symbol('title')
const ARTICLE_TITLE = Symbol('title')

if(ACADEMIC_TITLE === ARTICLE_TITLE) {
  // Никогда не будет true
}
```

Описание поможет вам получить информацию о символе во время разработки:

```
console.log(ACADEMIC_TITLE.description) // title
console.log(ACADEMIC_TITLE.toString()) // Symbol(title)
```

Символы — отличный вариант, если вы хотите иметь сопоставимые значения, которые являются исключительными и уникальными. Рассмотрим пример переключателей времени выполнения или сравнения режимов:

```
// Действительно плохая система журналирования
const LEVEL_INFO = Symbol('INFO')
const LEVEL_DEBUG = Symbol('DEBUG')
const LEVEL_WARN = Symbol('WARN')
const LEVEL_ERROR = Symbol('ERROR')

function log(msg, level) {
  switch(level) {
    case LEVEL_WARN:
      console.warn(msg); break
    case LEVEL_ERROR:
      console.error(msg); break;
    case LEVEL_DEBUG:
      console.log(msg);
      debugger; break;
    case LEVEL_INFO:
      console.log(msg);
  }
}
```

Символы работают и как ключи свойств, но не являются итерируемыми, поэтому их очень удобно использовать для сериализации:

```
const print = Symbol('print')

const user = {
  name: 'Stefan',
  age: 40,
  [print]: function() {
    console.log(` ${this.name} is ${this.age} years old` )
  }
}

JSON.stringify(user) // { name: 'Stefan', age: 40 }
user[print]() // Стефану 40 лет
```

Глобальный реестр символов позволяет получить доступ к токенам во всем приложении:

```
Symbol.for('print') // creates a global symbol

const user = {
  name: 'Stefan',
  age: 37,
  // использует глобальный символ
  [Symbol.for('print')]: function() {
    console.log(`${this.name} is ${this.age} years old`)
  }
}
```

При первом вызове `Symbol.for` создается символ, который используется и при втором вызове. Если вы сохраняете значение символа в переменной и хотите узнать ключ, то можете использовать `Symbol.keyFor()`:

```
const usedSymbolKeys = []

function extendObject(obj, symbol, value) {
  // Ой, что это за символ?
  const key = Symbol.keyFor(symbol)
  // Хорошо, давайте лучше сохраним его
  if(!usedSymbolKeys.includes(key)) {
    usedSymbolKeys.push(key)
  }
  obj[symbol] = value
}

// теперь пора извлечь их все
function printAllValues(obj) {
  usedSymbolKeys.forEach(key => {
    console.log(obj[Symbol.for(key)])
  })
}
```

Отлично!

TypeScript предоставляет полную поддержку символов, и они являются основными элементами системы типов. Сам `symbol` служит аннотацией к типу данных для всех возможных символов. Посмотрите на функцию `extendObject` в предыдущем блоке кода. Чтобы все символы могли расширять объект, можно использовать тип `symbol`:

```
const sym = Symbol('foo')

function extendObject(obj: any, sym: symbol, value: any) {
  obj[sym] = value
}

extendObject({}, sym, 42) // Работает со всеми символами
```

Существует также подтип `unique symbol` (уникальный символ), который тесно связан с объявлением. Он разрешен только в объявлениях `const` и ссылается исключительно на этот символ.

Вы можете представить номинальный тип в TypeScript, который обозначает номинальное значение в JavaScript.

Чтобы узнать тип `unique symbol`, нужно использовать оператор `typeof`:

```
const PROD: unique symbol = Symbol('Production mode')
const DEV: unique symbol = Symbol('Development mode')

function showWarning(msg: string, mode: typeof DEV | typeof PROD) {
  // ...
}
```

На момент написания книги единственным возможным номинальным типом является система структурных типов в TypeScript.

Символы — нечто среднее между номинальными и непрозрачными типами в TypeScript и JavaScript. Они ближе всего к проверке номинальных типов во время выполнения.

2.9. Понимание пространств имен значений и типов

Задача

В качестве аннотаций к типам одни имена вы можете использовать, а другие — нет. Этот факт обескураживает.

Решение

Узнайте о пространствах имен типов и значений, а также о том, какие имена на что влияют.

Обсуждение

TypeScript — надмножество JavaScript. Это значит, он добавляет новые возможности к существующему и определенному языку. Со временем вы научитесь определять, какие части относятся к JavaScript, а какие — к TypeScript.

Это действительно помогает воспринимать TypeScript как дополнительный слой типов к обычному JavaScript — тонкий слой метаинформации, который будет удален перед запуском вашего кода JavaScript в одной из доступных сред выполнения. Некоторые люди даже говорят о «стирании кода TypeScript в JavaScript» после компиляции.

TypeScript, являющийся этим слоем поверх JavaScript, также означает, что различный синтаксис используется на разных уровнях. В то время как `function` или `const` создают имя в части JavaScript, объявление `type` или `interface` создает имя на уровне TypeScript:

```
// Collection – это тип в языке TypeScript
type Collection = Person[]

// printCollection – это значение в языке JavaScript
function printCollection(coll: Collection) {
  console.log(...coll.entries)
}
```

Мы также говорим, что объявления добавляют имя либо в *пространство имен типов*, либо в *пространство имен значений*. Слой типов находится поверх слоя значений, поэтому можно использовать значения в слое типов, но не наоборот.

Для этого есть явные ключевые слова:

```
// значение
const person = {
  name: "Stefan",
};

// тип
type Person = typeof person;
```

`typeof` создает имя, доступное в слое типов, из слоя значений, расположенного ниже.

Это раздражает, когда существуют типы объявлений, которые создают и типы, и значения. Например, классы могут использоваться в слое TypeScript как тип, а в JavaScript — как значение:

```
// объявление
class Person {
  name: string;

  constructor(n: string) {
    this.name = n;
  }
}

// используется как значение
const person = new Person("Stefan");

// используется как тип
type Collection = Person[];

function printPersons(coll: Collection) {
  //...
}
```

А соглашения об именовании могут ввести вас в заблуждение. Обычно мы определяем классы, типы, интерфейсы, перечисления и т. д. с заглавной буквы. И даже если они могут хранить значения, то наверняка содержат типы. Ну, до тех пор, пока вы не напишете функции, используя верхний регистр для своего приложения React, как того требует соглашение.

Если вы привыкли использовать имена в качестве типов и значений, то будете озадачены, если вдруг получите хорошо известную ошибку TS2749: *YourType* refers to a value, but is being used as a type (TS2749: *YourType* ссылается на значение, но используется как тип):

```
type PersonProps = {
  name: string;
};

function Person({ name }: PersonProps) {
  // ...
}

type PrintComponentProps = {
  collection: Person[];
  //      ^- 'Person' refers to a value,
  //      but is being used as a type
}
```

Именно здесь TypeScript может по-настоящему запутаться. Что такое тип, а что — значение, зачем их разделять и почему это не работает так, как в других языках программирования? Внезапно вы сталкиваетесь с вызовами `typeof` или даже со вспомогательным типом `InstanceType`, поскольку понимаете, что классы на самом деле делятся на два типа (см. главу 11).

Классы добавляют имя в пространство имен типов, а поскольку TypeScript — система структурных типов, то допускает значения, которые имеют ту же структуру, что и экземпляр определенного класса. Таким образом, это разрешено:

```
class Person {
  name: string;

  constructor(n: string) {
    this.name = n;
  }
}

function printPerson(person: Person) {
  console.log(person.name);
}

printPerson(new Person("Stefan")); // хорошо
printPerson({ name: "Stefan" }); // тоже хорошо
```

Однако проверки `instanceof`, которые полностью работают в пространстве имен значений и имеют значение только в пространстве имен типов, будут завершаться неудачей, так как объекты с одинаковой структурой могут иметь одинаковые свойства, но не являться фактическими экземплярами класса:

```
function checkPerson(person: Person) {
    return person instanceof Person;
}

checkPerson(new Person("Stefan")); // true
checkPerson({ name: "Stefan" }); // false
```

Поэтому полезно понимать, что работает с типами, а что — со значениями. В табл. 2.1, адаптированной из документации TypeScript, наглядно показана эта разница.

Таблица 2.1. Пространства имен типов и значений

Тип объявления	Тип	Значение
Класс	X	X
Перечисление	X	X
Интерфейс	X	
Псевдоним типа	X	
Функция		X
Переменная		X

Если в самом начале вы будете работать с функциями, интерфейсами (или псевдонимами типов, см. рецепт 2.5) и переменными, то получите представление о том, что и где можно использовать. Если же будете работать с классами, то подумайте о последствиях чуть дольше.

Система типов

https://t.me/it_books/2

В предыдущей главе вы познакомились с основными элементами, которые позволяют сделать ваш код на JavaScript более выразительным. Но если у вас есть опыт работы с JavaScript, то вы понимаете, что базовые типы и аннотации TypeScript — лишь малая часть присущей ему гибкости.

TypeScript призван сделать намерения JavaScript более понятными, и ему необходимо сделать это, не жертвуя гибкостью, тем более что она позволяет разработчикам создавать великолепные API, которые используют и любят миллионы. Считайте, что TypeScript — это скорее способ формализовать JavaScript, а не ограничить его. Введите систему типов TypeScript.

В этой главе вы разработаете ментальную модель восприятия типов. Вы узнаете, как определять наборы значений широко или узко в зависимости от ваших потребностей и как изменять их область действия на протяжении всего потока управления. Кроме того, мы поговорим о том, как использовать структурную систему типов и когда следует отступить от правил.

В этой главе проводится граница между основами TypeScript и расширенными приемами работы с типами. Но независимо от того, кем вы являетесь: опытным разработчиком TypeScript или новичком, — эта ментальная модель послужит фундаментом для всего дальнейшего.

3.1. Моделирование данных с помощью типов объединений и пересечений

Задача

У вас есть сложная модель данных, которую вы хотите описать с помощью TypeScript.

Решение

Используйте типы объединения и пересечения для моделирования данных, а литеральные типы — для определения конкретных вариантов.

Обсуждение

Предположим, вы создаете модель данных для магазина игрушек. Каждый товар в нем имеет несколько основных свойств: название, количество и рекомендуемый минимальный возраст. Дополнительные свойства применимы только к каждому конкретному типу игрушек, для чего необходимо создать несколько вариантов:

```
type BoardGame = {
  name: string;
  price: number;
  quantity: number;
  minimumAge: number;
  players: number;
};
```

```
type Puzzle = {
  name: string;
  price: number;
  quantity: number;
  minimumAge: number;
  pieces: number;
};
```

```
type Doll = {
  name: string;
  price: number;
  quantity: number;
  minimumAge: number;
  material: string;
};
```

Для создаваемых функций вам нужен тип, который является репрезентативным для всех игрушек, — супертип, содержащий только основные свойства, общие для всех игрушек:

```
type ToyBase = {
  name: string;
  price: number;
  quantity: number;
  minimumAge: number;
};

function printToy(toy: ToyBase) {
  /* ... */
}

const doll: Doll = {
  name: "Mickey Mouse",
```



```
    price: 9.99,  
    quantity: 10000,  
    minimumAge: 2,  
    material: "plush",  
  };  
  
printToy(doll); // работает
```

Код работает, так как с помощью этой функции можно распечатать информацию обо всех куклах, настольных играх или пазлах, но учтите один момент: вы потеряете информацию об исходной игрушке в `printToy`. Вы можете вводить только общие свойства, но не конкретные.

Для типа, представляющего все возможные игрушки, можно создать *типы объединения*:

```
// Объединение Toy  
type Toy = Doll | BoardGame | Puzzle;  
  
function printToy(toy: Toy) {  
  /* ... */  
}
```

Лучше всего рассматривать тип как набор совместимых значений. TypeScript проверяет, совместимо ли значение, аннотированное или нет, с определенным типом. В случае объектов это касается значений с большим количеством свойств, чем определено в их типе. В результате выведения (inference) значениям с большим количеством свойств присваивается подтип в системе структурных типов. А значения подтипов входят в набор супертипов.

Типы объединения — это объединение множеств. Количество совместимых значений расширяется, а также происходит некоторое перекрытие типов. Например, объект, у которого есть и `material`, и `players`, может быть совместим и с `Doll`, и с `BoardGame`. На эту особенность следует обратить внимание, а метод работы с этой особенностью вы сможете увидеть в рецепте 3.2.

На рис. 3.1 показана концепция типов объединения в форме диаграммы Венна. Аналогии теории множеств хорошо работают и здесь.

Вы можете создавать типы объединения везде, в том числе и с помощью примитивных типов:

```
function takesNumberOrString(value: number | string) {  
  /* ... */  
}  
  
takesNumberOrString(2); // хорошо  
takesNumberOrString("Hello"); // хорошо
```

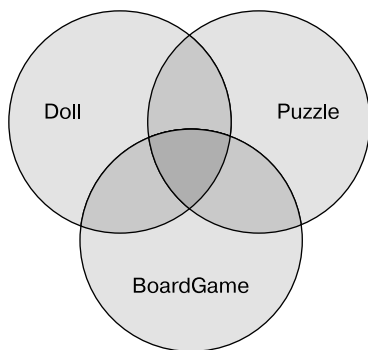


Рис. 3.1. Визуализация типов объединения. Каждый тип представляет набор совместимых значений, а тип объединения — объединение наборов

Это позволяет вам расширить набор значений так сильно, насколько вы захотите.

В примере с магазином игрушек вы также видите некоторую избыточность: свойства `ToyBase` повторяются. Было бы гораздо лучше, если бы вы могли использовать `ToyBase` в качестве основы каждой части объединения. И это можно сделать с помощью пересечения типов:

```
type ToyBase = {
  name: string;
  price: number;
  quantity: number;
  minimumAge: number;
};

// Пересечение ToyBase и { players: number }
type BoardGame = ToyBase & {
  players: number;
};

// Пересечение ToyBase и { pieces: number }
type Puzzle = ToyBase & {
  pieces: number;
};

// Пересечение ToyBase и { material: string }
type Doll = ToyBase & {
  material: string;
};
```

Как и типы объединения, *типы пересечения* напоминают свои аналоги из теории множеств. Они сообщают TypeScript, что совместимые значения должны быть типа А и типа В. Теперь тип принимает более узкий набор значений, который со-

держит все свойства обоих типов, в том числе их подтипы. На рис. 3.2 показана визуализация пересечения типов.

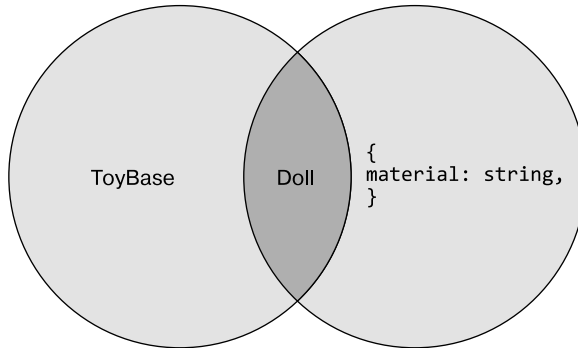


Рис. 3.2. Визуализация двух типов пересечения. Набор возможных значений сужается

Типы пересечения работают и с примитивными типами, но они бесполезны. Пересечение `string` и `number` приводит к результату `never`, так как ни одно значение одновременно не удовлетворяет свойствам `string` и `number`.



Вместо того чтобы использовать псевдонимы и пересечения типов, вы можете определять свои модели с помощью интерфейсов. В рецепте 2.5 рассказывается об их различиях, и среди них есть несколько, на которые вам следует обратить внимание. Таким образом, тип `BoardGame = ToyBase & { /* ... */ }` можно легко описать как `interface BoardGame extends ToyBase { /* ... */ }`. Однако вы не можете определить интерфейс, который является типом объединения. Тем не менее вы можете определить объединение интерфейсов.

Эти способы моделирования данных в TypeScript уже хороши, но можно пойти еще дальше. В TypeScript литеральные значения могут быть представлены в виде литерального типа. Мы можем определить тип, который будет просто, например, числом 1, и единственным совместимым значением окажется 1:

```
type One = 1;
const one: One = 1; // ничего другого присвоить нельзя
```

Такой тип называется *литеральным*. Сам по себе он кажется не очень полезным, но он хорош, когда вы объединяете несколько литеральных типов. Например, для типа `Doll` мы можем явно установить допустимые значения для `material`:

```
type Doll = ToyBase & {
  material: "plush" | "plastic";
};
```

```
function checkDoll(doll: Doll) {
  if (doll.material === "plush") {
    // сделать что-нибудь с plush
  } else {
    // doll.material – это "plastic", других вариантов нет
  }
}
```

Такой подход делает невозможным присвоение любого значения, кроме "plush" или "plastic", и значительно повышает надежность кода.

С помощью объединения, пересечения типов и литеральных типов намного проще определять даже сложные модели.

3.2. Явное определение моделей с размеченными типами объединения

Задача

Части моделируемого вами типа объединения сильно перекрываются по своим свойствам, поэтому различать их в потоке управления становится затруднительно.

Решение

Добавьте свойство `kind` к каждой части объединения с типом строкового литерала и проверяйте его содержимое.

Обсуждение

Рассмотрим модель данных, аналогичную той, которую мы создали в рецепте 3.1. На этот раз мы хотим определить различные формы для графического программного обеспечения:

```
type Circle = {
  radius: number;
};

type Square = {
  x: number;
};

type Triangle = {
  x: number;
```

```
    y: number;
  };

type Shape = Circle | Triangle | Square;
```

Типы чем-то похожи, но при этом информации, позволяющей различать их в функции `area`, достаточно:

```
function area(shape: Shape) {
  if ("radius" in shape) {
    // Круг
    return Math.PI * shape.radius * shape.radius;
  } else if ("y" in shape) {
    // Треугольник
    return (shape.x * shape.y) / 2;
  } else {
    // Квадрат
    return shape.x * shape.x;
  }
}
```

Это работает, но с некоторыми оговорками. Тип `Circle` единственный имеет свойство `radius`, а у `Triangle` и `Square` есть общее свойство `x`. Поскольку `Square` состоит только из свойства `x`, это делает `Triangle` подтипом `Square`.

Учитывая, что мы определили поток управления так, чтобы сначала проверялось отличительное свойство подтипа `y`, это не проблема. Но слишком легко проверить только `x` и создать ветвь в потоке управления, которая вычислит площадь и для `Triangle`, и для `Square` одинаковым способом, что неправильно.

Расширить `Shape` также непросто. Если мы посмотрим на необходимые свойства прямоугольника, то увидим, что он содержит те же свойства, что и `Triangle`:

```
type Rectangle = {
  x: number;
  y: number;
};

type Shape = Circle | Triangle | Square | Rectangle;
```

Не существует четкого способа различать каждую часть объединения. Чтобы убедиться в том, что каждая часть объединения различима, нужно расширить модели с помощью идентифицирующего свойства, которое абсолютно ясно показывает, с чем мы имеем дело.

Это можно сделать путем добавления свойства `kind`. Оно принимает тип строкового литерала, идентифицирующий часть модели.

Как было показано в рецепте 3.1, TypeScript позволяет преобразовать примитивные типы, такие как `string`, `number`, `bigint` и `boolean`, в конкретные значения. Это означает, что каждое значение также является типом, набором, состоящим ровно из одного совместимого значения.

Итак, чтобы наша модель была четко определена, мы добавляем свойство `kind` к каждой части модели и присваиваем ему точный литеральный тип, идентифицирующий эту часть:

```
type Circle = {
  radius: number;
  kind: "circle";
};

type Square = {
  x: number;
  kind: "square";
};

type Triangle = {
  x: number;
  y: number;
  kind: "triangle";
};

type Shape = Circle | Triangle | Square;
```

Обратите внимание, что мы задаем значение `kind` не как `string`, а как *точный* литеральный тип `"circle"` (или `"square"` и `"triangle"` соответственно). Это тип, а не значение, но единственным совместимым значением является литеральная строка.

Добавление свойства `kind` к строковым литеральным типам гарантирует, что части объединения не будут перекрываться, поскольку литеральные типы несовместимы друг с другом. Этот прием называется *размеченными типами объединения* и эффективно отделяет каждое множество, входящее в тип объединения `Shape`, указывая на точное множество.

Данный прием отлично подходит для функции `area`, так как мы можем эффективно различать типы, например, в операторе `switch`:

```
function area(shape: Shape) {
  switch (shape.kind) {
    case "circle": // Круг
      return Math.PI * shape.radius * shape.radius;
    case "triangle": // Треугольник
      return (shape.x * shape.y) / 2;
  }
}
```

```
    case "square": // Квадрат
      return shape.x * shape.x;
    default:
      throw Error("not possible");
  }
}
```

Становится абсолютно ясно, с чем мы имеем дело, и при этом образуется очень хорошая защита от предстоящих изменений, как мы увидим в рецепте 3.3.

3.3. Проверка полноты с помощью приема `assertNever`

Задача

Используемые вами размеченные типы объединений со временем меняются путем добавления к объединению новых частей. Становится трудно отслеживать все случаи, когда код необходимо адаптировать к этим изменениям.

Решение

Создайте проверки полноты, в которых вы с помощью функции `assertNever` утверждаете, что все остальные случаи никогда не могут произойти.

Обсуждение

Рассмотрим полный пример из рецепта 3.2:

```
type Circle = {
  radius: number;
  kind: "circle";
};

type Square = {
  x: number;
  kind: "square";
};

type Triangle = {
  x: number;
  y: number;
  kind: "triangle";
};
```

```

type Shape = Circle | Triangle | Square;

function area(shape: Shape) {
  switch (shape.kind) {
    case "circle": // Круг
      return Math.PI * shape.radius * shape.radius;
    case "triangle": // Треугольник
      return (shape.x * shape.y) / 2;
    case "square": // Квадрат
      return shape.x * shape.x;
    default:
      throw Error("not possible");
  }
}

```

Благодаря размеченным типам объединения мы можем различать каждую часть объединения. Функция `area` использует оператор `switch-case`, чтобы обрабатывать каждый случай отдельно. Благодаря строковым литеральным типам свойство `kind` не может иметь дублирования типов.

Когда все варианты исчерпаны, в случае по умолчанию код выдает ошибку, указывающую на то, что мы достигли недопустимой ситуации, которая никогда не должна возникать. Если наши типы верны во всей кодовой базе, то данная ошибка никогда не появится.

Даже система типов говорит нам, что случай по умолчанию — это невозможная ситуация. Если мы добавим `shape` в стандартный случай и наведем на нее курсор, то TypeScript скажет нам, что `shape` имеет тип `never`:

```

function area(shape: Shape) {
  switch (shape.kind) {
    case "circle": // Круг
      return Math.PI * shape.radius * shape.radius;
    case "triangle": // Треугольник
      return (shape.x * shape.y) / 2;
    case "square": // Квадрат
      return shape.x * shape.x;
    default:
      console.error("Shape not defined:", shape); // shape имеет тип never
      throw Error("not possible");
  }
}

```

Тип `never` вызывает интерес. Это *нижний тип* TypeScript, то есть он находится в самом низу иерархии типов. Если `any` и `unknown` содержат все возможные значения, то ни одно значение несовместимо с `never`. Это пустое множество, что объясняет его название. Если одно из ваших значений имеет тип `never`, то вы окажетесь в ситуации, которая *никогда не* должна произойти.

Тип `shape` в случаях по умолчанию немедленно меняется, если мы расширяем тип `shape`, например, до `Rectangle`:

```
type Rectangle = {
  x: number;
  y: number;
  kind: "rectangle";
};

type Shape = Circle | Triangle | Square | Rectangle;

function area(shape: Shape) {
  switch (shape.kind) {
    case "circle": // Круг
      return Math.PI * shape.radius * shape.radius;
    case "triangle": // Треугольник
      return (shape.x * shape.y) / 2;
    case "square": // Квадрат
      return shape.x * shape.x;
    default:
      console.error("Shape not defined:", shape); // Прямоугольник
      throw Error("not possible");
  }
}
```

Вот анализ потока управления в успешных случаях: TypeScript в каждый момент времени точно знает, какие типы у ваших значений. В ветке по умолчанию `shape` имеет тип `Rectangle`, но предполагается, что мы будем работать с прямоугольниками. Разве не было бы хорошо, если бы TypeScript мог сообщить, что мы упустили из виду потенциальный тип? С этим изменением мы теперь сталкиваемся каждый раз, когда вычисляем форму прямоугольника. Случай по умолчанию был предназначен для обработки (с точки зрения системы типов) невозможных ситуаций. Мы хотели бы сохранить его в таком виде.

Это уже плохо в одной ситуации и становится еще хуже, если вы используете паттерн проверки полноты несколько раз в своей кодовой базе. Вы не можете с уверенностью сказать, что не пропустили ни одного места, в котором ваша программа в итоге даст сбой.

Один из способов убедиться, что вы обработали все возможные случаи, — создать вспомогательную функцию, которая будет утверждать, что все варианты исчерпаны. Она должна гарантировать, что единственным возможным значением является отсутствие значений:

```
function assertNever(value: never) {
  console.error("Unknown value", value);
  throw Error("Not possible");
}
```

Обычно `never` говорит о том, что вы находитесь в безвыходной ситуации. В нашем же случае мы используем его как явную аннотацию типа для сигнатуры функции. Вы можете спросить: какие значения мы должны передавать? Ответ таков: никаких! В лучшем случае эта функция никогда не будет вызвана.

Но если мы заменим исходный вариант по умолчанию из нашего примера на `assertNever`, то сможем использовать систему типов, чтобы гарантировать совместимость всех возможных значений, даже если значений нет:

```
function area(shape: Shape) {
  switch (shape.kind) {
    case "circle": // Круга
      return Math.PI * shape.radius * shape.radius;
    case "triangle": // Треугольник
      return (shape.x * shape.y) / 2;
    case "square": // Квадрат
      return shape.x * shape.x;
    default: // Прямоугольник
      assertNever(shape);
  }
  // ^-- Error: Argument of type 'Rectangle' is not
  // assignable to parameter of type 'never'
}
```

Отлично! Теперь мы получаем красные волнистые линии всякий раз, когда забываем исчерпать все параметры. TypeScript не скомпилирует этот код без ошибки, и теперь легко обнаружить в кодовой базе все вхождения, когда нужно добавить случай `Rectangle`:

```
function area(shape: Shape) {
  switch (shape.kind) {
    case "circle": // Круг
      return Math.PI * shape.radius * shape.radius;
    case "triangle": // Треугольник
      return (shape.x * shape.y) / 2;
    case "square": // Квадрат
      return shape.x * shape.x;
    case "rectangle":
      return shape.x * shape.y;
    default: // shape имеет тип never
      assertNever(shape); // shape может быть передана в assertNever!
  }
}
```

Тип `never` не имеет совместимых значений и используется для обозначения невозможной ситуации в системе типов, но мы можем использовать `type` как аннотацию типа, чтобы не забывать о *возможных* ситуациях. Представление о типах как о наборах совместимых значений, которые могут расширяться или сужаться в зависимости от потока управления, приводит нас к таким приемам, как `assertNever` — очень полезной маленькой функции, которая может повысить качество кодовой базы.

3.4. Закрепление типов с помощью Const Context

Задача

Вы не можете присваивать объектные литералы своим тщательно смоделированным размеченным типам объединения.

Решение

Определите тип ваших литералов, используя утверждения типа и *контекст const*.

Обсуждение

В TypeScript каждое значение можно использовать как отдельный тип. Такие типы называются литеральными и позволяют вам разбивать большие множества на несколько допустимых значений.

Использование литеральных типов в TypeScript — не только хороший способ указать на конкретные значения, но и важная часть работы системы типов. Это становится очевидным, когда вы присваиваете значения примитивных типов различным привязкам, используя `let` или `const`.

Если вы присваиваете одно и то же значение дважды: один раз с помощью `let` и один раз с помощью `const`, то TypeScript выводит два разных типа. При привязке `let` TypeScript выведет более широкий примитивный тип:

```
let name = "Stefan"; // Имя — строка
```

При привязке `const` TypeScript выведет точный тип литерала:

```
const name = "Stefan"; // Имя — "Stefan"
```

Типы объектов ведут себя немного по-другому. Привязки `let` по-прежнему выведут более широкий набор:

```
// person is { name: string }  
let person = { name: "Stefan" };
```

Но то же самое делают и привязки `const`:

```
// person is { name: string }  
let person = { name: "Stefan" };
```

Причина этого кроется в JavaScript, хотя сама привязка постоянна (*constant*); значит, я не могу переопределить `person`, а значения свойств объекта могут меняться:

```
// person is { name: string }  
const person = { name: "Stefan" };  
  
person.name = "Not Stefan"; // работает!
```

Такое поведение правильно в том смысле, что отражает поведение JavaScript, но может вызвать проблемы, когда вы очень точно определяете модели данных.

В предыдущих рецептах мы моделировали данные с помощью типов объединения и пересечения. Мы использовали *размеченные типы объединения*, чтобы различать слишком похожие типы.

Проблема в том, что, когда мы используем литералы для данных, TypeScript обычно выводит более широкий набор, что делает значения несовместимыми с определенными типами. Это приводит к появлению очень длинного сообщения об ошибке:

```
type Circle = {
  radius: number;
  kind: "circle";
};

type Square = {
  x: number;
  kind: "square";
};

type Triangle = {
  x: number;
  y: number;
  kind: "triangle";
};

type Shape = Circle | Triangle | Square;

function area(shape: Shape) {
  /* ... */
}

const circle = {
  radius: 2,
  kind: "circle",
};

area(circle);
// ^-- Argument of type '{ radius: number; kind: string; }'
// is not assignable to parameter of type 'Shape'.
// Type '{ radius: number; kind: string; }' is not
// assignable to type 'Circle'.
// Types of property 'kind' are incompatible.
// Type 'string' is not assignable to type '"circle"'.
```

Есть несколько способов решить эту проблему. Так, можно использовать явные аннотации для проверки типа. Как было описано в рецепте 2.1, каждая аннотация

является проверкой типа, то есть значение с правой стороны проверяется на совместимость. Здесь нет выведения типа, поэтому TypeScript будет рассматривать точные значения, чтобы решить, совместим ли объектный литерал:

```
// Exact type
const circle: Circle = {
  radius: 2,
  kind: "circle",
};
```

```
area(circle); // Работает!
```

```
// Broader set
const circle: Shape = {
  radius: 2,
  kind: "circle",
};
```

```
area(circle); // Тоже работает!
```

Вместо того чтобы использовать аннотации типов, можно выполнять утверждения типов в конце присваивания:

```
// Type assertion
const circle = {
  radius: 2,
  kind: "circle",
} as Circle;
```

```
area(circle); // Работает!
```

Однако иногда аннотации могут ограничивать ваши возможности. Это особенно актуально, когда вам приходится работать с литералами, которые содержат больше информации и используются в разных местах с различной семантикой.

С того момента, как вы аннотируете или выполняете утверждение типа `Circle`, привязка всегда будет `circle`, независимо от того, какие значения на самом деле оно содержит.

Но утверждения типа позволяют вам быть более точными. Вместо того чтобы утверждать, что весь объект имеет определенный тип, вы можете указать, что отдельные свойства относятся к определенному типу:

```
const circle = {
  radius: 2,
  kind: "circle" as "circle",
};
```

```
area(circle); // Работает!
```

Другой способ утверждать точные значения — использовать *контекст const* с утверждением типа `as const`. TypeScript фиксирует значение как литеральный тип:

```
const circle = {  
  радиус: 2,  
  kind: "circle" as const,  
};  
  
area(circle); // Работает!
```

Кроме того, если вы применяете контекст `const` ко всему объекту, то гарантируете, что значения доступны только для чтения и не будут изменены:

```
const circle = {  
  radius: 2,  
  kind: "circle",  
} as const;  
  
area2(circle); // Работает!  
  
circle.kind = "rectangle";  
//    ^-- Cannot assign to 'kind' because  
//       it is a read-only property.
```

Утверждения типа контекст `const` — очень удобный инструмент, если вы хотите привязать значения к их точному литеральному типу и сохранить их такими. Если в вашей кодовой базе много объектных литералов, которые не должны изменяться, но их необходимо использовать в различных случаях, то контекст `const` может помочь!

3.5. Сужение типов с помощью предикатов типа

Задача

На основании определенных условий можно утверждать, что значение имеет более узкий тип, чем первоначально назначенный, но TypeScript не может сузить его за вас.

Решение

Добавьте предикаты типов в сигнатуру вспомогательной функции, чтобы указать действие логического условия на систему типов.

Обсуждение

С помощью литеральных типов и типов объединений TypeScript позволяет вам определять очень специфические наборы значений. Например, вы можете легко определить игральную кость с шестью гранями:

```
type Dice = 1 | 2 | 3 | 4 | 5 | 6;
```

Эта нотация является выразительной, и система типов может точно указать вам, какие значения будут допустимыми, но получить данный тип можно, только проделав определенную работу.

Представим, что у нас есть некая игра, в которой пользователям разрешено вводить любое число. Если оно обозначает допустимое количество точек, то мы выполняем определенные действия.

Мы записываем проверку условий, чтобы узнать, является ли вводимое число частью набора значений:

```
function rollDice(input: number) {  
  if ([1, 2, 3, 4, 5, 6].includes(input)) {  
    // `input` все еще `number`, хотя мы это знаем  
    // должно быть Dice  
  }  
}
```

Проблема состоит в том, что даже если мы проверяем, известен ли набор значений, TypeScript все равно обрабатывает `input` как `number`. Система типов никак не может установить связь между проверкой и изменением в системе типов.

Но системе типов можно помочь. Для начала извлеките свою проверку в отдельную вспомогательную функцию:

```
function isDice(value: number): boolean {  
  return [1, 2, 3, 4, 5, 6].includes(value);  
}
```

Обратите внимание, что эта проверка возвращает значение `boolean`. Это условие либо истинно, либо ложно. Для функций, возвращающих логическое значение, мы можем изменить возвращаемый тип в сигнатуре функции на предикат типа.

Мы сообщаем TypeScript, что если эта функция возвращает `true`, то мы знаем больше о значении, которое было передано в функцию. В нашем случае `value` имеет тип `Dice`:

```
function isDice(value: number): value is Dice {  
  return [1, 2, 3, 4, 5, 6].includes(value);  
}
```

Благодаря этому TypeScript получает подсказку о том, каковы фактические типы наших значений, что позволяет выполнять с ними более детальные операции:

```
function rollDice(input: number) {
  if (isDice(input)) {
    // Отлично! `input` теперь `Dice`
  } else {
    // input все еще `number`
  }
}
```

TypeScript имеет ограничения и не допускает никаких утверждений с предикатами типов. Это должен быть тип более узкий, чем исходный. Например, если получить на вход `string` и утверждать, что на выходе будет подмножество `number`, то возникнет ошибка:

```
type Dice = 1 | 2 | 3 | 4 | 5 | 6;
```

```
function isDice(value: string): value is Dice {
  // Ошибка: тип предиката типа должен быть назначен
  // его типу параметра. Тип 'number' не может быть назначен типу 'string'.
  return ["1", "2", "3", "4", "5", "6"].includes(value);
}
```

Этот отказоустойчивый механизм дает некоторую гарантию на уровне типов, но есть оговорка: он не будет проверять, имеют ли ваши условия смысл. Первоначальная проверка в `isDice` гарантирует, что переданное значение входит в массив допустимых чисел.

Значения в этом массиве вы выбираете сами. Если укажете неправильное число, то TypeScript все равно будет считать, что `value` — это правильный `Dice`, даже если результат вашей проверки не совпадет:

```
// Правильный на уровне типа
// неправильный набор значений на уровне значений
function isDice(value: number): value is Dice {
  return [1, 2, 3, 4, 5, 7].includes(value);
}
```

Здесь возможна ошибка. Условие в примере 3.1 верно для целых чисел, но неверно, если вы передаете число с плавающей запятой. Например, `3.1415` будет допустимым количеством точек `Dice`!

Пример 3.1. Неверная логика для `isDice` для чисел с плавающей запятой

```
// Правильно на уровне типа, неверная логика
function isDice(value: number): value is Dice {
  return value >= 1 && value <= 6;
}
```

На самом деле для TypeScript работает любое условие. Верните значение `true`, и TypeScript будет считать, что `value` — это `Dice`:


```
function isDice(value: number): value is Dice {  
  return true;  
}
```

TypeScript дает возможность утверждать типы. Ваша обязанность — убедиться, что эти утверждения верны и обоснованы. Если вы активно используете утверждения типов с помощью предикатов типов, то убедитесь, что соответствующим образом провели их тестирование.

3.6. Понимание void

Задача

Вы знакомы с понятием типа `void` из других языков программирования, но в TypeScript он может вести себя немного по-другому.

Решение

Используйте `void` в качестве заменяемого типа для обратных вызовов.

Обсуждение

Возможно, вы знакомы с понятием типа `void` из таких языков программирования, как Java или C#, где он указывает на отсутствие возвращаемого значения. Данный тип существует и в TypeScript и на первый взгляд делает то же самое: если ваши функции или методы ничего не возвращают, то тип возвращаемого значения — `void`.

Но если присмотреться к поведению `void` более пристально, то оно немного сложнее, как и его положение в системе типов. В TypeScript `void` — подтип `undefined`. Функции в JavaScript всегда что-то возвращают. Функция либо явно возвращает значение, либо неявно возвращает `undefined`:

```
function iHaveNoReturnValue(i) {  
  console.log(i);  
}
```

```
let check = iHaveNoReturnValue(2);  
// check — типа undefined
```

Если бы мы создали тип для `iHaveNoReturnValue`, то он отображал бы тип функции с `void` в качестве возвращаемого типа:

```
function iHaveNoReturnValue(i) {  
  console.log(i);  
}  
  
type Fn = typeof iHaveNoReturnValue;  
// type Fn = (i: any) => void
```

VOID В JAVASCRIPT

Тип `void` существует как оператор в JavaScript и демонстрирует совершенно особое поведение. Он оценивает выражение, стоящее за ним, но гарантирует возврат `undefined`:

```
let i = void 2; // i === undefined
```

Каковы сценарии использования `void`?

Во-первых, в ECMAScript 3 вы могли бы переопределить `undefined` и дать ему фактическое значение. Тип `void` всегда возвращал настоящее `undefined`.

Во-вторых, это хороший способ запускать сразу вызываемые функции:

```
// выполняется сразу
void function() {
  console.log('Hey');
}();
```

И все это без засорения глобального пространства имен:

```
void function aRecursion(i) {
  if(i > 0) {
    console.log(i--);
    aRecursion(i);
  }
}(3);
```

```
console.log(typeof aRecursion); // не определено
```

Тип `void` всегда возвращает `undefined` и оценивает выражение, следующее за ним, поэтому у вас есть очень короткий способ выполнить возврат из функции, обойдясь без возврата значения, но при этом вызывая функцию обратного вызова, например:

```
// Возвращение чего-то другого, кроме undefined, приведет к сбою приложения
function middleware(nextCallback) {
  if(conditionApplies()) {
    return void nextCallback();
  }
}
```

Это подводит меня к самому важному сценарию использования `void`: это шлюз безопасности для вашего приложения. Если ваша функция всегда должна возвращать `undefined`, вы можете убедиться, что это всегда так:

```
button.onclick = () => void doSomething();
```

Кроме того, `void` в качестве типа может использоваться для параметров и всех других объявлений. Единственное значение, которое может быть передано, — это `undefined`:

```
function iTakeNoParameters(x: void): void { }

iTakeNoParameters(); // работает
iTakeNoParameters(undefined); // работает
iTakeNoParameters(void 2); // работает
```

`void` и `undefined` — практически одно и то же. Но есть одно существенное различие: `void` как тип возвращаемого значения может быть заменен на другие типы, что позволяет использовать расширенные паттерны обратного вызова. Создадим, например, функцию `fetch`. Ее задача — получить набор чисел и передать результаты в функцию обратного вызова, указанную в качестве параметра:

```
function fetchResults(
  callback: (statusCode: number, results: number[]) => void
) {
  // получить результаты откуда-то...
  callback(200, results);
}
```

Функция обратного вызова имеет два параметра в своей сигнатуре: код состояния и результаты, а тип возвращаемого значения — `void`. Мы можем вызвать `fetchResults` с помощью функций обратного вызова, которые точно соответствуют типу `callback`:

```
function normalHandler(statusCode: number, results: number[]): void {
  // сделать что-нибудь с обоими параметрами
}

fetchResults(normalHandler);
```

Но если в типе функции указан возвращаемый тип `void`, то принимаются функции и с другим, более специфичным типом возвращаемого значения:

```
function handler(statusCode: number): boolean {
  // определить код состояния ...
  return true;
}

fetchResults(handler); // компилируется, нет проблем!
```

Сигнатуры функций не совпадают в точности, но код все равно компилируется. Во-первых, в сигнатурах функций можно указывать более короткий список аргументов. JavaScript может вызывать функции с избыточными параметрами, и если они не указаны в функции, то просто игнорируются. Нет необходимости указывать больше параметров, чем вам действительно нужно.

Во-вторых, тип возвращаемого значения — `boolean`, но TypeScript все равно передаст эту функцию. Это полезно при объявлении возвращаемого типа `void`. Исходная вызывающая функция `fetchResults` не ожидает возвращаемого значения при вызове функции обратного вызова. Таким образом, для системы типов возвращаемое значение `callback` все еще `undefined`, хотя может быть чем-то другим.

Пока система типов не позволяет работать с возвращаемым значением, ваш код должен быть в безопасности:

```
function fetchResults(  
  callback: (statusCode: number, results: number[]) => void  
) {  
  // получить результаты откуда-то...  
  const didItWork = callback(200, results);  
  // didItWork имеет значение `undefined` в системе типов,  
  // даже если должно быть логическим с `handler`.  
}
```

Именно поэтому можно передавать обратные вызовы с любым типом возвращаемого значения. Даже если обратный вызов что-то возвращает, это значение не будет использовано и переходит в `void`. Все решает вызывающая функция, которая лучше знает, чего ожидать от функции обратного вызова. А если вызывающая функция вообще не требует от обратного вызова возвратного значения, то можно делать что угодно!

TypeScript называет это свойство *заменяемостью*: возможность заменить одно другим там, где это имеет смысл. На первый взгляд это может показаться странным. Но когда вы работаете с библиотеками, авторами которых не являетесь, эта возможность окажется для вас очень ценной.

3.7. Работа с типами ошибок в предложениях `catch`

Задача

Вы не можете аннотировать явные типы ошибок в блоках `try-catch`.

Решение

Добавляйте аннотации с `any` или `unknown` и используйте предикаты типа (см. рецепт 3.5, чтобы сузить круг ошибок до конкретных типов).

Обсуждение

Если вы ранее работали в таких языках, как Java, C++ или C#, то привыкли выполнять обработку ошибок, генерируя исключения и впоследствии перехватывая

их в каскаде предложений `catch`. Возможно, существуют более эффективные способы обработки ошибок, но этот способ применяется уже много лет и, учитывая историю и влияние, нашел свой путь в JavaScript¹.

«Выбрасывать» ошибки (`throw`) и «перехватывать» их (`catch`) — допустимый способ обработки ошибок в JavaScript и TypeScript, но существует большая разница, когда дело доходит до указания предложений `catch`. Когда вы пытаетесь перехватить определенный тип ошибки, TypeScript выдаст ошибку.

В примере 3.2 в целях демонстрации проблемы используется Axios — популярная библиотека для сбора данных (<https://axios-http.com/>).

Пример 3.2. Перехват явных типов ошибок не работает

```
try {
  // например, с популярной библиотекой для сбора информации Axios
} catch(e: AxiosError) {
  //      ^^^^^^^^^^^ Error 1196: Catch clause variable
  //                type annotation must be 'any' or
  //                'unknown' if specified.
}
```

Это происходит по нескольким причинам.

Тип `any` может быть выброшен

В JavaScript разрешено выбрасывать любые выражения. Конечно, вы можете выбросить «исключения» (или ошибки, как мы называем их в JavaScript), но допускается использовать и любые другие значения:

```
throw "What a weird error"; // OK
throw 404; // OK
throw new Error("What a weird error"); // OK
```

Выбросить можно любое допустимое значение, поэтому возможные значения для перехвата уже шире, чем обычный подтип `Error`.

В JavaScript есть только одно предложение `catch`

В JavaScript есть только одно предложение `catch` для каждого оператора `try`. В прошлом были мысли использовать несколько предложений `catch` (<https://oreil.ly/NMn8O>) и даже условных выражений, но из-за отсутствия интереса к JavaScript в начале 2000-х эти идеи так и не были реализованы.

Вместо этого следует использовать одно предложение `catch` и выполнять проверки `instanceof` и `typeof`, как предлагается в MDN (<https://oreil.ly/ipzoR>).

¹ Например, язык программирования Rust получил высокую оценку за обработку ошибок.

Пример ниже — единственно правильный способ сужения типов для предложений `catch` в TypeScript:

```
try {
  myroutine(); // Здесь выдается несколько ошибок
} catch (e) {
  if (e instanceof TypeError) {
    // Ошибка TypeError
  } else if (e instanceof RangeError) {
    // Обработка ошибки RangeError
  } else if (e instanceof EvalError) {
    // Вы угадали: EvalError
  } else if (typeof e === "string") {
    // Ошибка — это строка
  } else if (axios.isAxiosError(e)) {
    // axios проверяет ошибки за нас!
  } else {
    // все остальное
    logMyErrors(e);
  }
}
```

Выброшены могут быть все возможные значения, а для их обработки есть только одно предложение `catch` в операторе `try`, поэтому диапазон типов `e` исключительно широк.

Любое исключение может произойти

Вы знаете о каждой возможной ошибке, поэтому не будет ли так же хорошо работать правильный тип объединения со всеми возможными «выбрасываемыми» ошибками? Теоретически — да. На практике же нет возможности определить, какие типы будут иметь исключение.

Помимо всех ваших пользовательских исключений и ошибок, система может выдать ошибку, когда что-то не так с памятью, обнаруживается несоответствие типов или одна из ваших функций не была определена. Простой вызов функции может вызвать печально известное переполнение стека.

Широкий набор возможных значений, единственное предложение `catch` и неопределенность возникающих ошибок допускают только два типа для `e`: `any` и `unknown`.

Все причины применимы, если вы отклоняете `Promise`. TypeScript позволяет вам указать лишь тип выполненного `Promise`. Отклонение может произойти по вашей вине или из-за системной ошибки:

```
const somePromise = () =>
  new Promise((fulfil, reject) => {
    if (someConditionIsValid()) {
      fulfil(42);
    } else {
```

```
    reject("Oh no!");
  }
});

somePromise()
.then((val) => console.log(val)) // val – число
.catch((e) => console.log(e)); // может быть чем угодно;
```

Код становится понятнее, если вызвать тот же Promise в потоке async/await:

```
try {
  const z = await somePromise(); // z is number
} catch(e) {
  // то же самое, e может быть чем угодно!
}
```

Если вы хотите определять собственные ошибки и соответствующим образом их отлавливать, то можете либо написать классы ошибок и выполнять экземпляры проверок, либо создать вспомогательные функции, которые проверяют определенные свойства и сообщают правильный тип с помощью предикатов типа. Axios снова является хорошим примером:

```
function isAxiosError(payload: any): payload is AxiosError {
  return payload !== null
    && typeof payload === 'object'
    && payload.isAxiosError;
}
```

Обработка ошибок в JavaScript и TypeScript может оказаться «ложным другом», если вы работали в других языках программирования с похожими функциями. Помните о различиях и доверяйте команде TypeScript и системе проверки типов, которые предоставят вам правильный поток управления и обеспечат эффективную обработку ваших ошибок.

3.8. Создание моделей исключяющего ИЛИ с помощью приема optional never

Задача

Ваша модель требует наличия взаимоисключающих частей объединения, но ваш API не может использовать свойство kind для различения.

Решение

Воспользуйтесь приемом «Необязательный never» (optional never), чтобы исключить определенные свойства.

Обсуждение

Вы хотите написать функцию, которая обрабатывает результат операции `select` в вашем приложении. Эта операция выдает два списка с перечислением возможных и выбранных параметров. Данная функция может работать как с вызовами операции `select`, которая выдает только одно значение, так и с операцией `select`, которая выдает несколько значений.

Вам необходимо адаптироваться к существующему API, поэтому ваша функция должна иметь возможность обрабатывать оба варианта и принимать решения для одного и нескольких случаев внутри функции.



Конечно, существуют более эффективные способы моделирования API, и об этом можно говорить бесконечно. Но иногда вам приходится иметь дело с существующими API, которые изначально не так уж хороши. TypeScript предоставляет приемы и методы, позволяющие корректно определить типы данных в подобных сценариях.

Ваша модель отражает этот API, поскольку вы можете передавать как одно значение — свойство `value`, так и несколько — свойство `values`:

```
type SelectBase = {
  options: string[];
};

type SingleSelect = SelectBase & {
  value: string;
};

type MultipleSelect = SelectBase & {
  values: string[];
};

type SelectProperties = SingleSelect | MultipleSelect;

function selectCallback(params: SelectProperties) {
  if ("value" in params) {
    // обработка отдельных случаев
  } else if ("values" in params) {
    // обработка нескольких случаев
  }
}

selectCallback({
  options: ["dracula", "monokai", "vscode"],
  value: "dracula",
});
```



```
selectCallback({
  options: ["dracula", "monokai", "vscode"],
  values: ["dracula", "vscode"],
});
```

Все работает как задумано, но не забывайте об особенностях системы структурных типов в TypeScript. Определение `SingleSelect` как типа допускает использование значения всех подтипов; то есть объекты, имеющие оба свойства: и `value`, и `values`, также совместимы с `SingleSelect`. То же самое относится и к `MultipleSelect`. Ничто не мешает вам использовать функцию `selectCallback` с объектом, содержащим оба свойства:

```
selectCallback({
  options: ["dracula", "monokai", "vscode"],
  values: ["dracula", "vscode"],
  value: "dracula",
}); // Все еще работает! Какой выбрать?
```

Значение, которое вы передаете здесь, допустимо, но в вашем приложении оно не имеет смысла. Вы не смогли определить, какая это операция: множественного или одиночного выбора.

В подобных случаях снова нужно разделить два набора значений ровно настолько, чтобы модель стала более понятной. Это можно сделать с помощью приема *optional never*¹. Он состоит в том, чтобы взять свойства, которые являются эксклюзивными для каждой ветви объединения, и добавить их в качестве необязательных (`optional`) свойств типа `never` к другим ветвям:

```
type SelectBase = {
  options: string[];
};

type SingleSelect = SelectBase & {
  value: string;
  values?: never;
};

type MultipleSelect = SelectBase & {
  value?: never;
  values: string[];
};
```

Вы сообщаете TypeScript, что это свойство является необязательным в данной ветке и, когда оно установлено, для него не существует совместимого значения.

¹ Выражаем благодарность Дэнну Вандеркаму, который первым назвал этот прием `optional never` в своем фантастическом блоге Effective TypeScript (<https://effectivetypescript.com/>).

Таким образом, все объекты, содержащие оба свойства, становятся недопустимыми для `SelectProperties`:

```
selectCallback({
  options: ["dracula", "monokai", "vscode"],
  values: ["dracula", "vscode"],
  value: "dracula",
});
// ^ Argument of type '{ options: string[]; values: string[]; value: string; }'
// is not assignable to parameter of type 'SelectProperties'.
```

Типы объединения снова разделяются, но без добавления свойства `kind`. Такая схема отлично работает для моделей, в которых отличительных свойств всего несколько. Если в вашей модели слишком много отличительных свойств и вы можете позволить себе добавить свойство `kind`, то используйте *размеченные типы объединения*, как было показано в рецепте 3.2.

3.9. Эффективное использование утверждений типа

Задача

Ваш код выдает правильные результаты, но типы слишком широкие. Вы знаете лучше!

Решение

Используйте утверждения типов, чтобы сузить набор до меньшего размера, добавив ключевое слово `as`, указывающее на небезопасную операцию.

Обсуждение

Представьте, что бросаете кубик и получаете число от одного до шести. Функция JavaScript состоит из одной строки и использует библиотеку `Math`. Вы хотите работать с суженным типом, объединением шести литеральных типов чисел, обозначающих результаты. Однако ваша операция выдает тип `number`, а он слишком широкий для ваших результатов:

```
type Dice = 1 | 2 | 3 | 4 | 5 | 6;

function rollDice(): Dice {
  let num = Math.floor(Math.random() * 6) + 1;
  return num;
//^ Type 'number' is not assignable to type 'Dice'.(2322)
}
```

Поскольку `number` допускает большее количество значений, чем `Dice`, то TypeScript не позволит вам сузить тип, просто добавив аннотацию к сигнатуре функции. Способ работает только в том случае, если тип является более широким, супертипом:

```
// Все dice – это числа
function asNumber(dice: Dice): number {
  return dice;
}
```

Вместо этого, как и в случае с предикатами типов из рецепта 3.5, мы можем сообщить TypeScript, что знаем лучше, утверждая, что тип более узкий, чем ожидалось:

```
type Dice = 1 | 2 | 3 | 4 | 5 | 6;

function rollDice(): Dice {
  let num = Math.floor(Math.random() * 6) + 1;
  return num as Dice;
}
```

Как и предикаты типов, утверждения типов работают только в пределах супертипов и подтипов предполагаемого типа. Мы можем либо присвоить этому значению более широкий супертип, либо изменить его на более узкий подтип. TypeScript не позволяет нам переключать наборы:

```
function asString(num: number): string {
  return num as string;
}
//      ^- Conversion of type 'number' to type 'string' may
//      be a mistake because neither type sufficiently
//      overlaps with the other.
//      If this was intentional, convert the expression to 'unknown' first.
}
```

Использовать синтаксис `as Dice` довольно удобно. Он указывает на изменение типа, за которое несем ответственность мы, разработчики. Это значит, если что-то пойдет не так, то мы можем легко просканировать код на наличие ключевого слова `as` и найти возможных виновников.



В повседневном языке люди склонны называть утверждения типов приведениями типов. Возможно, это происходит из-за сходства с реальными, явными приведениями типов в языках C, Java и подобных им. Однако утверждение типа сильно отличается от приведения типа. Последнее изменяет не только набор совместимых значений, но и структуру памяти и даже сами значения. Приведение числа с плавающей запятой к целому числу повлечет за собой обрезание мантиссы. Утверждение типа в TypeScript, с другой стороны, изменяет только набор совместимых значений. Само значение остается прежним. Это называется утверждением типа, поскольку вы утверждаете, что тип является чем-то либо более узким, либо более широким, давая больше подсказок системе типов. Поэтому, если вы обсуждаете изменение типов, называйте их утверждениями, а не приведениями.

Утверждения часто используются, когда вы собираете свойства объекта. Вы знаете, что фигура будет, например, `Person`, но сначала нужно задать свойства:

```
type Person = {
  name: string;
  age: number;
};

function createDemoPerson(name: string) {
  const person = {} as Person;
  person.name = name;
  person.age = Math.floor(Math.random() * 95);
  return person;
}
```

Утверждение типа сообщает TypeScript, что пустой объект должен быть `Person` в конечном счете. Впоследствии TypeScript позволяет вам задавать свойства. Это *небезопасная* операция, поскольку вы можете забыть, что задали свойство, и TypeScript не будет жаловаться. Что еще хуже, `Person` может измениться и получить больше свойств, а вы не увидите никаких уведомлений о том, что у вас не хватает свойств:

```
type Person = {
  name: string;
  age: number;
  profession: string;
};

function createDemoPerson(name: string) {
  const person = {} as Person;
  person.name = name;
  person.age = Math.floor(Math.random() * 95);
  // Где профессия?
  return person;
}
```

В подобных ситуациях лучше предпочесть *безопасное* создание объекта. Ничто не мешает вам сделать аннотацию и убедиться, что вы установили все необходимые свойства при присваивании:

```
type Person = {
  name: string;
  age: number;
};

function createDemoPerson(name: string) {
  const person: Person = {
    name,
    age: Math.floor(Math.random() * 95),
  };
  return person;
}
```

Аннотации типов более безопасны, чем утверждения типов, однако в ситуациях, подобных `rollDice`, лучшего выбора нет. В других случаях, когда используется TypeScript, у вас есть выбор, но вы можете предпочесть утверждения типов, даже если у вас есть возможность задать аннотации.

Когда мы используем API `fetch`, например, получаем данные JSON от серверной части, то можем вызвать `fetch` и присвоить результаты аннотированному типу:

```
type Person = {
  name: string;
  age: number;
};
```

```
const ppl: Person[] = await fetch("/api/people").then((res) => res.json());
```

`res.json()` возвращает `any`, и все, что является `any`, можно изменить на любой другой тип с помощью аннотации типа. Нет никакой гарантии, что результат действительно является `Person[]`. Мы можем написать ту же строку по-другому, утверждая, что результатом является `Person[]`, сужая `any` до чего-то более конкретного:

```
const ppl = await fetch("/api/people").then((res) => res.json()) as Person[];
```

Для системы типов это означает то же самое действие, но мы можем легко просмотреть ситуации, в которых могут возникнуть проблемы. Что, если модель в `"/api/people"` изменится? Сложнее обнаружить ошибки, если мы просто ищем аннотации. Утверждение здесь — это индикатор *небезопасной* операции.

Что действительно помогает, так это создание набора моделей, которые работают в рамках вашего приложения. В тот момент, когда вы используете что-то внешнее, например API или правильное вычисление числа, утверждения типа могут показать, что вы пересекли границу.

Как и при использовании предикатов типов (см. рецепт 3.5), утверждения типов возлагают на вас ответственность за правильный тип. Используйте их разумно.

3.10. Использование сигнатур индекса

Задача

Вы хотите работать с объектами, тип значений которых вам известен, но не знаете заранее имен всех свойств.

Решение

Используйте индексные сигнатуры для определения открытого набора ключей, но с конкретными типами значений.

Обсуждение

В веб-интерфейсах есть стиль, при котором вы получаете коллекции в виде объекта JavaScript, где имя свойства примерно соответствует уникальному идентификатору, а значения имеют одинаковую форму. Этот стиль отлично подходит, если вас больше всего интересуют *ключи*, так как простой вызов `Object.keys` дает вам все соответствующие идентификаторы, позволяя быстро фильтровать и индексировать нужные вам значения.

Представим обзор производительности всех ваших сайтов, где вы собираете соответствующие показатели и группируете их по имени домена:

```
const timings =
  {
    "fettblog.eu": {
      ttfb: 300,
      fcp: 1000,
      si:1200,
      lcp: 1500,
      tti: 1100,
      tbt: 10,
    },
    "typescript-book.com": {
      ttfb: 400,
      fcp: 1100,
      si:1100,
      lcp:2200,
      tti:1100,
      tbt: 0,
    },
  },
};
```

Если мы хотим найти домен с наименьшим временем для заданного показателя, то можем создать функцию, в которой будем перебирать все ключи, индексировать каждую запись показателя и сравнивать:

```
function findLowestTiming(collection, metric) {
  let result = {
    domain: "",
    value: Number.MAX_VALUE,
  };
  for (const domain in collection) {
    const timing = collection[domain];
    if (timing[metric] < result.value) {
      result.domain = domain;
      result.value = timing[metric];
    }
  }
  return result.domain;
}
```

Мы хорошие программисты, поэтому хотим соответствующим образом типизировать нашу функцию, чтобы быть уверенными, что не передаем никаких данных, которые не соответствуют нашему представлению о коллекции показателей. Ввести значения для показателей в правой части довольно просто:

```
type Metrics = {
  // Время до первого байта
  ttfb: number;
  // Первая отрисовка содержимого
  fcp: number;
  // Коэффициент скорости
  si: number;
  // Самая большая отрисовка содержимого
  lcp: number;
  // Время до интерактивности
  tti: number;
  // Общее время блокировки
  tbt: number;
};
```

Определить структуру, которая имеет еще не определенный набор ключей, сложнее, но в TypeScript есть инструмент для этого: индексные сигнатуры. Мы можем сообщить TypeScript, что не знаем имен свойств, но знаем, что они будут иметь тип `string` и указывать на `Metrics`:

```
type MetricCollection = {
  [domain: string]: Timings;
};
```

И это все, что нам нужно для определения типа `findLowestTiming`. Мы задаем аннотацию `MetricCollection` для `collection` и обеспечиваем передачу только ключей `Metrics` для второго параметра:

```
function findLowestTiming(
  collection: MetricCollection,
  key: keyof Metrics
): string {
  let result = {
    domain: "",
    value: Number.MAX_VALUE,
  };
  for (const domain in collection) {
    const timing = collection[domain];
    if (timing[key] < result.value) {
      result.domain = domain;
      result.value = timing[key];
    }
  }
  return result.domain;
}
```

Это замечательно, но учтите важный момент. TypeScript позволяет вам читать свойства любой строки, но не проверяет, действительно ли свойство доступно, так что будьте внимательны:

```
const emptySet: MetricCollection = {};
let timing = emptySet["typescript-cookbook.com"].fcp * 2;
// Никаких ошибок типа!
```

Изменение типа сигнатуры индекса на `Metrics` или `undefined` является более реалистичным представлением. Это говорит о том, что вы можете индексировать все возможные строки, но значения может не быть. Это приводит к нескольким дополнительным мерам предосторожности, но в конечном счете является правильным выбором:

```
type MetricCollection = {
  [domain: string]: Metrics | undefined;
};

function findLowestTiming(
  collection: MetricCollection,
  key: keyof Metrics
): string {
  let result = {
    domain: "",
    value: Number.MAX_VALUE,
  };
  for (const domain in collection) {
    const timing = collection[domain]; // Metrics | undefined
    // дополнительная проверка на наличие значений undefined
    if (timing && timing[key] < result.value) {
      result.domain = domain;
      result.value = timing[key];
    }
  }
  return result.domain;
}
```

```
const emptySet: MetricCollection = {};
// доступ с объединением в цепочку optional и оператора нулевого слияния
let timing = (emptySet["typescript-cookbook.com"]?.fcp ?? 0) * 2;
```

Значение `Metrics` или `undefined` не совсем похоже на отсутствующее свойство, но довольно близко, и его достаточно для данного сценария использования. О нюансах между отсутствующими свойствами и неопределенными значениями можно прочесть в рецепте 3.11. Чтобы установить ключи свойства как `optional`, вы сообщаете TypeScript, что `domain` — это не все множество `string`, а подмножество `string` с так называемым *сопоставленным типом*:

```
type MetricCollection = {
  [domain in string]?: Metrics;
};
```


Вы можете определить сигнатуры индексов для всего, что является допустимым ключом свойства: `string`, `number` или `symbol`, а с помощью сопоставленных типов — и для всего, что является их подмножеством. Например, вы можете определить тип для индексации только допустимых граней кубика:

```
type Throws = {
  [x in 1 | 2 | 3 | 4 | 5 | 6]: number;
};
```

Вы также можете добавить свойства к своему типу. Возьмем, к примеру, `ElementCollection`, который позволяет индексировать элементы по номеру, имеет дополнительные свойства для функций `get` и `filter`, а также свойство `length`:

```
type ElementCollection = {
  [y: number]: HTMLElement | undefined;
  get(index: number): HTMLElement | undefined;
  length: number;
  filter(callback: (element: HTMLElement) => boolean): ElementCollection;
};
```

Если вы объединяете свои сигнатуры индексов с другими свойствами, то вам нужно убедиться, что более широкий набор вашей сигнатуры индекса содержит типы из конкретных свойств. В предыдущем примере нет совпадения между индексной сигнатурой числа и строковыми ключами других свойств, но если вы определите индексную сигнатуру строк, которая сопоставлена со `string`, и захотите, чтобы рядом с ней было свойство `count` типа `number`, то TypeScript выдаст ошибку:

```
type StringDictionary = {
  [index: string]: string;
  count: number;
  // Error: Property 'count' of type 'number' is not assignable
  // to 'string' index type 'string'.(2411)
};
```

И это логично: если все строковые ключи указывают на `string`, то зачем `count` указывать на что-то другое? Возникает двусмысленность, и TypeScript этого не допустит. Вам придется расширить тип сигнатуры индекса, чтобы убедиться в том, что меньшее множество является частью большего:

```
type StringOrNumberDictionary = {
  [index: string]: string | number;
  count: number; // работает
};
```

Теперь подмножества `count` содержат как тип из сигнатуры индекса, так и тип значения свойства.

Сигнатуры индекса и сопоставленные типы — мощные инструменты, позволяющие работать с веб-интерфейсами API, а также со структурами данных, которые делают возможным гибкий доступ к элементам. Элементы JavaScript, которые мы знаем и любим, теперь надежно типизированы в TypeScript.

3.11. Различение отсутствующих свойств и значений `undefined`

Задача

Отсутствующие свойства и значения `undefined` — не одно и то же! Вы столкнетесь с ситуациями, когда эта разница будет иметь значение.

Решение

Активируйте `exactOptionalPropertyTypes` в `tsconfig`, чтобы обеспечить более строгую обработку необязательных свойств.

Обсуждение

В нашем программном обеспечении есть пользовательские настройки, где мы можем определить язык пользователя и его предпочтительные цветовые настройки. Это дополнительная тема, то есть основные цвета уже установлены в стиле `default`. Это означает, что пользовательская настройка для `theme` не является обязательной: либо она есть, либо ее нет. Для этого мы используем `optional` свойства TypeScript:

```
type Settings = {
  language: "en" | "de" | "fr";
  theme?: "dracula" | "monokai" | "github";
};
```

Когда параметр `strictNullChecks` активен, обращение к `theme` где-то в вашем коде расширяет количество возможных значений. У вас есть не только три переопределения `theme`, но и возможность `undefined`:

```
function applySettings(settings: Settings) {
  // theme имеет значения "dracula" | "monokai" | "github" | undefined
  const theme = settings.theme;
}
```

Это отличное поведение, поскольку вы действительно хотите быть уверены, что это свойство задано. В противном случае это может привести к ошибкам во время выполнения. Добавление в TypeScript значения `undefined` в список возможных значений необязательных свойств — это хорошо, но не полностью отражает поведение JavaScript. *Необязательные свойства* означают, что данный ключ отсутствует в объекте; это незаметно, но важно. Например, при проверке свойств из-за отсутствия ключа будет возвращено значение `false`:

```
function getTheme(settings: Settings) {
  if ('theme' in settings) { // значение true, если задано это свойство
    return settings.theme;
  }
}
```

```
    }
    return 'default';
}

const settings: Settings = {
  language: "de",
};

const settingsUndefinedTheme: Settings = {
  language: "de",
  theme: undefined,
};

console.log(getTheme(settings)) // "по умолчанию"
console.log(getTheme(settingsUndefinedTheme)) // не определено
```

Здесь мы получаем совершенно разные результаты, несмотря на то что два объекта настроек кажутся похожими. Хуже всего то, что тема `undefined` — это значение, которое мы не считаем допустимым. Однако TypeScript не обманывает нас, так как прекрасно понимает, что проверка на `in` сообщает нам только о том, доступно ли свойство. Возможные возвращаемые значения `getTheme` также содержат `undefined`:

```
type Fn = typeof getTheme;
// type Fn = (settings: Settings)
// => "dracula" | "monokai" | "github" | "default" | undefined
```

И возможно, есть более эффективные методы проверки правильности приведенных здесь значений. При использовании оператора *нулевого слияния* `??` (nullish coalescing) предыдущий код принимает такой вид:

```
function getTheme(settings: Settings) {
  return settings.theme ?? "default";
}

type Fn = typeof getTheme;
// type Fn = (settings: Settings)
// => "dracula" | "monokai" | "github" | "default"
```

Тем не менее проверки `in` действительно и используются разработчиками, а то, как TypeScript интерпретирует необязательные свойства, может вызвать неоднозначную реакцию. Чтение значения `undefined` из необязательного свойства является правильным, но установка необязательного свойства в `undefined` — нет.

Включив параметр `exactOptionalPropertyTypes`, TypeScript изменяет это поведение:

```
// exactOptionalPropertyTypes равно true
const settingsUndefinedTheme: Settings = {
  language: "de",
  theme: undefined,
};
```

```
// Error: Type '{ language: "de"; theme: undefined; }' is
// not assignable to type 'Settings' with 'exactOptionalPropertyTypes: true'.
// Consider adding 'undefined' to the types of the target's properties.
// Types of property 'theme' are incompatible.
// Type 'undefined' is not assignable to type
// '"dracula" | "monokai" | "github"'.(2375)
```

Установка флага `exactOptionalPropertyTypes` еще больше приближает поведение TypeScript к JavaScript. Однако этот флаг не задан в режиме `strict`, поэтому вам нужно установить его самостоятельно, если вы столкнетесь с подобными проблемами.

3.12. Работа с перечислениями

Задача

Перечисления в TypeScript — хорошая абстракция, но, похоже, они ведут себя совсем по-другому по сравнению с остальной системой типов.

Решение

Используйте их с осторожностью, отдавайте предпочтение перечислениям `const`, знайте об их недостатках и, возможно, выберите вместо них типы объединения.

Обсуждение

Перечисления в TypeScript позволяют разработчику определить набор именованных констант, что упрощает документирование намерений или создание набора отдельных вариантов.

Перечисления определяются с помощью ключевого слова `enum`:

```
enum Direction {
  Up,
  Down,
  Left,
  Right,
};
```

Как и классы, перечисления участвуют в пространствах имен `value` и `type`; это значит, вы можете использовать `Direction` при аннотировании типов или в вашем коде JavaScript в качестве значений:

```
// используется как тип
function move(direction: Direction) {
  // ...
}

// используется как значение
move(Direction.Up);
```

Перечисления — синтаксическое расширение JavaScript, то есть не только работают на уровне системы типов, но и генерируют код JavaScript:

```
var Direction;
(function (Direction) {
  Direction[Direction["Up"] = 0] = "Up";
  Direction[Direction["Down"] = 1] = "Down";
  Direction[Direction["Left"] = 2] = "Left";
  Direction[Direction["Right"] = 3] = "Right";
})(Direction || (Direction = {}));
```

Когда вы определяете свое перечисление как `const enum`, TypeScript пытается заменить его использование фактическими значениями, избавляясь от генерируемого кода:

```
const enum Direction {
  Up,
  Down,
  Left,
  Right,
};

// При наличии const enum
// TypeScript транпилирует move(Direction.Up) в следующее:
move(0 /* Direction.Up */);
```

TypeScript поддерживает как строковые, так и числовые перечисления, и оба варианта ведут себя совершенно по-разному.

Перечисления TypeScript по умолчанию являются числовыми, то есть каждому варианту перечисления присвоено числовое значение начиная с 0. Начальное и фактическое значения вариантов перечисления могут быть заданы по умолчанию или пользователем:

```
// По умолчанию
enum Direction {
  Up, // 0
  Down, // 1
  Left, // 2
  Right, // 3
};
```

```
enum Direction {
  Up = 1, // 1
  Down, // 2
  Left, // 3
  Right = 5, // 5
};
```

В некотором смысле числовые перечисления определяют тот же набор, что и тип объединения чисел:

```
type Direction = 0 | 1 | 2 | 3;
```

Но есть и существенные различия. Если тип объединения чисел допускает только строго определенный набор значений, то числовое перечисление позволяет присвоить любое значение:

```
function move(direction: Direction) { /* ... */ }
```

```
move(30); // Это ok!
```

Причина в том, что существует вариант использования флагов с числовыми перечислениями:

```
// Возможных свойств person может быть несколько
enum Traits {
  None, // 0000
  Friendly = 1, // 0001 или 1 << 0
  Mean = 1 << 1, // 0010
  Funny = 1 << 2, // 0100
  Boring = 1 << 3, // 1000
}

// (0010 | 0100) === 0110
let aPersonsTraits = Traits.Mean | Traits.Funny;

if ((aPersonsTraits & Traits.Mean) === Traits.Mean) {
  // Person – это mean, помимо всего прочего
}
```

Перечисления обеспечивают синтаксический уровень для этой ситуации. Чтобы компилятору было проще понять, какие значения допустимы, TypeScript расширяет совместимые значения для числовых перечислений до всего набора `number`.

Варианты перечисления также могут быть инициализированы с помощью строк вместо чисел, что позволяет создать строковое перечисление. Если вы решите написать строковое перечисление, то вам необходимо определить каждый вариант, поскольку строки не могут увеличиваться:

```
enum Status {
  Admin = "Admin",
  User = "User",
  Moderator = "Moderator",
};
```

Строковые перечисления имеют более строгие ограничения, чем числовые. Они позволяют передавать только фактические варианты перечисления, а не весь набор строк. Однако они не позволяют передавать строковый эквивалент:

```
function closeThread(threadId: number, status: Status): {
  // ...
}

closeThread(10, "Admin");
//      ^-- Argument of type '"Admin"' is not assignable to
//      parameter of type 'Status'

closeThread(10, Status.Admin); // Это работает
```

В отличие от всех остальных типов в TypeScript, строковые перечисления являются *номинальными* типами. Это означает, что два перечисления с одинаковым набором значений несовместимы друг с другом:

```
enum Roles {
  Admin = "Admin",
  User = "User",
  Moderator = "Moderator",
};

closeThread(10, Roles.Admin);
//      ^-- Argument of type 'Roles.Admin' is not
//      assignable to parameter of type 'Status'
```

Такая ситуация может стать источником путаницы и разочарования, особенно если значения поступают из другого источника, который не знает о ваших перечислениях, но содержит правильные строковые значения.

Используйте перечисления разумно и знайте об их нюансах. Перечисления отлично подходят для флагов функций и набора именованных констант, когда вы намеренно хотите, чтобы люди использовали структуру данных, а не просто значения.



Начиная с TypeScript 5.0 интерпретация числовых перечислений стала гораздо более строгой. Теперь они, как и перечисления строк, ведут себя как номинальные типы и не содержат весь набор чисел в качестве значений. Тем не менее вы все еще можете встретить кодовые базы, которые используют уникальные возможности числовых перечислений, существовавших до версии 5.0, так что будьте внимательны!

Старайтесь по возможности отдавать предпочтение перечислениям `const`, так как неконстантные перечисления могут увеличить размер вашей кодовой базы, который может оказаться избыточным. Мне встречались проекты с более чем двумя тысячами флагов в неконстантных перечислениях, что приводило к огромным накладным расходам на инструментарий, долгой компиляции и, как следствие, затратам времени выполнения.

Лучше не используйте перечисления вообще. Простой тип объединения работает аналогично и гораздо лучше соответствует остальной системе типов:

```
type Status = "Admin" | "User" | "Moderator";

function closeThread(threadId: number, status: Status) {
  // ...
}

closeThread(10, "Admin"); // Все хорошо
```

Вы получаете все преимущества перечислений, такие как надлежащий инструментарий и безопасность типов, не делая лишних действий и не рискуя вывести код, который вам не нужен. Кроме того, становится понятнее, что нужно передавать и откуда брать значение.

Если вы хотите писать код в стиле перечислений, с объектом и именованным идентификатором, то объект `const` со вспомогательным типом `Values` может просто дать вам желаемое поведение и будет *намного* больше соответствовать JavaScript. Тот же метод применим и к объединениям строк:

```
const Direction = {
  Up: 0,
  Down: 1,
  Left: 2,
  Right: 3,
} as const;

// Получить константные значения направления
type Direction = (typeof Direction)[keyof typeof Direction];

// (typeof Direction)[keyof typeof Direction] yields 0 | 1 | 2 | 3
function move(direction: Direction) {
  // ...
}

move(30); // Это ломает код!

move(0); // Это работает!

move(Direction.Left); // Это тоже работает!
```

Эта строчка особенно интересна:

```
// = 0 | 1 | 2 | 3
type Direction = (typeof Direction)[keyof typeof Direction];
```

Происходит несколько не совсем обычных событий.

- Мы объявляем тип с тем же именем, что и значение. Это возможно благодаря тому, что в TypeScript есть отдельные пространства имен значений и типов.

- Используя оператор `typeof`, мы получаем тип из `Direction`. Поскольку `Direction` находится в контексте `const`, мы получаем литеральный тип.
- Мы индексируем тип `Direction` с помощью его собственных ключей, оставляя все значения в правой части объекта: `0`, `1`, `2` и `3`. Короче говоря, это тип объединения чисел.

Использование типов объединения не вызывает никаких неожиданностей:

- вы *знаете*, какой код в итоге получите на выходе;
- вы не столкнетесь с изменением поведения из-за того, что кто-то решит перейти от строкового перечисления к числовому;
- вы обеспечите безопасность типов там, где это необходимо;
- вы предоставляете своим коллегам и пользователям те же удобства, которые возможны благодаря перечислениям.

Но, честно говоря, используя простой тип объединения строк, вы получаете именно то, что вам нужно: безопасность типов, автозаполнение и предсказуемое поведение.

3.13. Определение номинальных типов в системе структурных типов

Задача

В вашем приложении есть несколько типов, которые являются псевдонимами одного и того же примитивного типа, но имеют совершенно разную семантику. Структурная типизация считает их одинаковыми, но это не так!

Решение

Используйте классы-обертки или создайте пересечение вашего примитивного типа с литеральным объектным типом и задействуйте его для различения двух целых чисел.

Обсуждение

Система типов в TypeScript является структурной. Это означает, что если два типа имеют схожую структуру, то их значения совместимы друг с другом:

```
type Person = {
  name: string;
  age: number;
};
```

```
type Student = {
  name: string;
  age: number;
};

function acceptsPerson(person: Person) {
  // ...
}

const student: Student = {
  name: "Hannah",
  age: 27,
};

acceptsPerson(student); // все в порядке
```

JavaScript довольно активно использует объектные литералы, а TypeScript пытается определить их тип или *структуру*. Система структурных типов имеет большой смысл в этой ситуации, поскольку значения могут поступать откуда угодно и должны быть совместимы с определениями интерфейсов и типов.

Однако бывают случаи, когда необходимо более точно определить типы. Что касается типов объектов, то мы узнали о таких методах, как *размеченные* типы *объединения* со свойством `kind` в рецепте 3.2, *исключающее ИЛИ* с необязательным `never` в рецепте 3.8. Перечисления `string` тоже являются номинальными, как мы видели в рецепте 3.12.

Эти измерения достаточно хороши для типов объектов и перечислений, но не решают проблему, если у вас есть два независимых типа, которые используют тот же набор значений, что и примитивные типы. Что делать, если ваш восьмизначный номер счета и ваш баланс указывают на тип `number`, а вы их перепутали? Получение восьмизначного числа на своем балансе — это приятный сюрприз, но, скорее всего, что-то пошло не так.

Или, возможно, вам нужно проверить строки, вводимые пользователем, и вы хотите быть уверены, что используете в своей программе только проверенный пользовательский ввод, а не возвращаетесь к исходной, возможно небезопасной, строке.

TypeScript позволяет имитировать номинальные типы в системе типов в целях повышения безопасности. Хитрость состоит в том, чтобы разделить наборы возможных значений с разными свойствами ровно настолько, чтобы гарантировать, что одни и те же значения не попадают в один набор.

Для достижения этой цели можно воспользоваться обергиванием классов. Вместо того чтобы работать со значениями напрямую, мы обернем каждое значение в класс. С помощью свойства `private kind` мы гарантируем, что они не пересекаются:

```
class Balance {
  private kind = "balance";
  value: number;
```

```
    constructor(value: number) {  
        this.value = value;  
    }  
}
```

```
class AccountNumber {  
    private kind = "account";  
    value: number;  
  
    constructor(value: number) {  
        this.value = value;  
    }  
}
```

Здесь интересно вот что: мы используем `private` свойства, поэтому TypeScript будет различать эти два класса. Сейчас оба свойства имеют тип `string`. Несмотря на то что у них разные значения, их можно изменять внутренне.

Но классы работают по-разному. При наличии членов `private` или `protected` TypeScript считает два типа совместимыми, если они происходят из одного объявления. В противном случае они не считаются совместимыми.

Это позволяет нам усовершенствовать данный паттерн с помощью более общего подхода. Вместо того чтобы определять член `kind` и присваивать ему значение, мы задаем член `nominal` в каждом объявлении класса, который имеет тип `void`. Это в достаточной степени разделяет оба класса, но не дает нам использовать `_nominal`. Тип `void` позволяет только установить `_nominal` в значение `undefined`, а оно является фальшивым и потому совершенно бесполезно:

```
class Balance {  
    private _nominal: void = undefined;  
    value: number;  
  
    constructor(value: number) {  
        this.value = value;  
    }  
}  
  
class AccountNumber {  
    private _nominal: void = undefined;  
    value: number;  
  
    constructor(value: number) {  
        this.value = value;  
    }  
}  
  
const account = new AccountNumber(12345678);  
const balance = new Balance(10000);  
  
function acceptBalance(balance: Balance) {  
    // ...  
}
```

```

acceptBalance(balance); // ok
acceptBalance(account);
// ^ Argument of type 'AccountNumber' is not
// assignable to parameter of type 'Balance'.
// Types have separate declarations of a
// private property '_nominal'.(2345)

```

Теперь мы можем различать два типа, которые будут иметь одинаковый набор значений. Единственным недостатком этого подхода является то, что мы обертываем исходный тип, то есть каждый раз, когда мы хотим работать с исходным значением, нам нужно его развернуть.

Другой способ имитации номинальных типов — пересечение примитивного типа с фирменным типом объекта со свойством `kind`. Таким образом, мы сохраняем все операции исходного типа, но нам нужно потребовать, чтобы утверждения типа сообщали TypeScript, что мы хотим использовать эти типы по-другому.

Как мы узнали в рецепте 3.9, можно безопасно утверждать другой тип, если он является подтипом или супертипом исходного:

```

type Credits = number & { _kind: "credits" };

type AccountNumber = number & { _kind: "accountNumber" };

const account = 12345678 as AccountNumber;
let balance = 10000 as Credits;
const amount = 3000 as Credits;

function increase(balance: Credits, amount: Credits): Credits {
  return (balance + amount) as Credits;
}

balance = increase(balance, amount);
balance = increase(balance, account);
// ^ Argument of type 'AccountNumber' is not
// assignable to parameter of type 'Credits'.
// Type 'AccountNumber' is not assignable to type '{ _kind: "credits"; }'.
// Types of property '_kind' are incompatible.
// Type '"accountNumber"' is not assignable to type '"credits"'.(2345)

```

Обратите внимание еще и на то, что сложение `balance` и `amount` по-прежнему работает так, как было задумано изначально, но снова выдает число. Поэтому нам нужно добавить еще одно утверждение:

```

const result = balance + amount; // результатом будет число
const credits = (balance + amount) as Credits; // credits – это Credits

```

У обоих подходов есть свои преимущества и недостатки, и то, какой из них предпочесть, зависит в основном от вашей ситуации. Оба подхода представляют собой

обходные пути и приемы, разработанные членами сообщества на основе их понимания поведения системы типов.

В системе отслеживания проблем TypeScript на GitHub (<https://oreil.ly/XxmUV>) ведутся дискуссии об открытии системы типов для номинальных типов, и эта возможность постоянно изучается. Одна из идей — использовать для разграничения ключевое слово `unique` из символов:

```
// Гипотетический код, это не работает!  
type Balance = unique number;  
type AccountNumber = unique number;
```

На момент написания книги эта идея, как и многие другие, оставалась перспективной.

3.14. Включение свободного автозаполнения для подмножеств строк

Задача

Ваш API позволяет передавать любую строку, но вы все равно хотите показать несколько строковых значений для автозаполнения.

Решение

Добавьте `string & {}` в тип объединения строковых литералов.

Обсуждение

Допустим, вы определяете API для доступа к системе управления контентом. Существуют предопределенные типы контента, такие как `post`, `page` и `asset`, но разработчики могут определять собственные.

Вы создаете функцию `retrieve` с единственным параметром — типом контента, который позволяет загружать записи:

```
type Entry = {  
  // будет определено  
};  
  
function retrieve(contentType: string): Entry[] {  
  // будет определено  
}
```

Это работает достаточно хорошо, но вы хотите дать пользователям подсказку о стандартных вариантах типа контента. Можно создать вспомогательный тип, в котором перечислены все predetermined типы контента в виде строковых литералов в объединении со `string`:

```
type ContentType = "post" | "page" | "asset" | string;

function retrieve(content: ContentType): Entry[] {
  // будет определено
}
```

Код очень хорошо описывает вашу ситуацию, но имеет и недостаток: `post`, `page` и `asset` являются подтипами `string`, поэтому их объединение со `string` фактически помещает подробную информацию в более широкий набор.

Это означает, что вы не получаете подсказки по завершении оператора через редактор, как показано на рис. 3.3.

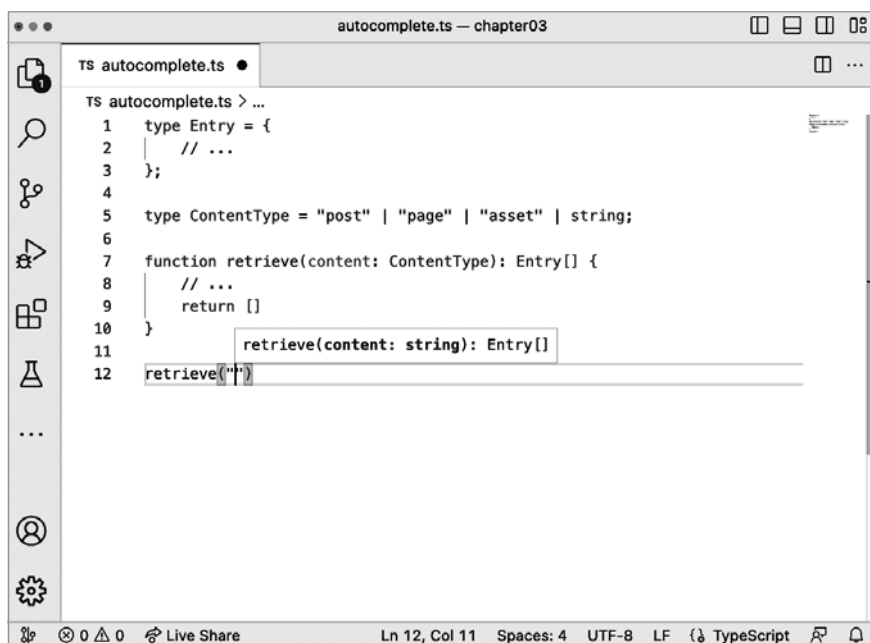


Рис. 3.3. TypeScript расширяет `ContentType` до всего множества `string`, тем самым поглощая информацию об автозаполнении

Чтобы сохранить информацию об автозаполнении и литеральные типы, нужно пересечь `string` с пустым типом объекта `{}`:

```
type ContentType = "post" | "page" | "asset" | string & {};
```

Это изменение дает более мягкий эффект. Оно не изменяет количества совместимых значений `ContentType`, но переводит TypeScript в режим, который предотвращает сокращение подтипов и сохраняет литеральные типы.

Эффект вы можете увидеть на рис. 3.4, где `ContentType` не преобразуется в `string`, и поэтому все буквенные значения доступны для завершения оператора в текстовом редакторе.

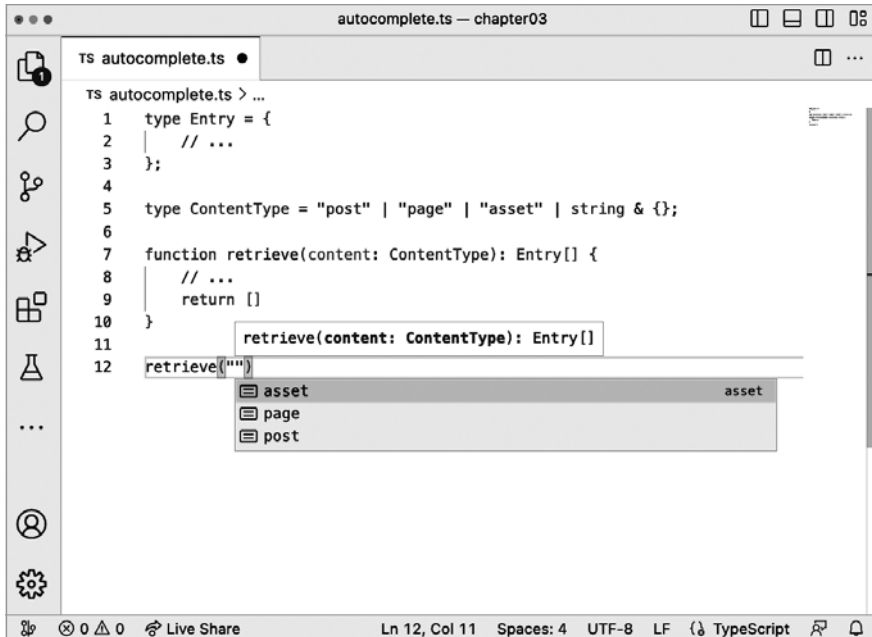


Рис. 3.4. При пересечении `string` с пустым объектом сохраняются подсказки по завершении инструкции

Тем не менее каждая строка является допустимым `ContentType`. Это просто меняет опыт работы разработчиков с вашим API и дает подсказки там, где это необходимо.

Данный прием используется в таких популярных библиотеках, как `CSSType` (<https://oreil.ly/lwtC5>), или при определении типов Definitely Typed для React (<https://oreil.ly/epbLV>).

ГЛАВА 4

Обобщенные типы

До сих пор нашей главной целью было использовать гибкость, присущую JavaScript, и найти способ формализовать ее с помощью системы типов. Мы добавили статические типы для языка с динамической типизацией, чтобы сообщать о намерениях, получать инструментарий и выявлять ошибки до их появления.

Однако некоторые части JavaScript на самом деле не заботятся о статических типах. Например, функция `isKeyAvailableInObject` должна проверять только то, доступен ли ключ в объекте; ей не нужно знать о конкретных типах. Чтобы правильно формализовать подобную функцию, мы можем использовать систему структурных типов TypeScript и описать только один тип: либо очень широкий (пожертвовав информацией), либо очень строгий (пожертвовав гибкостью).

Но мы не хотим ничем жертвовать. Нам нужны и гибкость, и информация. Обобщенные типы (generics) в TypeScript — именно то, что нам нужно. Мы можем описывать сложные отношения и формализовать структуру для данных, которые еще не определены.

Обобщенные типы в сочетании с сопоставленными и вспомогательными типами, а также картами и модификаторами типов открывают возможность метатипизации, позволяющей создавать новые типы на основе старых и сохранять отношения между типами неизменными, в то время как вновь созданные типы проверяют исходный код на наличие возможных ошибок.

Итак, мы переходим к расширенным концепциям TypeScript.

4.1. Обобщение сигнатур функций

Задача

У вас есть две функции, которые работают одинаково, но с разными и в значительной степени несовместимыми типами.

Решение

Обобщите их поведение, используя обобщенные типы.

Обсуждение

Вы пишете приложение, которое хранит в объекте несколько языковых файлов (например, субтитры). Ключи — это коды языков, а значения — URL. Вы загружаете языковые файлы, выбирая их с помощью кода языка, который поступает из какого-либо API или пользовательского интерфейса в виде строки. Чтобы убедиться, что код языка правильный и допустимый, вы добавляете функцию `isLanguageAvailable`, которая выполняет входную проверку и устанавливает правильный тип с помощью предиката типа:

```
type Languages = {
  de: URL;
  en: URL;
  pt: URL;
  es: URL;
  fr: URL;
  ja: URL;
};

function isLanguageAvailable(
  collection: Languages,
  lang: string
): lang is keyof Languages {
  return lang in collection;
}

function loadLanguage(collection: Languages, lang: string) {
  if (isLanguageAvailable(collection, lang)) {
    // lang — это ключ Languages
    collection[lang]; // доступ есть!
  }
}
```

То же приложение, другая ситуация, совершенно другой файл. Вы загружаете мультимедийные данные в HTML-элемент: либо аудио, либо видео, либо их комбинацию с определенными анимациями в элементе `canvas`. Все элементы уже есть в приложении, но вам нужно выбрать из них правильный, основываясь на входных данных, полученных от API. Опять же, выбор представляет собой строку, и вы пишете функцию `isElementAllowed`, чтобы убедиться, что входные данные действительно являются допустимым ключом вашей коллекции `AllowedElements`:

```
type AllowedElements = {
  video: HTMLVideoElement;
```

```

    audio: HTMLAudioElement;
    canvas: HTMLCanvasElement;
};

function isElementAllowed(
    collection: AllowedElements,
    elem: string
): elem is keyof AllowedElements {
    return elem in collection;
}

function selectElement(collection: AllowedElements, elem: string) {
    if (isElementAllowed(collection, elem)) {
        // elem — это ключ AllowedElements
        collection[elem]; // доступ есть
    }
}

```

Вы можете увидеть, что оба сценария очень похожи. Особенно это касается функций защиты типа. Если убрать всю информацию о типах и выровнять имена, то они будут идентичными:

```

function isAvailable(obj, key) {
    return key in obj;
}

```

Два из них существуют благодаря информации о типе, которую мы получаем. Не из-за входных параметров, а из-за предикатов типа. В обоих сценариях мы можем узнать больше о входных параметрах, утверждая определенный тип `keyof`.

Проблема в том, что оба типа входных данных для коллекции совершенно разные и никак не пересекаются, за исключением пустого объекта, для которого мы не получим столько ценной информации, если создадим тип `keyof`. А `keyof {}` — это на самом деле `never`.

Но в коде есть информация о типе, которую мы можем обобщить. Мы знаем, что первый входной параметр — это объект, а второй — ключ свойства. Если эта проверка завершается значением `true`, то мы понимаем, что первый параметр является ключом второго.

Чтобы обобщить эту функцию, мы можем добавить к `isAvailable` *параметр обобщенного типа* `Obj`, заключенный в угловые скобки. Это заполнитель для фактического типа, который будет заменен при добавлении `isAvailable`. Мы можем использовать данный параметр так же, как мы использовали бы `AllowedElements` или `Languages`, и можем добавить предикат типа. Поскольку `Obj` может быть заменен *любым* типом, то `key` должен содержать все возможные ключи свойств — `string`, `symbol` и `number`:

```
function isAvailable<Obj>(  
  obj: Obj,  
  key: string | number | symbol  
) : key is keyof Obj {  
  return key in obj;  
}  
  
function loadLanguage(collection: Languages, lang: string) {  
  if (isAvailable(collection, lang)) {  
    // lang – это ключ Languages  
    collection[lang]; // доступ есть!  
  }  
}  
  
function selectElement(collection: AllowedElements, elem: string) {  
  if (isAvailable(collection, elem)) {  
    // elem – это ключ AllowedElements  
    collection[elem]; // доступ есть!  
  }  
}
```

Вот и все: одна функция, которая работает в обоих сценариях, независимо от того, какие типы мы заменяем на `Obj`. Точно так же, как работает JavaScript! Мы по-прежнему получаем ту же функциональность и правильную информацию о типе. Доступ к индексам становится безопасным без ущерба для гибкости.

Что самое интересное? Мы можем использовать `isAvailable` точно так же, как мы использовали бы нетипизированный эквивалент JavaScript. Это связано с тем, что TypeScript выводит типы для параметров обобщенного типа в процессе использования. А это сопровождается некоторыми интересными побочными эффектами. Более подробную информацию об этом вы можете прочитать в рецепте 4.3.

4.2. Создание связанных аргументов функций

Задача

Вы пишете функции, в которых второй параметр зависит от первого.

Решение

Аннотируйте каждый параметр с помощью обобщенного типа и создайте связь между ними, используя обобщенные ограничения.

Обсуждение

Как и в рецепте 4.1, наше приложение хранит список субтитров в объекте типа `Languages`, который содержит набор ключей, описывающих код языка, и `URL` в качестве значения:

```
type Languages = {
  de: URL;
  en: URL;
  pt: URL;
  es: URL;
  fr: URL;
  ja: URL;
};
```

```
const languages: Languages = { /* ... */ };
```

В нашем приложении есть несколько подобных списков, и мы можем объединить их в тип `URLList`, индексные сигнатуры которого позволяют использовать любой ключ `string`:

```
type URLList = {
  [x: string]: URL;
};
```

`URLList` — супертип `Languages`: каждое значение типа `Languages` — это `URLList`, но не каждый `URLList` — `Languages`. Тем не менее мы можем использовать `URLList` для написания функции `fetchFile`, в которой загружаем определенную запись из этого списка:

```
function fetchFile(urls: URLList, key: string) {
  return fetch(urls[key]).then((res) => res.json());
}
```

```
const de = fetchFile(languages, "de");
const it = fetchFile(languages, "it");
```

Проблема состоит в том, что тип `string` для `key` допускает слишком много записей. Например, не определены итальянские субтитры, но `fetchFile` не мешает нам загружать `"it"` в качестве кода языка. Кроме того, когда мы загружаем элементы из определенного `URLList`, было бы хорошо знать, к каким ключам мы можем обращаться.

Мы можем решить эту проблему, заменив более широкий тип на обобщенный тип и установив *обобщенное ограничение*, чтобы гарантировать передачу подтипа `URLList`. Таким образом, сигнатура функции ведет себя по-прежнему, но мы можем работать с подтипами гораздо более успешно. Мы определяем параметр обобщенного типа `List`, который является подтипом `URLList`, и устанавливаем для `key` значение `keyof List`:

```
function fetchFile<List extends URLList>(urls: List, key: keyof List) {
  return fetch(urls[key]).then((res) => res.json());
}
```

```
const de = fetchFile(languages, "de");
const it = fetchFile(languages, "it");
//                ^
// Argument of type '"it"' is not assignable to
// parameter of type 'keyof Languages'.(2345)
```

В момент вызова `fetchFile` обобщенный тип `List` будет заменен на фактический тип, и мы знаем, что `"it"` не является частью ключей `Languages`. TypeScript покажет, когда мы допустили опечатку или выбрали элементы, которые не являются частью наших типов данных.

Эта схема работает и в том случае, если мы загружаем много ключей. Те же ограничения, тот же эффект:

```
function fetchFiles<List extends URLList>(urls: List, keys: (keyof List)[]) {
  const els = keys.map((el) =>
    fetch(urls[el])
      .then((res) => res.json())
      .then((data) => [el, data])
  );
  return els;
}
```

```
const de_and_fr = fetchFiles(languages, ["de", "fr"]); // Promise<any[]>[]
const de_and_it = fetchFiles(languages, ["de", "it"]);
//                ^
// Тип '"it"' не может быть назначен типу 'keyof Languages'.(2322)
```

Мы сохраняем результаты в кортеже с ключом языка в качестве первого элемента и данными — в качестве второго. Но когда мы получаем результат, то он представляет собой массив `Promises`, который разрешается в `any[]`. Это понятно, ведь функция `fetch` ничего не говорит нам о загруженных данных; а поскольку `data` имеет тип `any` и, следовательно, самый широкий тип, то он просто поглощает `el`, который является `keyof List`.

Однако на данном этапе мы знаем больше. Например, нам известно, что `[el, data]` — это не массив, а кортеж. Есть тонкое, но важное различие, которое показано в рецепте 2.4. Если мы аннотируем результат с помощью типа `tuple`, то получаем дополнительную информацию из возвращаемых значений:

```
function fetchFiles<List extends URLList>(urls: List, keys: (keyof List)[]) {
  const els = keys.map((el) =>
    fetch(urls[el])
      .then((res) => res.json())
      .then((data) => {
        const entry: [keyof List, any] = [el, data];
      })
  );
  return els;
}
```

```

    return entry;
  })
);
return els;
}

```

```
const de_and_fr = fetchFiles(languages, ["de", "fr"]);
```

Функция `fetchFiles` теперь возвращает массив `Promises` типа `[keyof List, any]`. Таким образом, как только мы заменим `List` на `Languages`, то узнаем, что единственными возможными ключами могут быть коды языков.

Однако здесь все же есть один нюанс. Как видно из предыдущего примера кода, единственными доступными языками в `de_and_fr` являются немецкий и французский, но компилятор не предупреждает нас о том, что позже мы должны проверить наличие английского языка. Компилятор должен иметь возможность делать это, поскольку данное условие всегда будет возвращать `false`:

```

for (const result of de_and_fr) {
  if (result[0] === "en") {
    // Английский?
  }
}

```

Проблема в том, что мы снова имеем дело со слишком широким типом. Да, `keyof List` намного уже, чем `string`, но мы также можем заменить все ключи меньшим набором.

Нам нужно повторить тот же процесс:

- 1) создать новый параметр обобщенного типа;
- 2) установить более широкий тип в качестве ограничения для вновь созданного параметра обобщенного типа;
- 3) использовать параметр в сигнатуре функции, чтобы заменить фактический тип.

И точно так же мы можем заменить `keyof List` подтипом: `"de" | "fr"`:

```

function fetchFiles<List extends URLList, Keys extends keyof List>(
  urls: List,
  keys: Keys[]
) {
  const els = keys.map((e1) =>
    fetch(urls[e1])
      .then((res) => res.json())
      .then((data) => {
        const entry: [Keys, any] = [e1, data];
        return entry;
      })
  );
}

```

```
    return els;
}
```

```
const de_and_fr = fetchFiles(languages, ["de", "fr"]);
```

Этот прием хорош тем, что мы можем устанавливать связи между параметрами обобщенных типов. Второй параметр типа может быть ограничен какими-то элементами первого параметра обобщенного типа. Это позволяет очень точно сузить диапазон, пока мы не заменим его реальными значениями. Каков эффект? Мы знаем о возможных значениях наших типов в любом месте кода. Поэтому не будем проверять наличие английского языка, если уже можем сказать, что никогда не запрашивали загрузку английского:

```
for (const entry of de_and_fr) {
  const result = await entry;
  if (result[0] === "en") {
    // Это условие всегда будет возвращать 'false', так как типы
    // ". "de" | "fr" не "en" не пересекаются.(2367)
  }
}
```

Одна проверка, от которой мы не избавились, — это какой язык находится в позиции 0.

Один нюанс, который мы не учли, — это *создание экземпляров обобщенного типа*. Мы позволяем заменять параметры типа реальными значениями в момент использования точно так же, как и при выведении типов. Но мы могли бы заменить их и явно с помощью аннотаций:

```
const de_and_ja = fetchFiles<Languages, "ja" | "de">(languages, ["de"]);
```

Здесь типы сообщают, что могут быть и японские субтитры, хотя из использования видно, что мы загружаем только немецкие. Пусть это послужит напоминанием, а больше информации вы найдете в рецепте 4.4.

4.3. Избавление от `any` и `unknown`

Задача

Похоже, что параметры обобщенного типа и типы `any` и `unknown` описывают очень широкие наборы значений. В каких случаях следует использовать `any`?

Решение

Используйте параметры обобщенного типа, когда в конце концов дойдете до фактического типа. В рецепте 2.2 описано, как выбрать между `any` и `unknown`.

Обсуждение

Когда мы используем обобщенные типы, то может показаться, что они заменяют типы `any` и `unknown`. Возьмем функцию `identity`: ее единственная задача состоит в том, чтобы вернуть значение, переданное в качестве входного параметра:

```
function identity(value: any): any {
  return value;
}
```

```
let a = identity("Hello!");
let b = identity(false);
let c = identity(2);
```

Она принимает значения любого типа, и возвращаемый тип тоже может быть любым. Мы можем написать ту же функцию, используя `unknown`, если хотим получить безопасный доступ к свойствам:

```
function identity(value: unknown): unknown {
  return value;
}
```

```
let a = identity("Hello!");
let b = identity(false);
let c = identity(2);
```

Мы можем даже смешивать и сопоставлять `any` и `unknown`, но результат всегда один и тот же: информация о типе теряется. Тип возвращаемого значения — тот, каким мы его определяем.

Теперь напишем ту же функцию, содержащую обобщенные типы вместо типов `any` или `unknown`. В аннотации к ее типу указано, что обобщенный тип является еще и типом возвращаемого значения:

```
function identity<T>(t: T): T {
  return t;
}
```

Мы можем использовать эту функцию для передачи любого значения и посмотреть, какой тип выводит TypeScript:

```
let a = identity("Hello!"); // a – это строка
let b = identity(2000); // b – это число
let c = identity({ a: 2 }); // c – это { a: число }
```

Присвоение привязке значения с помощью `const` вместо `let` дает несколько иные результаты:

```
const a = identity("Hello!"); // a – "Hello!"
const b = identity(2000); // b – 2000
const c = identity({ a: 2 }); // c – это { a: число }
```


Для примитивных типов TypeScript заменяет параметр обобщенного типа фактическим. Мы можем с успехом использовать этот принцип в более сложных ситуациях.

Кроме того, обобщенные типы TypeScript позволяют *аннотировать* параметр обобщенного типа:

```
const a = identity<string>("Hello!"); // a – это строка
const b = identity<number>(2000); // b – это число
const c = identity<{ a: 2 }>({ a: 2 }); // c – это { a: 2 }
```

Если данное поведение напоминает вам аннотацию и выводение, описанные в рецепте 3.4, то вы абсолютно правы. Это очень похоже, но в функциях есть параметры обобщенного типа.

При использовании обобщенных типов без ограничений мы можем писать функции, которые работают со значениями любого типа. Внутри они ведут себя как unknown, а это значит, что мы можем использовать защиту типа, чтобы сузить его. Самое большое различие — что после использования функции мы заменяем наши обобщенные типы реальными, не теряя при этом никакой информации о типизации.

Это позволяет нам быть немного более понятными с нашими типами, а не просто разрешать все. Функция `pairs`, показанная ниже, принимает два аргумента и создает кортеж:

```
function pairs(a: unknown, b: unknown): [unknown, unknown] {
  return [a, b];
}

const a = pairs(1, "1"); // [unknown, unknown]
```

Благодаря параметрам обобщенного типа мы получаем хороший тип кортежа:

```
function pairs<T, U>(a: T, b: U): [T, U] {
  return [a, b];
}

const b = pairs(1, "1"); // [number, string]
```

Используя тот же параметр обобщенного типа, мы можем быть уверены, что получаем кортежи, в которых каждый элемент имеет один и тот же тип:

```
function pairs<T>(a: T, b: T): [T, T] {
  return [a, b];
}

const c = pairs(1, "1");
//           ^
// Argument of type 'string' is not assignable to parameter of type 'number'
```

Итак, стоит ли использовать обобщенные типы везде? Не обязательно. В этой главе приведено множество решений, которые основаны на получении необходимой информации о типе в нужное время. Если вас устраивает более широкий набор значений и вы можете рассчитывать на совместимость подтипов, то вам вообще не нужно использовать обобщенные типы. Если в вашем коде есть тип `any` или `unknown`, то подумайте, нужно ли вам в какой-то момент добавлять фактический тип. Вместо этого вы можете добавить параметр обобщенного типа.

4.4. Создание экземпляров обобщенного типа

Задача

Вы понимаете, как обобщенные типы заменяются реальными, но иногда ошибки наподобие `Foo is assignable to the constraint of type Bar, but could be instantiated with a different subtype of constraint Baz` (`Foo` может быть назначен ограничению типа `Bar`, но может быть создан с помощью другого подтипа ограничения `Baz`) приводят вас в замешательство.

Решение

Помните, что значения обобщенного типа могут быть явно и неявно заменены различными подтипами. Пишите код, который может работать с подтипами.

Обсуждение

Вы создаете логику фильтрации для своего приложения. У вас есть различные правила фильтрации, которые вы можете компоновать с помощью комбинаторов `"and" | "or"`. Кроме того, вы можете связать обычные правила фильтрации с результатами *комбинаторных фильтров*. Вы создаете свои типы на основе этого поведения:

```
type FilterRule = {
  field: string;
  operator: string;
  value: any;
};

type CombinatorialFilter = {
  combinator: "and" | "or";
  rules: FilterRule[];
};

type ChainedFilter = {
  rules: (CombinatorialFilter | FilterRule)[];
};

type Filter = CombinatorialFilter | ChainedFilter;
```

Теперь вы хотите написать функцию `reset`, которая, основываясь на уже предоставленном фильтре, сбрасывает все правила. Вы используете защиту типов, чтобы различать `CombinatorialFilter` и `ChainedFilter`:

```
function reset(filter: Filter): Filter {
  if ("combinator" in filter) {
    // фильтр – CombinatorialFilter
    return { combinator: "and", rules: [] };
  }
  // фильтр – ChainedFilter
  return { rules: [] };
}

const filter: CombinatorialFilter = { rules: [], combinator: "or" };
const resetFilter = reset(filter); // resetFilter – это Filter
```

Поведение соответствует желаемому, но тип возвращаемого значения функции `reset` слишком широк. Передавая `CombinatorialFilter`, мы должны быть уверены, что фильтр сброса является еще и `CombinatorialFilter`. Здесь это тип объединения, как и указано в сигнатуре нашей функции. Но вы хотите быть уверены, что если передаете фильтр определенного типа, то и возвращаете фильтр того же типа. Поэтому вы заменяете широкий тип объединения параметром обобщенного типа, который ограничивается `Filter`. Тип возвращаемого значения работает как предполагалось, но реализация вашей функции приводит к ошибкам:

```
function reset<F extends Filter>(filter: F): F {
  if ("combinator" in filter) {
    return { combinator: "and", rules: [] };
  }
  // ^ '{ combinator: "and"; rules: never[]; }' is assignable to
  //   the constraint of type 'F', but 'F' could be instantiated
  //   with a different subtype of constraint 'Filter'.
}
return { rules: [] };
//^ '{ rules: never[]; }' is assignable to the constraint of type 'F',
//   but 'F' could be instantiated with a different subtype of
//   constraint 'Filter'.
}

const resetFilter = reset(filter); // resetFilter is CombinatorialFilter
```

Вы хотите различать две части объединения, но TypeScript мыслит более широко. Он знает, что вы можете передать объект, который *структурно совместим* с `Filter`, но имеет больше свойств и, следовательно, является подтипом.

Это означает, что вы можете вызвать `reset` с экземпляром `F`, созданным для подтипа, и ваша программа переопределит все лишние свойства. Это неправильно, и TypeScript сообщает вам об этом:

```
const onDemandFilter = reset({
  combinator: "and",
```

```

    rules: [],
    evaluated: true,
    result: false,
  });
  /* filter is {
    combinator: "and";
    rules: never[];
    evaluated: boolean;
    result: boolean;
  }; */

```

Решить эту проблему можно, написав код, который может работать с подтипами. Клонировать входной объект (все еще типа F), установите свойства, которые необходимо соответствующим образом изменить, и вернуть то, что по-прежнему имеет тип F:

```

function reset<F extends Filter>(filter: F): F {
  const result = { ...filter }; // результат – F
  result.rules = [];
  if ("combinator" in result) {
    result.combinator = "and";
  }
  return result;
}

const resetFilter = reset(filter); // resetFilter – CombinatorialFilter

```

Обобщенные типы могут быть одними из многих в объединении, но их может быть гораздо больше. Система структурных типов TypeScript позволяет вам работать с различными подтипами, и ваш код должен это отражать.

Рассмотрим еще один сценарий, который отличается от предыдущего, но имеет аналогичный результат. Мы хотим создать древовидную структуру данных и написать рекурсивный тип, в котором будут храниться все элементы дерева. Данный тип может иметь подтип, поэтому мы пишем функцию `createRootItem` с параметром обобщенного типа, поскольку хотим создать ее экземпляр с правильным подтипом:

```

type TreeItem = {
  id: string;
  children: TreeItem[];
  collapsed?: boolean;
};

function createRootItem<T extends TreeItem>(): T {
  return {
    id: "root",
    children: [],
  };
}
// '{ id: string; children: never[]; }' можно присвоить ограничению
// типа 'T', но 'T' может быть создан с другим подтипом

```

```
// ограничения 'TreeItem'.(2322)
}
```

```
const root = createRootItem(); // корень – TreeItem
```

Мы получаем аналогичную ошибку, поскольку не можем утверждать, что возвращаемое значение будет совместимо со всеми подтипами. Чтобы решить эту проблему, нужно избавиться от обобщенного типа! Мы знаем, как будет выглядеть возвращаемый тип, — это `TreeItem`:

```
function createRootItem(): TreeItem {
  return {
    id: "root",
    children: [],
  };
}
```

Самые простые решения часто оказываются лучшими. Но теперь мы хотим расширить программу, чтобы иметь возможность добавлять дочерние элементы типа или подтипа `TreeItem` во вновь созданный корневой каталог. Мы пока не добавляем никакие обобщенные типы и немного недовольны:

```
function attachToRoot(children: TreeItem[]): TreeItem {
  return {
    id: "root",
    children,
  };
}
```

```
const root = attachToRoot([]); // TreeItem
```

Элемент `root` имеет тип `TreeItem`, но мы теряем любую информацию о подтипах дочерних элементов. Даже если мы добавим параметр обобщенного типа только для дочерних элементов, ограниченный типом `TreeItem`, то не сохраним эту информацию в дальнейшем:

```
function attachToRoot<T extends TreeItem>(children: T[]): TreeItem {
  return {
    id: "root",
    children,
  };
}
```

```
const root = attachToRoot([
  {
    id: "child",
    children: [],
    collapsed: false,
    marked: true,
  },
]); // корень – TreeItem
```

Начиная добавлять обобщенный тип в качестве возвращаемого типа, мы сталкиваемся с уже известными нам проблемами. Чтобы решить их, нужно разделить корневой тип элемента и тип дочернего элемента, открыв `TreeItem` как обобщенный тип, где мы можем задать `Children` как подтип `TreeItem`.

Мы хотим избежать каких-либо циклических ссылок, поэтому нам нужно установить для элементов `Children` значение `BaseTreeItem` по умолчанию, чтобы мы могли использовать `TreeItem` как ограничение и для элементов `Children`, и для `attachToRoot`:

```
type BaseTreeItem = {
  id: string;
  children: BaseTreeItem[];
};

type TreeItem<Children extends TreeItem = BaseTreeItem> = {
  id: string;
  children: Children[];
  collapsed?: boolean;
};

function attachToRoot<T extends TreeItem>(children: T[]): TreeItem<T> {
  return {
    id: "root",
    children,
  };
}

const root = attachToRoot([
  {
    id: "child",
    children: [],
    collapsed: false,
    marked: true,
  },
]);
/*
root is TreeItem<{
  id: string;
  children: never[];
  collapsed: false;
  marked: boolean;
}>
*/
```

Опять же, мы пишем параметры, которые могут работать с подтипами, и рассматриваем наши входные параметры как собственные, вместо того чтобы делать предположения.

4.5. Создание новых типов объектов

Задача

В вашем приложении есть тип, связанный с моделью. Каждый раз при изменении модели необходимо менять и типы.

Решение

Используйте обобщенные сопоставленные типы для создания новых типов объектов на основе исходного типа.

Обсуждение

Вернемся к магазину игрушек из рецепта 3.1. Благодаря типам объединения, типам пересечения и размеченным типам объединения мы смогли довольно хорошо смоделировать наши данные:

```
type ToyBase = {
  name: string;
  description: string;
  minimumAge: number;
};

type BoardGame = ToyBase & {
  kind: "boardgame";
  players: number;
};

type Puzzle = ToyBase & {
  kind: "puzzle";
  pieces: number;
};

type Doll = ToyBase & {
  kind: "doll";
  material: "plush" | "plastic";
};

type Toy = Doll | Puzzle | BoardGame;
```

Где-то в коде нам нужно сгруппировать все игрушки из модели в структуру данных, которую можно описать с помощью типа `GroupedToys`. Он содержит свойство для каждой категории (или "kind") и массив `Toy` в качестве значения.

Функция `groupToys` принимает несортированный список игрушек и группирует их по виду:

```
type GroupedToys = {
  boardgame: Toy[];
  puzzle: Toy[];
  doll: Toy[];
};

function groupToys(toys: Toy[]): GroupedToys {
  const groups: GroupedToys = {
    boardgame: [],
    puzzle: [],
    doll: [],
  };
  for (let toy of toys) {
    groups[toy.kind].push(toy);
  }
  return groups;
}
```

В этом коде уже есть некоторые тонкости. Для начала, мы используем явную аннотацию типа при объявлении `groups`. Это позволяет нам быть уверенными в том, что мы не забудем ни одну категорию. Кроме того, ключи для `GroupedToys` совпадают с типом объединения "kind" в `Toy`, поэтому мы можем легко проиндексировать доступ к `groups` по `toy.kind`.

Со временем нам снова нужно слегка изменить модель. В магазине игрушек теперь продаются оригинальные или, может быть, альтернативные варианты игрушечных кубиков. Мы подключаем новый тип `Bricks` к нашей модели `Toy`:

```
type Bricks = ToyBase & {
  kind: "bricks",
  pieces: number;
  brand: string;
}

type Toy = Doll | Puzzle | BoardGame | Bricks;
```

Поскольку `groupToys` тоже приходится работать с `Bricks`, мы получаем подробное сообщение об ошибке, так как `GroupedToys` не имеет понятия о типе "bricks":

```
function groupToys(toys: Toy[]): GroupedToys {
  const groups: GroupedToys = {
    boardgame: [],
    puzzle: [],
    doll: [],
  };
  for (let toy of toys) {
    groups[toy.kind].push(toy);
  }
}
```



```
// ^- Element implicitly has an 'any' type because expression
//   of type '"boardgame" | "puzzle" | "doll" | "bricks"' can't
//   be used to index type 'GroupedToys'.
//   Property 'bricks' does not exist on type 'GroupedToys'.(7053)
}
return groups;
}
```

Это желательное поведение в TypeScript: знать, когда типы больше не совпадают. Такие случаи должны привлечь наше внимание. Обновим `GroupedToys` и `GroupToys`:

```
type GroupedToys = {
  boardgame: Toy[];
  puzzle: Toy[];
  doll: Toy[];
  bricks: Toy[];
};

function groupToys(toys: Toy[]): GroupedToys {
  const groups: GroupedToys = {
    boardgame: [],
    puzzle: [],
    doll: [],
    bricks: [],
  };
  for (let toy of toys) {
    groups[toy.kind].push(toy);
  }
  return groups;
}
```

Есть одна неприятная особенность: задача группировки игрушек всегда одна и та же. Независимо от того, насколько сильно изменится модель, мы всегда будем выбирать элемент по типу и помещать в массив. Нам нужно будет поддерживать `groups` при каждом изменении, но если мы изменим наше представление о группах, то сможем оптимизировать их с учетом изменений. Сначала мы изменим тип `GroupedToys`, чтобы он содержал необязательные свойства. Затем инициализируем каждую группу с помощью пустого массива, если инициализация еще не проводилась:

```
type GroupedToys = {
  boardgame?: Toy[];
  puzzle?: Toy[];
  doll?: Toy[];
  bricks?: Toy[];
};

function groupToys(toys: Toy[]): GroupedToys {
  const groups: GroupedToys = {};
  for (let toy of toys) {
    // Инициализировать, если нет значения
  }
}
```

```

    groups[toy.kind] = groups[toy.kind] ?? [];
    groups[toy.kind]?.push(toy);
  }
  return groups;
}

```

Нам больше не нужно поддерживать `groupToys`. Единственное, что нуждается в поддержке, — это тип `GroupedToys`. Если мы внимательно посмотрим на `GroupedToys`, то увидим, что он неявно связан с `Toy`. Каждый ключ свойства — часть `Toy["kind"]`. Сделаем эту связь *явной*. С помощью *сопоставленного типа* мы создадим новый тип объекта на основе каждого типа в `Toy["kind"]`.

Тип `Toy["kind"]` представляет собой объединение таких строковых литералов: `"boardgame" | "puzzle" | "doll" | "bricks"`. У нас очень ограниченный набор строк, поэтому каждый элемент этого объединения будет выступать ключом свойства. На минутку задумаемся: мы можем использовать *тип* в качестве *ключа свойства* вновь созданного типа. Каждое свойство имеет необязательный модификатор типа и указывает на `Toy[]`:

```

type GroupedToys = {
  [k in Toy["kind"]]?: Toy[];
};

```

Потрясающе! Каждый раз, меняя `Toy`, мы сразу же изменяем `Toy[]`. Наш код вообще не нуждается ни в каких изменениях; мы по-прежнему можем группировать по виду, как и раньше.

Этот паттерн мы можем обобщить. Создадим тип `Group`, который будет принимать коллекцию и группировать ее по определенному селектору. Мы хотим создать обобщенный тип с двумя параметрами типа:

- `Collection` может быть чем угодно;
- `Selector` — это ключ `Collection`, позволяющий создавать соответствующие свойства.

Первым делом мы попробовали бы взять то, что у нас есть в `GroupedToys`, и заменить конкретные типы параметрами типа. В результате то, что нам нужно, будет создано, но код приведет к ошибке:

```

// Как это работает
type GroupedToys = Group<Toy, "kind">;

type Group<Collection, Selector extends keyof Collection> = {
  [x in Collection[Selector]]?: Collection[];
  //   ^ Type 'Collection[Selector]' is not assignable
  //     to type 'string | number | symbol'.
  //   Type 'Collection[keyof Collection]' is not
  //     assignable to type 'string | number | symbol'.
  //   Type 'Collection[string] | Collection[number]

```

```
//      | Collection[symbol]' is not assignable to
//      type 'string | number | symbol'.
//      Type 'Collection[string]' is not assignable to
//      type 'string | number | symbol'.(2322)
};
```

TypeScript предупреждает нас, что `Collection[string] | Collection[number] | Collection[symbol]` может привести к чему угодно, а не только к тому, что можно использовать в качестве ключа. Это правда, и нам нужно подготовиться к такому повороту. У нас есть два варианта.

Во-первых, мы можем использовать ограничение типа на `Collection`, которое указывает на `Record<string, any>`. Тип `Record` — служебный; он генерирует новый объект, где первый параметр дает все ключи, а второй — типы:

```
// Этот тип встроенный!
type Record<K extends string | number | symbol, T> = { [P in K]: T; };
```

Такой код превращает `Collection` в объект с подстановочными знаками, фактически отключая проверку типов в `Groups`. Это нормально, ведь если какой-то тип ключа свойства окажется непригодным для использования, то TypeScript все равно отбросит его. Таким образом, последний тип `Group` имеет два параметра с ограниченным типом:

```
type Group<
  Collection extends Record<string, any>,
  Selector extends keyof Collection
> = {
  [x in Collection[Selector]]: Collection[];
};
```

Во-вторых, мы можем выполнить проверку для каждого ключа, чтобы определить, является ли он допустимым строковым ключом. Можно воспользоваться *условным типом*, чтобы проверить, действительно ли `Collection[Selector]` — допустимый тип для ключа. В противном случае мы удалили бы этот тип, выбрав `never`. Условные типы — нечто особенное, и мы подробно рассмотрим их в рецепте 5.4:

```
type Group<Collection, Selector extends keyof Collection> = {
  [k in Collection[Selector] extends string
    ? Collection[Selector]
    : never]?: Collection[];
};
```

Обратите внимание, что мы убрали модификатор необязательного типа. Это сделано потому, что в задачи группировки не входит превращение ключей в необязательные. Для этого есть `Partial<T>` — еще один отображаемый тип, который делает необязательным каждое свойство в типе объекта:

```
// Этот тип встроенный!
type Partial<T> = { [P in keyof T]?: T[P] };
```

Независимо от того, какой вспомогательный тип `Group` вы создадите, теперь вы можете создать объект `GroupedToys`, сообщив TypeScript, что вам нужен `Partial` объект (изменяя все свойства на необязательные) типа коллекции `Group` для типа `Toys`, сгруппированного по полю `"kind"`:

```
type GroupedToys = Partial<Group<Toy, "kind">>;
```

Теперь код читабелен.

4.6. Изменение объектов с помощью сигнатур утверждений

Задача

После того как определенная функция в вашем коде будет выполнена, вы знаете, что тип значения изменился.

Решение

Используйте сигнатуры утверждений для изменения типов независимо от операторов `if` и `switch`.

Обсуждение

JavaScript — очень гибкий язык. Его возможности динамической типизации позволяют изменять объекты во время выполнения, добавляя новые свойства по ходу работы. И разработчики используют это. Бывают ситуации, когда вы, например, просматриваете коллекцию элементов и вам нужно подтвердить определенные свойства. Затем вы сохраняете свойство `checked` и устанавливаете его в `true` просто чтобы знать, что вы прошли определенную проверку:

```
function check(person: any) {  
  person.checked = true;  
}
```

```
const person = {  
  name: "Stefan",  
  age: 27,  
};
```

```
check(person); // теперь у person есть проверенное свойство
```

```
person.checked; // это правда!
```

Вы хотите отразить это поведение в системе типов. В противном случае вам придется постоянно выполнять дополнительные проверки на наличие определенных свойств в объекте, даже если вы можете быть уверены в том, что они есть.

Один из способов утверждать, что определенные свойства существуют, — использовать утверждения типа. Мы говорим, что в определенный момент времени это свойство имеет другой тип:

```
(person as typeof person & { checked: boolean }).checked = true;
```

Хорошо, но вам придется использовать это утверждение типа снова и снова, поскольку оно не меняет исходный тип `person`. Другой способ подтвердить наличие определенных свойств — создать предикаты типа, подобные тем, которые были показаны в рецепте 3.5:

```
function check<T>(obj: T): obj is T & { checked: true } {  
  (obj as T & { checked: boolean }).checked = true;  
  return true;  
}
```

```
const person = {  
  name: "Stefan",  
  age: 27,  
};
```

```
if (check(person)) {  
  person.checked; // проверка успешна!  
}
```

Однако в данном случае ситуация несколько иная, из-за чего функция `check` выглядит негибкой: вам нужно выполнить проверку дополнительного условия и вернуть `true` в функции предиката. Это кажется неправильным.

К счастью, в TypeScript есть еще один метод, который можно использовать в подобных ситуациях: сигнатуры утверждений. Они позволяют изменять тип значения в потоке управления, не используя условия. Сигнатуры утверждений были смоделированы для функции `assert` в Node.js, которая принимает условие и выдает ошибку, если оно неверно. Это означает, что после вызова `assert` вы можете иметь больше информации, чем раньше. Например, если вы вызываете функцию `assert` и проверяете, имеет ли значение тип `string`, то знаете, что после ее выполнения значение должно быть `string`:

```
function assert(condition: any, msg?: string): asserts condition {  
  if (!condition) {  
    throw new Error(msg);  
  }  
}
```

```
function yell(str: any) {  
  assert(typeof str === "string");  
}
```

```
// str – это строка
return str.toUpperCase();
}
```

Пожалуйста, обратите внимание, что функция завершает работу, если условие ложно. Она выдает ошибку в случае `never`. Если же функция выполняется успешно, то вы действительно можете подтвердить условие.

Сигнатуры утверждений были смоделированы для Node.js функции `assert`, но вы можете утверждать любой тип по желанию. Например, у вас может быть функция, которая принимает любое значение для сложения, но вы утверждаете, что продолжение возможно, только если значения — типа `number`:

```
function assertNumber(val: any): asserts val is number {
  if (typeof val !== "number") {
    throw Error("value is not a number");
  }
}

function add(x: unknown, y: unknown): number {
  assertNumber(x); // x – это число
  assertNumber(y); // y – это число
  return x + y;
}
```

Все примеры, которые вы найдете по сигнатурам утверждений, основаны на утверждениях и коротких замыканиях (`shortcircuit`) в случае ошибок. Но мы можем воспользоваться тем же методом, чтобы сообщить TypeScript о наличии дополнительных свойств. Мы пишем функцию, которая очень похожа на функцию `check` в функции предиката, однако на этот раз нам не нужно возвращать `true`. Мы устанавливаем свойство, а поскольку в JavaScript объекты передаются по значению, то мы можем утверждать, что после вызова этой функции все, что мы передаем свойству `checked`, имеет значение `true`:

```
function check<T>(obj: T): asserts obj is T & { checked: true } {
  (obj as T & { checked: boolean }).checked = true;
}

const person = {
  name: "Stefan",
  age: 27,
};

check(person);
```

Благодаря этому можно изменять тип значения по ходу работы. Это малоизвестный метод, который может вам очень помочь.

4.7. Сопоставление типов с помощью карт типов

Задача

Вы пишете фабричную функцию, которая создает объект определенного подтипа на основе строкового идентификатора, и существует множество возможных подтипов.

Решение

Храните все подтипы в карте типов, расширьте ее с помощью индексного доступа и используйте сопоставленные типы, такие как `Partial<T>`.

Обсуждение

Фабричные функции позволяют создавать варианты сложных объектов на основе некой базовой информации. Один из сценариев, который вы, возможно, знаете по опыту работы с браузерным JavaScript, — это создание элементов. Функция `document.createElement` принимает имя тега элемента, и вы получаете объект, в котором можно изменять все необходимые свойства.

Вы хотите добавить в эту схему фабричную функцию `createElement`. Она не только принимает имя тега элемента, но и создает список свойств, чтобы вам не приходилось устанавливать каждое свойство по отдельности:

```
// Использование createElement

// a — это HTMLAnchorElement
const a = createElement("a", { href: "https://fettblog.eu" });
// b — это HTMLVideoElement
const b = createElement("video", { src: "/movie.mp4", autoplay: true });
// c — это HTMLElement
const c = createElement("my-element");
```

Вы хотите создать подходящие типы, поэтому вам нужно:

- убедиться, что вы создаете только допустимые HTML-элементы;
- указать тип, который принимает подмножество свойств HTML-элемента.

Сначала разберемся с допустимыми HTML-элементами. Существует около 140 возможных HTML-элементов, что очень много. У каждого из них есть имя тега, которое можно представить в виде строки, и соответствующий объект-прототип

в DOM. Используя библиотеку `dom` в вашем `tsconfig.json`, TypeScript имеет информацию об этих объектах-прототипах в виде типов. И вы можете выяснить все 140 имен элементов.

Хороший способ обеспечить соответствие между именами тегов элементов и объектами-прототипами — использовать *карту типов*. Это метод, при котором вы берете псевдоним типа или интерфейс и позволяете ключам указывать на соответствующие варианты типов. Затем вы можете получить правильный вариант типа, используя индексный доступ к строковому литеральному типу:

```
type AllElements = {
  a: HTMLAnchorElement;
  div: HTMLDivElement;
  video: HTMLVideoElement;
  // ... и еще более 140!
};

// HTMLAnchorElement
type A = AllElements["a"];
```

Это похоже на доступ к свойствам объекта JavaScript с помощью индексного доступа, но помните, что мы все еще работаем на уровне типов. Это значит, индексный доступ может быть широким:

```
type AllElements = {
  a: HTMLAnchorElement;
  div: HTMLDivElement;
  video: HTMLVideoElement;
  // ... и еще более 140!
};

// HTMLAnchorElement | HTMLDivElement
type AandDiv = AllElements["a" | "div"];
```

Используем эту карту, чтобы определить тип функции `createElement`. Мы применяем параметр обобщенного типа, ограниченный всеми ключами `AllElements`, что позволяет нам передавать только допустимые HTML-элементы:

```
function createElement<T extends keyof AllElements>(tag: T): AllElements[T] {
  return document.createElement(tag as string) as AllElements[T];
}

// a — это HTMLAnchorElement
const a = createElement("a");
```

Используйте здесь обобщенные типы, чтобы привязать строковый литерал к литеральному типу, который можно использовать для индексации нужного варианта HTML-элемента из карты типов. Обратите внимание еще и на то, что при исполь-

зовании `document.createElement` требует двух утверждений типа. Одно делает набор более широким (от `T` до `string`), а другое — более узким (от `HTMLElement` до `AllElements[T]`). Оба утверждения указывают на то, что вам придется иметь дело с API, находящимся вне вашего контроля, как было установлено в рецепте 3.9. Мы рассмотрим эти утверждения позже.

Теперь мы хотим предоставить возможность передавать дополнительные свойства для указанных HTML-элементов, устанавливая `href` для `HTMLAnchorElement` и т. д. Все свойства уже есть в соответствующих вариантах `HTMLElement`, но они обязательны, а нам нужны необязательные. Мы можем сделать все свойства необязательными с помощью встроеного типа `Partial<T>`. Это сопоставленный тип, который принимает все свойства определенного типа и добавляет модификатор типа:

```
type Partial<T> = { [P in keyof T]? T[P] };
```

Мы расширяем функцию с помощью необязательного аргумента `props`, который является `Partial` индексированного элемента из `AllElements`. Таким образом, мы знаем, что если передадим "a", то сможем установить только те свойства, которые доступны в `HTMLAnchorElement`:

```
function createElement<T extends keyof AllElements>(
  tag: T,
  props?: Partial<AllElements[T]>
): AllElements[T] {
  const elem = document.createElement(tag as string) as AllElements[T];
  return Object.assign(elem, props);
}

const a = createElement("a", { href: "https://fettblog.eu" });
const x = createElement("a", { src: "https://fettblog.eu" });
//
//           ^--
// Argument of type '{ src: string; }' is not assignable to parameter
// of type 'Partial<HTMLAnchorElement>'.
// Object literal may only specify known properties, and 'src' does not
// exist in type 'Partial<HTMLAnchorElement>'.(2345)
```

Потрясающе! Теперь вам предстоит разобраться во всех 140 HTML-элементах.

Или нет. Кто-то уже проделал эту работу и поместил `HTMLElementTagNameMap` в `lib.dom.ts`. Так что воспользуемся этой возможностью:

```
function createElement<T extends keyof HTMLElementTagNameMap>(
  tag: T,
  props?: Partial<HTMLElementTagNameMap[T]>
): HTMLElementTagNameMap[T] {
  const elem = document.createElement(tag);
  return Object.assign(elem, props);
}
```

Этот же интерфейс используется в `document.createElement`, поэтому нет никаких противоречий между вашей фабричной функцией и встроенной. Дополнительные утверждения не требуются.

Есть один нюанс. Вы можете использовать только 140 элементов, предоставляемых `HTMLElementTagNameMap`. А что будет, если вы хотите создавать элементы SVG или веб-компоненты, которые могут иметь полностью настраиваемые имена элементов? Ваша фабричная функция вдруг становится слишком ограниченной.

Чтобы обеспечить больше возможностей — как это делает `document.createElement`, — вам придется снова добавить все возможные строки в набор.

`HTMLElementTagNameMap` — это интерфейс. Таким образом, вы можете использовать *объединение деклараций*, чтобы расширить интерфейс с помощью *индексированной сигнатуры*, в которой все оставшиеся строки будут сопоставлены с `HTMLUnknownElement`:

```
interface HTMLElementTagNameMap {
  [x: string]: HTMLUnknownElement;
};

function createElement<T extends keyof HTMLElementTagNameMap>(
  tag: T,
  props?: Partial<HTMLElementTagNameMap[T]>
): HTMLElementTagNameMap[T] {
  const elem = document.createElement(tag);
  return Object.assign(elem, props);
}

// a — это HTMLAnchorElement
const a = createElement("a", { href: "https://fettblog.eu" });
// b — это HTMLUnknownElement
const b = createElement("my-element");
```

Теперь вы можете:

- использовать отличную фабричную функцию для создания типизированных HTML-элементов;
- задавать свойства элементов с помощью всего одного объекта конфигурации;
- гибко создавать большее количество элементов, чем определено.

Последний пункт замечателен, но что, если вы хотите разрешить только веб-компоненты? По поводу них есть соглашение: они должны иметь *тип* в имени тега. Вы можете смоделировать это, используя сопоставленный тип с *типом литералов шаблонов строк*. О таких типах мы поговорим в главе 6.

На данный момент вам нужно знать лишь то, что вы создаете набор строк, в котором шаблоном является *любая строка*, за которой следует *тире*, а за ним — *любая строка*. Этого достаточно, чтобы гарантировать, что вы передаете только правильные имена элементов.

Сопоставленные типы работают лишь с псевдонимами типов, а не с объявлениями интерфейсов, поэтому нам нужно снова определить тип `AllElements`:

```
type AllElements = HTMLElementTagNameMap &
  {
    [x in `${string}-${string}`]: HTMLElement;
  };

function createElement<T extends keyof AllElements>(
  tag: T,
  props?: Partial<AllElements[T]>
): AllElements[T] {
  const elem = document.createElement(tag as string) as AllElements[T];
  return Object.assign(elem, props);
}

const a = createElement("a", { href: "https://fettblog.eu" }); // OK
const b = createElement("my-element"); // OK

const c = createElement("thisWillError");
//           ^
// Argument of type '"thisWillError"' is not
// assignable to parameter of type '`${string}-${string}`
// | keyof HTMLElementTagNameMap'.(2345)
```

Великолепно. Благодаря типу `AllElements` мы получаем обратно утверждения типов, которые нам не очень нравятся. В этом случае вместо утверждения мы можем использовать перегрузку функции, определив два объявления: одно для пользователей, а другое для нас, чтобы реализовать функцию. Больше информации об этом методе перегрузки функций вы можете узнать в рецептах 2.6 и 12.7:

```
function createElement<T extends keyof AllElements>(
  tag: T,
  props?: Partial<AllElements[T]>
): AllElements[T];
function createElement(tag: string, props?: Partial<HTMLElement>): HTMLElement {
  const elem = document.createElement(tag);
  return Object.assign(elem, props);
}
```

Все готово. Мы определили карту типов с сопоставленными типами и сигнатурами индексов, используя параметры обобщенных типов, чтобы четко обозначить наши намерения. Отличное сочетание нескольких элементов инструментария TypeScript.

4.8. Использование `ThisType` для определения `this` в объектах

Задача

Вашему приложению требуются сложные объекты конфигурации с методами, где `this` имеет разный контекст в зависимости от использования.

Решение

Используйте встроенный обобщенный тип `ThisType<T>`, чтобы определить правильное значение `this`.

Обсуждение

Такие фреймворки, как VueJS (<https://vuejs.org/>), довольно активно используют фабричные функции, которым вы передаете комплексный объект конфигурации в целях определения начальных данных, вычисляемых свойств и методов для каждого экземпляра.

Вы хотите, чтобы компоненты вашего приложения вели себя аналогичным образом. Идея состоит в том, чтобы предоставить объект конфигурации с тремя свойствами.

Функция data

Возвращаемое значение — это начальные данные экземпляра. Вы не должны иметь доступ к каким-либо другим свойствам объекта конфигурации в этой функции.

Свойство computed

Относится к вычисляемым свойствам, которые основаны на начальных данных. Вычисляемые свойства объявляются с помощью функций. Они могут получить доступ к начальным данным так же, как к обычным свойствам.

Свойство methods

Методы могут быть вызваны и могут обращаться к вычисляемым свойствам, а также к начальным данным. К вычисляемым свойствам методы обращаются так же, как и к обычным: нет необходимости вызывать функцию.

Если посмотреть на используемый объект конфигурации, то `this` можно интерпретировать тремя разными способами. В функции `data` у `this` вообще нет никаких свойств. В свойстве `computed` каждая функция может получить доступ к возвращаемому значению `data` через `this`, как если бы оно было частью ее объекта.

В свойстве `methods` каждый метод может получить доступ к свойствам `computed` и `data` с помощью `this` таким же образом:

```
const instance = create({
  data() {
    return {
      firstName: "Stefan",
      lastName: "Baumgartner",
    };
  },
  computed: {
    fullName() {
      // имеет доступ к возвращаемому объекту данных
      return this.firstName + " " + this.lastName;
    },
  },
  methods: {
    hi() {
      // используйте вычисляемые свойства так же, как и обычные
      alert(this.fullName.toLowerCase());
    },
  },
});
```

Такое поведение является особенным, но не редким. И в его случае мы определенно хотим использовать хорошие типы.



В текущем уроке мы сосредоточимся только на типах, а не на фактической реализации, так как эта тема выходит за рамки данной главы.

Создадим типы для каждого свойства. Определим тип `Options`, который будем постепенно совершенствовать. Начнем с функции `data`. Ее может определить пользователь, поэтому мы хотим задать эту функцию с помощью параметра обобщенного типа. Данные, которые мы ожидаем, определяются типом возвращаемого значения функции `data`:

```
type Options<Data> = {
  data(this: {})? : Data;
};
```

Поэтому, как только мы укажем фактическое возвращаемое значение в функции `data`, заполнитель `Data` будет заменен типом реального объекта. Обратите также внимание, что мы определяем `this`, указывающий на пустой объект, и, следовательно, не получим доступа ни к какому другому свойству объекта конфигурации.

Далее мы определяем `computed`, который является объектом функций. Мы добавляем еще один параметр обобщенного типа под названием `Computed` и позволяем его значению быть типизированным путем использования. Здесь `this` изменяется

на все свойства `Data`. Мы не можем установить `this`, как это делается в функции `data`, поэтому можем использовать встроенный вспомогательный тип `ThisType` и установить для него параметр обобщенного типа `Data`:

```
type Options<Data, Computed> = {
  data(this: {})? : Data;
  computed?: Computed & ThisType<Data>;
};
```

Это позволяет нам получить доступ, например, к `this.firstName`, как в предыдущем примере.

И наконец, мы хотим указать `methods`. Это свойство снова является особенным, поскольку с его помощью мы получаем доступ не только к `Data` через `this`, но и ко всем методам и вычисляемым свойствам.

`Computed` хранит все вычисляемые свойства в виде функций. Однако нам потребуется их возвращаемое значение. Если мы получаем доступ к `fullName` через доступ к свойству, то ожидаем, что это будет `string`.

Для этого мы создаем вспомогательный тип `MapFnToProps`, который принимает тип, являющийся объектом функций, и сопоставляет его с типами возвращаемых значений. Встроенный вспомогательный тип `ReturnType` идеально подходит для этого сценария:

```
// Объект, состоящий из функций...
type FnObj = Record<string, () => any>;

// ... с объектом возвращаемых типов
type MapFnToProps<FunctionObj extends FnObj> = {
  [K in keyof FunctionObj]: ReturnType<FunctionObj[K]>;
};
```

Мы можем использовать `MapFnToProps`, чтобы установить `ThisType` для недавно добавленного параметра обобщенного типа под названием `Methods`. Мы также добавляем в набор `Data` и `Methods`. Чтобы параметр обобщенного типа `Computed` можно было передать в `MapFnToProps`, он должен быть ограничен значением `FnObj`, тем же ограничением, что и первый параметр `FunctionObj` в `MapFnToProps`:

```
type Options<Data, Computed extends FnObj, Methods> = {
  data(this: {})? : Data;
  computed?: Computed & ThisType<Data>;
  methods?: Methods & ThisType<Data & MapFnToProps<Computed> & Methods>;
};
```

И это тип! Мы берем все свойства обобщенного типа и добавляем их в фабричную функцию `create`:

```
declare function create<Data, Computed extends FnObj, Methods>(
  options: Options<Data, Computed, Methods>
): any;
```

При использовании будут заменены все параметры обобщенного типа. А благодаря тому, как типизированы Options, мы получаем все необходимые возможности автозаполнения, позволяющие гарантировать, что у нас не возникнут проблемы (рис. 4.1).

```

96 create({
97   data() {
98     // @ts-expect-error
99     this.firstName;
100    // @ts-expect-error
101    this.getRandom();
102    // @ts-expect-error
103    this.data();
104
105    return {
106      firstName: "Stefan",
107      lastName: "Baumgartner",
108      age: 40,
109    };
110  },
111  computed: {
112    fullName() {
113      return `${this.firstName} ${this.lastName}`;
114    },
115  },
116  methods: {
117    getRandom() {
118      return Math.random();
119    },
120    hi() {
121      alert(this.lastName);
122      alert(this.fullName);
123      alert(this.getRandom());
124    },
125    test() {
126      console.log(this.);
127    },
128  },
129 });
130

```

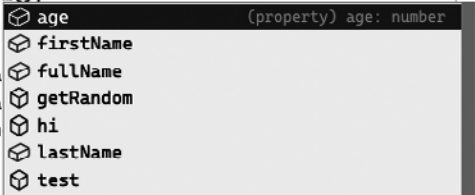


Рис. 4.1. Настройка методов в заводской функции, имеющей полный доступ к нужным свойствам

Этот пример прекрасно показывает, как с помощью TypeScript можно типизировать сложные API, в которых происходит множество взаимодействий с объектами¹.

¹ Отдельное спасибо создателям Type Challenges (<https://oreil.ly/pHc9j>) за этот прекрасный пример.

4.9. Добавление контекста `const` к параметрам обобщенного типа

Задача

Когда вы передаете в функцию сложные литеральные значения, TypeScript расширяет тип до более общего. Хотя во многих случаях это желательное поведение, есть ситуации, когда лучше работать с литеральными типами, а не с расширенным типом.

Решение

Добавьте модификатор `const` перед параметром обобщенного типа, чтобы сохранить переданные значения в контексте `const`.

Обсуждение

Фреймворки одностраничных приложений (single-page application, SPA), как правило, переопределяют многие функциональные возможности браузера на JavaScript. Например, такие функции, как History API (<https://oreil.ly/КМВgј>), позволили переопределить обычное поведение навигации, которое SPA-фреймворки используют для переключения между страницами без реальной перезагрузки страницы, путем замены содержимого страницы и изменения URL в браузере.

Представьте, что работаете над минималистичным SPA-фреймворком, который использует так называемый маршрутизатор для перемещения по страницам. Страницы определяются как компоненты, а интерфейс `ComponentConstructor` знает, как создавать и отображать новые элементы на вашем сайте:

```
interface ComponentConstructor {  
  new(): Component;  
}
```

```
interface Component {  
  render(): HTMLElement;  
}
```

Маршрутизатор должен получать список компонентов и связанных с ними путей, хранящихся в виде `string`. Если маршрутизатор создается с помощью функции `router`, то она должна возвращать объект, позволяющий перемещаться с помощью функции `navigate` по нужному пути:

```
type Route = {  
  path: string;
```



```
    component: ComponentConstructor;
};

function router(routes: Route[]) {
    return {
        navigate(path: string) {
            // ...
        },
    };
}
```

То, как реализована навигация, нас сейчас не интересует. Вместо этого мы хотим сосредоточиться на типах интерфейса функций.

Маршрутизатор работает как предполагалось: принимает массив объектов `Route` и возвращает объект с функцией `navigate`, которая позволяет запустить переход с одного URL на другой и отображает новый компонент:

```
const rtr = router([
    {
        path: "/",
        component: Main,
    },
    {
        path: "/about",
        component: About,
    },
])

rtr.navigate("/faq");
```

Сразу заметно, что типы слишком широки. Если мы разрешим навигацию по каждой доступной `string`, то сможем беспрепятственно использовать фиктивные маршруты, которые никуда не приведут. Нам нужно будет реализовать какую-то обработку ошибок для информации, которая уже готова и доступна. Так почему бы не использовать ее?

Наша первая идея — заменить конкретный тип параметром обобщенного типа. TypeScript работает с обобщенными подстановками так: если у нас есть литеральный тип, то TypeScript соответствующим образом создаст подтип. Введение `T` для `Route` и использование `T["path"]` вместо `string` приближается к тому, чего мы хотим добиться:

```
function router<T extends Route>(routes: T[]) {
    return {
        navigate(path: T["path"]) {
            // ...
        },
    };
}
```

Теоретически это должно сработать. Если мы вспомним, что TypeScript в этом случае делает с литеральными типами примитивов, то будем ждать, что значение окажется сужено до литерального типа:

```
function getPath<T extends string>(route: T): T {
    return route;
}

const path = getPath("/"); // "/"
```

Подробнее об этом вы можете прочитать в рецепте 4.3. Есть одна важная деталь: `path` в предыдущем примере находится в контексте `const`, поскольку возвращаемое значение является неизменяемым.

Единственная проблема состоит в том, что мы работаем с объектами и массивами, а TypeScript имеет тенденцию расширять типы объектов и массивов до чего-то более общего, чтобы обеспечить возможность изменения значений. Если мы посмотрим на новый пример, который является аналогичным, но имеет вложенный объект, то увидим, что TypeScript вместо этого принимает более широкий тип:

```
type Routes = {
    paths: string[];
};

function getPaths<T extends Routes>(routes: T): T["paths"] {
    return routes.paths;
}

const paths = getPaths({ paths: ["/", "/about"] }); // string[]
```

В случае объектов контекст `const` для `paths` предназначен только для привязки переменной, а не для ее содержимого. В итоге это приводит к потере части информации, с помощью которой указывается правильный тип `navigate`.

Способ обойти это ограничение — вручную применить контекст `const`; для этого нужно переопределить входной параметр, чтобы он был доступен только как `readonly`:

```
function router<T extends Route>(routes: readonly T[]) {
    return {
        navigate(path: T["path"]) {
            history.pushState({}, "", path);
        },
    };
}

const rtr = router([
    {
        path: "/",
```

```
    component: Main,
  },
  {
    path: "/about",
    component: About,
  },
] as const);

rtr.navigate("/about");
```

Этот код работает, но требует, чтобы при его написании мы не забывали об очень важной детали. А активное запоминание обходных путей — это всегда путь к катастрофе.

К счастью, TypeScript позволяет нам запрашивать контекст `const` из параметров обобщенного типа. Вместо того чтобы применять его к значению, мы заменяем этот параметр конкретным значением, *но* в контексте `const`, добавляя модификатор `const` к параметру обобщенного типа:

```
function router<const T extends Route>(routes: T[]) {
  return {
    navigate(path: T["path"]) {
      // будет определено
    },
  };
}
```

После этого мы можем использовать наш маршрутизатор так, как привыкли, и даже получить возможность автозаполнения для возможных путей:

```
const rtr = router([
  {
    path: "/",
    component: Main,
  },
  {
    path: "/about",
    component: About,
  },
])

rtr.navigate("/about");
```

Еще лучше то, что мы получаем правильные ошибки, когда передаем что-то поддельное:

```
const rtr = router([
  {
    path: "/",
    component: Main,
  },
```

```
{
  path: "/about",
  component: About,
},
])

rtr.navigate("/faq");
//           ^
// Argument of type '"/faq"' is not assignable to
// parameter of type '"/" | "/about"'.(2345)
```

Самое прекрасное здесь то, что все это скрыто в API функции. Ожидаемое поведение становится более понятным, интерфейс сообщает об ограничениях, и нам не нужно делать ничего лишнего при использовании `router` в целях обеспечения безопасности типов.

Условные типы

В этой главе мы подробно рассмотрим уникальную функциональность TypeScript: *условные типы*. Они позволяют выбирать типы на основе проверок подтипов, что дает возможность перемещаться в пространстве типов и делает проектирование интерфейсов и сигнатур функций еще более гибким.

Условные типы — мощный инструмент, который позволяет создавать типы в ходе работы. Это делает систему типов TypeScript полной по Тьюрингу, как показано в обсуждении на GitHub (<https://oreil.ly/igPhB>), что одновременно и замечательно, и немного пугающе. Имея столько возможностей, легко перестать понимать, какие типы действительно нужны. Это может запутать вас или побудить создать слишком сложные для восприятия типы. На протяжении всей книги мы будем подробно обсуждать использование условных типов, постоянно проверяя, действительно ли то, что мы делаем, приводит к желаемой цели.

Обратите внимание, что данная глава намного короче остальных. Это не потому, что об условных типах особо нечего сказать. Совсем наоборот. Скорее это потому, что мы увидим примеры хорошего применения условных типов в последующих главах. А в текущей мы сосредоточимся на основных принципах и определимся с терминологией, которую вы сможете использовать и на которую будете ссылаться всякий раз, когда вам понадобятся возможности, предоставляемые типами.

5.1. Управление сигнатурами сложных функций

Задача

Вы создаете функцию с различными параметрами и типами возвращаемых значений. Управлять всеми вариантами с помощью перегрузок функций все тяжелее.

Решение

Используйте условные типы, чтобы определить набор правил для типов параметров и возвращаемых значений.

Обсуждение

Вы создаете программное обеспечение, которое представляет определенные атрибуты в виде меток на основе введенных пользователем данных. Вы различаете `StringLabel` и `NumberLabel`, чтобы сделать возможными разные виды операций фильтрации и поиска:

```
type StringLabel = {
  name: string;
};

type NumberLabel = {
  id: number;
};
```

Пользователь вводит либо строку, либо число. Функция `createLabel` принимает вводимые данные как примитивный тип и создает объект `StringLabel` или `NumberLabel`:

```
function createLabel(input: number | string): NumberLabel | StringLabel {
  if (typeof input === "number") {
    return { id: input };
  } else {
    return { name: input };
  }
}
```

После того как базовая функциональность будет реализована, вы увидите, что ваши типы слишком широки. Если ввести `number`, то тип возвращаемого значения `createLabel` по-прежнему будет `NumberLabel | StringLabel`, тогда как это может быть только `NumberLabel`. Как можно решить эту проблему? Добавив перегрузки функций для явного определения взаимосвязей типов, как мы делали в рецепте 2.6:

```
function createLabel(input: number): NumberLabel;
function createLabel(input: string): StringLabel;
function createLabel(input: number | string): NumberLabel | StringLabel {
  if (typeof input === "number") {
    return { id: input };
  } else {
    return { name: input };
  }
}
```

Перегрузки функций работают следующим образом: сами перегрузки определяют типы для использования, а последнее объявление функции — типы для реализации тела функции. С помощью `createLabel` мы можем передать `string` и получить `StringLabel` или передать `number` и получить `NumberLabel`, поскольку это типы, доступные извне.

Это проблематично в случаях, когда мы не можем заранее сузить тип ввода. Нам не хватает внешнего типа функции, который позволял бы передавать на вход либо `number`, либо `string`:

```
function inputToLabel(input: string | number) {
  return createLabel(input);
  //           ^
  // No overload matches this call. (2769)
}
```

Чтобы обойти это ограничение, можно добавить еще одну перегрузку, которая отражает сигнатуру функции реализации для очень широкого круга типов входных данных:

```
function createLabel(input: number): NumberLabel;
function createLabel(input: string): StringLabel;
function createLabel(input: number | string): NumberLabel | StringLabel;
function createLabel(input: number | string): NumberLabel | StringLabel {
  if (typeof input === "number") {
    return { id: input };
  } else {
    return { name: input };
  }
}
```

Здесь мы видим, что описание базового поведения этой функциональности требует трех перегрузок и четырех объявлений сигнатур функций. И с этого момента ситуация лишь ухудшается.

Мы хотим расширить нашу функцию, чтобы она имела возможность копировать существующие объекты `StringLabel` и `NumberLabel`. В итоге это означает больше перегрузок:

```
function createLabel(input: number): NumberLabel;
function createLabel(input: string): StringLabel;
function createLabel(input: StringLabel): StringLabel;
function createLabel(input: NumberLabel): NumberLabel;
function createLabel(input: string | StringLabel): StringLabel;
function createLabel(input: number | NumberLabel): NumberLabel;
function createLabel(
  input: number | string | StringLabel | NumberLabel
): NumberLabel | StringLabel;
function createLabel(
  input: number | string | StringLabel | NumberLabel
): NumberLabel | StringLabel {
  if (typeof input === "number") {
    return { id: input };
  } else if (typeof input === "string") {
    return { name: input };
  } else if ("id" in input) {
    return { id: input.id };
  }
}
```

```

    } else {
      return { name: input.name };
    }
  }
}

```

По правде говоря, в зависимости от того, насколько выразительными мы хотим видеть наши подсказки типов, можно написать не только меньше, но и гораздо больше перегрузок функций. Проблема по-прежнему очевидна: большее разнообразие приводит к более сложным сигнатурам функций.

В подобных ситуациях может помочь один из инструментов в арсенале TypeScript: условные типы. Они позволяют выбирать тип на основе определенных проверок подтипов. Мы спрашиваем, принадлежит ли параметр обобщенного типа к определенному подтипу, и если да, возвращаем тип из ветви `true`; в противном случае возвращаем тип из ветви `false`.

Например, следующий тип возвращает входной параметр, если `T` является подтипом `string` (что означает все строки или очень конкретные). В противном случае он возвращает `never`:

```
type IsString<T> = T extends string ? T : never;
```

```

type A = IsString<string>; // строка
type B = IsString<"hello" | "world">; // строка
type C = IsString<1000>; // тип never

```

TypeScript заимствует этот синтаксис из тернарного оператора JavaScript. Как и в случае с тернарным оператором JavaScript, он проверяет выполнение определенных условий. Но вместо типичного набора условий, знакомого вам по языку программирования, система типов TypeScript проверяет только то, входят ли значения входного типа в набор значений, который подвергается проверке.

С помощью этого инструмента можно написать условный тип `GetLabel<T>`. Мы проверяем, являются ли входные данные `string` или `StringLabel`. Если да, то мы возвращаем `StringLabel`; в противном случае мы знаем, что это должна быть `NumberLabel`:

```
type GetLabel<T> = T extends string | StringLabel ? StringLabel : NumberLabel;
```

Этот тип проверяет только то, являются ли входные данные `string`, `StringLabel`, `number` и `NumberLabel` в ветви `else`. Если мы хотим подстраховаться, то добавим проверку возможных входных данных, которые создают `NumberLabel`, путем вложения условных типов:

```

type GetLabel<T> = T extends string | StringLabel
  ? StringLabel
  : T extends number | NumberLabel
  ? NumberLabel
  : never;

```


Теперь пришло время подключить наши обобщенные типы. Мы добавляем в `createLabel` новый параметр обобщенного типа `T`, который ограничен всеми возможными входными типами. Этот параметр `T` служит входными данными для `GetLabel<T>`, где будет создавать соответствующий тип возвращаемого значения:

```
function createLabel<T extends number | string | StringLabel | NumberLabel>(
  input: T
): GetLabel<T> {
  if (typeof input === "number") {
    return { id: input } as GetLabel<T>;
  } else if (typeof input === "string") {
    return { name: input } as GetLabel<T>;
  } else if ("id" in input) {
    return { id: input.id } as GetLabel<T>;
  } else {
    return { name: input.name } as GetLabel<T>;
  }
}
```

Теперь мы готовы обрабатывать все возможные комбинации типов и по-прежнему будем получать правильный тип возвращаемого значения из `getLabel` — и все это в одной строке кода.

Вы можете увидеть, что нам пришлось обойти проверку типа возвращаемого значения. К сожалению, TypeScript не способен правильно анализировать поток управления при работе с обобщенными и условными типами. Небольшое утверждение типа сообщает TypeScript, что мы имеем дело с правильным типом возвращаемого значения.

Другим обходным решением будет рассматривать сигнатуру функции с условными типами как перегрузку исходной, широко типизированной функции:

```
function createLabel<T extends number | string | StringLabel | NumberLabel>(
  input: T
): GetLabel<T>;
function createLabel(
  input: number | string | StringLabel | NumberLabel
): NumberLabel | StringLabel {
  if (typeof input === "number") {
    return { id: input };
  } else if (typeof input === "string") {
    return { name: input };
  } else if ("id" in input) {
    return { id: input.id };
  } else {
    return { name: input.name };
  }
}
```

Таким образом, у нас есть гибкий тип для внешнего мира, благодаря которому мы точно понимаем, что получим на выходе, основываясь на наших входных данных. А для реализации доступны гибкие возможности, которые вы знаете по широкому набору типов.

Означает ли это, что во всех сценариях следует отдавать предпочтение условным типам, а не перегрузкам функций? Не обязательно. В рецепте 12.7 мы рассмотрим ситуации, в которых лучше выбрать перегрузку функций.

5.2. Фильтрация с помощью типа `never`

Задача

У вас есть объединение различных типов, но вы просто хотите получить все под-типы `string`.

Решение

Используйте распределительный условный тип, чтобы отфильтровать нужный тип.

Обсуждение

Допустим, в вашем приложении есть унаследованный код, в котором вы пытались воссоздать фреймворки наподобие *jQuery*. У вас есть собственный тип `ElementList`, который содержит вспомогательные функции, служащие для добавления и удаления имен классов к объектам типа `HTMLElement` или для привязки слушателей событий к самим событиям.

Кроме того, вы можете получить доступ к каждому элементу вашего списка с помощью индексного доступа. Тип для такого `ElementList` можно описать, используя индексный тип доступа для доступа по числовому индексу, а также обычные ключи свойств `string`:

```
type ElementList = {
  addClass: (className: string) => ElementList;
  removeClass: (className: string) => ElementList;
  on: (event: string, callback: (ev: Event) => void) => ElementList;
  length: number;
  [x: number]: HTMLElement;
};
```

Эта структура данных была разработана так, чтобы сделать возможным гибкий интерфейс. Это означает, что при вызове таких методов, как `addClass` или

`removeClass`, вы получите обратно тот же объект, поэтому можете выстраивать цепочки вызовов методов.

Пример реализации этих методов может выглядеть следующим образом:

```
// начало фрагмента
addClass: function (className: string): ElementList {
  for (let i = 0; i < this.length; i++) {
    this[i].classList.add(className);
  }
  return this;
},
removeClass: function (className: string): ElementList {
  for (let i = 0; i < this.length; i++) {
    this[i].classList.remove(className);
  }
  return this;
},
on: function (event: string, callback: (ev: Event) => void): ElementList {
  for (let i = 0; i < this.length; i++) {
    this[i].addEventListener(event, callback);
  }
  return this;
},
// конец фрагмента
```

В качестве расширения встроенных коллекций, таких как `Array` или `NodeList`, изменение параметров в наборе объектов `HTMLElement` становится действительно удобным:

```
declare const myCollection: ElementList;

myCollection
  .addClass("toggle-off")
  .removeClass("toggle-on")
  .on("click", (e) => {});
```

Допустим, вам нужно выполнять сопровождение вашего заменителя `jQuery`, и вы обнаружили, что прямой доступ к элементам оказался несколько небезопасным. Когда части вашего приложения могут изменять что-то напрямую, вам становится сложнее понять, откуда берутся изменения, если не из вашей тщательно разработанной структуры данных `ElementList`:

```
myCollection[1].classList.toggle("toggle-on");
```

Вы не можете изменить исходный код библиотеки (от него зависит слишком много отделов), поэтому решаете обернуть исходный `ElementList` в `Proxy`.

Объекты `Proxy` содержат исходный целевой объект и объект-обработчик, который определяет, как обрабатывать доступ. В следующей реализации показан `Proxy`,

который разрешает доступ только для чтения и лишь в том случае, если ключ свойства имеет тип `string`, а не строку, которая является строковым представлением числа:

```
const safeAccessCollection = new Proxy(myCollection, {
  get(target, property) {
    if (
      typeof property === "string" &&
      property in target &&
      "" + parseInt(property) !== property
    ) {
      return target[property as keyof typeof target];
    }
    return undefined;
  },
});
```



Объекты-обработчики в объектах `Proxy` получают только свойства типа `string` или `symbol`. Если вы осуществляете доступ к индексу с помощью числа, например `0`, то JavaScript преобразует его в строку «`0`».

Описанная выше схема прекрасно работает в JavaScript, но наши типы больше не совпадают. Тип возвращаемого значения конструктора `Proxy` снова является `ElementList`, то есть доступ к числовому индексу сохраняется:

```
// Работает в TypeScript, выдает ошибку в JavaScript
safeAccessCollection[0].classList.toggle("toggle-on");
```

Нам нужно сообщить TypeScript, что теперь мы имеем дело с объектом без доступа к числовому индексу, определив новый тип.

Рассмотрим ключи `ElementList`. Если мы используем оператор `keyof`, то получим тип объединения всех возможных методов доступа к объектам типа `ElementList`:

```
// разрешается в "addClass" | "removeClass" | "on" | "length" | number
type ElementListKeys = keyof ElementList;
```

Он содержит четыре строки, а также все возможные числа. Теперь, когда у нас есть это объединение, мы можем создать условный тип, который избавляется от всего, что не является строкой:

```
type JustStrings<T> = T extends string ? T : never;
```

`JustStrings<T>` — это то, что мы называем *распределительным условным типом*. В условии `T` существует сам по себе, а не заключен в объект или массив, поэтому TypeScript будет рассматривать условный тип объединения как объединение условных типов. По сути, TypeScript выполняет одну и ту же проверку условий для каждого члена объединения `T`.

В нашем случае она проходит через все элементы `keyof ElementList`:

```
type JustElementListStrings =
  | "addClass" extends string ? "addClass" : never
  | "removeClass" extends string ? "removeClass" : never
  | "on" extends string ? "on" : never
  | "length" extends string ? "length" : never
  | number extends string ? number : never;
```

Единственное условие, которое переходит в ветвь `false`, — последнее; в нем мы проверяем, является ли `number` подтипом `string`, но это не так. Если мы разрешим каждое условие, то в итоге получим новый тип объединения:

```
type JustElementListStrings =
  | "AddClass"
  | "removeClass"
  | "on"
  | "length"
  | never;
```

Объединение с `never` фактически отменяет `never`. Если у вас есть набор, в котором нет возможных значений, и вы присоединяете к нему набор значений, то значения остаются:

```
type JustElementListStrings =
  | "AddClass"
  | "removeClass"
  | "on"
  | "length";
```

Это именно тот список ключей, доступ к которому мы считаем безопасным! Используя вспомогательный тип `Pick`, мы можем создать тип, который фактически является супертипом `ElementList`. Для этого выберем все ключи, имеющие тип `string`:

```
тип SafeAccess = Pick<ElementList, JustStrings<keyof ElementList>>;
```

Если мы наведем на него курсор, то увидим, что полученный тип — это именно то, что нам нужно:

```
type SafeAccess = {
  addClass: (className: string) => ElementList;
  removeClass: (className: string) => ElementList;
  on: (event: string, callback: (ev: Event) => void) => ElementList;
  length: number;
};
```

Добавим данный тип в качестве аннотации к `safeAccessCollection`. Его можно присвоить супертипу, так что TypeScript с этого момента будет рассматривать `safeAccessCollection` как тип без доступа по числовому индексу:

```
const safeAccessCollection: Pick<
  ElementList,
  JustStrings<keyof ElementList>
```

```
> = new Proxy(myCollection, {
  get(target, property) {
    if (
      typeof property === "string" &&
      property in target &&
      "" + parseInt(property) !== property
    ) {
      return target[property as keyof typeof target];
    }
    return undefined;
  },
});
```

Теперь, когда мы попытаемся получить доступ к элементам из `safeAccessCollection`, TypeScript выдаст сообщение об ошибке:

```
safeAccessCollection[1].classList.toggle("toggle-on");
// ^ Element implicitly has an 'any' type because expression of
// type '1' can't be used to index type
// 'Pick<ElementList, "addClass" | "removeClass" | "on" | "length">'.
```

Это именно то, что нам нужно. Распределительные условные типы хороши тем, что мы меняем члены объединения. Другой пример мы увидим в рецепте 5.3, в котором поработаем со встроенными вспомогательными типами.

5.3. Группировка элементов по типу

Задача

Ваш тип `Group` из рецепта 4.5 работает нормально, но тип для каждой записи группы слишком широк.

Решение

Используйте вспомогательный тип `Extract`, чтобы выбрать правильный элемент из типа объединения.

Обсуждение

Вернемся к примеру с магазином игрушек из рецептов 3.1 и 4.5. Мы начали с тщательно продуманной модели, содержащей размеченные типы объединения, позволяющие получить точную информацию о каждом возможном значении:

```
type ToyBase = {
  name: string;
```

```

    description: string;
    minimumAge: number;
};

type BoardGame = ToyBase & {
    kind: "boardgame";
    players: number;
};

type Puzzle = ToyBase & {
    kind: "puzzle";
    pieces: number;
};

type Doll = ToyBase & {
    kind: "doll";
    material: "plush" | "plastic";
};

type Toy = Doll | Puzzle | BoardGame;

```

Затем мы нашли способ *вывести* (derive) из Toy другой тип GroupedToys, в котором используем элементы типа объединения свойства kind в качестве ключей свойств для сопоставленного типа, каждое свойство которого имеет тип Toy[]:

```

type GroupedToys = {
    [k in Toy["kind"]]: Toy[];
};

```

Благодаря обобщенным типам мы смогли определить вспомогательный тип Group<Collection, Selector>, позволяющий повторно использовать один и тот же шаблон в различных сценариях:

```

type Group<
    Collection extends Record<string, any>,
    Selector extends keyof Collection
> = {
    [K in Collection[Selector]]: Collection[];
};

type GroupedToys = Partial<Group<Toy, "kind">>;

```

Вспомогательный тип отлично работает, но есть один нюанс. Если мы наведем курсор на сгенерированный тип, то увидим, что, хотя Group<Collection, Selector> может правильно выбирать дискриминант типа объединения Toy, все свойства указывают на очень широкий Toy[]:

```

type GroupedToys = {
    boardgame?: Toy[] | undefined;
    puzzle? Toy[] | undefined;
    doll? Toy[] | undefined;
};

```

Но разве мы не должны знать больше? Например, почему `boardgame` указывает на `Toy[]`, когда единственным реалистичным типом должен быть `BoardGame[]`? То же самое касается головоломок `puzzle` и кукол `doll`, а также всех последующих игрушек, которые мы хотим добавить в свою коллекцию. Ожидаемый нами тип должен выглядеть примерно так:

```
type GroupedToys = {
  boardgame?: BoardGame[] | undefined;
  puzzle?: Puzzle[] | undefined;
  doll?: Doll[] | undefined;
};
```

Мы можем получить данный тип, *извлекая* соответствующий член из типа объединения `Collection`. К счастью, для этого существует вспомогательный тип: `Extract<T, U>`, где `T` — это коллекция, а `U` — часть `T`.

`Extract<T, U>` определяется так:

```
type Extract<T, U> = T extends U ? T : never;
```

Поскольку `T` в условии является голым типом (*naked*), то `T` — *распределительный условный тип*. Это значит, TypeScript проверяет, является ли каждый элемент `T` подтипом `U`, и если да, то сохраняет этот элемент в типе объединения. Как это будет работать в ситуации выбора нужной группы игрушек из `Toy`?

Допустим, мы хотим выбрать группу `Doll` из `Toy`. У нее есть несколько свойств, но `kind` явно выделяется. Таким образом, для типа, который будет искать только `Doll`, это будет означать, что мы извлечем *из Toy* каждый тип, в котором `{ kind: "doll" }`:

```
type ExtractedDoll = Extract<Toy, { kind: "doll" }>;
```

В случае распределительных условных типов условный тип объединения является объединением условных типов, поэтому каждый член `T` проверяется на соответствие `U`:

```
type ExtractedDoll =
  BoardGame extends { kind: "doll" } ? BoardGame : never |
  Puzzle extends { kind: "doll" } ? Puzzle : never |
  Doll extends { kind: "doll" } ? Doll : never;
```

И `BoardGame`, и `Puzzle` не являются подтипами `{ kind: "doll" }`, поэтому разрешаются в `never`. Но `Doll` *является* подтипом `{ kind: "doll" }`, поэтому разрешается в `Doll`:

```
type ExtractedDoll = never | never | Doll;
```

При объединении с `never` этот тип просто исчезает. Поэтому итоговый тип — `Doll`:

```
type ExtractedDoll = Doll;
```


Это именно то, что нам нужно. Добавим данную проверку в наш вспомогательный тип `Group`. К счастью, у нас есть все возможности для извлечения определенного типа из коллекции группы:

- сама `Collection`, заполнитель, который в конечном счете будет заменен на `Toy`;
- свойство `discriminant` в `Selector`, которое в итоге заменяется на `"kind"`;
- тип дискриминанта, который мы хотим извлечь, является строковым типом и по совпадению еще и ключом свойства, который мы отобразили в `Group`: `K`.

Таким образом, общая версия `Extract<Toy, { kind: "doll" }>` внутри `Group<Collection, Selector>` выглядит следующим образом:

```
type Group<
  Collection extends Record<string, any>,
  Selector extends keyof Collection
> = {
  [K in Collection[Selector]]: Extract<Collection, { [P in Selector]: K }>[];
};
```

Если мы заменим `Collection` на `Toy`, а `Selector` на `"kind"`, то тип будет выглядеть так:

```
[K in Collection[Selector]]
```

Используйте каждый член `Toy["kind"]` — в данном случае `"boardgame"`, `"puzzle"` и `"doll"` — в качестве ключа свойства для нового типа объекта.

```
Extract<Collection, ...>
```

Извлеките из `Collection` тип объединения `Toy`, каждый член которого является подтипом из...

```
{ [P in Selector]: K }
```

Посмотрите каждый член `Selector` — в нашем случае это просто `"kind"` — и создайте тип объекта, который будет указывать на `"boardgame"`, когда ключ свойства `"boardgame"`, на `"puzzle"`, когда ключ свойства `"puzzle"`, и т. д.

Вот как мы выбираем для каждого ключа свойства нужный член `Toy`. Результат соответствует ожиданиям:

```
type GroupedToys = Partial<Group<Toy, "kind">>;
// разрешается в:
type GroupedToys = {
  boardgame?: BoardGame[] | undefined;
  puzzle?: Puzzle[] | undefined;
  doll?: Doll[] | undefined;
};
```

Великолепно! Теперь тип стал гораздо более понятным, и мы можем быть уверены, что нам не придется иметь дело с головоломками, когда мы выбираем настольные игры. Но появились некоторые новые проблемы.

Типы каждого свойства гораздо более детализированы и не указывают на очень широкий тип `Toy`, поэтому TypeScript испытывает трудности при правильном определении каждой коллекции в нашей группе:

```
function groupToys(toys: Toy[]): GroupedToys {
  const groups: GroupedToys = {};
  for (let toy of toys) {
    groups[toy.kind] = groups[toy.kind] ?? [];
  // ^ Type 'BoardGame[] | Doll[] | Puzzle[]' is not assignable to
  // type '(BoardGame[] & Puzzle[] & Doll[]) | undefined'. (2322)
    groups[toy.kind]?.push(toy);
  //           ^
  // Argument of type 'Toy' is not assignable to
  // parameter of type 'never'. (2345)
  }
  return groups;
}
```

Проблема вот в чем: TypeScript все еще считает, что `toy` потенциально относится ко всем игрушкам, в то время как каждое свойство `group` указывает на некоторые конкретные игрушки. Решить эту проблему можно с помощью трех способов.

Во-первых, мы могли бы снова проверить каждый член по отдельности. TypeScript считает `toy` очень широким типом, поэтому сужение снова проясняет взаимосвязь:

```
function groupToys(toys: Toy[]): GroupedToys {
  const groups: GroupedToys = {};
  for (let toy of toys) {
    switch (toy.kind) {
      case "boardgame":
        groups[toy.kind] = groups[toy.kind] ?? [];
        groups[toy.kind]?.push(toy);
        break;
      case "doll":
        groups[toy.kind] = groups[toy.kind] ?? [];
        groups[toy.kind]?.push(toy);
        break;
      case "puzzle":
        groups[toy.kind] = groups[toy.kind] ?? [];
        groups[toy.kind]?.push(toy);
        break;
    }
  }
  return groups;
}
```

Этот код работает, но мы хотим избежать большого количества дублирования.

Во-вторых, мы можем использовать утверждение типа для расширения типов `groups[toy.kind]`, чтобы TypeScript мог обеспечить доступ к индексу:

```
function groupToys(toys: Toy[]): GroupedToys {
  const groups: GroupedToys = {};
  for (let toy of toys) {
    (groups[toy.kind] as Toy[]) = groups[toy.kind] ?? [];
    (groups[toy.kind] as Toy[])?.push(toy);
  }
  return groups;
}
```

Код работает эффективно, как и до нашего изменения `GroupedToys`, и утверждение типа сообщает нам, что мы намеренно изменили тип здесь, чтобы избавиться от ошибок типа.

В-третьих, мы можем использовать небольшой окольный путь. Вместо того чтобы добавлять `toy` непосредственно в группу, мы применим вспомогательную функцию `assign`, в которой используем обобщенные типы:

```
function groupToys(toys: Toy[]): GroupedToys {
  const groups: GroupedToys = {};
  for (let toy of toys) {
    assign(groups, toy.kind, toy);
  }
  return groups;
}

function assign<T extends Record<string, K[]>, K>(
  groups: T,
  key: keyof T,
  value: K
) {
  // Инициализировать, если нет значения
  groups[key] = groups[key] ?? [];
  groups[key]?.push(value);
}
```

Здесь мы сужаем правый член объединения `Toy` с помощью общей подстановки TypeScript:

- `groups` — это `T`, а `Record<string, K[]>`. `K[]` может быть потенциально широким;
- `key` — ключ свойства `T`;
- значение имеет тип `K`.

Все три параметра функции связаны друг с другом, и способ, с помощью которого мы разработали отношения типов, позволяет нам безопасно получать доступ к `groups[key]` и помещать `value` в массив.

Кроме того, типы каждого параметра при вызове `assign` будут удовлетворять общим ограничениям типа, которые мы только что установили. Больше информации об этом методе вы найдете в рецепте 12.6.

5.4. Удаление определенных свойств объекта

Задача

Вы хотите создать вспомогательный обобщенный тип для объектов, в котором свойства выбираются на основе их типа, а не имени свойства.

Решение

Выполните фильтрацию с помощью условных типов и утверждений типов при сопоставлении свойств.

Обсуждение

TypeScript позволяет создавать типы на основе других типов, что дает возможность поддерживать их в актуальном состоянии, не поддерживая все их производные. Примеры мы уже рассматривали выше, в частности в рецепте 4.5. В следующем сценарии мы хотим адаптировать существующий тип объекта на основе типов его свойств. Рассмотрим тип для `Person`:

```
type Person = {
  name: string;
  age: number;
  profession?: string;
};
```

Он состоит из двух строк: `profession` и `name`, а также числа: `age`. Мы хотим создать тип, состоящий только из свойств строкового типа:

```
type PersonStrings = {
  name: string;
  profession?: string;
};
```

В TypeScript уже есть некоторые вспомогательные типы для фильтрации имен свойств. Например, сопоставленный тип `Pick<T>` использует подмножество ключей объекта, чтобы создать новый объект, содержащий только эти ключи:

```
type Pick<T, K extends keyof T> = {
  [P in K]: T[P];
}

// Включает только "name"
type PersonName = Pick<Person, "name">;
```

```
// Включает "name" и "profession"
type PersonStrings = Pick<Person, "name" | "profession">;
```

Если мы хотим удалить определенные свойства, то можем использовать `Omit<T>`, работающий так же, как `Pick<T>`, с той небольшой разницей, что мы сопоставляем свойства с помощью слегка измененного набора, из которого удаляются имена ненужных нам свойств:

```
type Omit<T, K extends string | number | symbol> = {
  [P in Exclude<keyof T, K>]: T[P];
}

// Не указан age, поэтому включены "name" и "profession"
type PersonWithoutAge = Omit<Person, "age">;
```

Чтобы выбрать нужные свойства на основе их типа, а не имени, нужно создать аналогичный вспомогательный тип, в котором мы сопоставляем динамически генерируемый набор имен свойств, указывающих только на нужные типы. Из рецепта 5.2 мы знаем, что при использовании условных типов вместо типов объединения мы можем с помощью `never` фильтровать элементы из этого объединения.

Итак, первая возможность состоит в том, что мы сопоставляем все ключи свойств `Person` и проверяем, является ли `Person[K]` подмножеством нужного нам типа. Если да, то мы возвращаем тип; в противном случае возвращаем `never`:

```
// Пока еще нет
type PersonStrings = {
  [K in keyof Person]: Person[K] extends string ? Person[K] : never;
};
```

Это хорошо, но есть нюанс: типы, которые мы проверяем, не входят в объединение, а являются типами из сопоставленного типа. Таким образом, вместо фильтрации ключей свойств мы получим свойства, указывающие на тип `never`; это значит, мы вообще запретим устанавливать определенные свойства.

Другая идея состоит в том, чтобы установить тип в `undefined`, рассматривая свойство как своего рода необязательное, но, как мы узнали в рецепте 3.11, отсутствующие свойства и неопределенные значения — не одно и то же.

На самом деле мы хотим удалить ключи свойств, указывающих на определенный тип. Этого можно добиться, поместив условие не в правую сторону объекта, а в левую, где создаются свойства.

Как и в случае с типом `Omit`, мы должны убедиться, что сопоставляем определенный набор свойств. При сопоставлении `keyof Person` можно изменить тип ключа свойства с помощью утверждения типа. Как и при использовании обычных утверждений типа, существует своего рода отказоустойчивый механизм, который означает следующее: вы просто не можете утверждать, что это может быть какой угодно элемент, — он должен находиться в пределах ключа свойства.

Мы хотим утверждать, что `K` является частью множества, когда `Person[K]` имеет тип `string`. Если это верно, то мы сохраняем `K`; в противном случае фильтруем элемент множества с помощью `never`. Поскольку `never` находится на левой стороне объекта, то свойство отбрасывается:

```
type PersonStrings = {
  [K in keyof Person as Person[K] extends string ? K : never]: Person[K];
};
```

Таким образом, мы выбираем только те ключи свойств, которые указывают на строковые значения. Есть один нюанс: необязательные строковые свойства имеют более широкий тип, чем обычные строки, поскольку в число возможных значений добавлено и значение `undefined`. Используя типы объединения, вы можете быть уверены, что необязательные свойства тоже будут сохранены:

```
type PersonStrings = {
  [K in keyof Person as Person[K] extends string | undefined
    ? K
    : never]: Person[K];
};
```

Следующий шаг — сделать этот тип обобщенным типом. Мы создадим тип `Select<O, T>`, заменив `Person` на `O`, а `string` на `T`:

```
type Select<O, T> = {
  [K in keyof O as O[K] extends T | undefined ? K : never]: O[K];
};
```

Этот новый вспомогательный тип универсален. С его помощью мы можем выбирать свойства определенного типа из наших собственных типов объектов:

```
type PersonStrings = Select<Person, string>;
type PersonNumbers = Select<Person, number>;
```

Но мы также можем выяснить, например, какие функции в прототипе `string` возвращают число:

```
type StringFnsReturningNumber = Select<String, (...args: any[]) => number>;
```

Обратный вспомогательный тип `Remove<O, T>`, в котором мы хотим удалить ключи свойств определенного типа, очень похож на `Select<O, T>`.

Единственное различие состоит в переключении условия и возврате `never` в ветвь `true`:

```
type Remove<O, T> = {
  [K in keyof O as O[K] extends T | undefined ? never : K]: O[K];
};

type PersonWithoutStrings = Remove<Person, string>;
```

Такая возможность особенно полезна, если вы создаете сериализуемую версию типов объектов:

```
type User = {
  name: string;
  age: number;
  profession?: string;
  posts(): string[];
  greeting(): string;
};

type SerializeableUser = Remove<User, Function>;
```

Зная, что вы можете использовать условные типы при сопоставлении ключей, вы неожиданно получаете доступ к широкому спектру потенциальных вспомогательных типов. Подробнее об этом поговорим в главе 8.

5.5. Выведение типов в условных выражениях

Задача

Вы хотите создать класс для сериализации объектов, который удаляет все не-сериализуемые свойства объекта, например функции. Если в вашем объекте есть функция `serialize`, то сериализатор принимает ее возвращаемое значение вместо того, чтобы сериализовать объект самостоятельно. С помощью каких типов вы можете это сделать?

Решение

Используйте рекурсивный условный тип, чтобы изменить существующий тип объекта. Для объектов, реализующих `serialize`, примените ключевое слово `infer`, чтобы привязать обобщенный тип возвращаемого значения к конкретному типу.

Обсуждение

Сериализация — это процесс преобразования структур данных и объектов в формат, который можно сохранить или передавать. Представьте, что берете объект JavaScript и сохраняете его данные на диске, чтобы позже снова получить их, десериализовав в JavaScript.

Объекты JavaScript могут содержать данные любого типа: примитивные типы, такие как строки или числа, составные типы, такие как объекты, и даже функции. Последние интересны тем, что содержат не данные, а поведение: то, что не может быть хорошо сериализовано. Один из подходов к сериализации объектов JavaScript состоит в том, чтобы полностью избавиться от функций. Именно это действие мы и хотим реализовать в данном уроке.

Мы начнем с простого объекта типа `Person`, содержащего обычные данные, которые мы хотим сохранить: имя и возраст человека. У него также есть метод `hello`, который выдает строку:

```
type Person = {
  name: string;
  age: number;
  hello: () => string;
};
```

Мы хотим сериализовать объекты этого типа. Класс `Serializer` содержит пустой конструктор и обобщенную функцию `serialize`. Обратите внимание, что мы добавляем параметр обобщенного типа в `serialize`, а не в класс. Таким образом, мы можем повторно использовать `serialize` для различных типов объектов. Тип возвращаемого значения указывает на обобщенный тип `Serialize<T>`, который будет результатом процесса сериализации:

```
class Serializer {
  constructor() {}
  serialize<T>(obj: T): Serialize<T> {
    // будет определено
  }
}
```

Реализацией мы займемся позже. Пока же сосредоточимся на типе `Serialize<T>`. Первая идея, которая приходит в голову, — просто отказаться от свойств, которые являются функциями. В рецепте 5.4 мы уже определили тип `Remove<O, T>`, который очень удобен, поскольку делает именно это — удаляет свойства определенного типа:

```
type Remove<O, T> = {
  [K in keyof O as O[K] extends T | undefined ? never : K]: O[K];
};
```

```
type Serialize<T> = Remove<T, Function>;
```

Первая итерация выполнена, и она работает для простых объектов глубиной в один уровень. Однако объекты могут быть сложными. Например, `Person` может содержать другие объекты, которые, в свою очередь, тоже могут иметь функции:

```
type Person = {
  name: string;
  age: number;
  profession: {
    title: string;
  };
};
```



```

    level: number;
    printProfession: () => void;
  };
  hello: () => string;
};

```

Чтобы решить эту проблему, нужно проверить, является ли каждое свойство другим объектом, и если да, то снова использовать тип `Serialize<T>`. Сопоставленный тип `NestSerialization` проверяет в условном типе, является ли каждое свойство типом `object`, и возвращает сериализованную версию этого типа в ветвь `true` и сам тип в ветвь `false`:

```

type NestSerialization<T> = {
  [K in keyof T]: T[K] extends object ? Serialize<T[K]> : T[K];
};

```

Мы переопределяем `Serialize<T>`, поместив исходный тип `Remove<T, Function>` из `Serialize<T>` в `NestSerialization`, фактически создавая *рекурсивный тип*. `Serialize<T>` использует `NestSerialization<T>`, который использует `Serialize<T>`, и т. д.:

```

type Serialize<T> = NestSerialization<Remove<T, Function>>;

```

TypeScript в известной степени может обрабатывать рекурсию типов. В данном случае он видит, что в `NestSerialization` буквально существует условие для выхода из рекурсии типов.

И это тип сериализации! Теперь перейдем к реализации функции, которая, как ни странно, является прямым переводом нашего объявления типа в JavaScript. Для каждого свойства мы проверяем, является ли оно объектом. Если да, то мы снова вызываем `serialize`. В противном случае переносим свойство, при условии, что оно не является функцией:

```

class Serializer {
  constructor() {}
  serialize<T>(obj: T): Serialize<T> {
    const ret: Record<string, any> = {};

    for (let k in obj) {
      if (typeof obj[k] === "object") {
        ret[k] = this.serialize(obj[k]);
      } else if (typeof obj[k] !== "function") {
        ret[k] = obj[k];
      }
    }
    return ret as Serialize<T>;
  }
}

```

Обратите внимание, что мы создаем новый объект в рамках `serialize`, поэтому начинаем с очень широкого типа `Record<string, any>`, который позволяет нам установить любой ключ строкового свойства практически на что угодно и в конце

утверждать, что мы создали объект, который соответствует нашему возвращаемому типу. Такой паттерн часто встречается при создании новых объектов, но в итоге требует от вас быть абсолютно уверенными в том, что вы все сделали правильно. Пожалуйста, тщательно протестируйте эту функцию.

После завершения первой реализации мы можем написать новый объект типа `Person` и передать его нашему свеже созданному сериализатору:

```
const person: Person = {
  name: "Stefan",
  age: 40,
  profession: {
    title: "Software Developer",
    level: 5,
    printProfession() {
      console.log(`${this.title}, Level ${this.level}`);
    },
  },
  hello() {
    return `Hello ${this.name}`;
  },
};

const serializer = new Serializer();
const serializedPerson = serializer.serialize(person);
console.log(serializedPerson);
```

Результат ожидаемый: в типе `serializedPerson` отсутствует вся информация о методах и функциях. И если мы запишем в лог `serializedPerson`, то увидим, что все методы и функции исчезли. Тип соответствует результату реализации:

```
[LOG]: {
  "name": "Stefan",
  "age": 40,
  "profession": {
    "title": "Software Developer",
    "level": 5
  }
}
```

Но мы еще не закончили. У сериализатора есть специальная функция. Объекты могут реализовывать метод `serialize`, и если он реализован, то сериализатор использует выходные данные этого метода вместо того, чтобы сериализовать объект самостоятельно. Расширим тип `Person` и добавим в него метод `serialize`:

```
type Person = {
  name: string;
  age: number;
  profession: {
    title: string;
    level: number;
```

```

    printProfession: () => void;
  };
  hello: () => string;
  serialize: () => string;
};

const person: Person = {
  name: "Stefan",
  age: 40,
  profession: {
    title: "Software Developer",
    level: 5,
    printProfession() {
      console.log(`${this.title}, Level ${this.level}`);
    },
  },
  hello() {
    return `Hello ${this.name}`;
  },
  serialize() {
    return `${this.name}: ${this.profession.title} L${this.profession.level}`;
  },
};

```

Нам необходимо адаптировать тип `Serialize<T>`. Перед запуском `NestSerialization` мы проверяем в условном типе, реализует ли объект метод `serialize`. Для этого мы спрашиваем, является ли `T` подтипом типа, содержащего метод `serialize`. Если да, то нам нужно перейти к возвращаемому типу, поскольку это результат сериализации.

Именно здесь мы используем ключевое слово `infer`. Оно позволяет нам взять тип из условия и применить в качестве параметра типа в ветви `true`.

Мы сообщаем TypeScript: если это условие истинно, то возьми тот тип, который ты здесь нашел, и сделай его доступным для нас:

```

type Serialize<T> = T extends { serialize(): infer R }
  ? R
  : NestSerialization<Remove<T, Function>>;

```

Сначала представьте, что `R` — это `any`. Если мы проверим `Person` на соответствие `{serialize(): any}`, то попадем в ветвь `true`, поскольку у `Person` есть функция `serialize`, что делает его допустимым подтипом. Но `any` — широкое понятие, а нас интересует конкретный тип в позиции `any`. Ключевое слово `infer` может указать на этот тип. Таким образом, `Serialize<T>` теперь читается так:

- если `t` содержит метод `serialize`, то получи тип его возвращаемого значения и верни его;
- в противном случае начни сериализацию с полного удаления всех свойств, имеющих тип `Function`.

Мы хотим отразить поведение этого типа и в нашей реализации JavaScript. Мы выполняем несколько проверок типа: проверяем, доступна ли функция `serialize`, является ли она функцией, и в итоге вызываем ее. TypeScript требует, чтобы мы явно указали защиту типа, чтобы быть абсолютно уверенными в том, что эта функция существует:

```
class Serializer {
  constructor() {}
  serialize<T>(obj: T): Serialize<T> {
    if (
      // является объектом
      typeof obj === "object" &&
      // не null
      obj &&
      // доступна сериализация
      "serialize" in obj &&
      // и функция
      typeof obj.serialize === "function"
    ) {
      return obj.serialize();
    }

    const ret: Record<string, any> = {};

    for (let k in obj) {
      if (typeof obj[k] === "object") {
        ret[k] = this.serialize(obj[k]);
      } else if (typeof obj[k] !== "function") {
        ret[k] = obj[k];
      }
    }
    return ret as Serialize<T>;
  }
}
```

После того как это изменение будет внесено, тип `serializedPerson` станет `string`, а результат будет таким, как ожидалось:

```
[LOG]: "Stefan: Software Developer L5"
```

Этот мощный инструмент очень помогает при создании объектов. И есть своя прелесть в том, что мы создаем тип, используя декларативный метаязык, которым является система типов TypeScript, чтобы в итоге увидеть тот же самый процесс, который написан на JavaScript, но императивно.

Типы литералов шаблонов строк

В системе типов TypeScript каждое значение является еще и типом. Мы называем такие типы литеральными, и в сочетании с другими литеральными типами вы можете определить тип, в котором четко определено, какие значения он может принимать. В качестве примера возьмем подмножество `string`. Вы можете точно определить, какие строки должны быть частью вашего набора, и исключить массу ошибок. На другой чаше весов снова будет весь набор строк.

Но что, если есть нечто среднее? Что, если мы можем определить типы, которые проверяют наличие определенных паттернов строк, а остальное пусть будет более гибким? *Типы литералов строковых шаблонов* делают именно это. Они позволяют определять типы, в которых конкретные части строки предопределены, а все остальное может применяться в различных сценариях использования.

Более того, в сочетании с условными типами можно разбивать строки на отдельные фрагменты и повторно использовать те же фрагменты для новых типов. Это невероятно мощный инструмент, особенно если вспомнить, в каком объеме кода JavaScript используются шаблоны внутри строк.

В этой главе мы рассмотрим различные сценарии использования литеральных типов шаблонов строк: от следования простым шаблонам строк до извлечения параметров и типов на основе строк формата. Вы увидите, насколько эффективен синтаксический анализ строк как типов.

Но мы придерживаемся реалистичности. Все, что вы увидите здесь, взято из реальных примеров. Возможности, которые вы можете реализовать с помощью литеральных типов шаблонов строк, кажутся бесконечными. Разработчики используют типы литералов шаблонов строк крайне разнообразно, создавая программы проверки орфографии (<https://oreil.ly/63z2Y>) или реализуя парсеры SQL (<https://oreil.ly/fo5vx>). Кажется, нет предела тому, что можно сделать с помощью этой потрясающей функции.

6.1. Определение пользовательской системы событий

Задача

Вы создаете пользовательскую систему событий и хотите убедиться, что название каждого события соответствует определенным правилам и начинается с "on".

Решение

Используйте типы литералов шаблонов строк для описания шаблонов строк.

Обсуждение

Как правило, в системах событий JavaScript используется какой-либо префикс, указывающий на то, что определенная строка является событием. Обычно строки события или обработчика событий начинаются с `on`, но в зависимости от реализации может использоваться и другое обозначение.

Мы хотим создать собственную систему событий, соблюдая это соглашение. С помощью строковых типов TypeScript можно принимать либо все возможные строки, либо подмножество типов строковых литералов в типе объединения. Первый вариант слишком широк, а второй недостаточно гибок для наших нужд. Мы не хотим заранее определять все возможные имена событий, а планируем придерживаться определенного шаблона.

К счастью, *типы литералов шаблонов строк* или просто *типы литеральных шаблонов* — это именно то, что нам нужно. Они позволяют определять строковые литералы, оставляя некоторые части гибкими.

Например, тип, который принимает все строки, начинающиеся с `on`, может выглядеть следующим образом:

```
type EventName = `on${string}`;
```

Синтаксически литеральные типы шаблонов заимствованы из *шаблонных строк* JavaScript. Они начинаются и заканчиваются знаком обратной кавычки, за которым следует любая строка.

Использование специального синтаксиса `${}` позволяет добавлять в строки выражения JavaScript, такие как переменные, вызовы функций и т. п.:

```
function greet(name: string) {  
    return `Hi, ${name}!`;  
}
```

```
greet("Stefan"); // "Hi, Stefan!"
```

Литеральные типы шаблонов в TypeScript очень похожи. Вместо выражений JavaScript они позволяют добавлять набор значений в виде типов. Тип, определяющий строковое представление всех доступных элементов заголовка в HTML, может выглядеть следующим образом:

```
type Levels = 1 | 2 | 3 | 4 | 5 | 6;

// разрешается в "H1" | "H2" | "H3" | "H4" | "H5" | "H6"
type Headings = `H${Levels}`;
```

`Levels` — это подмножество `number`, а `Headings` читается как «начинается с H, за которым следует значение, совместимое с `Levels`». Вы не можете указать здесь все типы — только те, которые имеют строковое представление.

Вернемся к `EventName`:

```
type EventName = `on${string}`;
```

Определенный таким образом `EventName` читается как «начинается с "on", за которым следует любая строка». Он содержит пустую строку. Используем `EventName` для создания простой системы событий. Сначала мы хотим собрать только функции обратного вызова.

Для этого мы определяем тип `Callback`, который является типом функции с одним параметром: `EventObject` — обобщенным типом, который содержит значение с информацией о событии:

```
type EventObject<T> = {
  val: T;
};

type Callback<T = any> = (ev: EventObject<T>) => void;
```

Кроме того, нам нужен тип для хранения всех зарегистрированных обратных вызовов событий, `Events`:

```
type Events = {
  [x: EventName]: Callback[] | undefined;
};
```

Мы используем `EventName` в качестве индекса доступа, поскольку он является допустимым подтипом `string`. Каждый индекс указывает на массив обратных вызовов. Определив типы, мы создаем класс `EventSystem`:

```
class EventSystem {
  events: Events;
  constructor() {
    this.events = {};
  }

  defineEventHandler(ev: EventName, cb: Callback): void {
    this.events[ev] = this.events[ev] ?? [];
  }
}
```

```

    this.events[ev]?.push(cb);
  }

  trigger(ev: EventName, value: any) {
    let callbacks = this.events[ev];
    if (callbacks) {
      callbacks.forEach((cb) => {
        cb({ val: value });
      });
    }
  }
}

```

Конструктор создает новое хранилище событий, а `defineEventHandler` принимает `EventName` и `Callback` и сохраняет их в указанном хранилище. Кроме того, `trigger` принимает `EventName` и, если обратные вызовы зарегистрированы, выполняет каждый зарегистрированный обратный вызов с `EventObject`.

Первый шаг сделан. Теперь определение событий становится безопасным для типов:

```

const system = new EventSystem();
system.defineEventHandler("click", () => {});
// ^ Argument of type '"click"' is not assignable to parameter
//. of type '`on${string}`'.(2345)
system.defineEventHandler("onClick", () => {});
system.defineEventHandler("onChange", () => {});

```

В рецепте 6.2 мы рассмотрим, как можно использовать типы манипуляций со строками и с помощью переназначения ключей улучшить нашу систему.

6.2. Создание обратных вызовов событий с помощью типов манипуляций со строками и переназначения ключей

Задача

Вы хотите предоставить функцию `watch`, которая принимает любой объект и добавляет функции прослушателя событий для каждого свойства, позволяя вам определять обратные вызовы событий.

Решение

Используйте *переназначения ключей* для создания новых ключей свойств строк, а *типы манипуляций со строками* — для того чтобы обеспечить правильный верблужий регистр (camel casing) для функций-прослушателей.

Обсуждение

Наша система событий из рецепта 6.1 обретает форму. Мы можем регистрировать обработчики событий и инициировать события. Теперь мы хотим добавить функциональность отслеживания. Идея состоит в том, чтобы расширить допустимые объекты с помощью методов для регистрации обратных вызовов, которые будут выполняться каждый раз, когда свойство изменяется. Например, определяя объект `person`, мы должны иметь возможность прослушивать события `onAgeChanged` и `onNameChanged`:

```
let person = {
  name: "Stefan",
  age: 40,
};

const watchedPerson = system.watch(person);

watchedPerson.onAgeChanged((ev) => {
  console.log(ev.val, "changed!!");
});

watchedPerson.age = 41; // запускает обратные вызовы
```

Таким образом, для каждого свойства будет реализован метод, начинающийся с `on`, заканчивающийся `Changed` и принимающий функции обратного вызова с параметрами объекта события.

Чтобы определить новые методы обработчика событий, мы создаем вспомогательный тип `WatchedObject<T>`, в который добавляем специальные методы:

```
type WatchedObject<T> = {
  [K in string & keyof T as `on${K}Changed`]: (
    ev: Callback<T[K]>
  ) => void;
};
```

Рассмотрим код поэтапно.

1. Мы определяем *сопоставленный тип*, перебирая все ключи из `T`. Нам интересуют только ключи свойств типа `string`, поэтому мы используем пересечение `string & keyof T`, чтобы избавиться от возможных символов или чисел.
2. Далее мы *переназначаем* этот ключ новой строке, определяемой *литеральным типом шаблона строки*. Он начинается с `on`, затем берется ключ `K` из нашего процесса сопоставления и к нему добавляется `Changed`.
3. Ключ свойства указывает на функцию, принимающую обратный вызов. Сам вызов имеет в качестве аргумента объект события, и, правильно заместив его обобщенные типы, мы можем убедиться, что данный объект события содержит исходный тип нашего объекта наблюдения. Это означает, что, когда мы вызываем `onAgeChanged`, объект события будет содержать `number`.

Это уже нечто удивительное, но здесь не хватает существенных деталей. Когда мы используем `WatchedObject` для `person`, во всех сгенерированных методах обработчика событий после `on` не используется символ верхнего регистра. Чтобы решить эту проблему, мы можем задействовать один из встроенных *типов манипуляций со строками* для выделения заглавных букв в строковых типах:

```
type WatchedObject<T> = {
  [K in string & keyof T as `on${Capitalize<K>}Changed`]: (
    ev: Callback<T[K]>
  ) => void;
};
```

Помимо `Capitalize`, доступны также варианты `Lowercase`, `Uppercase` и `Uncapitalize`. Если мы наведем курсор на `WatchedObject<typeof person>`, то увидим, как выглядит сгенерированный тип:

```
type WatchedPerson = {
  onNameChanged: (ev: Callback<string>) => void;
  onAgeChanged: (ev: Callback<number>) => void;
};
```

Установив типы, приступаем к реализации. Сначала создадим две вспомогательные функции:

```
function capitalize(inp: string) {
  return inp.charAt(0).toUpperCase() + inp.slice(1);
}

function handlerName(name: string): EventName {
  return `on${capitalize(name)}Changed` as EventName;
}
```

Обе вспомогательные функции нужны, чтобы имитировать поведение TypeScript при переназначении строк и манипулировании ими. Функция `capitalize` изменяет первую букву строки на ее эквивалент в верхнем регистре, а `handlerName` добавляет к ней префикс и суффикс. С помощью `handlerName` нам нужно выполнить небольшое утверждение типа, чтобы сообщить TypeScript об изменении типа. Учитывая множество способов преобразования строк в JavaScript, TypeScript не сможет предвидеть, что это приведет к созданию версии с заглавной буквой.

Далее мы реализуем функцию `watch` в системе событий. Мы создаем обобщенную функцию, которая принимает любой объект и возвращает объект, содержащий как исходные свойства, так и свойства наблюдателя.

Чтобы успешно реализовать срабатывание обработчиков событий при изменении свойств, мы используем объекты `Proxy` для перехвата вызовов `get` и `set`:

```
class EventSystem {
  // сокращено для краткости
  watch<T extends object>(obj: T): T & WatchedObject<T> {
```

```

const self = this;
return new Proxy(obj, {
  get(target, property) {
    // (1)
    if (
      typeof property === "string" &&
      property.startsWith("on") &&
      property.endsWith("Changed")
    ) {
      // (2)
      return (cb: Callback) => {
        self.defineEventHandler(property as EventName, cb);
      };
    }
    // (3)
    return target[property as keyof T];
  },
  // будет сделано...
}) as T & WatchedObject<T>;
}
}

```

Вызовы `get`, которые мы хотим перехватить, выполняются всякий раз, когда мы обращаемся к свойствам `WatchedObject<T>`:

- они начинаются с `on` и заканчиваются `Changed`;
- в этом случае мы возвращаем функцию, принимающую обратные вызовы. Сама функция добавляет обратные вызовы в хранилище событий с помощью `defineEventHandler`;
- во всех остальных случаях мы выполняем обычный доступ к свойствам.

Теперь каждый раз, устанавливая значение исходного объекта, мы хотим запускать сохраненные события. Именно поэтому мы изменяем все вызовы `set`:

```

class EventSystem {
  // ... сокращено для краткости
  watch<T extends object>(obj: T): T & WatchedObject<T> {
    const self = this;
    return new Proxy(obj, {
      // получено сверху...
      set(target, property, value) {
        if (property in target && typeof property === "string") {
          // (1)
          target[property as keyof T] = value;
          // (2)
          self.trigger(handlerName(property), value);
          return true;
        }
        return false;
      },
    });
  },
}

```

```

    }) as T & WatchedObject<T>;
  }
}

```

Процесс выглядит так.

1. Установите значение. В любом случае нужно обновить объект.
2. Вызовите функцию `trigger`, чтобы выполнить все зарегистрированные обратные вызовы.

Обратите внимание, что нам нужна пара утверждений типа, чтобы указать TypeScript правильное направление. В конце концов, мы создаем новые объекты.

Вот и все! Попробуйте воспроизвести пример с самого начала, чтобы увидеть, как работает ваша система событий:

```

let person = {
  name: "Stefan",
  age: 40,
};

const watchedPerson = system.watch(person);

watchedPerson.onAgeChanged((ev) => {
  console.log(ev.val, "changed!!");
});

watchedPerson.age = 41; // записывает "41 changed!!"

```

Типы литералов шаблонов строк, а также типы манипуляций со строками и переназначения ключей позволяют создавать типы для новых объектов прямо во время работы. Эти мощные инструменты делают процесс использования расширенного создания объектов JavaScript более надежным.

6.3. Написание функции форматирования

Задача

Вам нужно создать типизацию для функции, которая принимает строку форматирования и заменяет заполнители фактическими значениями.

Решение

Создайте условный тип, который выводит имя заполнителя из литерального типа шаблона строки.

Обсуждение

В вашем приложении вы можете определять строки формата, задавая заполнители с помощью фигурных скобок. Второй параметр принимает объект с подстановками, так что для каждого заполнителя, определенного в строке формата, существует один ключ свойства с соответствующим значением:

```
format("Hello {world}. My name is {you}.", {
  world: "World",
  you: "Stefan",
});
```

Создадим типизации для этой функции, чтобы убедиться, что пользователи не забудут добавить необходимые свойства. В качестве первого шага мы определим интерфейс функции с помощью некоторых очень широких типов. Строка форматирования имеет тип `string`, а параметры форматирования находятся в записи `Record` ключей типа `string` и буквально любого значения. Сначала мы сосредоточимся на типах, а реализацию тела функции рассмотрим позже:

```
function format(fmtString: string, params: Record<string, any>): string {
  throw "unimplemented";
}
```

В качестве следующего шага мы хотим привязать аргументы функции к конкретным значениям или литеральным типам, добавив обобщенные типы. Мы изменим тип `fmtString` на обобщенный тип `T`, который является подтипом `string`. Это позволяет нам по-прежнему передавать строки в функцию, но в момент передачи литеральной строки мы можем анализировать тип литерала и искать шаблоны (более подробную информацию см. в рецепте 4.3):

```
function format<T extends string>(
  fmtString: T,
  params: Record<string, any>
): string {
  throw "unimplemented";
}
```

Теперь, зафиксировав `T`, мы можем передать его в качестве параметра типа в обобщенный тип `FormatKeys`. Это условный тип, который будет проверять нашу строку формата на наличие фигурных скобок:

```
type FormatKeys<
  T extends string
> = T extends `${string}${string}${string}`
  ? T
  : never;
```

Здесь мы проверяем, соответствует ли строка формата следующим правилам:

- начинается со строки, которая может быть и пустой;
- содержит {, за которым следует любая строка, а затем };
- за закрывающей фигурной скобкой снова следует любая строка.

Фактически это означает, что мы проверяем, есть ли в строке формата ровно один заполнитель. Если да, то мы возвращаем всю строку формата, а если нет, то возвращаем `never`:

```
type A = FormatKeys<"Hello {world}">; // "Hello {world}"
type B = FormatKeys<"Hello">; // тип never
```

`FormatKeys` может сообщить нам, являются ли передаваемые строки строками формата, но в действительности нас гораздо больше интересует конкретная часть строки формата: фрагмент между фигурными скобками. Используя ключевое слово TypeScript `infer`, мы можем сообщить TypeScript, что если строка формата соответствует этому шаблону, то TypeScript может взять любой литеральный тип, который найдет между фигурными скобками, и поместить его в переменную типа:

```
type FormatKeys<
  T extends string
> = T extends `${string}{${infer Key}}${string}`
  ? Key
  : never;
```

Таким образом, мы можем извлекать подстроки и использовать их для наших нужд:

```
type A = FormatKeys<"Hello {world}">; // "world"
type B = FormatKeys<"Hello">; // тип never
```

Замечательно! Мы извлекли первое имя-заполнитель. Теперь перейдем к остальным. После него могут быть еще какие-то заполнители, поэтому мы берем все, что находится *после* первого заполнителя, и сохраняем в переменной `Rest`. Это условие будет всегда истинным, поскольку `Rest` либо является пустой строкой, либо содержит фактическую строку, которую мы можем проанализировать снова.

Мы берем `Rest` и в ветви `true` вызываем `FormatKeys<Rest>` в типе объединения `Key`:

```
type FormatKeys<
  T extends string
> = T extends `${string}{${infer Key}}${infer Rest}`
  ? Key | FormatKeys<Rest>
  : never;
```

Это *рекурсивный условный тип*. Результатом будет объединение заполнителей, которые мы можем использовать в качестве ключей для объекта форматирования:

```
type A = FormatKeys<"Hello {world}">; // "world"
type B = FormatKeys<"Hello {world}. I'm {you}.">; // "world" | "you"
type C = FormatKeys<"Hello">; // never
```

Теперь пришло время подключить `FormatKeys`. Мы уже заблокировали `T`, поэтому можем передать его в качестве аргумента `FormatKeys`, который можем использовать в качестве аргумента `Record`:

```
function format<T extends string>(
  fmtString: T,
  params: Record<FormatKeys<T>, any>
): string {
  throw "unimplemented";
}
```

И теперь все наши типизации готовы. Переходим к реализации! Она полностью соответствует тому, как мы определяли наши типы. Мы просматриваем все ключи из `params` и заменяем все вхождения в фигурных скобках на соответствующее значение:

```
function format<T extends string>(
  fmtString: T,
  params: Record<FormatKeys<T>, any>
): string {
  let ret: string = fmtString;
  for (let k in params) {
    ret = ret.replaceAll(`{${k}}`, params[k as keyof typeof params]);
  }
  return ret;
}
```

Обратите внимание на два нюанса.

- Нам нужно аннотировать `ret` с помощью `string`. `fmtString` соответствует `T`, подтипу `string`; таким образом, `ret` также будет `T`. Это будет означать, что мы не сможем изменять значения, поскольку изменится тип `T`. Его аннотация к более широкому типу `string` помогает изменить `ret`.
- Нам также необходимо утверждение, что ключ объекта `k` на самом деле является ключом `params`. Это неудачное обходное решение, которое связано с некоторыми механизмами безопасности TypeScript. Более подробная информация по этой теме содержится в рецепте 9.1.

Прочитайте рецепт 9.1, а затем вернитесь к текущему рецепту. Используя информацию из рецепта 9.1, мы можем переопределить `format`, чтобы избавиться от некоторых утверждений типа и получить окончательную версию функции `format`:

```
function format<T extends string, K extends Record<FormatKeys<T>, any>>(
  fmtString: T,
  params: K
): string {
  let ret: string = fmtString;
  for (let k in params) {
    ret = ret.replaceAll(`{${k}}`, params[k]);
  }
  return ret;
}
```

Возможность разделять строки и извлекать ключи свойств — чрезвычайно эффективная. Разработчики TypeScript по всему миру используют этот паттерн для усиления типов, например для веб-серверов типа Express (<https://expressjs.com/>). Вы еще увидите несколько примеров того, как с помощью этого инструмента получать более совершенные типы.

6.4. Извлечение типов параметров формата

Задача

Вы хотите расширить функцию форматирования из рецепта 6.3, добавив в нее возможность определения типов ваших заполнителей.

Решение

Создайте вложенный условный тип и найдите типы с помощью карты типов.

Обсуждение

Расширим пример из предыдущего урока. Теперь мы хотим не только знать все заполнители, но и иметь возможность определять конкретный набор типов с помощью заполнителей. Типы должны быть необязательными, обозначаться двоеточием после имени заполнителя и являться одним из примитивных типов JavaScript. Мы ожидаем, что при передаче значения неправильного типа будут возникать ошибки типа:

```
format("Hello {world:string}. I'm {you}, {age:number} years old.", {
  world: "World",
  age: 40,
  you: "Stefan",
});
```

Для справки рассмотрим исходную реализацию из рецепта 6.3:

```
type FormatKeys<
  T extends string
> = T extends `${string}${infer Key}${infer Rest}`
  ? Key | FormatKeys<Rest>
  : never;

function format<T extends string>(
  fmtString: T,
  params: Record<FormatKeys<T>, any>
): string {
```



```

let ret: string = fmtString;
for (let k in params) {
  ret = ret.replace(`${k}`, params[k as keyof typeof params]);
}
return ret;
}

```

Чтобы достичь этого, нам нужно выполнить два действия.

1. Изменить тип `params` с `Record<FormatKeys<T>, any>` на фактический тип объекта, который имеет правильные типы, связанные с каждым ключом свойства.
2. Адаптировать литеральный тип шаблона строки в `FormatKeys`, чтобы он мог извлекать примитивные типы JavaScript.

В качестве первого шага мы вводим новый тип `FormatObj<T>`. Он работает так же, как и `FormatKeys`, но вместо того, чтобы просто возвращать строковые ключи, сопоставляет те же ключи с новым типом объекта. Для этого нам необходимо связать рекурсию в цепочку, используя пересечение типов вместо типов объединения (мы добавляем больше свойств с каждой рекурсией), и изменить условие завершения с `never` на `{}`. Если бы мы использовали пересечение с `never`, то весь тип возвращаемого значения стал бы `never`. Таким образом, мы не добавляем никаких новых свойств к типу возвращаемого значения:

```

type FormatObj<
  T extends string
> = T extends `${string}${infer Key}${infer Rest}`
  ? { [K in Key]: any } & FormatObj<Rest>
  : {};

```

`FormatObj<T>` работает так же, как `Record<FormatKeys<T>, any>`. Мы по-прежнему не извлекаем ни один тип заполнителя, но теперь, контролируя весь тип объекта, можем легко установить тип для каждого заполнителя.

В качестве второго шага мы изменим условие синтаксического анализа в `FormatObj<T>`, чтобы оно также искало разделители двоеточий. Если мы находим символ `:`, то определяем тип последующего строкового литерала в поле `Type` и используем его как тип для сопоставленного ключа:

```

type FormatObj<
  T extends string
> = T extends `${string}${infer Key}:${infer Type}${infer Rest}`
  ? { [K in Key]: Type } & FormatObj<Rest>
  : {};

```

Мы очень близки к цели, но есть один нюанс. Мы выводим тип *строкового* литерала. Это означает, что если мы, например, проанализируем `{age: number}`, то типом `age` будет литеральная строка `"number"`. Нам нужно преобразовать эту строку

в фактический тип. Мы можем сделать еще один условный тип или использовать тип `map` для подстановки:

```
type MapFormatType = {
  string: string;
  number: number;
  boolean: boolean;
  [x: string]: any;
};
```

Таким образом, мы можем просто проверять, какой тип связан с тем или иным ключом, и получить отличный запасной вариант для всех остальных строк:

```
type A = MapFormatType["string"]; // строка
type B = MapFormatType["number"]; // число
type C = MapFormatType["notavailable"]; // что угодно
```

Соединим `MapFormatType` с `FormatObj<T>`:

```
type FormatObj<
  T extends string
> = T extends `${string}${infer Key}:${infer Type}${infer Rest}`
  ? { [K in Key]: MapFormatType[Type] } & FormatObj<Rest>
  : {};
```

Мы почти у цели! Есть одна проблема: мы ожидаем, что каждый заполнитель будет также определять тип. Мы хотим сделать типы необязательными. Но наше условие синтаксического анализа явно требует наличия разделителей `:`, поэтому каждый заполнитель, не определяющий тип, не создает и свойства.

Решение состоит в том, чтобы выполнять проверку типов *после* того, как мы проверим наличие заполнителя:

```
type FormatObj<
  T extends string
> = T extends `${string}${infer Key}${infer Rest}`
  ? Key extends `${infer KeyPart}:${infer TypePart}`
    ? { [K in KeyPart]: MapFormatType[TypePart] } & FormatObj<Rest>
    : { [K in Key]: any } & FormatObj<Rest>
  : {};
```

Тип читается следующим образом.

1. Проверяем наличие заполнителя.
2. Если заполнитель доступен, то проверяем, есть ли в нем аннотация типа. Если да, то сопоставьте ключ с типом формата, в противном случае сопоставьте исходный ключ с `any`.
3. Во всех остальных случаях возвращаем пустой объект.

И это все. Есть одно средство защиты от сбоев, которое мы можем добавить. Вместо того чтобы разрешить использовать тип `any` для заполнителей без определения типа, мы можем, по крайней мере, ожидать, что этот тип реализует `toString()`. Это позволяет нам быть уверенными в том, что мы всегда будем получать строковое представление:

```
type FormatObj<
  T extends string
> = T extends `${string}{${infer Key}}${infer Rest}`
  ? Key extends `${infer KeyPart}:${infer TypePart}`
    ? { [K in KeyPart]: MapFormatType[TypePart] } & FormatObj<Rest>
    : { [K in Key]: { toString(): string } } & FormatObj<Rest>
  : {};
```

После этого применим новый тип к функции `format` и изменим реализацию:

```
function format<T extends string, K extends FormatObj<T>>(
  fmtString: T,
  params: K
): string {
  let ret: string = fmtString;
  for (let k in params) {
    let val = `${params[k]}`;
    let searchPattern = new RegExp(`${k}:?.*?`, "g");
    ret = ret.replaceAll(searchPattern, val);
  }
  return ret;
}
```

Мы используем регулярное выражение для замены имен потенциальными аннотациями типов. Нет необходимости проверять типы внутри функции. TypeScript должно быть достаточно, чтобы помочь в этом случае.

Мы убедились, что условные типы в сочетании с литеральными типами строковых шаблонов и другими инструментами, такими как рекурсия и поиск типов, позволяют задавать сложные отношения с помощью пары строк кода. Наши типы улучшаются, код становится более надежным, а разработчикам приятно использовать подобные API.

6.5. Работа с ограничениями рекурсии

Задача

Вы создаете сложный литеральный тип строкового шаблона, который преобразует любую строку в допустимый ключ свойства. При настройке вспомогательных типов вы столкнулись с ограничениями рекурсии.

Решение

Используйте метод накопления, чтобы добавить оптимизацию конечных вызовов.

Обсуждение

Типы литералов шаблонов строк в TypeScript в сочетании с условными типами позволяют вам прямо в ходе работы создавать новые строковые типы, которые могут служить ключами свойств или проверять вашу программу на наличие допустимых строк.

Они работают с использованием рекурсии; это значит, что, как и функция, вы можете вызывать один и тот же тип снова и снова, до определенного предела.

Например, этот тип `Trim<T>` удаляет символы пробела в начале и в конце вашего строкового типа:

```
type Trim<T extends string> =
  T extends `${infer X}` ? Trim<X> :
  T extends `${infer X}` ? Trim<X> :
  T;
```

Он проверяет, есть ли пробелы в начале, выводит остальные и снова повторяет ту же проверку. Как только все пробельные символы в начале будут удалены, та же проверка выполняется для пробельных символов в конце. Удалив все пробельные символы в начале и в конце, тип `Trim<T>` завершает работу и переходит в последнюю ветвь, возвращая оставшуюся строку:

```
type Trimmed = Trim<" key ">; // "key"
```

Повторный вызов этого типа — это рекурсия, и написание его таким образом работает достаточно хорошо. TypeScript может видеть по типу, что рекурсивные вызовы выполняются сами по себе, и может оценить их как оптимизированные для конечного вызова, то есть может определить следующий шаг рекурсии в том же фрейме стека вызовов.



Если вы хотите узнать больше о стеке вызовов в JavaScript, то можете прочесть книгу Томаса Хантера *Distributed Systems with Node.js* (О'Рейли) (<https://learning.oreilly.com/library/view/distributed-systems-with/9781492077282>), в которой содержится отличное введение.

Мы хотим использовать возможность TypeScript рекурсивно вызывать условные типы, чтобы создать допустимый строковый идентификатор из любой строки, удалив пробелы и недопустимые символы.

Сначала мы напишем вспомогательный тип, аналогичный `Trim<T>`, который удаляет все найденные пробелы:

```
type RemoveWhiteSpace<T extends string> = T extends `${infer A} ${infer B}`
  ? RemoveWhiteSpace<`${Uncapitalize<A>}${Capitalize<B>}`>
  : T;
```

Он проверяет, есть ли пробел, выводит строки перед пробелом и после него (которые могут быть пустыми строками) и снова вызывает тот же тип с вновь сформированным строковым типом. Кроме того, он убирает заглавные буквы из первого вывода и записывает второй вывод с заглавной буквы, чтобы создать строковый идентификатор, имя которого будет написано в верблюжем регистре.

И так до тех пор, пока не исчезнут все пробельные символы:

```
type Identifier = RemoveWhiteSpace<"Hello World!">; // "helloWorld!"
```

Далее мы хотим проверить, являются ли оставшиеся символы допустимыми. Мы снова используем рекурсию, чтобы взять строку допустимых символов, разделить ее на отдельные типы строк, содержащие только один символ, и создать версию с заглавной буквой и без нее:

```
type StringSplit<T extends string> = T extends `${infer Char}${infer Rest}`
  ? Capitalize<Char> | Uncapitalize<Char> | StringSplit<Rest>
  : never;
```

```
type Chars = StringSplit<"abcdefghijklmnopqrstuvwxy">;
// "a" | "A" | "b" | "B" | "c" | "C" | "d" | "D" | "e" | "E" |
// "f" | "F" | "g" | "G" | "h" | "H" | "i" | "I" | "j" | "J" |
// "k" | "K" | "l" | "L" | "m" | "M" | "n" | "N" | "o" | "O" |
// "p" | "P" | "q" | "Q" | "r" | "R" | "s" | "S" | "t" | "T" |
// "u" | "U" | "v" | "V" | "w" | "W" | "x" | "X" | "y" | "Y" |
// "z" | "Z"
```

Мы вычеркиваем первый найденный символ, записываем его с заглавной буквы, превращаем заглавные буквы в строчные и делаем то же самое с остальными, пока не останется строк. Обратите внимание, что эту рекурсию нельзя оптимизировать для конечных вызовов, поскольку мы помещаем рекурсивный вызов в типы объединения с результатами каждого шага рекурсии. Здесь мы достигли бы предела рекурсии, когда набрали 50 символов (жесткое ограничение, установленное компилятором TypeScript). С базовыми символами все в порядке!

Но мы столкнемся с первыми ограничениями, когда будем выполнять следующий шаг — создавать `Identifier`. Здесь мы проверяем наличие допустимых символов. Сначала вызываем тип `RemoveWhiteSpace<T>`, который позволяет избавиться от пробелов, а остальные перевести в верблюжий регистр. Затем мы проверяем результат на соответствие допустимым символам.

Как и в случае с `StringSplit<T>`, мы удаляем первый символ, но выполняем еще одну проверку типа в рамках выведения. Мы проверяем, является ли удаленный символ одним из допустимых. Затем мы получаем остальные. Мы снова объединяем ту же строку, но выполняем рекурсивную проверку оставшейся строки.

Если первый символ не является допустимым, то мы вызываем `CreateIdentifier<T>` с остальными:

```
type CreateIdentifier<T extends string> =
  RemoveWhiteSpace<T> extends `${infer A extends Chars}${infer Rest}`
    ? `${A}${CreateIdentifier<Rest>}`
// ^ Type instantiation is excessively deep and possibly infinite.(2589)_
: RemoveWhiteSpace<T> extends `${infer A}${infer Rest}`
  ? CreateIdentifier<Rest>
  : T;
```

И здесь мы сталкиваемся с первым пределом рекурсии. TypeScript выдает предупреждение, что реализация этого типа, возможно, бесконечна и чрезмерно глубока. Похоже, что, если мы используем рекурсивный вызов внутри литерального типа строкового шаблона, это может привести к ошибкам в стеке вызовов и сбою в работе. Таким образом, работа TypeScript нарушается. Здесь невозможно выполнить оптимизацию конечных вызовов.



`CreateIdentifier<T>` может по-прежнему выдавать корректные результаты, даже несмотря на ошибки TypeScript при написании вашего типа. Эти ошибки трудно обнаружить, поскольку они могут возникнуть в неожиданный для вас момент. Следите за тем, чтобы TypeScript не выдавал никаких результатов при возникновении ошибок.

Есть один способ обойти эту проблему. Чтобы оптимизацию конечного вызова можно было активировать, рекурсивный вызов должен быть автономным. Мы можем добиться этого, используя так называемый *прием аккумулятора*. Здесь мы передаем второй параметр типа `Acc`, который имеет тип `string` и создается как пустая строка. Мы используем его в качестве аккумулятора, в котором сохраняем промежуточный результат, передавая его снова и снова при следующем вызове:

```
type CreateIdentifier<T extends string, Acc extends string = ""> =
  RemoveWhiteSpace<T> extends `${infer A extends Chars}${infer Rest}`
    ? CreateIdentifier<Rest, `${Acc}${A}`>
  : RemoveWhiteSpace<T> extends `${infer A}${infer Rest}`
    ? CreateIdentifier<Rest, Acc>
  : Acc;
```

Таким образом, рекурсивный вызов снова выполняется сам по себе, а результатом является второй параметр. Завершая рекурсивный вызов, то есть выполняя ветвь, прерывающую рекурсию, мы возвращаем аккумулятор, поскольку это наш конечный результат:

```
type Identifier = CreateIdentifier<"Hello Wor!ld!">; // "helloWorld"
```

Возможно, существуют более хитроумные способы получения идентификаторов из любой строки, но учтите, что то же самое может случиться в любом сложном условном типе, в котором используется рекурсия. Использование приема аккумулятора — хороший способ решения подобных проблем.

6.6. Использование литералов шаблонов строк в качестве дискриминантов

Задача

Вы моделируете запросы к серверной части как конечный автомат, переходящий из состояния *ожидания* в состояние *ошибки* или *успеха*. Эти состояния должны работать для разных серверных запросов, но базовые типы должны быть одинаковыми.

Решение

Используйте литералы шаблонов строк в качестве дискриминантов для размеченного типа объединения.

Обсуждение

Способ, с помощью которого мы получаем данные от серверной части, всегда соответствует одной и той же схеме. Мы отправляем запрос, и он ожидает, что будет либо выполнен и вернет некие данные (это состояние успеха), либо отклонен и вернет сообщение об ошибке. Например, для входа пользователя в систему все возможные состояния могут выглядеть следующим образом:

```
type UserRequest =  
  | {  
    state: "USER_PENDING";  
  }  
  | {  
    state: "USER_ERROR";  
    message: string;  
  }  
  | {  
    state: "USER_SUCCESS";  
    data: User;  
  };
```

Когда мы получаем заказ пользователя, нам доступны те же состояния. Единственное различие состоит в смысле состояния успеха и названиях каждого состояния, которые соответствуют типу запроса:

```
type OrderRequest =  
  | {  
    state: "ORDER_PENDING";  
  }  
  | {  
    state: "ORDER_ERROR";  
    message: string;  
  }
```

```
| {  
  state: "ORDER_SUCCESS";  
  data: Order;  
};
```

Взаимодействуя с глобальным механизмом обработки состояний, таким как Redux (<https://redux.js.org/>), мы хотим дифференцировать их, используя идентификаторы, подобные этому. Мы по-прежнему надеемся ограничить их, используя соответствующие типы состояний!

TypeScript позволяет создавать размеченные типы объединений, где дискриминантом является литеральный тип строкового шаблона. Таким образом, мы можем суммировать все возможные серверные запросы, используя один и тот же шаблон:

```
type Pending = {  
  state: `${Uppercase<string>}_PENDING`;  
};  
  
type Err = {  
  state: `${Uppercase<string>}_ERROR`;  
  message: string;  
};  
  
type Success = {  
  state: `${Uppercase<string>}_SUCCESS`;  
  data: any;  
};  
  
type BackendRequest = Pending | Err | Success;
```

Эта возможность дает нам определенное преимущество. Мы знаем, что свойство `state` каждого элемента типа объединения должно начинаться со строки, написанной в верхнем регистре, за которой следуют символ подчеркивания и соответствующее состояние в виде строки. И мы можем сузить это свойство до подтипов, как уже делали:

```
function execute(req: BackendRequest) {  
  switch (req.state) {  
    case "USER_PENDING":  
      // req: Pending  
      console.log("Login pending...");  
      break;  
    case "USER_ERROR":  
      // req: Err  
      throw new Error(`Login failed: ${req.message}`);  
    case "USER_SUCCESS":  
      // req: Success  
      login(req.data);  
      break;  
    case "ORDER_PENDING":  
      // req: Pending
```



```

    console.log("Fetching orders pending");
    break;
  case "ORDER_ERROR":
    // req: Err
    throw new Error(`Fetching orders failed: ${req.message}`);
  case "ORDER_SUCCESS":
    // req: Success
    displayOrder(req.data);
    break;
}
}
}

```

Иметь весь набор строк в качестве первой части дискриминанта может быть немного избыточно. Мы можем выделить подмножество известных запросов и использовать типы манипуляций со строками, чтобы получить правильные подтипы:

```

type RequestConstants = "user" | "order";

type Pending = {
  state: `${Uppercase<RequestConstants>}_PENDING`;
};

type Err = {
  state: `${Uppercase<RequestConstants>}_ERROR`;
  message: string;
};

type Success = {
  state: `${Uppercase<RequestConstants>}_SUCCESS`;
  data: any;
};

```

Вот как можно избавиться от опечаток! Есть и еще более удачная возможность: предположим, что храним все данные в глобальном объекте состояния типа `Data`. Из него мы можем получить все возможные типы `BackendRequest`. Используя `keyof Data`, мы получим строковые ключи, составляющие состояние `BackendRequest`:

```

type Data = {
  user: User | null;
  order: Order | null;
};

type RequestConstants = keyof Data;

type Pending = {
  state: `${Uppercase<RequestConstants>}_PENDING`;
};

type Err = {
  state: `${Uppercase<RequestConstants>}_ERROR`;
  message: string;
};

```

Этот код хорошо работает для `Pending` и `Err`, но в случае `Success` мы хотим, чтобы фактический тип данных был связан с `"user"` или `"order"`.

Первым вариантом было бы использовать индексный доступ, чтобы получить правильные типы для свойства `data` из `Data`:

```
type Success = {
  state: `${Uppercase<RequestConstants>}_SUCCESS`;
  data: NonNullable<Data[RequestConstants]>;
};
```



`NonNullable<T>` избавляется от `null` и `undefined` в типе объединения. При установленном флаге компилятора `strictNullChecks` из всех типов исключаются значения `null` и `undefined`. Это значит, вам нужно вручную добавлять их, если у вас есть нулевые состояния, и вручную исключать их, если вы хотите быть уверены, что их нет.

Но это будет означать, что `data` может быть как `User`, так и `Order` для всех серверных запросов, а также других, если мы добавим новые. Чтобы не разрывать соединение между идентификатором и связанным с ним типом данных, мы сопоставляем все `RequestConstants`, создаем объекты состояния, а затем снова используем индексный доступ к `RequestConstants` для создания типа объединения:

```
type Success = {
  [K in RequestConstants]: {
    state: `${Uppercase<K>}_SUCCESS`;
    data: NonNullable<Data[K]>;
  };
}[RequestConstants];
```

`Success` теперь равен созданному вручную типу объединения:

```
type Success = {
  state: "USER_SUCCESS";
  data: User;
} | {
  state: "ORDER_SUCCESS";
  data: Order;
};
```

Вариативные типы кортежей

Типы кортежей — это массивы с фиксированной длиной, в которых определен тип каждого элемента. Кортежи широко используются в таких библиотеках, как React, поскольку их легко деструктурировать и присваивать имена элементам, но за пределами React они тоже получили признание как хорошая альтернатива объектам.

Вариативный тип кортежа имеет одинаковые свойства (определенную длину и тип каждого элемента), но его *точная структура* еще не определена. По сути, такие типы сообщают системе, что будут какие-то элементы, но мы пока еще не знаем, какие именно. Они являются обобщенными типами и предназначены для замены реальными типами.

То, что кажется довольно скучной функцией, становится гораздо более интересным, когда мы понимаем, что кортежные типы можно использовать и для описания сигнатур функций, поскольку кортежи можно распределить по вызовам функций в качестве аргументов. Это значит, что с помощью переменных типов кортежей можно получить максимальную информацию о функциях и вызовах функций, а также о функциях, которые принимают функции в качестве параметров.

В этой главе приведено множество примеров того, как использовать типы кортежей с переменным количеством элементов для описания нескольких сценариев, в которых мы используем функции в качестве параметров и от которых требуется получить максимум информации. Без типов кортежей с переменным количеством элементов эти сценарии было бы трудно или вовсе невозможно разработать. Прочитав эту главу, вы поймете, что типы кортежей с переменным количеством элементов являются ключевой особенностью паттернов функционального программирования.

7.1. Типизация функции `concat`

Задача

У вас есть функция `concat`, которая принимает два массива и объединяет их. Вы хотите иметь точные типы, но использование перегрузок функций — слишком трудоемкий процесс.

Решение

Используйте вариативные типы кортежей.

Обсуждение

Функция `concat` — вспомогательная; она берет два массива и объединяет их. Она использует расширение массивов и является короткой, красивой и легко читаемой:

```
function concat(arr1, arr2) {
  return [...arr1, ...arr2];
}
```

Создание типов для этой функции может оказаться сложной задачей, особенно если вы чего-то ждете от своих типов. Передача параметров в двух массивах проста, но как должен выглядеть тип возвращаемого значения? Вас устраивает один тип возвращаемого массива или вы хотите знать типы каждого элемента в этом массиве?

Остановимся на последнем варианте: нам нужны кортежи, чтобы знать тип каждого элемента, который мы передаем в эту функцию. Правильная типизация такой функции, учитывающей все возможные крайние случаи, вызвала бы огромное количество перегрузок:

```
// 7 перегрузок для пустого второго массива
function concat(arr1: [], arr2: []): [];
function concat<A>(arr1: [A], arr2: []): [A];
function concat<A, B>(arr1: [A, B], arr2: []): [A, B];
function concat<A, B, C>(arr1: [A, B, C], arr2: []): [A, B, C];
function concat<A, B, C, D>(arr1: [A, B, C, D], arr2: []): [A, B, C, D];
function concat<A, B, C, D, E>(
  arr1: [A, B, C, D, E],
  arr2: []
): [A, B, C, D, E];
function concat<A, B, C, D, E, F>(
  arr1: [A, B, C, D, E, F],
  arr2: []
): [A, B, C, D, E, F];
// Еще 7 для arr2, имеющего один элемент
function concat<A2>(arr1: [], arr2: [A2]): [A2];
function concat<A1, A2>(arr1: [A1], arr2: [A2]): [A1, A2];
function concat<A1, B1, A2>(arr1: [A1, B1], arr2: [A2]): [A1, B1, A2];
function concat<A1, B1, C1, A2>(
  arr1: [A1, B1, C1],
  arr2: [A2]
): [A1, B1, C1, A2];
function concat<A1, B1, C1, D1, A2>(
  arr1: [A1, B1, C1, D1],
```

```

    arr2: [A2]
  ): [A1, B1, C1, D1, A2];
function concat<A1, B1, C1, D1, E1, A2>(
  arr1: [A1, B1, C1, D1, E1],
  arr2: [A2]
): [A1, B1, C1, D1, E1, A2];
function concat<A1, B1, C1, D1, E1, F1, A2>(
  arr1: [A1, B1, C1, D1, E1, F1],
  arr2: [A2]
): [A1, B1, C1, D1, E1, F1, A2];
// и т. д. и т. п.

```

И этот код учитывает только массивы, содержащие до шести элементов. Комбинации для типизации подобной функции с перегрузками утомительны. Но есть более простой способ: использовать вариативные типы кортежей.

Тип кортежа в TypeScript — это массив со следующими характеристиками:

- длина массива определена;
- тип каждого элемента известен (и необязательно должен быть одинаковым).

Например, это тип кортежа:

```

type PersonProps = [string, number];

const [name, age]: PersonProps = ['Stefan', 37];

```

Как было сказано выше, вариативный тип кортежа имеет одинаковые свойства (определенную длину и тип каждого элемента), но его точная структура еще не определена. Тип и длина нам пока неизвестны, поэтому мы можем использовать вариативные типы кортежей только в обобщенных типах:

```

type Foo<T extends unknown[]> = [string, ...T, number];

type T1 = Foo<[boolean]>; // [string, boolean, number]
type T2 = Foo<[number, number]>; // [string, number, number, number]
type T3 = Foo<[]>; // [string, number]

```

Это похоже на элементы `rest` в функциях, но существенное различие состоит в том, что вариативные типы кортежей могут неоднократно встречаться в любом месте кортежа:

```

type Bar<
  T extends unknown[],
  U extends unknown[]
> = [...T, string, ...U];

type T4 = Bar<[boolean], [number]>; // [boolean, string, number]
type T5 = Bar<[number, number], [boolean]>; // [number, number, string, boolean]
type T6 = Bar<[], []>; // [string]

```

Когда мы применяем этот код к функции `concat`, нам приходится вводить два параметра обобщенного типа, по одному для каждого массива. Оба параметра должны быть ограничены массивами. Затем мы можем создать тип возвращаемого значения, который объединяет оба типа массивов во вновь созданный тип кортежа:

```
function concat<T extends unknown[], U extends unknown[]>(
  arr1: T,
  arr2: U
): [...T, ...U] {
  return [...arr1, ...arr2];
}

// const test: (string | number)[]
const test = concat([1, 2, 3], [6, 7, "a"]);
```

Синтаксис прекрасен; он очень похож на реальную конкатенацию в JavaScript. Результат также очень хорош: мы получаем `(string | number)[]`, а это уже то, с чем можно работать.

Но мы работаем с кортежными типами. Если мы хотим знать *точно*, какие элементы конкатенируем, то должны преобразовать типы массивов в типы кортежей, распространив обобщенный тип массива на тип кортежа:

```
function concat<T extends unknown[], U extends unknown[]>(
  arr1: [...T],
  arr2: [...U]
): [...T, ...U] {
  return [...arr1, ...arr2];
}
```

И при этом мы также получаем тип кортежа в ответ:

```
// const test: [number, number, number, number, number, string]
const test = concat([1, 2, 3], [6, 7, "a"]);
```

Хорошая новость — мы ничего не теряем. Если мы передаем массивы, каждый элемент которых нам неизвестен заранее, то все равно получим в ответ типы массивов:

```
declare const a: string[]
declare const b: number[]

// const test: (string | number)[]
const test = concat(a, b);
```

Возможность описать такое поведение в одном типе, безусловно, обеспечивает удобочитаемость и гораздо большую гибкость, чем написание всех возможных комбинаций в перегрузке функции.

7.2. Типизация функции promisify

Задача

Вы хотите преобразовать функции обратного вызова в промисы (Promises) и сделать возможной их идеальную типизацию.

Решение

Аргументы функции представляют собой типы кортежей. Сделайте их обобщенными типами, используя вариативные типы кортежей.

Обсуждение

До появления промисов в JavaScript было очень распространено асинхронное программирование с использованием обратных вызовов. Функции обычно принимали список аргументов, за которыми следовала функция обратного вызова, которая выполнялась после получения результатов, например функции для загрузки файла или выполнения очень упрощенного HTTP-запроса:

```
function loadFile(
  filename: string,
  encoding: string,
  callback: (result: File) => void
) {
  // TODO
}

loadFile("./data.json", "utf-8", (result) => {
  // что-то делаем с файлом
});

function request(url: URL, callback: (result: JSON) => void) {
  // TODO
}

request("https://typescript-cookbook.com", (result) => {
  // TODO
});
```

Оба варианта строятся по одной и той же схеме: сначала аргументы, затем обратный вызов с результатом. Это работает, но может быть неудобно, если у вас много асинхронных вызовов, которые приводят к обратным вызовам внутри обратных вызовов — явлению, которое известно под названием «пирамида судьбы» (<https://oreil.ly/Ye3Qr>):

```
loadFile("./data.txt", "utf-8", (file) => {
  // псевдоAPI
  file.readText((url) => {
```

```

    request(url, (data) => {
      // что-то делаем с данными
    })
  })
})

```

Промисы заботятся об этом. Они не только находят способ связывать асинхронные вызовы в цепочку, вместо того чтобы использовать вложенность, но и являются плюсом для `async/await`, позволяя писать асинхронный код в синхронной форме:

```

loadFilePromise("./data.txt", "utf-8")
  .then((file) => file.text())
  .then((url) => request(url))
  .then((data) => {
    // что-то делаем с данными
  });

// с использованием async/await
const file = await loadFilePromise("./data.txt", "utf-8");
const url = await file.text();
const data = await request(url);
// что-то делаем с данными

```

Код теперь гораздо приятнее! К счастью, есть возможность преобразовать каждую функцию, которая придерживается паттерна обратного вызова, в `Promise`. Мы хотим создать функцию `promisify`, которая будет делать это за нас автоматически:

```

function promisify(fn: unknown): Promise<unknown> {
  // Будет реализовано
}

const loadFilePromise = promisify(loadFile);
const requestPromise = promisify(request);

```

Но как это типизировать? С помощью вариативных типов кортежей!

Каждый заголовок функции может быть описан как тип кортежа. Например,

```
declare function hello(name: string, msg: string): void;
```

это то же самое, что:

```
declare function hello(...args: [string, string]): void;
```

И мы получаем очень много возможностей его определения:

```

declare function h(a: string, b: string, c: string): void;
// эквивалентно
declare function h(a: string, b: string, ...r: [string]): void;
// эквивалентно
declare function h(a: string, ...r: [string, string]): void;
// эквивалентно
declare function h(...r: [string, string, string]): void;

```


Это также известно как *элемент rest*, который есть в JavaScript. Он позволяет определять функции с почти неограниченным списком аргументов, последний элемент которого — `rest` — поглощает все лишние аргументы.

Например, эта обобщенная функция кортежа принимает список аргументов любого типа и создает из него кортеж:

```
function tuple<T extends any[]>(...args: T): T {
  return args;
}

const numbers: number[] = getArrayOfNumbers();
const t1 = tuple("foo", 1, true); // [string, number, boolean]
const t2 = tuple("bar", ...numbers); // [string, ...number[]]
```

Дело в том, что элементы `rest` всегда должны быть последними. В JavaScript невозможно определить почти бесконечный список аргументов где-то посередине. Однако используя вариативные типы кортежей, мы можем сделать это в TypeScript!

Еще раз рассмотрим функции `loadFile` и `request`. Если бы мы описывали параметры обеих функций в виде кортежей, то они выглядели бы так:

```
function loadFile(...args: [string, string, (result: File) => void]) {
  // TODO
}
function request2(...args: [URL, (result: JSON) => void]) {
  // TODO
}
```

Поищем сходство. Оба кортежа завершаются обратным вызовом с различным типом результата. Мы можем согласовать типы для обоих обратных вызовов, заменив вариативный тип обобщенным. Позже, в процессе использования, мы заменяем обобщенные типы фактическими. Таким образом, `JSON` и `File` становятся параметрами обобщенного типа `Res`.

Теперь о параметрах *перед* `Res`. Возможно, они совершенно разные, но даже у них есть что-то общее: они являются элементами в составе кортежа. Для этого требуется вариативный тип кортежа. Мы знаем, что они будут иметь конкретную длину и конкретные типы, но сейчас просто используем для них заполнитель. Назовем их `Args`.

Таким образом, тип функции, описывающий обе сигнатуры функций, может выглядеть следующим образом:

```
type Fn<Args extends unknown[], Res> = (
  ...args: [...Args, (result: Res) => void]
) => void;
```

Опробуйте свой новый тип в деле:

```
type LoadFileFn = Fn<[string, string], File>;
type RequestFn = Fn<[URL], JSON>;
```

Это именно то, что нам нужно для функции `promisify`. Мы можем извлечь все необходимые параметры — те, что были до обратного вызова, и тип результата — и привести их в новый порядок.

Начнем с того, что добавим вновь созданный тип функции непосредственно в сигнатуру функции `promisify`:

```
function promisify<Args extends unknown[], Res>(
  fn: (...args: [...Args, (result: Res) => void]) => void
): (...args: Args) => Promise<Res> {
  // скоро
}
```

Теперь `promisify` выглядит так:

- есть два параметра обобщенного типа: `Args`, который должен быть массивом (или кортежем), и `Res`;
- параметром `promisify` является функция, первые аргументы которой — элементы `Args`, а последний аргумент — функция с параметром типа `Res`;
- `promisify` возвращает функцию, которая принимает `Args` в качестве параметров и возвращает `Promise` типа `Res`.

Если вы попытаете ввести новые типы для `promisify`, то увидите, что получаете именно тот тип, который вам нужен.

Но дела обстоят еще лучше. Если посмотреть на сигнатуру функции, то абсолютно понятно, какие аргументы мы ожидаем, даже если они являются вариативными и будут заменены на реальные типы. Мы можем использовать те же типы для реализации `promisify`:

```
function promisify<Args extends unknown[], Res>(
  fn: (...args: [...Args, (result: Res) => void]) => void
): (...args: Args) => Promise<Res> {
  return function (...args: Args) { ❶
    return new Promise((resolve) => { ❷
      function callback(res: Res) { ❸
        resolve(res);
      }
      fn.call(null, ...[...args, callback]); ❹
    });
  };
}
```

Итак, что же делает эта функция?

- ❶ Мы возвращаем функцию, которая принимает все параметры, кроме обратного вызова.
- ❷ Эта функция возвращает только что созданный `Promise`.

- ❸ У нас еще нет обратного вызова, поэтому нужно его создать. Что он делает? Вызывает функцию `resolve` из `Promise`, выдавая результат.
- ❹ То, что было разделено, нужно снова объединить! Мы добавляем обратный вызов к аргументам и вызываем исходную функцию.

Вот и все. Мы получаем работающую функцию `promisify` для функций, которые придерживаются паттерна обратного вызова. Идеально типизированную. И мы даже сохранили имена параметров.

7.3. Типизация функции `curry`

Задача

Вы пишете функцию `curry`. *Каррирование* (`currying`) — это прием, который позволяет преобразовать функцию, принимающую несколько аргументов, в последовательность функций, каждая из которых принимает один аргумент.

Вы хотите предоставить отличные типы.

Решение

Комбинируйте условные типы с вариативными типами кортежей, всегда удаляя первый параметр.

Обсуждение

Каррирование — очень известный прием функционального программирования. Лежащая в его основе концепция называется «частичное применение аргументов функции». Мы используем ее, чтобы максимизировать повторное использование функций. Программа типа «Hello, World!» в каррировании реализует функцию `add`, которая может частично применить второй аргумент позже:

```
function add(a: number, b: number) {  
  return a + b;  
}
```

```
const curriedAdd = curry(add); // преобразование: (a: number) => (b: number) => number  
const add5 = curriedAdd(5); // использование первого аргумента. (b: number) => number  
const result1 = add5(2); // второй аргумент. Результат: 7  
const result2 = add5(3); // второй аргумент. Результат: 8
```

Хотя этот пример на первый взгляд кажется условным, его метод оказывается полезным при работе с длинными списками аргументов. Следующая обобщенная функция либо добавляет, либо удаляет классы в `HTMLElement`.

Мы можем подготовить все, кроме заключительного события:

```
function applyClass(
  this: HTMLElement, // только для TypeScript
  method: "remove" | "add",
  className: string,
  event: Event
) {
  if (this === event.target) {
    this.classList[method](className);
  }
}

const applyClassCurried = curry(applyClass); // преобразование
const removeToggle = applyClassCurried("remove")("hidden");

document.querySelector(".toggle")?.addEventListener("click", removeToggle);
```

Таким образом, мы можем повторно использовать `removeToggle` для нескольких событий в ряде элементов. Кроме того, мы можем использовать `applyClass` для многих других ситуаций.

Применение каррирования — фундаментальная концепция языка программирования Haskell, отдающего дань уважения математику Хаскеллу Бруксу Карри, в честь которого названы как язык программирования, так и этот прием. В языке Haskell каждая операция является каррированной, и программисты используют эту концепцию довольно активно.

JavaScript в значительной степени заимствован из функциональных языков программирования, и благодаря встроенной в него функции привязки можно реализовать частичное применение:

```
function add(a: number, b: number, c: number) {
  return a + b + c;
}

// Частичное применение
const partialAdd5And3 = add.bind(this, 5, 3);
const result = partialAdd5And3(2); // третий аргумент
```

Функции — полноправные элементы языка JavaScript, поэтому мы можем создать функцию `curry`, которая принимает функцию в качестве аргумента и собирает все аргументы перед ее выполнением:

```
function curry(fn) {
  let curried = (...args) => {
    // если вы не указали достаточно аргументов
    if (fn.length !== args.length) {
      // частично применяем аргументы
      // и возвращаем сборщик данных
      return curried.bind(null, ...args);
    }
  }
}
```

```

    // в ином случае вызываем все функции
    return fn(...args);
  };
  return curried;
}

```

Хитрость состоит в том, что каждая функция хранит количество определенных аргументов в своем свойстве `length`. Таким образом, мы можем рекурсивно собрать все необходимые аргументы, прежде чем применить их к переданной функции.

Так чего же не хватает? Типов! Создадим тип, который будет работать с паттерном каррирования, где каждая функция последовательности может принимать ровно один аргумент. Для этого мы создадим условный тип, который будет выполнять действие, обратное тому, что делает функция `curried` внутри функции `curry`: удалять аргументы.

Итак, создадим тип `Curried<F>`. Первое, что нужно сделать, — это проверить, действительно ли данный тип является функцией:

```

type Curried<F> = F extends (...args: infer A) => infer R
  ? /* to be done */
  : never; // это не функция, такого не должно быть

```

Мы также определяем аргументы как `A` и тип возвращаемого значения как `R`. Далее удаляем первый параметр как `F`, а все остальные параметры сохраняем в `L` (напоследок):

```

type Curried<F> = F extends (...args: infer A) => infer R
  ? A extends [infer F, ...infer L]
    ? /* должно быть сделано */
    : () => R
  : never;

```

Если аргументов нет, то мы возвращаем функцию, которая не принимает аргументы. Последняя проверка: мы проверяем, являются ли остальные параметры пустыми. Это означает, что мы достигли конца удаления аргументов из списка аргументов:

```

type Curried<F> = F extends (...args: infer A) => infer R
  ? A extends [infer F, ...infer L]
    ? L extends []
      ? (a: F) => R
      : (a: F) => Curried<(...args: L) => R>
    : () => R
  : never;

```

Если какие-то параметры остаются, то мы снова вызываем тип `Curried`, но уже с оставшимися параметрами. Таким образом, мы поэтапно избавляемся от параметров, и если внимательно присмотреться, то можно увидеть, что процесс практически идентичен тому, что мы делаем в функции `curried`. Там, где мы деконструируем параметры в `Curried<F>`, мы снова собираем их в `curried(fn)`.

Когда тип будет готов, добавим его к `curry`:

```
function curry<F extends Function>(fn: F): Curried<F> {
  let curried: Function = (...args: any) => {
    if (fn.length !== args.length) {
      return curried.bind(null, ...args);
    }
    return fn(...args);
  };
  return curried as Curried<F>;
}
```

Нам нужно несколько утверждений и `any` из-за гибкой природы типа. Но с помощью ключевых слов `as` и `any` мы помечаем, какие части считаются небезопасными типами.

Вот и все! Можем двигаться дальше!

7.4. Типизация гибкой функции `curry`

Задача

Функция `curry` из рецепта 7.3 дает возможность передавать произвольное количество аргументов, но ваша типизация позволяет принимать только один аргумент за раз.

Решение

Расширьте типизацию, чтобы создать перегрузки функций для всех возможных комбинаций кортежей.

Обсуждение

В рецепте 7.3 мы получили типы функций, которые позволяют применять аргументы функции по одному:

```
function addThree(a: number, b: number, c: number) {
  return a + b + c;
}

const adder = curried(addThree);
const add7 = adder(5)(2);
const result = add7(2);
```

Однако сама функция `curry` может принимать произвольный список аргументов:

```
function addThree(a: number, b: number, c: number) {
  return a + b + c;
}
```

```
const adder = curried(addThree);
const add7 = adder(5, 2); // вот в чем разница
const result = add7(2);
```

Это позволяет нам работать с теми же сценариями использования, но с гораздо меньшим количеством вызовов функций. Итак, адаптируем наши типы, чтобы в полной мере воспользоваться всеми преимуществами функции `curry`.



Этот пример хорошо показывает, как система типов работает в качестве тонкого слоя поверх JavaScript. Добавляя утверждения и `any` в нужных местах, мы определяем, как должна работать `curry`, в то время как сама функция становится гораздо более гибкой. Обратите внимание: определяя сложные типы поверх сложной функциональности, вы можете запутаться, и только от вас зависит, как типы будут работать в итоге. Протестируйте их соответствующим образом.

Наша цель — создать тип, который сможет создавать все возможные сигнатуры функций для каждого частичного применения. Для функции `addThree` все возможные типы будут выглядеть так:

```
type Adder = (a: number) => (b: number) => (c: number) => number;
type Adder = (a: number) => (b: number, c: number) => number;
type Adder = (a: number, b: number) => (c: number) => number;
type Adder = (a: number, b: number, c: number) => number;
```

На рис. 7.1 показаны все возможные графы вызовов.

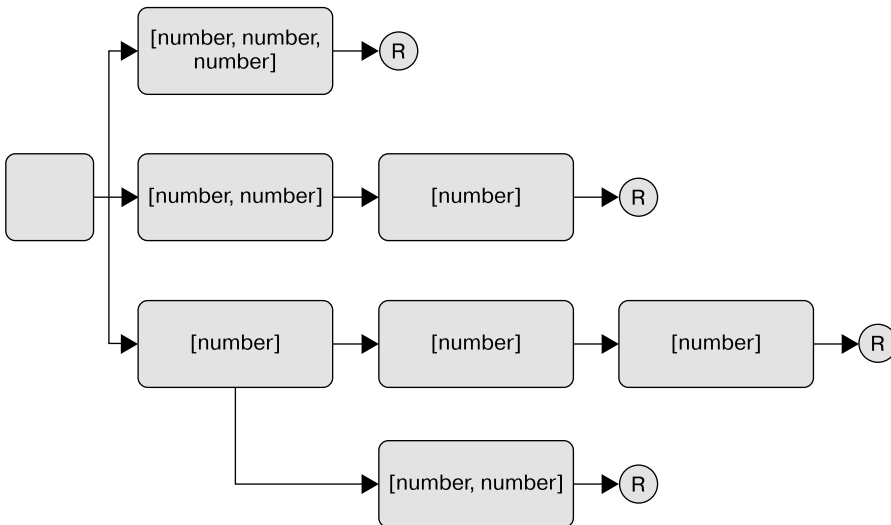


Рис. 7.1. Граф, показывающий все возможные комбинации вызовов функции `addThree` при ее выполнении; есть три начальные ветви, и возможна четвертая

Первое, что мы сделаем, — это немного адаптируем способ вызова вспомогательного типа `Curried`. В исходном типе мы выполняем выведение аргументов функции и типов возвращаемых значений во вспомогательном типе. Теперь нужно перенести возвращаемое значение через несколько вызовов типов, поэтому мы извлекаем возвращаемый тип и аргументы непосредственно в функции `curry`:

```
function curry<A extends any[], R extends any>(
  fn: (...args: A) => R
): Curried<A, R> {
  // см. ранее, мы не меняем реализацию
}
```

Далее мы переопределим тип `Curried`. Теперь он имеет два параметра обобщенного типа: `A` для аргументов и `R` для типа возвращаемого значения. Для начала мы проверяем, содержат ли аргументы элементы кортежа. Мы извлекаем первый элемент `F` и все оставшиеся элементы `L`. Если элементов не осталось, то возвращаем тип возвращаемого значения `R`:

```
type Curried<A extends any[], R extends any> = A extends [infer F, ...infer L]
  ? // должно быть сделано
  : R;
```

Невозможно извлечь несколько кортежей с помощью оператора `rest`. Поэтому нам все равно нужно удалить первый элемент и собрать оставшиеся в `L`. Но это нормально; нам нужен хотя бы *один* параметр, чтобы сделать частичное применение эффективным.

Находясь в ветви `true`, мы создаем определения функций. В предыдущем примере мы вернули функцию, которая возвращает рекурсивный вызов; теперь нам нужно предоставить все возможности частичного применения.

Аргументы функций — не что иное, как типы кортежей (см. рецепт 7.2), поэтому аргументы перегрузок функций можно описать как тип объединения кортежей. Тип `Overloads` принимает кортеж аргументов функции и создает все случаи частичного применения:

```
type Overloads<A extends any[]> = A extends [infer A, ...infer L]
  ? [A] | [A, ...Overloads<L>] | []
  : [];
```

Если мы передаем кортеж, то получаем объединение, начинающееся с пустого кортежа и затем расширяющееся до одного аргумента, затем до двух аргументов и т. д., вплоть до кортежа, содержащего все аргументы:

```
// type Overloaded = [] | [string, number, string] | [string] | [string, number]
type Overloaded = Overloads<[string, number, string]>;
```


Теперь, когда мы можем определить все перегрузки, мы берем оставшиеся аргументы из списка аргументов исходной функции и создаем все возможные вызовы функции, которые также содержат первый аргумент:

```
type Curried<A extends any[], R extends any> = A extends [infer F, ...infer L]
  ? <K extends Overloads<L>>(
    arg: F,
    ...args: K
  ) => /* to be done */
  : R;
```

Применительно к предыдущему примеру `addThree` эта часть создаст первый аргумент `F` как `number`, а затем объединит его с `[]`, `[number]` и `[number, number]`.

Теперь о типе возврата. Это снова рекурсивный вызов `Curried`, как и в рецепте 7.2. Помните, что мы связываем функции в последовательность. Мы передаем один и тот же тип возвращаемого значения — в конце концов, мы должны его получить, — но требуется передать и все оставшиеся аргументы, которые мы не распределили в перегрузках функций. Таким образом, если мы вызываем `addThree` только с `number`, то два оставшихся числа должны быть аргументами следующей итерации `Curried`. Так мы создаем дерево возможных вызовов.

Чтобы перейти к возможным комбинациям, нам нужно удалить из оставшихся аргументов те, которые мы уже описали в сигнатуре функции. Вспомогательный тип `Remove<T, U>` проходит через оба кортежа и удаляет из каждого по одному элементу, пока в одном из двух кортежей не закончатся элементы:

```
type Remove<T extends any[], U extends any[]> = U extends [infer _, ...infer UL]
  ? T extends [infer _, ...infer TL]
    ? Remove<TL, UL>
    : never
  : T;
```

Подключив это к `Curried`, мы получим окончательный результат:

```
type Curried<A extends any[], R extends any> = A extends [infer F, ...infer L]
  ? <K extends Overloads<L>>(
    arg: F,
    ...args: K
  ) => Curried<Remove<L, K>, R>
  : R;
```

`Curried<A, R>` теперь создает тот же граф вызовов, который описан на рис. 7.1, но обеспечивает гибкость для всех возможных функций, которые мы передаем в `curry`.

Благодарим пользователя GitHub Акире Мацузаки, который в своем решении `Type Challenges` предоставил недостающий фрагмент, позволивший обеспечить максимальную гибкость при надлежащей безопасности типов.

7.5. Типизация простейшей функции `curry`

Задача

Функции `curry` и их типизация впечатляют, но имеют множество нюансов. Есть ли более простые решения?

Решение

Создайте функцию `curry`, выполнив всего один последовательный шаг. TypeScript может самостоятельно определить нужные типы.

Обсуждение

В последней части трилогии о `curry` я хочу поговорить о том, что мы видели в рецептах 7.3 и 7.4. Мы создали очень сложные типы, которые работают почти как настоящая реализация благодаря возможностям метапрограммирования TypeScript. И хотя результаты впечатляют, есть несколько нюансов, о которых стоит упомянуть.

- Способы реализации типов в рецептах 7.3 и 7.4 различаются слегка, но результаты — весьма существенно! Тем не менее функция `curry` остается неизменной. Единственный способ, с помощью которого она работает, — это использование аргументов `any` и утверждения типа для типа возвращаемого значения. Это значит, что мы фактически отключаем проверку типов, заставляя TypeScript придерживаться нашего подхода. Это замечательно, что TypeScript может это делать, и иногда такие действия необходимы (например, при создании новых объектов), но они могут иметь неприятные последствия, особенно когда и реализация, и типы становятся очень сложными.

Тесты как для типов, так и для реализации являются обязательными. О тестировании типов мы поговорим в рецепте 12.4.

- Вы теряете информацию. Особенно при каррировании сохранять имена аргументов нужно для того, чтобы знать, какие аргументы уже были применены. Решения в предыдущих рецептах не могли сохранять имена аргументов, а по умолчанию использовали звучащие обобщенно `a` или `args`. Если вашими типами аргументов являются, например, все строки, то вы не сможете сказать, какую строку записываете в данный момент.
- В то время как результат, полученный в рецепте 7.4, позволяет выполнить правильную проверку типа, автозаполнение ограничено из-за особенностей типа. Вы знаете только, что второй аргумент необходим лишь в момент его ввода. Одна из главных особенностей TypeScript — предоставление вам необходимых инструментов и информации, которые сделают вашу работу более продуктивной. Гибкий тип `Curried` снова сводит вашу производительность к догадкам.

Опять же, эти варианты впечатляют, но нельзя отрицать, что они требуют значительных компромиссов. В связи с этим возникает вопрос: а стоит ли вообще идти на них? Думаю, все действительно зависит от того, чего вы пытаетесь достичь.

Каррирование и частичное применение привели к образованию двух лагерей. Представители первого любят шаблоны функционального программирования и пытаются использовать функциональные возможности JavaScript по максимуму. Эти люди хотят использовать частичное применение как можно чаще и нуждаются в расширенных функциях каррирования. Сторонники другой точки зрения видят, что шаблоны функционального программирования эффективны в определенных ситуациях — например, при ожидании, пока конечный параметр предоставит одну и ту же функцию нескольким событиям. Эти разработчики часто предпочитают применять шаблоны как можно больше, а затем предоставлять остальные функции на втором этапе.

До сих пор мы имели дело только с первым лагерем. Если вы относитесь ко второму, то вам, скорее всего, нужна только функция каррирования, которая частично применяет несколько параметров, так что вы можете передать остальные на втором этапе: никакой последовательности параметров одного аргумента и никакого гибкого применения сколь угодно большого количества аргументов. Идеальный интерфейс выглядел бы следующим образом:

```
function applyClass(  
  this: HTMLElement, // только для TypeScript  
  method: "remove" | "add",  
  className: string,  
  event: Event  
) {  
  if (this === event.target) {  
    this.classList[method](className);  
  }  
}  
  
const removeToggle = curry(applyClass, "remove", "hidden");  
  
document.querySelector("button")?.addEventListener("click", removeToggle);
```

Функция `curry` принимает в качестве аргумента другую функцию `f`, затем последовательность `t` параметров `f`. Она возвращает функцию, принимающую оставшиеся параметры `u` функции `f`, которая вызывает `f` со всеми возможными параметрами. В JavaScript функция может выглядеть так:

```
function curry(f, ...t) {  
  return (...u) => f(...t, ...u);  
}
```

Благодаря операторам `rest` и `spread` `curry` превращается в односложное предложение. Теперь типизируем эту функцию! Нам придется использовать обобщенные типы, поскольку мы имеем дело с параметрами, которые нам еще неизвестны. Существует тип возвращаемого значения `R`, а также обе части аргументов

функции, `T` и `U`. Последние представляют собой вариативные типы кортежей и должны быть определены как таковые.

Если параметры обобщенного типа `T` и `U` — это аргументы `f`, то тип для `f` выглядит так:

```
type Fn<T extends any[], U extends any[]> =
  (...args: [...T, ...U]) => any;
```

Аргументы функции могут быть описаны как кортежи, и здесь мы говорим, что эти аргументы функции должны быть разделены на две части. Добавим этот тип в `curry` и используем другой параметр обобщенного типа для типа возвращаемого значения `R`:

```
function curry<T extends any[], U extends any[], R>(
  f: (...args: [...T, ...U]) => R,
  ...t: T
) {
  return (...u: U) => f(...t, ...u);
}
```

И это все типы, которые нам нужны: простые, понятные и очень похожие на реальную реализацию. Используя несколько вариативных типов кортежей, TypeScript предоставляет следующие возможности.

- 100%-ная безопасность типов. TypeScript напрямую выводит обобщенные типы из вашего использования, и они верны. Никаких сложных типов, созданных с помощью условных типов и рекурсии.
- Автозаполнение для всех возможных решений. Как только вы добавите символ `a`, чтобы объявить о следующем шаге ваших аргументов, TypeScript адаптирует типы и даст вам подсказку о том, чего ожидать.
- Сохранение информации. Мы не создаем новых типов, поэтому TypeScript сохраняет метки исходного типа, и мы знаем, каких аргументов следует ожидать.

Да, функция `curry` не такая гибкая, как оригинальная версия, но может подойти для многих сценариев использования. Все дело в компромиссах, которые мы принимаем для нашего сценария использования.



Если вы часто работаете с кортежами, то можете присвоить элементам ваших кортежей имена: `type Person = [name: string, age: number];` Эти метки являются просто аннотациями и удаляются после транспиляции.

В конечном счете функция `curry` и множество ее различных реализаций означают множество способов, с помощью которых вы можете использовать TypeScript для решения конкретной задачи. Вы можете полностью использовать систему типов и применять ее для создания очень сложных и замысловатых типов, а можете немного уменьшить область действия и позволить компилятору сделать всю работу за вас. Ваш выбор зависит от ваших целей и того, чего вы пытаетесь достичь.

7.6. Создание перечисления из кортежа

Задача

Вам нравится, что перечисления упрощают выбор допустимых значений, но после прочтения рецепта 3.12 вы не хотите разбираться во всех нюансах.

Решение

Создавайте свои перечисления из кортежей. Используйте условные типы, вариативные типы кортежей и свойство "length" для создания типа структуры данных.

Обсуждение

В рецепте 3.12 мы обсудили все возможные нюансы, возникающие при использовании числовых и строковых перечислений. В итоге мы получили паттерн, который гораздо ближе к системе типов, но при этом предоставляет вам те же возможности для разработчика, которые вы можете получить, работая с обычными перечислениями:

```
const Direction = {
  Up: 0,
  Down: 1,
  Left: 2,
  Right: 3,
} as const;

// Получаем константные значения Direction
type Direction = (typeof Direction)[keyof typeof Direction];

// (typeof Direction)[keyof typeof Direction] возвращает 0 | 1 | 2 | 3

function move(direction: Direction) {
  // будет определено
}

move(30); // Это ломает код!

move(0); // Это работает!

move(Direction.Left); // Это тоже работает!
```

Это очень простой паттерн, в котором нет ничего неожиданного, но он может потребовать от вас значительного объема работы, если вы имеете дело с большим количеством записей, особенно если вы хотите использовать строковые перечисления:

```
const Commands = {
  Shift: "shift",
  Xargs: "xargs",
  Tail: "tail",
  Head: "head",
```

```

    Uniq: "uniq",
    Cut: "cut",
    Awk: "awk",
    Sed: "sed",
    Grep: "grep",
    Echo: "echo",
  } as const;

```

Происходит дублирование, которое чревато опечатками, а они, в свою очередь, могут привести к неопределенному поведению. Вспомогательная функция, создающая для вас подобное перечисление, помогает справиться с избыточностью и дублированием. Допустим, у вас есть коллекция элементов следующего вида:

```

const commandItems = [
  "echo",
  "grep",
  "sed",
  "awk",
  "cut",
  "uniq",
  "head",
  "tail",
  "xargs",
  "shift",
] as const;

```

Вспомогательная функция `createEnum` перебирает все элементы, создавая объект с заглавными ключами, которые указывают либо на строковое, либо на числовое значение в зависимости от ваших входных параметров:

```

function capitalize(x: string): string {
  return x.charAt(0).toUpperCase() + x.slice(1);
}

```

```

// Необходимо сделать типизацию
function createEnum(arr, numeric) {
  let obj = {};
  for (let [i, el] of arr.entries()) {
    obj[capitalize(el)] = numeric ? i : el;
  }
  return obj;
}

```

```

const Command = createEnum(commandItems); // перечисление строк
const CommandN = createEnum(commandItems, true); // перечисление числовых значений

```

Создадим типы для этой функции! Нам нужно выполнить два действия:

- создать объект из кортежа. Имена ключей пишутся заглавными буквами;
- установить для каждого ключа свойства либо строковое, либо числовое значение. Числовые значения должны начинаться с 0 и с каждым шагом увеличиваться на единицу.

Чтобы создать ключи объектов, нужен тип объединения, которое мы можем сопоставить. Чтобы получить все ключи объектов, мы должны преобразовать кортеж в тип объединения. Вспомогательный тип `TupleToUnion` принимает строковый кортеж и преобразует его в тип объединения. Почему только строковые кортежи? Потому что нам нужны ключи объектов, а строковые ключи использовать проще всего.

Тип `TupleToUnion<T>` — рекурсивный. Так же как в других уроках, мы удаляем отдельные элементы — на этот раз в конце кортежа — и затем снова вызываем тип с оставшимися элементами. Мы помещаем каждый вызов в объединение, фактически получая тип объединения элементов кортежа:

```
type TupleToUnion<T extends readonly string[]> = T extends readonly [
  ...infer Rest extends string[],
  infer Key extends string
]
? Key | TupleToUnion<Rest>
: never;
```

С помощью типа карты и типа манипуляции со строками мы можем создать версию строкового перечисления `Enum<T>`:

```
type Enum<T extends readonly string[], N extends boolean = false> = Readonly<
  {
    [K in TupleToUnion<T> as Capitalize<K>]: K
  }
>;
```

Для версии перечисления чисел нам нужно получить числовое представление каждого значения. Мы уже сохранили его где-то в наших исходных данных. Посмотрим, как `TupleToUnion` работает с четырехэлементным кортежем:

```
// Тип, который мы хотим преобразовать в тип объединения
type Direction = ["up", "down", "left", "right"];

// Вызов вспомогательного типа
type DirectionUnion = TupleToUnion<Direction>;

// Извлечение последнего, рекурсивный вызов TupleToUnion с помощью Rest
type DirectionUnion = "right" | TupleToUnion<["up", "down", "left"]>;

// Извлечение последнего, рекурсивный вызов TupleToUnion с помощью Rest
type DirectionUnion = "right" | "left" | TupleToUnion<["up", "down"]>;

// Извлечение последнего, рекурсивный вызов TupleToUnion с помощью Rest
type DirectionUnion = "right" | "left" | "down" | TupleToUnion<["up"]>;

// Извлекаем последний, рекурсивно вызывая TupleToUnion с пустым кортежем
type DirectionUnion = "right" | "left" | "down" | "up" | TupleToUnion<[]>;
```

```
// Условный тип переходит в ветвь else, добавляя never в тип объединения
type DirectionUnion = "right" | "left" | "down" | "up" | never;

// never в типе объединения поглощается
type DirectionUnion = "right" | "left" | "down" | "up";
```

Если присмотреться, то можно заметить, что длина кортежа уменьшается с каждым вызовом. Сначала это три элемента, потом два, затем один, и в конце концов элементов не остается. Кортежи определяются длиной массива и типом в каждой позиции массива. В TypeScript длина кортежей сохраняется в виде числа, доступного через свойство "length":

```
type DirectionLength = Direction["length"]; // 4
```

Таким образом, при каждом рекурсивном вызове мы можем получить количество оставшихся элементов и использовать его в качестве значения для перечисления. Вместо того чтобы просто возвращать ключи перечисления, мы возвращаем объект, содержащий ключ и его возможное числовое значение:

```
type TupleToUnion<T extends readonly string[]> = T extends readonly [
  ...infer Rest extends string[],
  infer Key extends string
]
? { key: Key; val: Rest["length"] } | TupleToUnion<Rest>
: never;
```

Мы используем этот вновь созданный объект, чтобы решить, хотим ли иметь числовые или строковые значения в нашем перечислении:

```
type Enum<T extends readonly string[], N extends boolean = false> = Readonly<
  {
    [K in TupleToUnion<T> as Capitalize<K["key"]>]: N extends true
      ? K["val"]
      : K["key"];
  }
>;
```

Вот и все! Мы подключаем наш новый тип Enum<T, N> к функции createEnum:

```
type Values<T> = T[keyof T];

function createEnum<T extends readonly string[], B extends boolean>(
  arr: T,
  numeric?: B
) {
  let obj: any = {};
  for (let [i, el] of arr.entries()) {
    obj[capitalize(el)] = numeric ? i : el;
  }
  return obj as Enum<T, B>;
}

const Command = createEnum(commandItems, false);
type Command = Values<typeof Command>;
```


Возможность иметь доступ к длине кортежа в системе типов — одно из скрытых преимуществ TypeScript. Оно позволяет сделать многое, как показано в этом примере, а также реализовать калькуляторы в системе типов. Как и в случае со всеми расширенными возможностями, используйте их разумно.

7.7. Разделение всех элементов сигнатуры функции

Задача

Вы знаете, как получать типы аргументов и типы возвращаемых значений из функций внутри функции, но хотите использовать те же типы и вне ее.

Решение

Используйте встроенные вспомогательные типы `Parameters<F>` и `ReturnType<F>`.

Обсуждение

В этой главе мы рассмотрели вспомогательные функции и то, как они могут получать информацию от функций, которые являются аргументами. Например, функция `defer` принимает функцию и все ее аргументы и возвращает другую функцию, которая ее выполнит. С помощью некоторых обобщенных типов мы можем получить все, что нам нужно:

```
function defer<Par extends unknown[], Ret>(
  fn: (...par: Par) => Ret,
  ...args: Par
): () => Ret {
  return () => fn(...args);
}

const log = defer(console.log, "Hello, world!");
log();
```

Код работает отлично, если мы передаем функции в качестве аргументов, поскольку мы можем легко выбрать элементы и использовать их повторно. Но в некоторых сценариях аргументы функции и ее возвращаемый тип нужны вне обобщенной функции. К счастью, мы можем использовать некоторые встроенные вспомогательные типы TypeScript. С помощью `Parameters<F>` мы получаем аргументы функции в виде кортежа, а с помощью `ReturnType<F>` — типы возвращаемых значений функции. Таким образом, представленную ранее функцию `defer` можно записать так:

```
type Fn = (...args: any[]) => any;

function defer<F extends Fn>(
  fn: F,
  ...args: Parameters<F>
```

```
) : () => ReturnType<F> {  
  return () => fn(...args);  
}
```

И `Parameters<F>`, и `ReturnType<F>` — условные типы, которые основаны на типах функций/кортежей и являются очень похожими. В `Parameters<F>` мы определяем аргументы, а в `ReturnType<F>` выводим типы возвращаемых значений:

```
type Parameters<F extends (...args: any) => any> =  
  F extends (...args: infer P) => any ? P : never;
```

```
type ReturnType<F extends (...args: any) => any> =  
  F extends (...args: any) => infer R ? R : any;
```

Мы можем использовать эти вспомогательные типы, например, для подготовки аргументов функций вне функций. Возьмем эту функцию `search`:

```
type Result = {  
  page: URL;  
  title: string;  
  description: string;  
};  
  
function search(query: string, tags: string[]): Promise<Result[]> {  
  throw "to be done";  
}
```

С помощью `Parameters<typeof search>` мы получаем представление о том, каких параметров следует ожидать. Мы определяем их вне вызова функции и передаем в качестве аргументов при вызове:

```
const searchParams: Parameters<typeof search> = [  
  "Variadic tuple types",  
  ["TypeScript", "JavaScript"],  
];  
  
search(...searchParams);  
const deferredSearch = defer(search, ...searchParams);
```

Оба вспомогательных типа пригодятся вам и при создании новых типов; пример см. в рецепте 4.8.

Вспомогательные ТИПЫ

Одна из сильных сторон TypeScript — возможность выводить типы из других типов. Это позволяет определять взаимосвязи между типами, при которых обновления в одном типе автоматически распространяются на все производные типы. Это сокращает время сопровождения и в итоге приводит к созданию более надежных наборов типов.

При создании производных типов мы обычно применяем одни и те же модификации типов, но в разных комбинациях. В TypeScript уже есть набор встроенных служебных типов (<https://oreil.ly/inM2y>), с частью которых вы уже познакомились в этой книге. Но иногда их бывает недостаточно. В некоторых ситуациях для получения желаемого результата требуется либо по-другому применить известные методы, либо углубиться во внутреннюю работу системы типов. Возможно, вам понадобится собственный набор вспомогательных типов.

В этой главе вы познакомитесь с концепцией вспомогательных типов и увидите несколько сценариев применения, когда пользовательский вспомогательный тип значительно расширяет ваши возможности получения типов из других. Каждый тип предназначен для работы в различных ситуациях и должен открывать вам новый аспект системы типов. Конечно, список типов, который вы видите здесь, ни в коем случае не является полным, но послужит вам хорошей отправной точкой и предоставит достаточно ресурсов, позволяющих развиваться дальше.

В итоге систему типов TypeScript можно рассматривать как собственный функциональный язык метапрограммирования, в котором вы комбинируете небольшие универсальные вспомогательные типы с более крупными вспомогательными типами, чтобы сделать использование производных типов таким же простым, как применение одного типа к существующим моделям.

8.1. Установка определенных необязательных свойств

Задача

Вы хотите создавать производные типы, для которых устанавливаете определенные необязательные свойства.

Решение

Создайте пользовательский вспомогательный тип `SetOptional`, который будет пересечением двух типов объектов: одного, который сопоставляется со всеми выбранными свойствами с помощью модификатора необязательного типа, и другого, который сопоставляется со всеми оставшимися свойствами.

Обсуждение

Все ваши модели в проекте TypeScript заданы и определены, и вы хотите ссылаться на них во всем своем коде:

```
type Person = {
  name: string;
  age: number;
  profession: string;
};
```

Довольно часто возникает ситуация, когда вам нужно что-то похожее на `Person`, но не требующее установки всех свойств; некоторые из них могут быть *необязательными* (optional).

Это сделает ваш API более открытым для других структур и типов с аналогичной структурой, но не имеющих одного или двух полей. Вы не хотите поддерживать различные типы (см. рецепт 12.1), а стремитесь вывести их из исходной модели, которая все еще используется. В TypeScript есть встроенный вспомогательный тип `Partial<T>`, который делает все свойства необязательными:

```
type Partial<T> = { [P in keyof T]? T[P]; };
```

Это *сопоставленный тип*, который сопоставляется со всеми ключами и использует *модификатор необязательного сопоставленного типа*, чтобы сделать каждое свойство необязательным. Первый этап создания типа `SetOptional` — сокращение набора ключей, которые могут быть установлены как необязательные:

```
type SelectPartial<T, K extends keyof T> = {
  [P in K]? T[P]
};
```



Модификатор необязательного сопоставленного типа применяет символ необязательного свойства — вопросительный знак — к набору свойств. Вы узнали о модификаторах сопоставленных типов в рецепте 4.5.

В `SelectPartial<T, K extends keyof T>` мы сопоставляем не все ключи, а только подмножество предоставленных ключей. Обобщенное ограничение `extends keyof T` позволяет нам быть уверенными в том, что мы передаем только допустимые ключи свойств. Если мы применим `SelectPartial` к `Person`, чтобы выбрать "age", то в итоге получим тип, в котором видим *только* свойство `age`, установленное как необязательное:

```
type Age = SelectPartial<Person, "age">;

// type Age = { age?: number | undefined };
```

Первая половина работы сделана: все, что мы хотим задать как необязательное, является необязательным. Но остальные свойства отсутствуют. Вернем их в тип объекта.

Самый простой способ расширить существующий тип объекта, добавив дополнительные свойства, — это создать тип пересечения с другим типом объекта. Так что в нашем случае мы берем то, что написали в `SelectPartial`, и пересекаем с типом, который содержит все оставшиеся ключи.

Мы можем получить все оставшиеся ключи с помощью вспомогательного типа `Exclude`. Тип `Exclude<T, U>` — *условный*; он сравнивает два набора. Если элементы из набора `T` находятся в `U`, то будут удалены с помощью `never`; в противном случае остаются в типе:

```
type Exclude<T, U> = T extends U ? never : T;
```

Это работает, в отличие от типа `Extract<T, U>`, который мы описали в рецепте 5.3. Он является *распределительным условным* (см. рецепт 5.2) и распределяет условный тип по каждому элементу объединения:

```
// В этом примере показано, как TypeScript определяет
// вспомогательный тип шаг за шагом.
```

```
type ExcludeAge = Exclude<"name" | "age", "age">;
```

```
// 1. Распределить
type ExcludeAge =
  "name" extends "age" ? never : "name" |
  "age" extends "age" ? never : "age";
```

```
// 2. Определить
type ExcludeAge = "name" | never;
```

```
// 3. Удалить ненужный `never`
type ExcludeAge = "name";
```

Это именно то, что нам нужно! В `SetOptional` мы создаем один тип, который *выбирает* все отмеченные ключи и делает их необязательными, а затем *исключаем* эти же ключи из большего набора всех ключей объекта:

```
type SetOptional<T, K extends keyof T> = {
  [P in K]?: T[P];
} &
{
  [P in Exclude<keyof T, K>]: T[P];
};
```

Пересечение обоих типов — новый тип объекта, который мы можем использовать с любой понравившейся моделью:

```
type OptionalAge = SetOptional<Person, "age">;

/*
type OptionalAge = {
  name: string;
  age?: number | undefined;
  profession: string;
};
*/
```

Если мы хотим сделать необязательными несколько ключей, то нам нужно предоставить тип объединения со всеми необходимыми ключами свойств:

```
type OptionalAgeAndProf = SetOptional<Person, "age" | "profession">;
```

TypeScript не только позволяет вам самостоятельно определять типы, подобные этому, но и имеет набор встроенных вспомогательных типов, которые вы можете легко комбинировать, чтобы достичь аналогичного эффекта. Мы могли бы написать тот же тип `SetOptional` исключительно на основе вспомогательных типов:

```
type SetOptional<T, K extends keyof T> = Partial<Pick<T, K>> & Omit<T, K>;
```

- `Pick<T, K>` выбирает ключи `K` из объекта `T`.
- `Omit<T, K>` выбирает из объекта `T` все, кроме `K` (скрыто используя `Exclude`).
- И мы уже узнали, что делает `Partial<T>`.

В зависимости от того, как вы предпочитаете читать типы, эту комбинацию вспомогательных типов может быть проще читать и понимать, тем более что встроенные типы гораздо лучше известны среди разработчиков.

Есть только одна проблема: если навести курсор мыши на только что созданные типы, то TypeScript покажет вам, как создан тип, а не каковы его фактические свойства. С помощью вспомогательного типа `Retain` из рецепта 8.3 мы можем сделать наши типы более читабельными и удобными для использования:

```
type SetOptional<T, K extends keyof T> = Remap<
  Partial<Pick<T, K>> & Omit<T, K>
>;
```

Если вы рассматриваете свои аргументы типа как функциональный интерфейс, то, возможно, вам стоит подумать и о параметрах типа. Одна из оптимизаций, которую можно сделать, — установить для второго аргумента (выбранных ключей объекта) значение по умолчанию:

```
type SetOptional<T, K extends keyof T = keyof T> = Remap<
  Partial<Pick<T, K>> & Omit<T, K>
>;
```

Используя `K extends keyof T = keyof T`, вы можете быть уверены, что устанавливаете все ключи свойств как необязательные и выбираете только определенные, если они вам нужны. Ваш вспомогательный тип только что стал чуть более гибким.

Таким же образом вы можете начать создавать типы для других ситуаций, например `SetRequired`, когда нужно убедиться, что некоторые ключи обязательны:

```
type SetRequired<T, K extends keyof T = keyof T> = Remap<
  Required<Pick<T, K>> & Omit<T, K>
>;
```

Или `OnlyRequired`, когда все указанные вами ключи являются обязательными, а остальные нет:

```
type OnlyRequired<T, K extends keyof T = keyof T> = Remap<
  Required<Pick<T, K>> & Partial<Omit<T, K>>
>;
```

Самое приятное: в итоге вы получаете целый арсенал вспомогательных типов, которые можно использовать в разных проектах.

8.2. Изменение вложенных объектов

Задача

Вспомогательные типы объектов, такие как `Partial`, `Required` и `Readonly`, изменяют только первый уровень объекта и не затрагивают свойства вложенных объектов.

Решение

Создайте рекурсивные вспомогательные типы, которые выполняют одну и ту же операцию с вложенными объектами.

Обсуждение

Допустим, в вашем приложении есть различные параметры, которые могут настроить пользователи. Чтобы со временем упростить расширение настроек, вы сохраняете только разницу между набором настроек по умолчанию и настройками, сконфигурированными пользователем:

```
type Settings = {
  mode: "light" | "dark";
  playbackSpeed: number;
  subtitles: {
    active: boolean;
    color: string;
  };
};

const defaults: Settings = {
  mode: "dark",
  playbackSpeed: 1.0,
  subtitles: {
    active: false,
    color: "white",
  },
};
```

Функция `applySettings` использует как настройки по умолчанию, так и настройки ваших пользователей. Вы определили их как `Partial<Settings>`, поскольку пользователю необходимо указать только *некоторые* ключи; остальные будут взяты из настроек по умолчанию:

```
function applySettings(
  defaultSettings: Settings,
  userSettings: Partial<Settings>
): Settings {
  return { ...defaultSettings, ...userSettings };
}
```

Схема действительно хорошо работает, если вам нужно задать определенные свойства на первом уровне:

```
let settings = applySettings(defaults, { mode: "light" });
```

Но она же вызывает проблемы, если вы хотите изменить определенные свойства в глубине объекта, например, установить `subtitles` в активное состояние:

```
let settings = applySettings(defaults, { subtitles: { active: true } });
//                                     ^
// Property 'color' is missing in type '{ active: true; }'
// but required in type '{ active: boolean; color: string; }'.(2741)
```


TypeScript жалуется, что `subtitles` необходимо предоставить весь объект. Это связано с тем, что `Partial<T>`, как и его родственные элементы `Required<T>` и `Readonly<T>`, изменяет только первый уровень объекта. Вложенные объекты будут обрабатываться как простые значения.

Чтобы изменить это, нужно создать новый тип `DeepPartial<T>`, который рекурсивно просматривает каждое свойство и применяет *модификатор необязательного сопоставленного типа* для каждого уровня:

```
type DeepPartial<T> = {
  [K in keyof T]?: DeepPartial<T[K]>;
};
```

Первый вариант кода работает хорошо благодаря тому, что TypeScript останавливает рекурсию на примитивных значениях, но потенциально может привести к нечитаемому результату. Простое условие, которое проверяет, что мы углубляемся только в том случае, если имеем дело с объектом, сильно увеличивает надежность типа и делает результат более читабельным:

```
type DeepPartial<T> = T extends object
  ? {
    [K in keyof T]? DeepPartial<T[K]>;
  }
  : T;
```

Например, `DeepPartial<Settings>` приводит к следующему результату:

```
type DeepPartialSettings = {
  mode?: "light" | "dark" | undefined;
  playbackSpeed?: number | undefined;
  subtitles?: {
    active?: boolean | undefined;
    color?: string | undefined;
  } | undefined;
};
```

Это именно то, к чему мы стремились. Если мы используем `DeepPartial<T>` в `applySettings`, то видим, что фактическое использование `applySettings` работает, но TypeScript выдает другую ошибку:

```
function applySettings(
  defaultSettings: Settings,
  userSettings: DeepPartial<Settings>
): Settings {
  return { ...defaultSettings, ...userSettings };
//      ^
// Type '{ mode: "light" | "dark"; playbackSpeed: number;
//   subtitles: { active?: boolean | undefined;
//   color?: string | undefined; }; }' is not assignable to type 'Settings'.
}
```

Здесь TypeScript жалуется, что не может объединить эти два объекта в нечто, способное привести к `Settings`, поскольку некоторые элементы набора `DeepPartial` не могут быть присвоены `Settings`. И это правда! Объединение объектов с помощью деструктуризации тоже работает лишь на первом уровне, как и было определено в `Partial<T>`. Это означает, что если бы мы вызвали `applySettings`, как и раньше, то получили бы совершенно другой тип, чем для `settings`:

```
let settings = applySettings(defaults, { subtitles: { active: true } });

// приводит к

let settings = {
  mode: "dark",
  playbackSpeed: 1,
  subtitles: {
    active: true
  }
};
```

Весь `color` исчез! Это одна из ситуаций, когда тип TypeScript может показаться неинтуитивным: почему глубина типов модификации объектов достигает только одного уровня? Потому что JavaScript имеет лишь один уровень глубины! Но в итоге он указывает на ошибки, которые иначе вы не обнаружили бы.

Чтобы обойти такую ситуацию, необходимо применять настройки рекурсивно. Реализовать это действие самостоятельно может быть очень сложно, поэтому мы используем `lodash` и его функцию `merge`:

```
import { merge } from "lodash";

function applySettings(
  defaultSettings: Settings,
  userSettings: DeepPartial<Settings>
): Settings {
  return merge(defaultSettings, userSettings)
}
```

В функции `merge` определен интерфейс для создания пересечения двух объектов:

```
function merge<TObject, TSource>(
  object: TObject, source: TSource
): TObject & TSource {
  // ...
}
```

Опять же, это именно то, что нам нужно. Пересечение `Settings` и `DeepPartial<Settings>` тоже дает пересечение обоих типов, которое в силу природы типов снова является `Settings`.

В итоге мы получаем типы, которые точно сообщают нам, чего именно ожидать, корректные результаты вывода и еще один вспомогательный тип в нашем арсенале. Аналогично можно создать `DeepReadOnly` и `DeepRequired`.

8.3. Переназначение типов

Задача

Благодаря конструированию типов вы получаете гибкие самоподдерживающиеся типы, но подсказки редактора оставляют желать лучшего.

Решение

Используйте вспомогательные типы `Remap<T>` и `DeepRemap<T>` для улучшения подсказок редактора.

Обсуждение

Когда вы задействуете систему типов TypeScript для создания новых типов, используя вспомогательные и сложные условные типы или даже простые пересечения, вы можете получить подсказки редактора, которые трудно расшифровать.

Рассмотрим `OnlyRequired` из рецепта 8.1. В этом типе используются четыре вспомогательных типа и одно пересечение для создания нового типа, в котором все ключи, указанные в качестве второго параметра типа, установлены как обязательные, а все остальные — как необязательные:

```
type OnlyRequired<T, K extends keyof T = keyof T> =  
  Required<Pick<T, K>> & Partial<Omit<T, K>>;
```

Такой способ написания типов позволяет вам получить хорошее представление о том, что происходит. Вы можете ознакомиться с функциональностью, основываясь на том, как вспомогательные типы komponуются друг с другом. Но когда вы будете использовать типы в своих моделях, вам может понадобиться и другая информация, помимо сведений о фактической конструкции типа:

```
type Person = {  
  name: string;  
  age: number;  
  profession: string;  
};
```

```
type NameRequired = OnlyRequired<Person, "name">;
```

Если вы наведете указатель мыши на `NameRequired`, то увидите, что TypeScript предоставляет вам информацию о том, как тип был создан на основе заданных вами параметров, но подсказка редактора не покажет вам результат, поскольку окончательный тип создается с помощью этих вспомогательных типов. Обратная связь редактора показана на рис. 8.1.

```
type NameRequired = Required<Pick<Person, "name">> &
Partial<Omit<Person, "name">>
type NameRequired = OnlyRequired<Person, "name">;
```

Рис. 8.1. Подсказки редактора для сложных типов раскрываются очень поверхностно; без знания типов, лежащих в их основе и их функциональности, становится трудно понять результат

Чтобы конечный результат выглядел как реальный тип и описывал все свойства, вы должны использовать простой, но эффективный тип `Remap`:

```
type Remap<T> = {
  [K in keyof T]: T[K];
};
```

`Remap<T>` — это просто тип объекта, который проходит через каждое свойство и сопоставляет его с определенным значением. Нет никаких модификаций, никаких фильтров, мы просто добавляем то, что было введено. TypeScript выводит каждое свойство сопоставленных типов, поэтому вместо отображения конструкции вы видите фактический тип (рис. 8.2).

```
type NameRequired = {
  name: string;
  age?: number | undefined;
  profession?: string | undefined;
}
type NameRequired = Remap<OnlyRequired<Person, "name">>;
```

Рис. 8.2. Использование `Remap<T>` значительно улучшает читаемость представления `NameRequired`

Прекрасно! Это стало основной частью библиотек типов TypeScript. Одни называют ее `Debug`, другие — `Simplify`. `Remap` — просто другое название того же

инструмента и эффекта: получение представления о том, как будет выглядеть ваш результат.

Как и другие сопоставленные типы `Partial<T>`, `ReadOnly<T>` и `Required<T>`, `Remap<T>` тоже работает только на первом уровне. Вложенный тип, например `Settings`, содержащий тип `Subtitles`, будет переназначен на тот же выход, и подсказки редактора будут такими же:

```
type Subtitles = {
  active: boolean;
  color: string;
};

type Settings = {
  mode: "light" | "dark";
  playbackSpeed: number;
  subtitles: Subtitles;
};
```

Но так же, как показано в рецепте 8.2, мы можем создать рекурсивный вариант, который переназначает *все* вложенные типы объектов:

```
type DeepRemap<T> = T extends object
  ? {
    [K in keyof T]: DeepRemap<T[K]>;
  }
  : T;
```

Кроме того, применение `DeepRemap<T>` к `Settings` расширит `Subtitles`:

```
type SettingsRemapped = DeepRemap<Settings>;

// приводит к

type SettingsRemapped = {
  mode: "light" | "dark";
  playbackSpeed: number;
  subtitles: {
    active: boolean;
    color: string;
  };
};
```

Использовать ли `Remap` — ваш выбор. Иногда вам хочется узнать о реализации, а иногда читать краткое представление вложенных типов более удобно, чем расширенные версии. Но есть ситуации, когда вам действительно важен сам результат. В таких случаях удобный и доступный вспомогательный тип `Remap<T>`, безусловно, будет очень полезен.

8.4. Получение всех необходимых ключей

Задача

Вы хотите создать тип, который извлекает из объекта все *необходимые* свойства.

Решение

Создайте сопоставленный вспомогательный тип `GetRequired<T>`, который фильтрует ключи на основе проверки подтипа на соответствие его требуемому аналогу.

Обсуждение

Необязательные свойства оказывают огромное влияние на совместимость типов. Простой модификатор типа, знак вопроса, значительно расширяет исходный тип. Такие свойства позволяют определять поля, которые могут присутствовать в исходном типе, но их можно использовать, только если мы выполним дополнительные проверки.

Это означает, что мы можем сделать наши функции и интерфейсы совместимыми с типами, не имеющими определенных свойств:

```
type Person = {
  name: string;
  age?: number;
};

function printPerson(person: Person): void {
  // ...
}

type Student = {
  name: string;
  semester: number;
};

const student: Student = {
  name: "Stefan",
  semester: 37,
};

printPerson(student); // Все хорошо!
```

Мы видим, что `age` определено в `Person`, но совсем не определено в `Student`. Данное поле необязательно, так что это не мешает нам использовать `printPerson` с объектами типа `Student`. Набор совместимых значений стал шире, так как мы можем использовать объекты типов, в которых `age` полностью отсутствует.

TypeScript решает данную проблему, добавляя `undefined` к свойствам, которые являются необязательными. Это наиболее точное представление «данное свойство может быть у объекта».

Этот факт важен, если мы хотим проверить, являются ли ключи свойств обязательными. Начнем с самой простой проверки. У нас есть объект, и мы хотим проверить, все ли ключи являются обязательными. Для этого мы используем вспомогательный тип `Required<T>`, который изменяет все свойства, делая их обязательными. Самая простая проверка — выяснить, является ли тип объекта, например `Name`, подмножеством его аналога `Required<T>`:

```
type Name = {
  name: string;
};

type Test = Name extends Required<Name> ? true : false;
// type Test = true
```

Здесь в результате выполнения `Test` выдается `true`, ведь если мы изменим все свойства на `required` с помощью `Required<T>`, то все равно получим тот же тип. Однако все меняется, если мы вводим необязательное свойство:

```
type Person = {
  name: string;
  age?: number;
};

type Test = Person extends Required<Person> ? true : false;
// type Test = false
```

Здесь выполнение `Test` дает результат `false`, поскольку тип `Person` с необязательным свойством `age` принимает гораздо более широкий набор значений, чем `Required<Person>`, где необходимо задать `age`. В отличие от этой проверки, если поменять местами `Person` и `Required<Person>`, мы увидим, что более узкий тип `Required<Person>` на самом деле является подмножеством `Person`:

```
type Test = Required<Person> extends Person ? true : false;
// type Test = true
```

До сих пор мы проверяли, содержит ли весь объект необходимые ключи. Однако на самом деле мы хотим получить объект, содержащий только те ключи свойств, для которых установлено значение `required`. Это означает, что нам нужно выполнить данную проверку для каждого ключа свойства. Необходимость подвергать одной и той же проверке набор ключей — хороший индикатор для сопоставленного типа.

Далее мы создадим сопоставленный тип, который будет выполнять проверку подмножества для каждого свойства, чтобы проверить, не содержат ли полученные значения `undefined`:

```
type RequiredPerson = {
  [K in keyof Person]: Person[K] extends Required<Person[K]> ? true : false;
};
```

```

/*
type RequiredPerson = {
  name: true;
  age?: true | undefined;
}
*/

```

Это хорошее предположение, но оно дает результаты, которые не работают. Каждое свойство принимает значение `true`; это значит, подмножество проверяет только типы значений *без* `undefined`. Это происходит потому, что `Required<T>` работает с объектами, а не с примитивными типами. Более надежные результаты дает проверка наличия в `Person[K]` каких-либо значений, *допускающих значение* `NULL`. `NonNullable<T>` удаляет `undefined` и `null`:

```

type RequiredPerson = {
  [K in keyof Person]: Person[K] extends NonNullable<Person[K]> ? true : false;
};

/*
type RequiredPerson = {
  name: true;
  age?: false | undefined;
}
*/

```

Этот код уже более хорош, но все еще не такой, как нам хотелось бы. Значение `undefined` снова появилось, так как добавляется с помощью модификатора свойства. Кроме того, свойство по-прежнему относится к типу, а мы хотим избавиться от него.

Что нужно сделать, так это уменьшить набор возможных ключей. Поэтому вместо того, чтобы проверять значения, мы выполняем условную проверку каждого свойства в процессе сопоставления ключей. Мы проверяем, является ли `Person[K]` подмножеством `Required<Person>[K]`, выполняя соответствующую проверку по большему подмножеству. В этом случае мы выводим ключ `K`, в противном случае удаляем свойство, используя `never` (см. рецепт 5.2):

```

type RequiredPerson = {
  [K in keyof Person as Person[K] extends Required<Person>[K]
    ? K
    : never]: Person[K];
};

```

Этот код дает желаемые результаты. Теперь мы подставляем `Person` вместо параметра обобщенного типа, и наш вспомогательный тип `GetRequired<T>` готов:

```

type GetRequired<T> = {
  [K in keyof T as T[K] extends Required<T>[K]
    ? K
    : never]: T[K];
};

```


С этого момента мы можем создавать такие варианты, как `GetOptional<T>`. Однако проверить что-либо на необязательность не так просто, как выяснить, являются ли некоторые ключи свойств обязательными, но мы можем использовать `GetRequired<T>` и оператор `keyof`, чтобы получить все необходимые ключи свойств:

```
type RequiredKeys<T> = keyof GetRequired<T>;
```

После этого мы используем `RequiredKeys<T>`, чтобы *исключить* их из нашего целевого объекта:

```
type GetOptional<T> = Omit<T, RequiredKeys<T>>;
```

Опять же, комбинация нескольких вспомогательных типов создает производные самоподдерживающиеся типы.

8.5. Разрешение хотя бы одного свойства

Задача

У вас есть тип, и вы хотите убедиться, что для него задано хотя бы одно свойство.

Решение

Создайте вспомогательный тип `Split<T>`, который разбивает объект на группы объектов с одним свойством.

Обсуждение

Ваше приложение хранит набор URL (например, для видеформатов) в объекте, каждый ключ которого идентифицирует отдельный формат:

```
type VideoFormatURLs = {  
  format360p: URL;  
  format480p: URL;  
  format720p: URL;  
  format1080p: URL;  
};
```

Вы хотите создать функцию `loadVideo`, которая сможет загружать видео из всех указанных URL видеформатов, но должна загрузить хотя бы один URL.

Если `loadVideo` принимает параметры типа `VideoFormatURLs`, то вам необходимо предоставить *все* URL видеформатов:

```
function loadVideo(formats: VideoFormatURLs) {  
  // будет определено  
}
```

```
loadVideo({
  format360p: new URL("..."),
  format480p: new URL("..."),
  format720p: new URL("..."),
  format1080p: new URL("..."),
});
```

Но некоторые видео могут отсутствовать, поэтому на самом деле вам нужно выбрать подмножество из всех доступных типов. `Partial<VideoFormatURLs>` дает вам такую возможность:

```
function loadVideo(formats: Partial<VideoFormatURLs>) {
  // будет определено
}

loadVideo({
  format480p: new URL("..."),
  format720p: new URL("..."),
});
```

Но все ключи являются необязательными, поэтому вы также можете разрешить пустой объект в качестве допустимого параметра:

```
loadVideo({});
```

Это приводит к неопределенному поведению. Вам нужен хотя бы один URL, чтобы вы могли загрузить данное видео.

Нужно найти тип, выражающий, что вы ожидаете по крайней мере один из доступных видеформатов: тип, который позволяет передавать все из них и некоторые из них, но не позволяет совсем не передавать ни один из них.

Начнем со случая «только один». Вместо того чтобы искать один тип, создадим тип объединения, которое сгруппирует все ситуации, когда имеется только один набор свойств:

```
type AvailableVideoFormats =
  | {
    format360p: URL;
  }
  | {
    format480p: URL;
  }
  | {
    format720p: URL;
  }
  | {
    format1080p: URL;
  };
```

Это позволяет нам передавать объекты, у которых задано только одно свойство. Далее добавим ситуации, когда заданы два свойства:

```
type AvailableVideoFormats =  
  | {  
    format360p: URL;  
  }  
  | {  
    format480p: URL;  
  }  
  | {  
    format720p: URL;  
  }  
  | {  
    format1080p: URL;  
  };
```

Как видим, это тот же самый тип! Именно так и работают типы объединения. Если они не различаются (см. рецепт 3.2), то тип объединения будет допускать значения, находящиеся на всех пересечениях исходного набора (рис. 8.3).

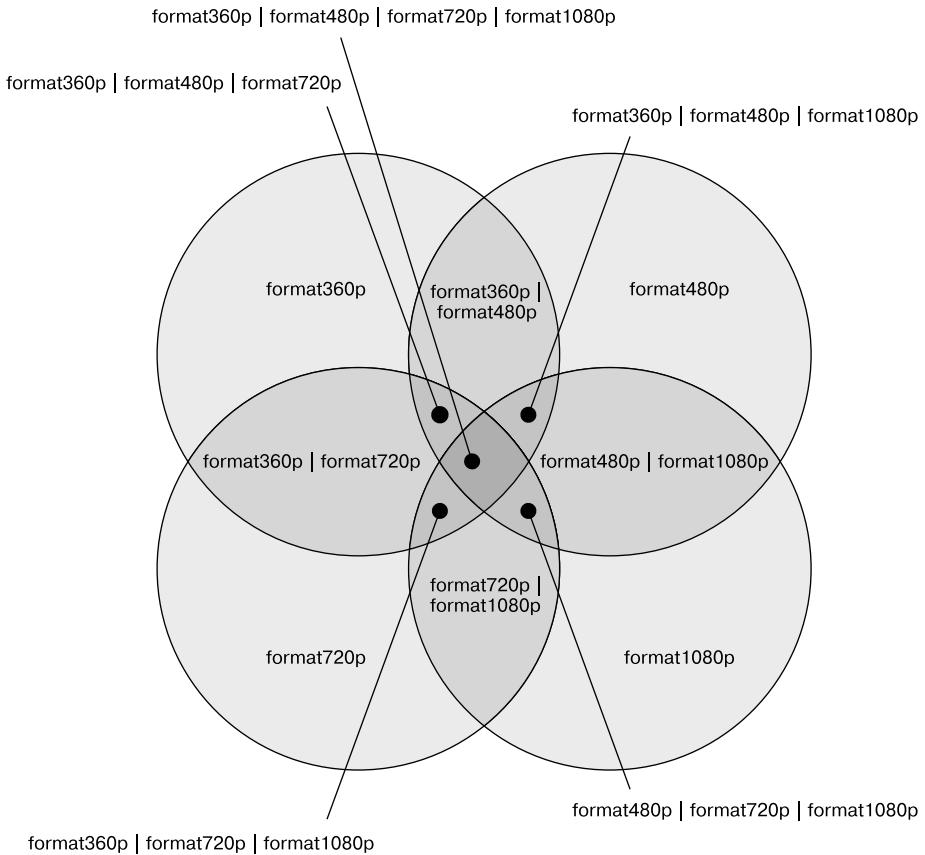


Рис. 8.3. Тип объединения `AvailableVideoFormats`

Каждый член объединения определяет набор возможных значений. Пересечения описывают значения, в которых оба типа перекрываются. С помощью этого объединения можно выразить все возможные комбинации.

Итак, теперь, когда мы знаем тип, было бы хорошо получить его из исходного типа. Мы хотим разделить тип объекта на типы объединения, каждый член которого содержит ровно одно свойство.

Один из способов получить тип объединения, связанный с `VideoFormatURLs`, — использовать оператор `keyof`:

```
type AvailableVideoFormats = keyof VideoFormatURLs;
```

В результате получается объединение ключей `"format360p" | "format480p" | "format720p" | "format1080p"`. Мы можем использовать оператор `keyof`, чтобы получить индексный доступ к исходному типу:

```
type AvailableVideoFormats = VideoFormatURLs[keyof VideoFormatURLs];
```

В результате получается URL, который представляет собой всего лишь один тип, но в действительности это тип объединения значений. Теперь нам осталось найти способ получить правильные значения, которые представляют фактический тип объекта и связаны с каждым ключом свойства.

Прочитайте этот фрагмент еще раз: «связаны с каждым ключом свойства». Для этого требуется сопоставленный тип! Мы можем сопоставить все `VideoFormatURL`, чтобы получить ключ свойства в правой части объекта:

```
type AvailableVideoFormats = {
  [K in keyof VideoFormatURLs]: K;
};
/* yields
type AvailableVideoFormats = {
  format360p: "format360p";
  format480p: "format480p";
  format720p: "format720p";
  format1080p: "format1080p";
}; */
```

Благодаря этому мы можем получить индексный доступ к сопоставленному типу и узнавать типы значений для каждого элемента. Но мы не только устанавливаем ключ в правой части, но и создаем другой тип объекта, который принимает эту строку в качестве ключа свойства и сопоставляет ее с соответствующим типом значения:

```
type AvailableVideoFormats = {
  [K in keyof VideoFormatURLs]: {
    [P in K]: VideoFormatURLs[P]
  };
};
```

```

/* результаты
type AvailableVideoFormats = {
  format360p: {
    format360p: URL;
  };
  format480p: {
    format480p: URL;
  };
  format720p: {
    format720p: URL;
  };
  format1080p: {
    format1080p: URL;
  };
};

```

Теперь мы можем снова использовать индексный доступ, чтобы собрать все типы значений из правой части в объединение:

```

type AvailableVideoFormats = {
  [K in keyof VideoFormatURLs]: {
    [P in K]: VideoFormatURLs[P]
  };
}[keyof VideoFormatURLs];

```

```

/* результаты
type AvailableVideoFormats =
  | {
    format360p: URL;
  }
  | {
    format480p: URL;
  }
  | {
    format720p: URL;
  }
  | {
    format1080p: URL;
  };
*/

```

И это то, что мы искали! В качестве следующего шага мы берем конкретные типы и заменяем их обобщенными типами, в результате чего получаем вспомогательный тип `Split<T>`:

```

type Split<T> = {
  [K in keyof T]: {
    [P in K]: T[P];
  };
}[keyof T];

```

Теперь в нашем арсенале есть еще один вспомогательный тип. Использование его с `loadVideo` дает именно то поведение, к которому мы стремились:

```
function loadVideo(formats: Split<VideoFormatURLs>) {
  // будет определено
}

loadVideo({});
//      ^
// Argument of type '{}' is not assignable to parameter
// of type 'Split<VideoFormatURLs>'

loadVideo({
  format480p: new URL("..."),
}); // все хорошо
```

Использование типа `Split<T>` — хороший способ увидеть, как базовая функциональность системы типов может существенно изменить поведение ваших интерфейсов и как с помощью некоторых простых методов типизации, таких как сопоставленные типы, типы индексного доступа и ключи свойств, можно получить крошечный, но эффективный вспомогательный тип.

8.6. Разрешение ровно одного свойства, всех свойств или ни одного

Задача

Помимо требования о наличии *хотя бы одного* свойства, как в рецепте 8.5, вы хотите предусмотреть сценарии, в которых пользователи предоставляют *ровно одно, все или ни одного*.

Решение

Создайте `ExactlyOne<T>` и `AllOrNone<T, K>`. Оба основаны на приеме «Необязательный `never`» в сочетании с производным от `Split<T>`.

Обсуждение

С помощью типа `Split<T>` из рецепта 8.5 мы создаем удобный вспомогательный тип, позволяющий описать сценарий, в котором мы хотим указать *хотя бы один* параметр. Это то, что не может обеспечить `Partial<T>`, но могут обычные типы объединения.

Исходя из этой идеи, мы можем столкнуться со сценариями, в которых хотим, чтобы наши пользователи предоставляли *ровно одно* свойство, следя за тем, чтобы они не добавляли слишком много параметров.

Один из приемов, который можно использовать здесь, — это «Необязательный `never`», который мы изучали в рецепте 3.8. Наряду со всеми свойствами, которые вы хотите разрешить, вы устанавливаете все свойства, которые не хотите разрешать, как необязательные и присваиваете им значение `never`. Это означает, что в тот момент, когда вы пишете имя свойства, TypeScript хочет, чтобы вы установили его значение, совместимое с типом `never`, чего вы не можете сделать, поскольку у данного типа нет значений.

Ключевым является тип объединения, в котором мы помещаем все имена свойств в отношении *исключительного ИЛИ*. Мы получаем тип объединения, в котором каждое свойство добавлено с помощью `Split<T>`:

```
type Split<T> =  
{ [K in keyof T]: {  
  [P в K]: T[P];  
  };  
}[keyof T];
```

Все, что нам нужно сделать, — сопоставить каждый элемент с оставшимися ключами и установить для них необязательное значение `never`:

```
type ExactlyOne<T> = {  
  [K in keyof T]: {  
    [P in K]: T[P];  
  } &  
  {  
    [P in Exclude<keyof T, K>]?: never; // необязательное значение never  
  };  
}[keyof T];
```

При этом итоговый тип получается более обширным, но точно указывает, какие свойства нужно исключить:

```
type ExactlyOneVideoFormat = ({  
  format360p: URL;  
} & {  
  format480p?: never;  
  format720p?: never;  
  format1080p?: never;  
}) | ({  
  format480p: URL;  
} & {  
  format360p?: never;  
  format720p?: never;
```

```

    format1080p?: never;
  }) | ({
    format720p: URL;
  } & {
    format320p?: never;
    format480p?: never;
    format1080p?: never;
  }) | ({
    format1080p: URL;
  } & {
    format320p?: never;
    format480p?: never;
    format720p?: never;
  });

```

И все работает так, как и ожидалось:

```

function loadVideo(formats: ExactlyOne<VideoFormatURLs>) {
  // будет определено
}

loadVideo({
  format360p: new URL("..."),
}); // работает

loadVideo({
  format360p: new URL("..."),
  format1080p: new URL("..."),
});
// ^
// Argument of type '{ format360p: URL; format1080p: URL; }'
// is not assignable to parameter of type 'ExactlyOne<VideoFormatURLs>'.

```

`ExactlyOne<T>` настолько похож на `Split<T>`, что мы могли бы подумать о расширении функциональности `Split<T>` за счет добавления необязательного паттерна `never`:

```

type Split<T, OptionalNever extends boolean = false> = {
  [K in keyof T]: {
    [P in K]: T[P];
  } &
  (OptionalNever extends false
    ? {}
    : {
      [P in Exclude<keyof T, K>]?: never;
    });
}[keyof T];

type ExactlyOne<T> = Split<T, true>;

```


Мы добавляем новый параметр обобщенного типа `OptionalNever`, значение которого по умолчанию установлено `false`. Затем мы пересекаем часть, в которой создаем новые объекты, с условным типом, который проверяет, действительно ли параметр `OptionalNever` равен `false`. Если это так, то мы пересекаемся с пустым объектом (оставляя исходный объект нетронутым); в противном случае добавляем к объекту часть `OptionalNever`. `ExactlyOne<T>` рефакторится в `Split<T, true>`, где мы активируем флаг `OptionalNever`.

Другой сценарий, очень похожий на `Split<T>` или `ExactlyOne<T>`, состоит в предоставлении всех аргументов или отсутствии аргументов. Вообразите сценарий разделения форматов видео на стандартное (SD: 360 и 480 пикселей) и видео высокой четкости (HD: 720 и 1080 пикселей). В своем приложении вы хотите убедиться, что если ваши пользователи предоставляют форматы SD, то должны предоставить все возможные форматы. Вполне допустимо использовать только один формат HD.

Здесь также используется прием «Необязательный `never`». Мы определяем тип, для которого *требуются* все выбранные ключи, или устанавливаем для них значение `never`, если предоставлен только один:

```
type AllOrNone<T, Keys extends keyof T> = (
  | {
    [K in Keys]-?: T[K]; // все доступно
  }
  | {
    [K in Keys]?: never; // или нет
  }
);
```

Если вы хотите убедиться, что предоставляете также *все* HD-форматы, то добавьте к ним остальные с помощью пересечения:

```
type AllOrNone<T, Keys extends keyof T> = (
  | {
    [K in Keys]-?: T[K];
  }
  | {
    [K in Keys]?: never;
  }
) & {
  [K in Exclude<keyof T, Keys>]: T[K] // остальное, как было определено
}
```

Если же форматы HD совершенно необязательны, добавьте их с помощью `Partial<T>`:

```
type AllOrNone<T, Keys extends keyof T> = (
  | {
    [K in Keys]-?: T[K];
```

```

    }
  | {
    [K in Keys]?: never;
  }
) & Partial<Omit<T, Keys>>; // все остальное, но необязательно

```

Но тогда мы столкнемся с той же проблемой, описанной и в рецепте 8.5, — мы можем предоставить значения, которые вообще не содержат никаких форматов. Пересечение варианта *«все или ничего»* с типом `Split<T>` — то решение, к которому мы стремимся:

```

type AllOrNone<T, Keys extends keyof T> = (
  | {
    [K in Keys]-?: T[K];
  }
  | {
    [K in Keys]?: never;
  }
) & Split<T>;

```

И код работает как задумано:

```

function loadVideo(
  formats: AllOrNone<VideoFormatURLs, "format360p" | "format480p">
) {
  // будет определено
}

loadVideo({
  format360p: new URL("..."),
  format480p: new URL("..."),
}); // OK

loadVideo({
  format360p: new URL("..."),
  format480p: new URL("..."),
  format1080p: new URL("..."),
}); // OK

loadVideo({
  format1080p: new URL("..."),
}); // OK

loadVideo({
  format360p: new URL("..."),
  format1080p: new URL("..."),
});
// ^ Argument of type '{ format360p: URL; format1080p: URL; }' is
// not assignable to parameter of type
// '{ format360p: URL; format480p: URL; } & ... (abbreviated)

```

Если мы внимательно посмотрим на то, что делает `AllOrNone`, то сможем легко переписать его с помощью встроенных вспомогательных типов:

```
type AllOrNone<T, Keys extends keyof T> = (  
  | Required<Pick<T, Keys>>  
  | Partial<Record<Keys, never>>  
) &  
  Split<T>;
```

Этот код, пожалуй, более читабелен и при этом больше соответствует сути метапрограммирования в системе типов. У вас есть набор вспомогательных типов, и, комбинируя их, вы можете создавать новые вспомогательные типы: почти как в функциональном языке программирования, но с использованием наборов значений в системе типов.

8.7. Преобразование типов объединения в типы пересечения

Задача

Ваша модель определена как тип объединения нескольких вариантов. Чтобы вывести из нее другие типы, сначала нужно преобразовать типы объединения в типы пересечения.

Решение

Создайте вспомогательный тип `UnionToIntersection<T>`, использующий контра-вариантные позиции.

Обсуждение

В рецепте 8.5 мы обсуждали, как можно разделить тип модели на объединение ее вариантов. В зависимости от того, как работает ваше приложение, вы можете с самого начала определить модель как тип объединения нескольких вариантов:

```
type BasicVideoData = {  
  // будет определено  
};  
  
type Format320 = { urls: { format320p: URL } };  
type Format480 = { urls: { format480p: URL } };  
type Format720 = { urls: { format720p: URL } };  
type Format1080 = { urls: { format1080p: URL } };  
  
type Video = BasicVideoData & (Format320 | Format480 | Format720 | Format1080);
```

Тип `Video` позволяет вам задать несколько форматов, но требует, чтобы вы определили хотя бы один из них:

```
const video1: Video = {
  // ...
  urls: {
    format320p: new URL("https://..."),
  },
}; // OK

const video2: Video = {
  // ...
  urls: {
    format320p: new URL("https://..."),
    format480p: new URL("https://..."),
  },
}; // OK

const video3: Video = {
  // ...
  urls: {
    format1080p: new URL("https://..."),
  },
}; // OK
```

Однако сбор их в один тип имеет некоторые побочные эффекты, например, когда вам нужны все доступные ключи:

```
type FormatKeys = keyof Video["urls"];
// FormatKeys = never

// Это не то, чего мы здесь хотим !
function selectFormat(format: FormatKeys): void {
  // будет определено
}
```

Можно было бы ожидать, что `FormatKeys` предоставит тип объединения всех ключей, вложенных в `urls`. Однако индексный доступ к типу объединения пытается найти наименьший общий знаменатель. А в данном случае его нет. Чтобы получить тип объединения всех ключей формата, необходимо, чтобы все ключи были одного типа:

```
type Video = BasicVideoData & {
  urls: {
    format320p: URL;
    format480p: URL;
    format720p: URL;
    format1080p: URL;
  };
};
```

```
type FormatKeys = keyof Video["urls"];
// type FormatKeys =
// "format320p" | "format480p" | "format720p" | "format1080p";
```

Чтобы создать подобный объект, нужно изменить типы объединения в типы пересечения.



В рецепте 8.5 было принято моделировать данные в одном типе; в этом рецепте мы видим, что используется моделирование данных в виде типов объединения. В действительности не существует единого ответа на вопрос, как определять модели. Используйте то представление, которое наилучшим образом соответствует вашей прикладной области и не слишком мешает вам. Важно иметь возможность создавать другие типы по мере необходимости. Это сокращает время сопровождения и позволяет вводить более надежные типы. В главе 12 и особенно в рецепте 12.1 мы рассмотрим принцип «простые в сопровождении типы».

Преобразование типов объединения в типы пересечения — своеобразная задача в TypeScript, требующая глубоких знаний о внутреннем устройстве системы типов. Чтобы изучить все эти понятия, мы рассмотрим готовый тип, а затем посмотрим, что происходит «под капотом»:

```
type UnionToIntersection<T> =
  (T extends any ? (x: T) => any : never) extends
  (x: infer R) => any ? R : never;
```

Здесь нужно разобрать *многое*.

- У нас есть два условных типа. Кажется, что первый из них всегда приводит к ветви `true`, так зачем он нужен?
- Первый условный тип оборачивает тип в аргумент функции, а второй — снова его разворачивает. Зачем это нужно?
- И как оба условных типа преобразуют типы объединения в типы пересечения?

Поэтапно проанализируем `UnionToIntersection<T>`.

В первом условном типе внутри `UnionToIntersection<T>` мы используем аргумент обобщенного типа как *голый тип* (naked type):

```
type UnionToIntersection<T> =
  (T extends any ? (x: T) => any : never) //...
```

Это значит, мы проверяем, находится ли `T` в условии подтипа, не оборачивая его в какой-либо другой тип:

```
type Naked<T> = T extends ...; // голый тип
```

```
type NotNaked<T> = { o: T } extends ...; // не голый тип
```

Голые типы в условных типах имеют определенную особенность. Если T — объединение, то они выполняют условный тип для каждой составляющей объединения. Таким образом, в случае с голым типом *условное объединение типов становится объединением условных типов*:

```
type WrapNaked<T> = T extends any ? { o: T } : never;

type Foo = WrapNaked<string | number | boolean>;

// Голый тип, поэтому это эквивалентно

type Foo =
  WrapNaked<string> | WrapNaked<number> | WrapNaked<boolean>;

// эквивалентно

type Foo =
  string extends any ? { o: string } : never |
  number extends any ? { o: number } : never |
  boolean extends any ? { o: boolean } : never;

type Foo =
  { o: string } | { o: number } | { o: boolean };
```

Сравним с версией без использования голого типа:

```
type WrapNaked<T> = { o: T } extends any ? { o: T } : never;

type Foo = WrapNaked<string | number | boolean>;

// Не голый тип, поэтому это эквивалентно

type Foo =
  { o: string | number | boolean } extends any ?
  { o: string | number | boolean } : never;

type Foo = { o: string | number | boolean };
```

Мы видим небольшие, но существенные различия для сложных типов!

В нашем примере мы используем голый тип и спрашиваем, расширяет ли он `any` (что всегда и происходит, так как `any` — высший тип, разрешающий все):

```
type UnionToIntersection<T> =
  (T extends any ? (x: T) => any : never) //...
```

Данное условие всегда выполняется, поэтому мы обертываем наш обобщенный тип в функцию, где T — тип параметра функции. Но почему мы это делаем?

Мы приходим ко второму условию:

```
type UnionToIntersection<T> =  
  (T extends any ? (x: T) => any : never) extends  
  (x: infer R) => any ? R : never
```

Первое условие всегда дает значение `true`; это значит, мы обернули наш тип в тип функции. Поэтому второе условие тоже всегда дает значение `true`. По сути, мы проверяем, является ли тип, который мы только что создали, подтипом самого себя. Но вместо того, чтобы передавать `T`, мы выводим новый тип `R` и возвращаем выведенный тип.

Что мы делаем, так это оборачиваем и разворачиваем тип `T` с помощью типа функции.

Выполнение этого действия с помощью аргументов функции приводит к тому, что новый выведенный тип `R` становится *контравариантным*.

Так что же означает *контравариантность*? Противоположностью *контравариантности* является *ковариантность*, и именно ее следует ожидать от обычного подтипирования:

```
declare let b: string;  
declare let c: string | number;  
  
c = b // ОК
```

Тип `string` является подтипом `string | number`; все элементы `string` появляются в `string | number`, поэтому мы можем присвоить `b` значению `c`. Тип `c` по-прежнему ведет себя так, как мы планировали изначально. Это и есть ковариация.

Однако код не работает:

```
type Fun<X> = (...args: X[]) => void;  
  
declare let f: Fun<string>;  
declare let g: Fun<string | number>;  
  
g = f // не может быть присвоено
```

Мы не можем присвоить `f` значению `g`, поскольку тогда смогли бы вызывать `f` с параметром типа `number`! Мы упускаем часть контракта `g`. Это и есть контравариантность.

Интересно то, что контравариантность фактически работает как пересечение: если `f` принимает `string`, а `g` — `string | number`, то тип, который принимается обоими, — это `(string | number) & string`, то есть `string`.

КОВАРИАНТНОСТЬ И КОНТРАВАРИАНТНОСТЬ

На сайте [Originate.com](https://originate.com) утверждается, что «дисперсия определяет, насколько экземпляры параметризованных типов являются подтипами или супертипами друг друга».

TypeScript использует дисперсию, чтобы проверить, можно ли обосновать типы для другого типа в выражении. Наряду с описанием этого рецепта приведен рис. 8.4, основанный на материалах Университета Райса (<https://oreil.ly/ftfP7>) и показывающий, как работают ковариантность и контравариантность.



Рис. 8.4. Ковариантность и контравариантность объясняются с помощью потребителей и производителей

Когда мы помещаем типы в контравариантные позиции внутри условного типа, TypeScript создает из них *пересечение*. Это означает следующее: мы *делаем вывод* из аргумента функции, поэтому TypeScript знает, что мы должны выполнить полный контракт, создавая пересечение всех составляющих в объединении.

По сути, мы движемся от *объединения к пересечению*.

Разберем это по порядку:

```
type UnionToIntersection<T> =
  (T extends any ? (x: T) => any : never) extends
    (x: infer R) => any ? R : never;

type Intersected = UnionToIntersection<Video["urls"]>;

// эквивалентно

type Intersected = UnionToIntersection<
  { format320p: URL } |
  { format480p: URL } |
  { format720p: URL } |
  { format1080p: URL }
>;
```

У нас есть голый тип; это значит, мы можем объединять условные выражения:

```
type Intersected =
  | UnionToIntersection<{ format320p: URL }>
  | UnionToIntersection<{ format480p: URL }>
  | UnionToIntersection<{ format720p: URL }>
  | UnionToIntersection<{ format1080p: URL }>;
```

Расширим `UnionToIntersection<T>`:

```
type Intersected =
  | ({ format320p: URL } extends any ?
    (x: { format320p: URL }) => any : never) extends
    (x: infer R) => any ? R : never
  | ({ format480p: URL } extends any ?
    (x: { format480p: URL }) => any : never) extends
    (x: infer R) => any ? R : never
  | ({ format720p: URL } extends any ?
    (x: { format720p: URL }) => any : never) extends
    (x: infer R) => any ? R : never
  | ({ format1080p: URL } extends any ?
    (x: { format1080p: URL }) => any : never) extends
    (x: infer R) => any ? R : never;
```

И вычислим первое условное выражение:

```
type Intersected =
  | ((x: { format320p: URL }) => any) extends (x: infer R) => any ? R : never
  | ((x: { format480p: URL }) => any) extends (x: infer R) => any ? R : never
  | ((x: { format720p: URL }) => any) extends (x: infer R) => any ? R : never
  | ((x: { format1080p: URL }) => any) extends (x: infer R) => any ? R : never;
```

Вычислим второе условное выражение, из которого выведем R:

```
type Intersected =
  | { format320p: URL } | { format480p: URL }
  | { format720p: URL } | { format1080p: URL };
```

Но подождите! R выводится из контравариантной позиции. Нам нужно сделать пересечение, в противном случае мы потеряем совместимость типов:

```
type Intersected =
  { format320p: URL } & { format480p: URL } &
  { format720p: URL } & { format1080p: URL };
```

И это именно то, что мы искали! Итак, применительно к нашему исходному примеру:

```
type FormatKeys = keyof UnionToIntersection<Video["urls"]>;
```

FormatKeys теперь "format320p" | "format480p" | "format720p" | "format1080p". Всякий раз, когда мы добавляем еще один формат к исходному объединению, тип FormatKeys автоматически обновляется. Сохранив один раз, вы можете использовать его везде.

8.8. Использование библиотеки type-fest

Задача

Вам так нравятся вспомогательные типы, что вы хотите создать библиотеку утилит для быстрого доступа к ним.

Решение

Скорее всего, в type-fest уже есть все, что вам нужно.

Обсуждение

Основная идея этой главы заключалась в том, чтобы познакомить вас с несколькими полезными вспомогательными типами, которые не являются частью стандартного TypeScript, но доказали свою высокую гибкость для многих сценариев использования: одноцелевые универсальные вспомогательные типы, которые можно комбинировать для получения производных типов на основе ваших существующих моделей. Вы пишете свои модели один раз, а все остальные типы обновляются автоматически. Эта идея создания *простых в сопровождении типов* (путем

выведения типов из других) уникальна для TypeScript и по достоинству оценена множеством разработчиков, создающих сложные приложения или библиотеки.

Возможно, вы будете часто использовать вспомогательные типы, поэтому для удобства начнете объединять их в служебную библиотеку, но, скорее всего, в одной из существующих библиотек уже есть все, что вам нужно. В использовании четко определенного набора вспомогательных типов нет ничего нового, и многие из них предоставляют вам все, что вы видели в этой главе. В одних случаях это те же элементы, но имеющие другое название; в других это похожая идея, но решенная иначе. Базовые элементы, скорее всего, есть во всех библиотеках типов, но одна библиотека, `type-fest` (<https://oreil.ly/Cw4Kc>), не только полезна, но и активно поддерживается, хорошо документирована и широко используется.

Эта библиотека имеет несколько отличительных особенностей, которые выделяют ее на фоне других. Во-первых, она имеет подробную документацию, в которой не только описывается *использование* того или иного вспомогательного типа, но и приведены примеры и сценарии, которые подскажут, где вы можете использовать этот вспомогательный тип. В качестве одного из примеров приведен `Integer<T>`, который гарантирует, что число, которое вы предоставите, не будет содержать дробной части.

Это служебный тип, который почти вошел в данную книгу, но я увидел, что фрагмент кода из `type-fest` расскажет вам все, что нужно знать об этом типе:

```
/**
A `number` that is an integer.
You can't pass a `bigint` as they are already guaranteed to be integers.
Use-case: Validating and documenting parameters.

@example
```
import type {Integer} from 'type-fest';
declare function setYear<T extends number>(length: Integer<T>): void;
```

@see NegativeInteger
@see NonNegativeInteger
@category Numeric
*/
// `${bigint}` is a type that matches a valid bigint
// literal without the `n` (ex. 1, 0b1, 0o1, 0x1)
// Because T is a number and not a string we can effectively use
// this to filter out any numbers containing decimal points

export type Integer<T extends number> = `${T}` extends `${bigint}` ? T : never;
```

В остальной части файла рассматриваются отрицательные целые числа, неотрицательные целые числа, числа с плавающей точкой и т. д. Это настоящий клад информации, если вы хотите узнать больше о том, как создаются типы.

Во-вторых, библиотека `type-fest` имеет дело с крайними случаями. В рецепте 8.2 вы узнали о рекурсивных типах и определили `DeepPartial<T>`. Его аналог `PartialDeep<T>`, описанный в стиле `type-fest`, немного более обширен:

```
export type PartialDeep<T, Opts extends PartialDeepOptions = {}> =
  T extends BuiltIns
    ? T
    : T extends Map<infer KeyType, infer ValueType>
      ? PartialMapDeep<KeyType, ValueType, Opts>
      : T extends Set<infer ItemType>
        ? PartialSetDeep<ItemType, Opts>
        : T extends ReadonlyMap<infer KeyType, infer ValueType>
          ? PartialReadonlyMapDeep<KeyType, ValueType, Opts>
          : T extends ReadonlySet<infer ItemType>
            ? PartialReadonlySetDeep<ItemType, Opts>
            : T extends (...arguments: any[]) => unknown
              ? T | undefined
              : T extends object
                ? T extends ReadonlyArray<infer ItemType>
                  ? Opts['recurseIntoArrays'] extends true
                    ? ItemType[] extends T
                      ? readonly ItemType[] extends T
                        ? ReadonlyArray<PartialDeep<ItemType | undefined, Opts>>
                          : Array<PartialDeep<ItemType | undefined, Opts>>
                        : PartialObjectDeep<T, Opts>
                      : T
                    : PartialObjectDeep<T, Opts>
                  : unknown;

/**
 * Same as `PartialDeep`, but accepts only `Map`s and as inputs.
 * Internal helper for `PartialDeep`.
 */
type PartialMapDeep<KeyType, ValueType, Options extends PartialDeepOptions> =
  {} & Map<PartialDeep<KeyType, Options>, PartialDeep<ValueType, Options>>;

/**
 * Same as `PartialDeep`, but accepts only `Set`s as inputs.
 * Internal helper for `PartialDeep`.
 */
type PartialSetDeep<T, Options extends PartialDeepOptions> =
  {} & Set<PartialDeep<T, Options>>;

/**
 * Same as `PartialDeep`, but accepts only `ReadonlyMap`s as inputs.
 * Internal helper for `PartialDeep`.
 */
type PartialReadonlyMapDeep<
  KeyType, ValueType,
  Options extends PartialDeepOptions
> = {} & ReadonlyMap<
```

```

    PartialDeep<KeyType, Options>,
    PartialDeep<ValueType, Options>
  >;

/**
Same as `PartialDeep`, but accepts only `ReadOnlySet`s as inputs.
Internal helper for `PartialDeep`.
*/
type PartialReadOnlySetDeep<T, Options extends PartialDeepOptions> =
  {} & ReadOnlySet<PartialDeep<T, Options>>;

/**
Same as `PartialDeep`, but accepts only `object`s as inputs.
Internal helper for `PartialDeep`.
*/
type PartialObjectDeep<
  ObjectType extends object,
  Options extends PartialDeepOptions
> = {
  [KeyType in keyof ObjectType]?: PartialDeep<ObjectType[KeyType], Options>
};

```

Нет необходимости подробно рассматривать всю эту реализацию, но она должна дать вам представление о том, насколько надежны реализации библиотеки `type-fest` для определенных типов утилит.



Тип `PartialDeep<T>` очень обширный и учитывает все возможные крайние случаи, но за это приходится расплачиваться тем, что он сложен и его трудно обрабатывать программе проверки типов TypeScript. В зависимости от вашего сценария использования, возможно, вам подойдет более простая версия из рецепта 8.2.

В-третьих, эти библиотеки не добавляют вспомогательные типы только ради того, чтобы их добавить. В файле `Readme` есть список отклоненных типов и обоснование отказа: либо случаи использования ограничены, либо существуют лучшие альтернативы. Как и во всем остальном, разработчики библиотеки `type-fest` очень, очень хорошо документируют свой выбор.

В-четвертых, библиотека `type-fest` рассказывает о существующих вспомогательных типах. Как таковые, вспомогательные типы были в TypeScript всегда, но в прошлом почти не документировались. Много лет назад мой блог (<https://oreil.ly/eRtx9>) пытался стать источником информации о встроенных вспомогательных типах, пока в официальной документации (<https://oreil.ly/K5cXq>) не была добавлена глава о служебных типах (<https://oreil.ly/K5cXq>). Служебные типы — не то, что можно легко освоить, просто используя TypeScript. Вы должны понимать, что они существуют, и читать о них. В документации к `type-fest` есть целый раздел, посвященный встроенным типам, с примерами и сценариями использования.

И последнее, но не менее важное: библиотека широко распространена и совершенствуется надежными разработчиками программного обеспечения с открытым исходным кодом. Ее создатель Синдре Сорхус (<https://oreil.ly/thSin>) работает над проектами с открытым исходным кодом уже несколько десятилетий, и многие из них совершенно удивительны. Библиотека `type-fest` — еще один гениальный ход. Скорее всего, бóльшая часть вашей работы зависит от работы Синдре.

Используя `type-fest`, вы получаете еще один ресурс вспомогательных типов, которые можете добавить в свой проект. Решите сами, чего вы хотите: сохранить небольшой набор вспомогательных типов или использовать реализации сообщества.

Стандартная библиотека и определения внешних типов

Ведущий архитектор TypeScript Андерс Хейлсберг однажды сказал, что, по его мнению, «TypeScript — это Швейцария JavaScript». Это значит, что он не отдает предпочтения какому-то одному фреймворку и не стремится к совместимости с ним, а скорее пытается удовлетворить потребности всех фреймворков JavaScript и их разновидностей. В прошлом TypeScript работал над реализацией декораторов, чтобы убедить Google отказаться от использования диалекта JavaScript AtScript (<https://oreil.ly/ZrcKR>) для Angular, который представлял собой TypeScript плюс декораторы. Вдобавок реализация декораторов TypeScript послужила шаблоном для соответствующего предложения ECMAScript по декораторам (<https://oreil.ly/76JuE>). Кроме того, TypeScript поддерживает расширение синтаксиса JSX, что позволяет таким фреймворкам, как React или Preact, использовать TypeScript без ограничений.

Но даже если TypeScript попытается удовлетворить потребности всех разработчиков JavaScript и приложит огромные усилия по интеграции новых и полезных функций для множества фреймворков, все равно есть то, что он не может или не хочет делать. Возможно, потому что определенная функция является нишевой или из-за того, что решение будет иметь огромные последствия для слишком большого количества разработчиков.

Именно поэтому TypeScript был спроектирован так, чтобы по умолчанию быть расширяемым. Многие его функции, такие как пространства имен, модули и интерфейсы, позволяют объединять декларации, что дает разработчикам возможность добавлять собственные определения типов.

В этой главе мы рассмотрим, как TypeScript работает со стандартными функциями JavaScript, такими как модули, массивы и объекты. Мы рассмотрим некоторые из их ограничений, проанализируем причины, лежащие в их основе, и исследуем разумные обходные пути. Вы увидите, что TypeScript был разработан таким образом, чтобы быть очень гибким для различных разновидностей JavaScript, начиная с разумных настроек по умолчанию и предоставляя возможность расширять его при необходимости.

9.1. Перебор объектов с помощью `Object.keys`

Задача

Когда вы пытаетесь получить доступ к свойствам объекта путем перебора его ключей, TypeScript выдает вам красные волнистые линии, сообщающие, что `'string'` не может быть использована для индексирования типа.

Решение

Используйте цикл `for-in` вместо `Object.keys` и заблокируйте свой тип с помощью параметров обобщенного типа.

Обсуждение

В TypeScript часто встречается попытка получить доступ к свойству объекта с помощью перебора его ключей. Этот паттерн весьма распространен в JavaScript, однако TypeScript, кажется, любой ценой удерживает разработчиков от его использования. Добавим эту простую строку кода для перебора свойств объекта:

```
Object.keys(person).map(k => person[k])
```

Это приводит к тому, что TypeScript выбрасывает красные волнистые линии, а разработчики изучают таблицы: «Элемент неявно имеет тип `'any'`, поскольку выражение типа `'string'` не может быть использовано для индексации типа `'Person'`». В такой ситуации опытным JavaScript разработчикам кажется, что TypeScript работает против них. Но, как и во всех его решениях, есть веская причина, по которой он ведет себя подобным образом.

Разберемся, почему он так делает. Взгляните на эту функцию:

```
type Person = {
  name: string;
  age: number;
};

function printPerson(p: Person) {
  Object.keys(p).forEach((k) => {
    console.log(k, p[k]);
    //           ^
    // Element implicitly has an 'any' type because expression
    // of type 'string' can't be used to index type 'Person'.
  });
}
```


Все, чего мы хотим, — распечатать поля `Person`, получив доступ к ним через его ключи. TypeScript не позволяет этого сделать. `Object.keys(p)` возвращает тип `string[]`, который слишком широк для доступа к определенной форме объекта `Person`.

Но почему так? Разве не очевидно, что мы обращаемся только к тем ключам, которые доступны? В этом весь смысл использования `Object.keys`! Да, но мы также можем передавать объекты, являющиеся подтипами `Person`, которые могут иметь больше свойств, чем определено в `Person`:

```
const me = {
  name: "Stefan",
  age: 40,
  website: "https://fettblog.eu",
};

printPerson(me); // Все хорошо!
```

`printPerson` по-прежнему должен работать корректно. Он выводит больше свойств, но не ломается. Это все еще ключи `p`, поэтому каждое свойство должно быть доступно. Но что, если вы обращаетесь не только к `p`?

Предположим, что `Object.keys` дает вам `(keyof Person)[]`. Вы можете легко написать что-то вроде этого:

```
function printPerson(p: Person) {
  const you: Person = {
    name: "Reader",
    age: NaN,
  };

  Object.keys(p).forEach((k) => {
    console.log(k, you[k]);
  });
}

const me = {
  name: "Stefan",
  age: 40,
  website: "https://fettblog.eu",
};

printPerson(me);
```

Если `Object.keys(p)` возвращает массив типа `keyof Person[]`, то вы сможете получить доступ и к другим объектам `Person`. Это может не сработать. В нашем примере мы просто выводим `undefined`. Но что, если вы попытаетесь что-то сделать с этими значениями? Это приведет к сбою во время выполнения.

TypeScript предотвращает подобные сценарии. Хотя мы можем подумать, что `Object.keys` — это `keyof Person`, на самом деле это может быть нечто большее.

Один из способов решения этой проблемы — использовать защиту типов:

```
function isKey<T>(x: T, k: PropertyKey): k is keyof T {
    return k in x;
}

function printPerson(p: Person) {
    Object.keys(p).forEach((k) => {
        if (isKey(p, k)) console.log(k, p[k]); // Все отлично!
    });
}
```

Но этот код добавляет дополнительный шаг, которого, честно говоря, не должно быть.

Есть еще один способ перебора объектов — с помощью циклов `for-in`:

```
function printPerson(p: Person) {
    for (let k in p) {
        console.log(k, p[k]);
    }
    //           ^
    // Element implicitly has an 'any' type because expression
    // of type 'string' can't be used to index type 'Person'.
}
```

TypeScript выдаст ту же ошибку по той же причине, поскольку вы все еще можете делать подобные вещи:

```
function printPerson(p: Person) {
    const you: Person = {
        name: "Reader",
        age: NaN,
    };

    for (let k in p) {
        console.log(k, you[k]);
    }
}

const me = {
    name: "Stefan",
    age: 40,
    website: "https://fettblog.eu",
};

printPerson(me);
```

И этот код приведет к поломке во время выполнения. Однако, написав его таким образом, вы получаете небольшое преимущество перед версией `Object.keys`. TypeScript может быть гораздо более точным в этом сценарии, если вы добавите обобщенный тип:

```
function printPerson<T extends Person>(p: T) {
  for (let k in p) {
    console.log(k, p[k]); // Это работает
  }
}
```

Вместо того чтобы требовать от `p` быть типа `Person` (и, следовательно, совместимым со всеми подтипами `Person`), мы добавляем новый параметр обобщенного типа `T`, который является подтипом `Person`. Это означает, что все типы, которые были совместимы с этой сигнатурой функции, по-прежнему совместимы, но в момент использования `p` мы имеем дело с явным подтипом, а не с более широким супертипом `Person`.

Мы заменяем `T` на то, что совместимо с `Person`, но при этом TypeScript знает, что оно достаточно конкретное, чтобы предотвратить ошибки.

Предыдущий код работает. `k` имеет тип `keyof T`. Поэтому мы можем получить доступ к `p`, который имеет тип `T`. И этот метод по-прежнему не позволяет нам получить доступ к типам, у которых отсутствуют определенные свойства:

```
function printPerson<T extends Person>(p: T) {
  const you: Person = {
    name: "Reader",
    age: NaN,
  };
  for (let k in p) {
    console.log(k, you[k]);
  }
  //           ^
  // Type 'Extract<keyof T, string>' cannot be used to index type 'Person'
}
```

Мы не можем получить доступ к `Person` с `keyof T`. Они могут быть разными. Но поскольку `T` — подтип `Person`, то мы все равно можем присваивать свойства, если знаем точные их имена:

```
p.age = you.age
```

И это именно то, что нам нужно.

TypeScript очень консервативно относится к своим типам, и на первый взгляд такая его черта может показаться странной, но она поможет вам в сценариях, о которых вы даже не подозревали. Думаю, именно в этой части разработчики JavaScript обычно ругают компилятор и думают, что «борются» с ним, но, возможно, TypeScript спас вас, а вы даже не подозревали об этом. В ситуациях, когда это начинает раздражать, TypeScript, по крайней мере, дает вам способы обойти это.

9.2. Явное выделение небезопасных операций с помощью утверждений типов и `unknown`

Задача

Синтаксический анализ произвольных данных с помощью операций JSON может привести к ошибкам, если данные будут неверными. Значения по умолчанию в TypeScript не предоставляют никаких мер защиты от таких небезопасных операций.

Решение

Явно выделяйте небезопасные операции, используя утверждения типов вместо аннотаций типов, и обеспечивайте их принудительное применение, исправляя исходные типы с помощью `unknown`.

Обсуждение

В рецепте 3.9 мы говорили о том, как эффективно использовать утверждения типов — явный вызов системы типов, призванный сообщить, что некий тип должен быть другим. На основании какого-то набора защитных механизмов (например, если не указано, что `number` на самом деле является `string`) TypeScript будет обрабатывать это конкретное значение как новый тип.

Благодаря богатой и разветвленной системе типов в TypeScript утверждения типов иногда неизбежны. Порой они даже необходимы, как показано в рецепте 3.9, где мы используем `API fetch` для получения JSON-данных от серверной части. Один из способов — вызвать функцию `fetch` и присвоить результаты аннотированному типу:

```
type Person = {
  name: string;
  age: number;
};
```

```
const ppl: Person[] = await fetch("/api/people").then((res) => res.json());
```

Функция `res.json()` выдает значение `any`¹, и все, что имеет это значение, можно изменить на любой другой тип с помощью аннотации типа. Нет никакой гарантии, что результат действительно равен `Person[]`.

¹ В то время, когда создавалось определение API, `unknown` еще не существовало. Кроме того, в TypeScript большое внимание уделяется продуктивности разработчиков, а поскольку `res.json()` является широко используемым методом, это привело бы к *сбоям* в работе бесчисленного множества приложений.

Другой способ — использовать утверждение типа вместо аннотации типа:

```
const ppl = await fetch("/api/people").then((res) => res.json()) as Person[];
```

С точки зрения системы типов это то же самое, но мы можем легко отследить ситуации, в которых могут возникнуть проблемы. Если мы не проверяем входящие значения на соответствие типам (например, с помощью `Zod`; см. рецепт 12.5), то наличие утверждения типа здесь — эффективный способ выявления небезопасных операций.

Небезопасные операции в системе типов — это ситуации, когда мы сообщаем системе типов, что ожидаем значения, имеющие определенный тип, но у нас нет никаких гарантий от самой системы, что это действительно будет так. В основном это происходит на границах нашего приложения, когда мы откуда-то загружаем данные, обрабатываем пользовательский ввод или анализируем данные с помощью встроенных методов.

Небезопасные операции можно выделить с помощью определенных ключевых слов, которые указывают на явное изменение в системе типов. Утверждения типов (`as`), предикаты типов (`is`) или сигнатуры утверждений (`asserts`) помогают нам находить такие ситуации. В некоторых случаях TypeScript даже заставляет нас либо соответствовать его представлению о типах, либо явно изменить правила в зависимости от наших ситуаций. Но не всегда.

Когда мы извлекаем данные из какого-либо серверного интерфейса, аннотировать их так же просто, как и написать утверждение типа. Подобные вещи можно упустить из виду, если не заставлять себя использовать правильный прием.

Но мы можем помочь TypeScript сделать все правильно. Проблема состоит в вызове `res.json()`, который поступает из интерфейса `Body` в файл `lib.dom.d.ts`:

```
interface Body {
  readonly body: ReadableStream<Uint8Array> | null;
  readonly bodyUsed: boolean;
  arrayBuffer(): Promise<ArrayBuffer>;
  blob(): Promise<Blob>;
  formData(): Promise<FormData>;
  json(): Promise<any>;
  text(): Promise<string>;
}
```

Вызов `json()` возвращает `Promise<any>`, а `any` — это нестрогий тип, в котором TypeScript просто игнорирует любую проверку типов. Нам понадобится `unknown`. Благодаря объединению деклараций мы можем переопределить определение типа `Body` и сделать `json()` чуть более строгим:

```
interface Body {
  json(): Promise<unknown>;
}
```

В тот момент, когда мы создаем аннотацию типа, TypeScript сообщит, что мы не можем присвоить `Person[]` значение `unknown`:

```
const ppl: Person[] = await fetch("/api/people").then((res) => res.json());  
//    ^  
// Type 'unknown' is not assignable to type 'Person[]'.ts(2322)
```

Но TypeScript все равно будет рад, если мы сделаем утверждение типа:

```
const ppl = await fetch("/api/people").then((res) => res.json()) as Person[];
```

И с помощью этого кода мы можем заставить TypeScript выделять небезопасные операции¹.

9.3. Работа с `defineProperty`

Задача

Вы определяете свойства в процессе работы, используя `Object.defineProperty`, но TypeScript не распознает изменения.

Решение

Создайте функцию-обертку и используйте утверждения сигнатуры, чтобы изменить тип объекта.

Обсуждение

В JavaScript вы можете определять свойства объектов динамически с помощью `Object.defineProperty`. Это полезно, если вы хотите, чтобы ваши свойства были доступны только для чтения. Представьте объект, хранящий максимальное значение, которое не следует перезаписывать:

```
const storage = {  
  currentValue: 0  
};  
  
Object.defineProperty(storage, 'maxValue', {  
  value: 9001,
```

¹ Благодарю Дэна Вандеркама за его блог Effective TypeScript (<https://effectivetypescript.com/>), вдохновивший меня на эту тему.

```

    writable: false
  });
  console.log(storage.maxValue); // 9001

  storage.maxValue = 2;

  console.log(storage.maxValue); // все еще 9001

```

`defineProperty` и дескрипторы свойств очень сложны. Они позволяют вам делать со свойствами все то, что обычно зарезервировано для встроенных объектов. Поэтому они распространены в больших кодовых базах. В TypeScript есть проблема с `defineProperty`:

```

const storage = {
  currentValue: 0
};

Object.defineProperty(storage, 'maxValue', {
  value: 9001,
  writable: false
});

console.log(storage.maxValue);
//           ^
// Property 'maxValue' does not exist on type '{ currentValue: number; }'.

```

Если мы явно не утверждаем новый тип, то не получим значения `maxValue`, привязанного к типу `storage`. Однако для простых сценариев использования мы можем помочь себе, используя утверждения сигнатуры.



Несмотря на то что в TypeScript могут отсутствовать изменения объектов при использовании `Object.defineProperty`, есть вероятность, что в будущем команда добавит типизацию или специальное поведение для подобных случаев. Например, проверка наличия у объекта определенного свойства с помощью ключевого слова `in` в течение многих лет не влияла на типы. Все изменилось в 2022 году после выхода TypeScript 4.9.

Представьте функцию `assertIsNumber`, с помощью которой можно убедиться, что какое-то значение имеет тип `number`. В противном случае выдается ошибка. Этот код аналогичен функции `assert` в `Node.js`:

```

function assertIsNumber(val: any) {
  if (typeof val !== "number") {
    throw new AssertionError("Not a number!");
  }
}

```

```
function multiply(x, y) {
  assertIsNumber(x);
  assertIsNumber(y);
  // at this point I'm sure x and y are numbers
  // if one assert condition is not true, this position
  // is never reached
  return x * y;
}
```

Чтобы соответствовать такому поведению, мы можем добавить утверждение сигнатуры, которое сообщает TypeScript, что мы знаем больше о типе после этой функции:

```
function assertIsNumber(val: any) : asserts val is number
  if (typeof val !== "number") {
    throw new AssertionError("Not a number!");
  }
}
```

Этот код во многом похож на предикаты типов (см. рецепт 3.5), но не имеет потока управления, свойственного структурам, основанным на условиях, таким как `if` или `switch`:

```
function multiply(x, y) {
  assertIsNumber(x);
  assertIsNumber(y);
  // Теперь TypeScript также знает, что и x, и y являются числами
  return x * y;
}
```

Если присмотреться, то можно увидеть, что утверждения сигнатуры могут *менять тип параметра или переменной динамически*. То же самое делает и `Object.defineProperty`.

Приведенный ниже вспомогательный инструмент не претендует на 100%-ную точность и полноту. В нем могут быть ошибки, и он может не охватывать все крайние случаи спецификации `defineProperty`.

Но этот инструмент предоставит базовую функциональность. Сначала мы определим новую функцию `defineProperty`, которая будет использоваться в качестве функции-обертки для `Object.defineProperty`:

```
function defineProperty<
  Obj extends object,
  Key extends PropertyKey,
  PDesc extends PropertyDescriptor>
  (obj: Obj, prop: Key, val: PDesc) {
  Object.defineProperty(obj, prop, val);
}
```


Мы работаем с тремя обобщенными типами:

- объект, который мы хотим изменить, имеет тип `Obj`, являющийся подтипом типа `object`;
- тип `Key`, который является подтипом `PropertyKey` (встроенного типа): `string | number | symbol`;
- `PDesc`, подтип `PropertyDescriptor` (встроенный тип). Это позволяет нам определить свойство со всеми его возможностями (возможность записи, перечисления, реконфигурации).

Мы используем обобщенные типы, поскольку TypeScript позволяет сузить их до очень конкретного типа объектов. Например, `PropertyKey` — это все числа, строки и символы. Но если мы используем `Key extends PropertyKey`, то можем точно определить, что `prop`, например, имеет тип `"maxValue"`. Это будет полезно, если мы захотим изменить исходный тип, добавив дополнительные свойства.

Функция `Object.defineProperty` либо изменяет объект, либо выдает ошибку, если что-то пошло не так. Это именно то, что делает функция утверждения. Наш пользовательский помощник `defineProperty` делает то же самое.

Добавим утверждение сигнатуры. После успешного выполнения `defineProperty` у нашего объекта появится еще одно свойство. Для этого мы создаем несколько вспомогательных типов. Сначала сигнатура:

```
function defineProperty<
  Obj extends object,
  Key extends PropertyKey,
  PDesc extends PropertyDescriptor>
  (obj: Obj, prop: Key, val: PDesc):
    asserts obj is Obj & DefineProperty<Key, PDesc> {
  Object.defineProperty(obj, prop, val);
}
```

Таким образом, `obj` имеет тип `Obj` (суженный с помощью обобщенного типа) и определенное нами новое свойство.

Это вспомогательный тип `DefineProperty`:

```
type DefineProperty<
  Prop extends PropertyKey,
  Desc extends PropertyDescriptor> =
  Desc extends { writable: any, set(val: any): any } ? never :
  Desc extends { writable: any, get(): any } ? never :
  Desc extends { writable: false } ? Readonly<InferValue<Prop, Desc>> :
  Desc extends { writable: true } ? InferValue<Prop, Desc> :
  Readonly<InferValue<Prop, Desc>>;
```

Сначала мы разберемся со свойством `writable` дескриптора `PropertyDescriptor`. Это набор условий для определения некоторых крайних случаев и условий работы исходных дескрипторов свойств.

- Если мы установим `writable` и любой аксессор свойства (`get`, `set`), то потерпим неудачу. Тип `never` сообщает нам, что произошла ошибка.
- Если мы установим для параметра `writable` значение `false`, то свойство будет доступно только для чтения. Мы используем вспомогательный тип `InferValue`.
- Если мы установим значение `writable` в `true`, то свойство не будет предназначено только для чтения. Мы также считаемся с этим.
- Последний случай по умолчанию такой же, как и `writable: false`, поэтому `ReadOnly<InferValue<Prop, Desc>>`. (`ReadOnly<T>` является встроенным.)

Это вспомогательный тип `InferValue`, работающий с аксессором `set` свойства `value`:

```
type InferValue<Prop extends PropertyKey, Desc> =
  Desc extends { get(): any, value: any } ? never :
  Desc extends { value: infer T } ? Record<Prop, T> :
  Desc extends { get(): infer T } ? Record<Prop, T> : never;
```

Снова появляется набор условий.

- Есть ли у нас геттер и набор значений? `Object.defineProperty` выдает ошибку, так что выдается значение `never`.
- Если мы установили значение, то определим тип этого значения и создадим объект с определенным нами ключом свойства и типом значения.
- Или мы выводим тип по возвращаемому типу геттера.
- Обо всем остальном мы забываем. TypeScript не позволит нам работать с объектом, поскольку он становится `never`.

Множество вспомогательных типов, но примерно 20 строк кода, чтобы сделать все правильно:

```
type InferValue<Prop extends PropertyKey, Desc> =
  Desc extends { get(): any, value: any } ? never :
  Desc extends { value: infer T } ? Record<Prop, T> :
  Desc extends { get(): infer T } ? Record<Prop, T> : never;
```

```
type DefineProperty<
  Prop extends PropertyKey,
  Desc extends PropertyDescriptor> =
  Desc extends { writable: any, set(val: any): any } ? never :
  Desc extends { writable: any, get(): any } ? never :
  Desc extends { writable: false } ? ReadOnly<InferValue<Prop, Desc>> :
```

```
Desc extends { writable: true } ? InferValue<Prop, Desc> :
  Readonly<InferValue<Prop, Desc>>
```

```
function defineProperty<
  Obj extends object,
  Key extends PropertyKey,
  PDesc extends PropertyDescriptor>
  (obj: Obj, prop: Key, val: PDesc):
    asserts obj is Obj & DefineProperty<Key, PDesc> {
  Object.defineProperty(obj, prop, val)
}
```

Посмотрим, что TypeScript сделает с нашими изменениями:

```
const storage = {
  currentValue: 0
};

defineProperty(storage, 'maxValue', {
  writable: false, value: 9001
});

storage.maxValue; // Это число
storage.maxValue = 2; // Ошибка! Только для чтения

const storageName = 'My Storage';
defineProperty(storage, 'name', {
  get() {
    return storageName
  }
});

storage.name; // Это строка!

// невозможно присвоить значение и геттер
defineProperty(storage, 'broken', {
  get() {
    return storageName
  },
  value: 4000
});

// storage – never, так как у нас есть вредоносный
// дескриптор свойства
storage;
```

Возможно, этот код может не охватить все ситуации, но с простыми определениями свойств уже можно работать.

9.4. Расширение типов для `Array.prototype.includes`

Задача

TypeScript не сможет найти элемент широкого типа, например `string` или `number`, в очень узком кортеже или массиве.

Решение

Создавайте обобщенные вспомогательные функции, используя предикаты типов, в которых изменяете взаимосвязь параметров типа.

Обсуждение

Мы создаем массив `actions`, содержащий набор действий в строковом формате, которые мы хотим выполнить. Итоговый тип этого массива `actions` — `string[]`. Функция `execute` принимает в качестве аргумента любую строку. Мы проверяем, является ли она допустимым действием, и если да, то делаем что-нибудь:

```
// actions: string[]
const actions = ["CREATE", "READ", "UPDATE", "DELETE"];

function execute(action: string) {
  if (actions.includes(action)) {
    // сделайте что-нибудь с action
  }
}
```

Все становится немного сложнее, если мы хотим сузить строку `string[]` до чего-то более конкретного, подмножества всех возможных строк. Добавив контекст `const` с помощью `as const`, мы можем сузить `actions` до типа `readonly ["CREATE", "READ", "UPDATE", "DELETE"]`. Это удобно, если мы хотим выполнить проверку на полноту, чтобы убедиться в наличии вариантов для всех доступных действий. Однако `actions.includes` с нами не согласен:

```
// Добавление контекста const
// actions: readonly ["CREATE", "READ", "UPDATE", "DELETE"]
const actions = ["CREATE", "READ", "UPDATE", "DELETE"] as const;

function execute(action: string) {
  if (actions.includes(action)) {
//      ^
// Argument of type 'string' is not assignable to parameter of type
// '"CREATE" | "READ" | "UPDATE" | "DELETE"'.(2345)
  }
}
```

Почему это так? Посмотрим на типы `Array<T>` и `readonly Array<T>` (мы работаем с последним из-за контекста `const`):

```
interface Array<T> {
  /**
   * Determines whether an array includes a certain element,
   * returning true or false as appropriate.
   * @param searchElement The element to search for.
   * @param fromIndex The position in this array at which
   * to begin searching for searchElement.
   */
  includes(searchElement: T, fromIndex?: number): boolean;
}

interface ReadonlyArray<T> {
  /**
   * Determines whether an array includes a certain element,
   * returning true or false as appropriate.
   * @param searchElement The element to search for.
   * @param fromIndex The position in this array at which
   * to begin searching for searchElement.
   */
  includes(searchElement: T, fromIndex?: number): boolean;
}
```

Элемент, который мы хотим найти (`searchElement`), должен быть того же типа, что и сам массив! Таким образом, если у нас есть `Array<string>` (или `string[]`, или `ReadonlyArray<string>`), то мы можем искать только строки. В нашем случае это означает, что `action` должно быть типа `"CREATE" | "READ" | "UPDATE" | "DELETE"`.

Внезапно наша программа перестала иметь смысл. Зачем искать что-то, если тип уже говорит нам, что это может быть только одна из четырех строк? Если мы изменим тип `action` на `"CREATE" | "READ" | "UPDATE" | "DELETE"`, то `actions.includes` станет ненужной. Если мы не изменим ее, то TypeScript выдаст нам ошибку, и это справедливо!

Одна из проблем состоит в том, что в TypeScript отсутствует возможность проверять наличие контравариантных типов с помощью, например, обобщенных типов с верхней границей. Мы можем определить, должен ли тип быть *подмножеством* типа `T`, с помощью таких конструкций, как `extends`; мы не можем проверить, является ли тип надмножеством `T`. По крайней мере, пока нет!

Итак, что же мы можем сделать?

Вариант 1. Повторно объявить массив `ReadonlyArray`

Один из вариантов, который приходит на ум, — это изменить поведение `includes` в `ReadonlyArray`. Благодаря объединению деклараций мы можем добавить наши собственные определения для `ReadonlyArray`, которые будут давать больше свободы в аргументах и определять более конкретные результаты, например, так:

```
interface ReadonlyArray<T> {
  includes(searchElement: any, fromIndex?: number): searchElement is T;
}
```

Это позволяет передавать более широкий набор значений `searchElement` (буквально любых!), и если условие истинно, то мы сообщаем TypeScript через *предикат* `includes`, что `searchElement` является `T` (подмножеством, которое мы ищем). Оказывается, этот код работает довольно хорошо:

```
const actions = ["CREATE", "READ", "UPDATE", "DELETE"] as const;

function execute(action: string) {
  if(actions.includes(action)) {
    // action: "CREATE" | "READ" | "UPDATE" | "DELETE"
  }
}
```

Но есть проблема. Решение работает, но требует предположения о том, что правильно, а что нужно проверить. Если вы измените `action` на `number`, то TypeScript обычно выдает ошибку, что вы не можете выполнить поиск по такому типу. `actions` состоит только из `string`, так зачем вообще искать `number`? Это ошибка, которую вы хотите выявить:

```
// тип number вообще не имеет отношения к actions
function execute(action: number) {
  if(actions.includes(action)) {
    // сделайте что-нибудь
  }
}
```

Из-за нашего изменения в `ReadonlyArray` мы теряем эту проверку, поскольку `searchElement` является `any`. Функциональность `action.includes` по-прежнему работает должным образом, но мы можем не заметить нужную *проблему*, если изменим сигнатуры функций по ходу дела.

Таким образом, что более важно, мы меняем поведение встроенных типов. Это может привести к изменению ваших проверок типов в других местах и вызвать проблемы в долгосрочной перспективе!



Если вы выполняете исправление типа, изменяя поведение с помощью стандартной библиотеки, то убедитесь, что этот модуль добавлен в область видимости, а не глобально.

Есть и другой способ.

Вариант 2. Вспомогательная функция с утверждениями типов

Как уже говорилось, одна из проблем состоит в том, что в TypeScript отсутствует возможность проверить, принадлежит ли значение к *надмножеству* параметра

обобщенного типа. С помощью вспомогательной функции мы можем изменить эту взаимосвязь:

```
function includes<T extends U, U>(coll: ReadonlyArray<T>, el: U): el is T {
  return coll.includes(el as T);
}
```

Функция `includes` принимает в качестве аргумента `ReadonlyArray<T>` и выполняет поиск элемента, который имеет тип `U`. С помощью наших общих границ мы проверяем, что `T` расширяет `U`; это значит, что `U` является *надмножеством* `T` (или `T` является *подмножеством* `U`). Если метод возвращает `true`, то мы можем с уверенностью сказать, что `el` относится к *более узкому* типу `U`.

Чтобы реализация работала, нам нужно выполнить только небольшое утверждение типа в тот момент, когда мы передаем `el` в `Array.prototype.includes`. Исходная проблема все еще существует! Однако утверждение типа `el` как `T` — это нормально, поскольку мы проверяем возможные проблемы уже в сигнатуре функции.

Это означает, что в тот момент, когда мы меняем, например, `action` на `number`, получаем правильные ошибки во всем нашем коде:

```
function execute(action: number) {
  if(includes(actions, action)) {
    //      ^
    // Argument of type 'readonly ["CREATE", "READ", "UPDATE", "DELETE"]'
    // is not assignable to parameter of type 'readonly number[]'.
  }
}
```

И это именно то поведение, которое нам нужно. Приятным моментом является то, что TypeScript хочет, чтобы мы изменили массив, а не элемент, который мы ищем. Это связано с взаимосвязью параметров обобщенного типа.



Эти же решения работают, если вы сталкиваетесь с похожими проблемами при работе с `Array.prototype.indexOf`.

TypeScript стремится обеспечить корректное использование всей стандартной функциональности JavaScript, но иногда приходится идти на компромиссы. Описанный выше случай требует компромиссов: допускаете ли вы, что список аргументов будет более свободным, чем вы ожидаете, или выдаете ошибки для типов, о которых вам нужно знать больше?

Утверждения типов, объединение деклараций и другие инструменты помогают обойти эти проблемы в ситуациях, когда система типов не может нам помочь. И так будет до тех пор, пока она не станет лучше, чем раньше, позволяя нам достигать большего в области типов.

9.5. Фильтрация нулевых значений

Задача

Вы хотите использовать логический конструктор для фильтрации нулевых значений из массива, но TypeScript по-прежнему возвращает те же типы, в том числе `null` и `undefined`.

Решение

Перегрузите метод `filter` объекта `Array`, используя объединение объявлений.

Обсуждение

Иногда в коллекциях могут встречаться *нулевые* значения (`undefined` или `null`):

```
// const array: (number | null | undefined)[]
const array = [1, 2, 3, undefined, 4, null];
```

Чтобы продолжить работу, вам необходимо удалить эти нулевые значения из вашей коллекции. Обычно это делается с помощью метода `filter` объекта `Array`, возможно, путем проверки *истинности* значения. Значения `null` и `undefined` являются *ложными*, поэтому отфильтровываются:

```
const filtered = array.filter((val) => !!val);
```

Удобный способ проверить истинность значения — передать его логическому конструктору. Этот код короткий, конкретный и очень элегантно читается:

```
// const array: (number | null | undefined)[]
const filtered = array.filter(Boolean);
```

Но, к сожалению, этот способ не меняет наш тип. У нас по-прежнему есть `null` и `undefined` в качестве возможных типов для отфильтрованного массива.

Открыв интерфейс `Array` и создав другое объявление для `filter`, мы можем добавить этот особый случай в качестве перегрузки:

```
interface Array<T> {
  filter(predicate: BooleanConstructor): NonNullable<T>[]
}

interface ReadonlyArray<T> {
  filter(predicate: BooleanConstructor): NonNullable<T>[]
}
```


Таким образом, мы избавляемся от нулевых типов и имеем более четкое представление о типе содержимого нашего массива:

```
// const array: number[]
const filtered = array.filter(Boolean);
```

Отлично! В чем нюанс? В литеральных кортежах и массивах.

`BooleanConstructor` фильтрует не только нулевые, но и ложные значения. Чтобы получить правильные элементы, мы должны не только вернуть `NonNullable<T>`, но и ввести тип, который проверяет наличие истинных значений:

```
type Truthy<T> = T extends "" | false | 0 | 0n ? never : T;

interface Array<T> {
  filter(predicate: BooleanConstructor): Truthy<NonNullable<T>>[];
}

interface ReadonlyArray<T> {
  filter(predicate: BooleanConstructor): Truthy<NonNullable<T>>[];
}

// const создает кортеж только для чтения
const array = [0, 1, 2, 3, ``, -0, 0n, false, undefined, null] as const;

// const отфильтрован: (1 | 2 | 3)[]
const filtered = array.filter(Boolean);

const nullOrOne: Array<0 | 1> = [0, 1, 0, 1];

// const onlyOnes: 1[]
const onlyOnes = nullOrOne.filter(Boolean);
```



В примере используется значение `0n`, которое равно `0` в типе `BigInt`. Этот тип доступен начиная с ECMAScript 2020.

Таким образом мы получаем правильное представление о том, какие типы следует ожидать. Но `ReadonlyArray<T>` принимает типы элементов кортежа, а не сам тип кортежа, поэтому мы теряем информацию о порядке следования типов внутри кортежа.

Как и в случае со всеми расширениями для существующих типов TypeScript, имейте в виду, что они могут вызвать побочные эффекты. Используйте их локально и будьте осторожны.

9.6. Расширение модулей

Задача

Вы работаете с библиотеками, которые предоставляют собственное представление HTML-элементов, например Preact или React. Но иногда их определения типов не содержат новейших функций. Вы хотите внести в них исправления.

Решение

Используйте объединение деклараций на уровне модулей и интерфейсов.

Обсуждение

JSX — это синтаксическое расширение JavaScript, представляющее XML-подобный способ описания и вложенности компонентов. В принципе, все, что может быть описано в виде дерева элементов, можно выразить в JSX. Создатели популярного фреймворка React ввели расширение JSX, чтобы сделать возможными написание и вложение компонентов в HTML-подобном виде в JavaScript, где он фактически транпилируется в серию вызовов функций:

```
<button onClick={() => alert('YES')}>Click me</button>
```

// Транспируется в:

```
React.createElement("button", { onClick: () => alert('YES') }, 'Click me');
```

С тех пор JSX был принят многими фреймворками, даже если они практически не связаны с React. Подробнее о JSX рассказывается в главе 10.

Типизация React для TypeScript содержит множество интерфейсов для всех возможных HTML-элементов. Но иногда ваши браузеры, фреймворки или ваш код немного опережают возможные варианты.

Допустим, вы хотите использовать новейшие графические функции в Chrome и загружать их в ленивом режиме. Это прогрессивное усовершенствование, поэтому только браузеры, которые понимают, что происходит, знают, как это интерпретировать. Другие браузеры в целях обеспечения достаточного уровня надежности не обращают на это внимания:

```

```

Но что происходит с JSX-кодом TypeScript? Возникают ошибки:

```
function Image({ src, alt }) {  
  // Свойство 'loading' не существует.  
  return <img src={src} alt={alt} loading="lazy" />;  
}
```

Чтобы предотвратить это, мы можем расширить доступные интерфейсы с помощью собственных свойств. Эта функция в TypeScript называется *объединением деклараций*.

Создадим папку `@types` и поместим в нее файл `jsx.d.ts`. Изменим конфигурацию TypeScript так, чтобы параметры компилятора позволяли использовать дополнительные типы:

```
{
  "compilerOptions": {
    ...
    /* Type declaration files to be included in compilation. */
    "types": ["@types/**"],
  },
  ...
}
```

Мы воссоздаем точную структуру модулей и интерфейсов:

- модуль называется "react";
- интерфейс `ImgHTMLAttributes<T>` расширяет `HTMLAttributes<T>`.

Мы знаем это из оригинальных типизаций. Здесь мы добавляем нужные нам свойства:

```
import "react";

declare module "react" {
  interface ImgHTMLAttributes<T> extends HTMLAttributes<T> {
    loading?: "lazy" | "eager" | "auto";
  }
}
```

И раз уж мы заговорили об этом, не будем забывать о замещающих текстах:

```
import "react";

declare module "react" {
  interface ImgHTMLAttributes<T> extends HTMLAttributes<T> {
    loading?: "lazy" | "eager" | "auto";
    alt: string;
  }
}
```

Такой вариант намного лучше! TypeScript возьмет исходное определение и объединит ваши объявления. Программа автозаполнения может предоставить вам все доступные варианты и выдаст ошибку, если вы забудете замещающий текст.

При работе с Preact (<https://preactjs.com/>) все немного сложнее. Исходный HTML-код очень широк и не так специфичен, как типы React. Вот почему при определении изображений нам приходится давать немного более явные декларации:

```
declare namespace JSX {
  interface IntrinsicElements {
```

```
img: HTMLAttributes & {
  alt: string;
  src: string;
  loading?: "lazy" | "eager" | "auto";
};
}
```

Этот код позволяет убедиться, что доступны как `alt`, так и `src`, и добавляет новый атрибут, называемый `loading`. Однако прием все тот же: объединение деклараций, которое работает на уровне пространств имен, интерфейсов и модулей.

9.7. Расширение глобальных переменных

Задача

Вы используете такую функцию браузера, как `ResizeObserver`, и видите, что она недоступна в вашей текущей конфигурации TypeScript.

Решение

Расширьте глобальное пространство имен, используя пользовательские определения типов.

Обсуждение

TypeScript хранит типы для всех API DOM в `lib.dom.d.ts`. Этот файл автоматически сгенерирован из файлов Web IDL. *Web IDL* расшифровывается как *Web Interface Definition Language* (язык описания веб-интерфейсов) и представляет собой формат, с помощью которого W3C и WHATWG определяют интерфейсы к веб-API. Он появился примерно в 2012 году и стал стандартом с 2016 года.

Когда вы читаете стандарты в W3C (<https://www.w3.org/>), например в `Resize Observer` (<https://oreil.ly/XeSUG>), то можете увидеть части определения или полное определение где-нибудь в спецификации. Например, вот это:

```
enum ResizeObserverBoxOptions {
  "border-box", "content-box", "device-pixel-content-box"
};

dictionary ResizeObserverOptions {
  ResizeObserverBoxOptions box = "content-box";
};
```

```

[Exposed=Window]
interface ResizeObserver {
  constructor(ResizeObserverCallback callback);
  void observe(Element target, optional ResizeObserverOptions options);
  void unobserve(Element target);
  void disconnect();
};

callback ResizeObserverCallback = void (
  sequence<ResizeObserverEntry> entries,
  ResizeObserver observer
);

[Exposed=Window]
interface ResizeObserverEntry {
  readonly attribute Element target;
  readonly attribute DOMRectReadOnly contentRect;
  readonly attribute FrozenArray<ResizeObserverSize> borderBoxSize;
  readonly attribute FrozenArray<ResizeObserverSize> contentBoxSize;
  readonly attribute FrozenArray<ResizeObserverSize> devicePixelContentBoxSize;
};

interface ResizeObserverSize {
  readonly attribute unrestricted double inlineSize;
  readonly attribute unrestricted double blockSize;
};

interface ResizeObservation {
  constructor(Element target);
  readonly attribute Element target;
  readonly attribute ResizeObserverBoxOptions observedBox;
  readonly attribute FrozenArray<ResizeObserverSize> lastReportedSizes;
};

```

Браузеры используют это определение в качестве руководства для реализации соответствующих API. TypeScript задействует эти IDL-файлы для генерации файла `lib.dom.d.ts`. Проект генератора библиотек TypeScript и JavaScript (<https://oreil.ly/WLcLB>) использует веб-стандарты и извлекает информацию IDL. Затем генератор *IDL в TypeScript* разбирает IDL-файл и генерирует правильные типы.

Страницы, которые необходимо проанализировать, обрабатываются вручную. Как только спецификация становится достаточно полной и поддерживается всеми основными браузерами, пользователи добавляют новый ресурс и видят, что их изменения опубликованы в следующей версии TypeScript. Так что это всего лишь вопрос времени, когда мы получим `ResizeObserver` в файле `lib.dom.d.ts`.

Если же мы не хотим ждать, то можем добавить типы самостоятельно, но только в тот проект, с которым сейчас работаем.

Предположим, что мы сгенерировали типы для `ResizeObserver`. Полученный результат мы сохраним в файле `resize-observer.d.ts`. Вот его содержимое:

```
type ResizeObserverBoxOptions =
  "border-box" |
  "content-box" |
  "device-pixel-content-box";

interface ResizeObserverOptions {
  box?: ResizeObserverBoxOptions;
}

interface ResizeObservation {
  readonly lastReportedSizes: ReadonlyArray<ResizeObserverSize>;
  readonly observedBox: ResizeObserverBoxOptions;
  readonly target: Element;
}

declare var ResizeObservation: {
  prototype: ResizeObservation;
  new(target: Element): ResizeObservation;
};

interface ResizeObserver {
  disconnect(): void;
  observe(target: Element, options?: ResizeObserverOptions): void;
  unobserve(target: Element): void;
}

export declare var ResizeObserver: {
  prototype: ResizeObserver;
  new(callback: ResizeObserverCallback): ResizeObserver;
};

interface ResizeObserverEntry {
  readonly borderBoxSize: ReadonlyArray<ResizeObserverSize>;
  readonly contentBoxSize: ReadonlyArray<ResizeObserverSize>;
  readonly contentRect: DOMRectReadOnly;
  readonly devicePixelContentBoxSize: ReadonlyArray<ResizeObserverSize>;
  readonly target: Element;
}

declare var ResizeObserverEntry: {
  prototype: ResizeObserverEntry;
  new(): ResizeObserverEntry;
};

interface ResizeObserverSize {
  readonly blockSize: number;
  readonly inlineSize: number;
}
```

```
declare var ResizeObserverSize: {
  prototype: ResizeObserverSize;
  new(): ResizeObserverSize;
};

interface ResizeObserverCallback {
  (entries: ResizeObserverEntry[], observer: ResizeObserver): void;
}
```

Мы объявляем множество интерфейсов и некоторые переменные, которые реализуют наши интерфейсы, например `declare var ResizeObserver`, который является объектом, определяющим прототип и функцию конструктора:

```
declare var ResizeObserver: {
  prototype: ResizeObserver;
  new(callback: ResizeObserverCallback): ResizeObserver;
};
```

Этот код уже очень помогает. Мы можем использовать (возможно) длинные объявления типов и помещать их непосредственно в тот файл, в котором они нам нужны. `ResizeObserver` найден! Однако мы хотим, чтобы он был доступен глобально.

Благодаря функции объединения деклараций в TypeScript мы можем расширять *пространства имен и интерфейсы* по мере необходимости. На этот раз мы расширяем *глобальное пространство имен*.

Глобальное пространство имен содержит все объекты и интерфейсы, которые доступны глобально. Это может быть объект `window` (и интерфейс `Window`), а также все остальное, что должно быть частью нашего контекста выполнения JavaScript. Мы расширяем глобальное пространство имен и добавляем в него объект `ResizeObserver`:

```
declare global { // открытие пространства имен
  var ResizeObserver: { // объединение с ним ResizeObserver
    prototype: ResizeObserver;
    new(callback: ResizeObserverCallback): ResizeObserver;
  }
}
```

Поместим файл `resize-observer.d.ts` в папку `@types`. Не забудьте добавить ее к исходным текстам, которые будет анализировать TypeScript, а также в список папок, содержащих объявления типов в файле `tsconfig.json`:

```
{
  "compilerOptions": {
    //...
    "typeRoots": ["@types", "./node_modules/@types"],
    //...
  },
  "include": ["src", "@types"]
}
```

Есть большая вероятность того, что `ResizeObserver` еще недоступен в вашем целевом браузере, поэтому убедитесь, что вы сделали объект `ResizeObserver` неопределенным (`undefined`). Это побудит вас проверить, доступен ли объект:

```
declare global {
  var ResizeObserver: {
    prototype: ResizeObserver;
    new(callback: ResizeObserverCallback): ResizeObserver;
  } | undefined
}
```

В вашем приложении:

```
if (typeof ResizeObserver !== 'undefined') {
  const x = new ResizeObserver((entries) => {});
}
```

Этот код делает работу с `ResizeObserver` максимально безопасной!

Возможно, TypeScript не распознает ваши файлы внешних объявлений и глобальное дополнение. Если это произошло, убедитесь, что:

- вы анализируете папку `@types` с помощью свойства `include` в файле `tsconfig.json`;
- ваши файлы объявления окружающих типов распознаются как таковые путем добавления их в `types` или `typeRoots` в параметрах компилятора `tsconfig.json`;
- вы добавляете `export {}` в конце файла внешней декларации (ambient declaration), чтобы TypeScript распознал этот файл как модуль.

9.8. Добавление модулей, которые отличаются от JS, в граф модулей

Задача

Вы используете сборщик, такой как `Webpack`, для загрузки файлов типа `.css` или изображений из JavaScript, но TypeScript не распознает эти файлы.

Решение

Глобально объявляйте модули на основе расширений имен файлов.

Обсуждение

В веб-разработке наблюдается тенденция к тому, чтобы сделать JavaScript точкой входа по умолчанию для всего приложения и позволить ему обрабатывать все необходимые ресурсы с помощью операторов `import`. Для этого вам нужен сборщик,

который анализирует ваш код и создает нужные артефакты. Популярным инструментом, выполняющим такие действия, является Webpack (<https://webpack.js.org/>), сборщик JavaScript, который позволяет объединять *все*: CSS, Markdown, SVG, JPEG — да все что угодно:

```
// это
import "./Button.css";

// или это
import styles from "./Button.css";
```

Webpack использует концепцию *загрузчиков* (loaders), просматривая окончания файлов и активируя определенные способы компоновки. Импорт `.css`-файлов в JavaScript не является нативным. Это часть Webpack (или любого другого используемого вами сборщика). Однако мы можем научить TypeScript понимать такие файлы.



Комитет по стандартам ECMAScript внес предложение разрешить импорт файлов, отличных от JavaScript, и использовать для этого определенные встроенные форматы. В итоге это отразится и на TypeScript. Вы можете прочитать об этом здесь: <https://oreil.ly/stAm5>.

TypeScript поддерживает *объявления внешних модулей* даже для модуля, который не находится там «физически», но пребывает в среде или доступен с помощью инструментов. В качестве примеров можно привести основные встроенные модули Node, такие как `url`, `http` или `path`, как описано в документации TypeScript:

```
declare module "path" {
  export function normalize(p: string): string;
  export function join(...paths: any[]): string;
  export var sep: string;
}
```

Данный код отлично подходит для модулей, название которых мы знаем точно. Этот же прием мы можем использовать и для паттернов с подстановочными знаками. Объявим обобщенный модуль среды для всех наших `.css`-файлов:

```
declare module '*.css' {
  // должно быть сделано
}
```

Паттерн готов. Он прослушивает все файлы `.css`, которые мы хотим импортировать. Мы ожидаем получить список имен классов, которые можем добавить в наши компоненты. Мы не знаем, какие классы определены в `.css`-файлах, поэтому воспользуемся объектом, который принимает каждый строковый ключ и возвращает строку:

```
declare module '*.css' {
  interface IClassNames {
```

```

    [className: string]: string
  }
  const classNames: IClassNames;
  export default classNames;
}

```

Это все, что нам нужно для повторной компиляции наших файлов. Единственным недостатком является невозможность использовать точные имена классов, чтобы получить автозаполнение и аналогичные преимущества. Решить эту проблему можно, автоматически генерируя файлы типов. В NPM (<https://oreil.ly/sDBv0>) есть пакеты, которые устраняют эту проблему. Не стесняйтесь выбрать тот, который вам больше нравится.

Будет немного проще, если мы решим импортировать что-то вроде MDX в наши модули. MDX позволяет нам писать Markdown (язык разметки), который преобразуется в обычные компоненты React (или JSX) (подробнее о React мы поговорим в главе 10).

Мы ожидаем, что функциональный компонент (которому мы можем передать свойства) вернет JSX-элемент:

```

declare module '*.mdx' {
  let MDXComponent: (props) => JSX.Element;
  export default MDXComponent;
}

```

И вуаля! Мы можем загружать файлы `.mdx` в JavaScript и использовать их в качестве компонентов:

```

import About from '../articles/about.mdx';

function App() {
  return <>
    <About/>
  </>
}

```

Если вы не знаете, чего ожидать, то данный код упростит вам жизнь. Все, что вам нужно сделать, — это объявить модуль. Не указывайте никакие типы. TypeScript разрешит загрузку, но не обеспечит никакую безопасность типов:

```

declare module '*.svg';

```

Чтобы сделать внешние модули доступными для вашего приложения, рекомендуется создать папку `@types` где-нибудь в проекте (возможно, на корневом уровне). Туда вы можете поместить любое количество файлов `.d.ts`, содержащих определения ваших модулей. Добавьте ссылку в ваш файл `tsconfig.json` — и TypeScript будет знать, что делать:

```
{
  ...
  "compilerOptions": {
    ...
    "typeRoots": [
      "./node_modules/@types",
      "./@types"
    ],
    ...
  }
}
```

Одна из главных особенностей TypeScript — возможность адаптации ко всем разновидностям JavaScript. Одни элементы являются встроенными, а другие требуют от вас внесения дополнительных исправлений.

TypeScript и React

React, пожалуй, одна из самых популярных библиотек JavaScript за последние годы. Ее простой подход к созданию компонентов изменил способ написания интерфейсных (и в некоторой степени серверных) приложений, позволяя декларативно писать код пользовательского интерфейса с помощью расширения синтаксиса JavaScript под названием JSX. Этот простой принцип легко усвоить и понять, к тому же он повлиял на десятки других библиотек.

JSX, несомненно, изменил правила игры в мире JavaScript, и поскольку цель TypeScript — удовлетворить потребности всех разработчиков JavaScript, то JSX нашел свое применение и в TypeScript. На самом деле TypeScript — полноценный компилятор JSX. Если вам не требуются дополнительные пакеты или инструменты, то TypeScript — это все, что вам нужно для запуска вашего приложения React. Кроме того, TypeScript пользуется огромной популярностью. На момент написания данной книги количество скачиваний React на NPM составляло 20 миллионов в неделю. Благодаря великолепному инструментарию VS Code и эффективным типам TypeScript стал выбором № 1 среди разработчиков React по всему миру.

Популярность TypeScript среди разработчиков React не снижается, однако есть обстоятельство, которое несколько затрудняет использование TypeScript в React: TypeScript не является наиболее предпочтительным вариантом для команды React. Другие библиотеки, основанные на JSX, в настоящее время в основном написаны на TypeScript и поэтому предоставляют отлично работающие типы «из коробки». А вот команда React работает с собственным средством статической проверки типов Flow (<https://flow.org/>), которое похоже на TypeScript, но в конечном счете несовместимо с ним. Это означает, что декларации типов React, которыми пользуются миллионы разработчиков, впоследствии создаются группой участников сообщества и публикуются на сайте Definitely Typed. Хотя `@types/react` считаются превосходными, они по-прежнему являются лишь лучшей попыткой типизировать такую сложную библиотеку, как React. Это неизбежно приводит к появлению пробелов.

Данная глава будет вам полезна там, где эти пробелы становятся заметными.

Мы рассмотрим ситуации, когда React должен быть простым, но TypeScript создает трудности, выдавая сложные сообщения об ошибках. Мы собираемся выяснить,

что означают эти сообщения, как их можно обойти и какие решения помогут в долгосрочной перспективе. Кроме того, мы поговорим о различных паттернах разработки и их преимуществах, а также о том, как использовать встроенную в TypeScript поддержку JSX.

Чего вы не найдете, так это базового руководства по настройке React и TypeScript. Экосистема настолько обширна и богата, что вы обязательно получите нужный результат. Откройте документацию вашего фреймворка и найдите информацию, касающуюся TypeScript. Кроме того, я предполагаю, что у вас есть некий опыт работы с React. В этой главе мы в основном будем говорить о типизации React.

Здесь мы сделаем акцент на React, но вы сможете использовать некоторые полученные знания и в других фреймворках и библиотеках на основе JSX.

10.1. Написание прокси-компонентов

Задача

Вы пишете много стандартных HTML-компонентов, но не хотите постоянно устанавливать все необходимые свойства.

Решение

Создайте прокси-компоненты и примените несколько паттернов, чтобы их можно было использовать в вашем сценарии.

Обсуждение

В большинстве веб-приложений используются кнопки. У них есть свойство `type`, которое по умолчанию имеет значение `submit`. Это разумное значение по умолчанию для форм, в которых вы выполняете действие по протоколу HTTP, отправляя содержимое в серверный API. Но если вы хотите, чтобы на вашем сайте были просто интерактивные элементы, то правильным типом для кнопок будет `button`. Этот выбор важен с точки зрения не только эстетики, но и доступности:

```
<button type="button">Click me!</button>
```

Когда вы пишете код React, есть вероятность, что вы редко отправляете форму на сервер с типом `submit`, но взаимодействуете с большим количеством кнопок типа `button`. Хороший способ справиться с подобными ситуациями — написать прокси-компоненты. Они имитируют HTML-элементы, но обладают рядом свойств:

```
function Button(props) {  
  return <button type="button" {...props} />;  
}
```

Идея состоит в том, что `Button` имеет те же свойства, что и HTML `button`, а атрибуты распространяются на HTML-элемент. Это, в свою очередь, является функцией, которая позволяет вам быть уверенными в том, что вы можете задать все свойства HTML, имеющиеся у элемента, не зная заранее, какие именно вы хотите задать. Но как их типизировать?

Все HTML-элементы, которые могут использоваться в JSX, определяются через внутренние элементы в пространстве имен `JSX`. Когда вы загружаете React, это пространство отображается как глобальное пространство имен в вашем файле, и вы можете получить доступ ко всем элементам, используя индексный доступ. Таким образом, правильные типы свойств для `Button` определены в `JSX.IntrinsicElements`.



Альтернативой `JSX.IntrinsicElements` служит `React.ElementType`, обобщенный тип в пакете `React`, который содержит компоненты классов и функций. Для прокси-компонентов достаточно `JSX.IntrinsicElements`, и он дает дополнительное преимущество: ваши компоненты остаются совместимыми с другими React-подобными фреймворками, такими как `Preact`.

`JSX.IntrinsicElements` — это тип в глобальном пространстве имен `JSX`. Как только это пространство окажется в области видимости, TypeScript сможет подобрать базовые элементы, совместимые с вашим фреймворком на основе `JSX`:

```
type ButtonProps = JSX.IntrinsicElements["button"];

function Button(props: ButtonProps) {
  return <button type="button" {...props} />;
}
```

Это относится и к дочерним элементам: мы распределяем их по порядку! Как видите, мы задали для кнопки тип `button`. Поскольку `props` — это просто объекты JavaScript, можно переопределить `type`, задав его в качестве атрибута в `props`. Если определены два ключа с одинаковыми именами, то победит последний. Это поведение может быть желательным, но в качестве альтернативы вы можете захотеть запретить себе и вашим коллегам переопределение `type`. С помощью вспомогательного типа `Omit<T, K>` вы можете использовать все свойства `button` JSX, но удалить все ключи, которые вы не хотите переопределять:

```
type ButtonProps = Omit<JSX.IntrinsicElements["button"], "type">;

function Button(props: ButtonProps) {
  return <button type="button" {...props} />;
}

const aButton = <Button type="button">Hi</Button>;
//
// Type '{ children: string; type: string; }' is not
// assignable to type 'IntrinsicAttributes & ButtonProps'.
// Property 'type' does not exist on type
// 'IntrinsicAttributes & ButtonProps'.(2322)
```

Если вам нужно выполнить `submit` для свойства `type`, то вы можете создать другой прокси-компонент:

```
type SubmitButtonProps = Omit<JSX.IntrinsicElements["button"], "type">;

function SubmitButton(props: SubmitButtonProps) {
  return <button type="submit" {...props} />;
}
```

Вы можете расширить эту возможность пропуска свойств, если хотите задать еще больше свойств. Возможно, вы придерживаетесь какой-то системы проектирования и не хотите, чтобы имена классов задавались произвольно:

```
type StyledButton = Omit<
  JSX.IntrinsicElements["button"],
  "type" | "className" | "style"
> & {
  type: "primary" | "secondary";
};

function StyledButton({ type, ...allProps }: StyledButton) {
  return <Button type="button" className={`btn-${type}`} {...allProps}/>;
}
```

Этот код даже позволяет повторно использовать имя свойства `type`.

Мы удалили некоторые свойства из определения типа и установили для них разумные значения по умолчанию. Теперь мы хотим убедиться, что наши пользователи не забудут установить некоторые свойства, такие как атрибут `alt` для изображения или атрибут `src`.

Для этого мы создаем вспомогательный тип `MakeRequired`, который снимает флаг необходимости:

```
type MakeRequired<T, K extends keyof T> = Omit<T, K> & Required<Pick<T, K>>;
```

И создаем собственные свойства:

```
type ImgProps
  = MakeRequired<
    JSX.IntrinsicElements["img"],
    "alt" | "src"
  >;

export function Img(props: ImgProps) {
  return <img {...props} />;
}

const anImage = <Img />;
//           ^
// Type '{}' is missing the following properties from type
// 'Required<Pick<DetailedHTMLProps<ImgHTMLAttributes<HTMLImageElement>,
// HTMLImageElement>, "alt" | "src">>': alt, src (2739)
```

Внеся всего лишь несколько изменений в тип исходного встроенного элемента и прокси-компонент, мы можем гарантировать, что наш код станет более надежным и доступным и менее подверженным ошибкам.

10.2. Написание контролируемых компонентов

Задача

Элементы формы, такие как поля ввода, создают еще одну сложность, поскольку нам нужно решить, где контролировать состояние: в браузере или в React.

Решение

Напишите прокси-компонент, который использует размеченные типы объединения и прием «Необязательный `never`», чтобы гарантировать, что вы не переключитесь с неконтролируемого компонента на контролируемый во время выполнения.

Обсуждение

React разделяет элементы формы на *контролируемые* и *неконтролируемые компоненты*. Когда вы используете обычные элементы, такие как `input`, `textarea` или `select`, вам необходимо помнить, что лежащие в их основе HTML-элементы сами контролируют собственное состояние. А в React состояние элемента определяется *непосредственно библиотекой*.

Если вы устанавливаете атрибут `value`, то React предполагает, что значение элемента контролируется системой управления состоянием React. То есть вы не сможете изменить это значение, если не будете поддерживать состояние элемента с помощью `useState` и соответствующей сеттер-функции.

Есть два способа решить эту проблему. Во-первых, вы можете выбрать в качестве свойства `defaultValue`, а не `value`. В этом случае `value` для поля ввода будет установлено только при первом отображении, а в дальнейшем все будет зависеть от браузера:

```
function Input({
  value = "", ...allProps
}: Props) {
  return (
    <input
      defaultValue={value}
      {...allProps}
    />
  );
}
```


Или же вы управляете `value` внутри системы с помощью управления состоянием React. Обычно достаточно просто сопоставить свойства исходного элемента ввода с вашим собственным типом. Мы удаляем `value` из встроенных элементов и добавляем его в качестве обязательного атрибута типа `string`:

```
type ControlledProps =  
  Omit<JSX.IntrinsicElements["input"], "value"> & {  
    value: string;  
  };
```

Затем мы обернем элемент ввода в прокси-компонент. Не рекомендуется сохранять состояние внутри прокси-компонента; скорее, вы должны управлять им извне с помощью `useState`. Кроме того, мы пересылаем обработчик `onChange`, который передаем из исходных свойств элемента ввода:

```
function Input({  
  value = "", onChange, ...allProps  
}: ControlledProps) {  
  return (  
    <input  
      value={value}  
      {...allProps}  
      onChange={onChange}  
    />  
  );  
}
```

```
function AComponentUsingInput() {  
  const [val, setVal] = useState("");  
  return <Input  
    value={val}  
    onChange={(e) => {  
      setVal(e.target.value);  
    }}  
  />  
}
```

React выдает интересное предупреждение при переключении с неуправляемого компонента на управляемый во время выполнения:

Компонент изменяет неконтролируемый вход на контролируемый. Вероятно, это вызвано тем, что значение меняется с неопределенного на определенное, чего не должно происходить. Примите решение об использовании контролируемого или неконтролируемого элемента ввода в течение всего срока службы компонента.

Мы можем предотвратить это предупреждение, убедившись во время компиляции, что всегда предоставляем либо определенный строковый атрибут `value`, либо

значение по умолчанию (`defaultValue`), но не то и другое сразу. Эту проблему можно решить, используя размеченный тип объединения с помощью приема «Необязательный `never`» (как показано в рецепте 3.8), а также используя вспомогательный тип `OnlyRequired` из рецепта 8.1 для извлечения возможных свойств из `JSX.IntrinsicElements["input"]`:

```
import React, { useState } from "react";

// Вспомогательный тип, задающий несколько свойств как обязательные
type OnlyRequired<T, K extends keyof T = keyof T> = Required<Pick<T, K>> &
  Partial<Omit<T, K>>;

// Ветвь 1: Сделаем обязательными "value" и "onChange", удалим `defaultValue`
type ControlledProps = OnlyRequired<
  JSX.IntrinsicElements["input"],
  "value" | "onChange"
> & {
  defaultValue?: never;
};

// Ветвь 2: Удалим `value` и `onChange`, сделаем `defaultValue` обязательным
type UncontrolledProps = Omit<
  JSX.IntrinsicElements["input"],
  "value" | "onChange"
> & {
  defaultValue: string;
  value?: never;
  onChange?: never;
};

type InputProps = ControlledProps | UncontrolledProps;

function Input({ ...allProps }: InputProps) {
  return <input {...allProps} />;
}

function Controlled() {
  const [val, setVal] = useState("");
  return <Input value={val} onChange={(e) => setVal(e.target.value)} />;
}

function Uncontrolled() {
  return <Input defaultValue="Hello" />;
}
```

Во всех остальных случаях наличие необязательного значения или значения по умолчанию (`defaultValue`) и попытка управлять значениями будут запрещены системой типов.

10.3. Типизация пользовательских хуков

Задача

Вы хотите определить пользовательские перехватчики (хуки, hooks) и получить правильные типы.

Решение

Используйте кортежные типы или контекст `const`.

Обсуждение

Создадим пользовательский хук в React и будем придерживаться соглашения об именовании, как это делают обычные хуки React: возвращаем массив (или кортеж), который может быть деструктурирован. Например, `useState`:

```
const [state, setState] = useState(0);
```

Почему мы вообще используем массивы? Потому что поля массива не имеют имени, и вы можете задать им собственные имена:

```
const [count, setCount] = useState(0);  
const [darkMode, setDarkMode] = useState(true);
```

Поэтому естественно, если у вас есть похожий паттерн, вы хотите возвращать массив. Пользовательский хук переключения может выглядеть следующим образом:

```
export const useToggle = (initialValue: boolean) => {  
  const [value, setValue] = useState(initialValue);  
  const toggleValue = () => setValue(!value);  
  return [value, toggleValue];  
}
```

Ничего необычного. Единственные типы, которые нам нужно задать, — это типы входных параметров. Попробуем:

```
export const Body = () => {  
  const [isVisible, toggleVisible] = useToggle(false)  
  return (  
    <>  
      <button onClick={toggleVisible}></button>  
      { /* Error. See below */ }  
      {isVisible && <div>World</div>}>>  
    </>  
  )  
}
```

```
// Error: Type 'boolean | (() => void)' is not assignable to
// type 'MouseEventHandler<HTMLButtonElement> | undefined'.
// Type 'boolean' is not assignable to type
// 'MouseEventHandler<HTMLButtonElement>'.(2322)
```

Так почему же работа кода завершается ошибкой? Сообщение о ней может быть загадочным, но нам следует обратить внимание на первый тип, который объявлен несовместимым: `boolean | (() => void)`'. Это происходит из-за того, что возвращается массив: список любой длины, который может содержать столько элементов, сколько практически возможно. По возвращаемому значению в `useToggle` TypeScript определяет тип массива. Тип `value` — это `boolean` (отлично!), а тип `toggleValue` — это `() => void` (функция, которая, как ожидается, ничего не вернет), поэтому TypeScript сообщает нам, что в этом массиве возможны оба типа.

Именно этот факт нарушает совместимость с `onClick`, так как `onClick` ожидает функцию.

Это нормально, но `toggleValue` (или `toggleVisible`) является функцией. Однако, согласно TypeScript, она также может быть и логическим значением! TypeScript выдает предупреждение о необходимости явно указывать тип или, по крайней мере, выполнять проверку типов.

Но нам не нужно выполнять дополнительные проверки типов. Наш код очень понятен. Неправильными являются типы. Мы имеем дело не с массивом, поэтому воспользуемся другим названием: кортеж. Массив — это список значений, который может быть любой длины, и мы точно знаем, сколько значений получим в нем. Обычно мы знаем и тип каждого элемента кортежа.

Поэтому в `UseToggle` мы должны возвращать не массив, а кортеж. Есть проблема: в JavaScript массив и кортеж неразличимы. В системе типов TypeScript мы можем их различать.

Первый вариант: определимся с типом возвращаемого значения. TypeScript — правильно! — выводит массив, поэтому мы должны сообщить TypeScript, что ожидаем кортеж:

```
// добавляем здесь возвращаемый тип
export const useToggle = (initialValue: boolean): [boolean, () => void] => {
  const [value, setValue] = useState(initialValue);
  const toggleValue = () => setValue(!value);
  return [value, toggleValue];
};
```

Используя в качестве возвращаемого типа `[boolean, () => void]`, TypeScript проверяет, что в этой функции мы возвращаем кортеж. TypeScript не выводит, а скорее проверяет, соответствует ли предполагаемый тип возвращаемого значения фактическим значениям. И вуаля — ваш код больше не выдает ошибки.

Второй вариант: используем контекст `const`. Мы знаем, сколько элементов кортежа ожидаем и каков их тип. Это похоже на задание по замораживанию типа с помощью утверждения `const`:

```
export const useToggle = (initialValue: boolean) => {
  const [value, setValue] = useState(initialValue);
  const toggleValue = () => setValue(!value);
  // здесь мы преобразуем массив в кортеж
  return [value, toggleValue] as const;
}
```

Тип возвращаемого значения теперь `readonly [boolean, () => void]`, поскольку `const` гарантирует, что ваши значения постоянны и не могут быть изменены. Этот тип немного отличается семантически, однако на самом деле вы не сможете изменить возвращаемые значения вне `useToggle`. Поэтому было бы правильнее использовать `readonly`.

10.4. Типизация обобщенных компонентов forwardRef

Задача

Вы используете `forwardRef` для своих компонентов, но вам нужно, чтобы они были обобщенными.

Решение

Существует несколько решений этой проблемы.

Обсуждение

Если вы создаете библиотеки компонентов и системы проектирования в React, то, возможно, у вас уже есть ссылки `forwardRef` на элементы DOM внутри ваших компонентов.

Это особенно полезно, если вы оборачиваете базовые компоненты или листья в *прокси-компоненты* (см. рецепт 10.1), но хотите использовать свойство `ref` так, как привыкли:

```
const Button = React.forwardRef((props, ref) => (
  <button type="button" {...props} ref={ref}>
    {props.children}
  </button>
));
```

```
// Использование: Вы можете использовать свой прокси точно так же,
// как вы используете обычную кнопку!
const reference = React.createRef();
<Button className="primary" ref={reference}>Hello</Button>
```

Предоставление типов для `React.forwardRef` — обычно довольно простая процедура. Типы, поставляемые `@types/react`, имеют переменные обобщенного типа, которые вы можете задать при вызове `React.forwardRef`. В этом случае лучше явно аннотировать типы:

```
type ButtonProps = JSX.IntrinsicElements["button"];

const Button = React.forwardRef<HTMLButtonElement, ButtonProps>(
  (props, ref) => (
    <button type="button" {...props} ref={ref}>
      {props.children}
    </button>
  )
);

// Использование
const reference = React.createRef<HTMLButtonElement>();
<Button className="primary" ref={reference}>Hello</Button>
```

Пока все хорошо. Но все немного усложняется, если у вас есть компонент, который принимает обобщенные свойства. Следующий компонент создает список элементов списка, в котором вы можете выбрать каждую строку с помощью элемента `button`:

```
type ClickableListProps<T> = {
  items: T[];
  onSelect: (item: T) => void;
};

function ClickableList<T>(props: ClickableListProps<T>) {
  return (
    <ul>
      {props.items.map((item, idx) => (
        <li>
          <button key={idx} onClick={() => props.onSelect(item)}>
            Choose
          </button>
          {item}
        </li>
      ))}
    </ul>
  );
}

// Использование
const items = [1, 2, 3, 4];
<ClickableList items={items}>
```

```

onSelect={(item) => {
  // элемент имеет тип number
  console.log(item);
} } />

```

Вам нужна дополнительная безопасность типов, чтобы вы могли работать с типобезопасным `item` при обратном вызове `onSelect`. Допустим, вы хотите создать ссылку `ref` на внутренний элемент `ul`: как вы поступите? Изменим компонент `ClickableList` на компонент с внутренней функцией, которая принимает `ForwardRef` и использует его в качестве аргумента в функции `React.forwardRef`:

```

// Исходный компонент расширен с помощью `ref`
function ClickableListInner<T>({
  props: ClickableListProps<T>,
  ref: React.ForwardedRef<HTMLUListElement>
}) {
  return (
    <ul ref={ref}>
      {props.items.map((item, i) => (
        <li key={i}>
          <button onClick={(el) => props.onSelect(item)}>Select</button>
          {item}
        </li>
      ))}
    </ul>
  );
}

```

```

// В качестве аргумента в `React.forwardRef`
const ClickableList = React.forwardRef(ClickableListInner)

```

Код компилируется, но имеет один недостаток: мы не можем назначить переменную обобщенного типа для `ClickableListProps`. По умолчанию она становится `unknown`. Это хорошо по сравнению с `any`, но немного раздражает. Используя `ClickableList`, мы знаем, какие элементы нужно передать, и хотим, чтобы они были типизированы соответствующим образом! Как же этого добиться? Ответ на этот вопрос сложен... и у вас есть несколько вариантов.

Первый вариант — выполнить утверждение типа, которое восстанавливает исходную сигнатуру функции:

```

const ClickableList = React.forwardRef(ClickableListInner) as <T>({
  props: ClickableListProps<T> & { ref?: React.ForwardedRef<HTMLUListElement> }
}) => ReturnType<typeof ClickableListInner>;

```

Утверждения типов отлично работают, если у вас есть всего несколько ситуаций, когда вам нужны обобщенные компоненты `forwardRef`, но они могут быть слишком громоздкими, когда вы работаете с большим количеством таких компонентов. Кроме того, вы вводите небезопасный оператор для того, что должно быть поведением по умолчанию.

Второй вариант — создать пользовательские ссылки с помощью компонентов-оберток. Хотя `ref` — это зарезервированное слово для компонентов React, вы можете использовать собственные свойства, чтобы имитировать подобное поведение. Это работает так же хорошо:

```
type ClickableListProps<T> = {
  items: T[];
  onSelect: (item: T) => void;
  mRef?: React.Ref<HTMLUListElement> | null;
};

export function ClickableList<T>(
  props: ClickableListProps<T>
) {
  return (
    <ul ref={props.mRef}>
      {props.items.map((item, i) => (
        <li key={i}>
          <button onClick={(el) => props.onSelect(item)}>Select</button>
          {item}
        </li>
      ))}
    </ul>
  );
}
```

Однако вы вводите новый API. Для справки: существует также возможность использования компонента-обертки, который позволяет задействовать `forwardRef` во *внутреннем* компоненте, чтобы предоставить доступ к пользовательскому свойству `ref` снаружи:

```
function ClickableListInner<T>(
  props: ClickableListProps<T>,
  ref: React.ForwardedRef<HTMLUListElement>
) {
  return (
    <ul ref={ref}>
      {props.items.map((item, i) => (
        <li key={i}>
          <button onClick={(el) => props.onSelect(item)}>Select</button>
          {item}
        </li>
      ))}
    </ul>
  );
}

const ClickableListWithRef = forwardRef(ClickableListInner);

type ClickableListWithRefProps<T> = ClickableListProps<T> & {
  mRef?: React.Ref<HTMLUListElement>;
};
```



```
export function ClickableList<T>({
  mRef,
  ...props
}: ClickableListWithRefProps<T>) {
  return <ClickableListWithRef ref={mRef} {...props} />;
}
```

Оба варианта допустимы, если единственное, чего вы хотите добиться, — это передать ссылку. Если же вы хотите иметь согласованный API, то вам стоит поискать другой способ.

Третий и последний вариант — дополнить `forwardRef` вашими собственными определениями типов. В TypeScript есть функция, называемая *выводом типа функции высшего порядка* (<https://oreil.ly/rVs9>), которая позволяет передавать параметры свободного типа во внешнюю функцию.

Схема очень похожа на то, что мы хотели бы получить от `forwardRef`, но не работает с текущими типами. Причина в том, что выведение типов функций высшего порядка работает только с простыми типами функций. Объявления функций внутри `forwardRef` добавляют свойства для `defaultProps` и т. д. Это пережитки времен компонентов классов, которые вы, возможно, не захотите использовать.

Таким образом, и без применения дополнительных свойств должна быть возможность использовать выведение типов функций высшего порядка!

Мы используем TypeScript, поэтому можем самостоятельно объявлять и переопределять глобальные объявления модулей, пространств имен и интерфейсов. Объединение деклараций — мощный инструмент, и мы собираемся им воспользоваться:

```
// Переопределяем forwardRef
declare module "react" {
  function forwardRef<T, P = {}>(<
    render: (props: P, ref: React.Ref<T>) => React.ReactElement | null
  >): (props: P & React.RefAttributes<T>) => React.ReactElement | null;
}
```

// Просто пишите свои компоненты так, как привыкли!

```
type ClickableListProps<T> = {
  items: T[];
  onSelect: (item: T) => void;
};
function ClickableListInner<T>(<
  props: ClickableListProps<T>,
  ref: React.ForwardedRef<HTMLUListElement>
>) {
  return (
    <ul ref={ref}>
      {props.items.map((item, i) => (
```

```

    <li key={i}>
      <button onClick={(e) => props.onSelect(item)}>Select</button>
      {item}
    </li>
  )}
</ul>
);
}

export const ClickableList = React.forwardRef(ClickableListInner);

```

Самым приятным моментом в этом решении является то, что вы снова пишете обычный JavaScript и работаете исключительно на уровне типов. Кроме того, повторные объявления выполняются в рамках модуля: никакого вмешательства в вызовы `forwardRef` из других модулей!

10.5. Предоставление типов для контекстного API

Задача

Вы хотите использовать контекстный API для глобальных переменных в своем приложении, но не знаете, как лучше всего работать с определениями типов.

Решение

Либо определите свойства по умолчанию для контекста и дайте возможность вывести тип, либо создайте часть свойств вашего контекста и явно добавьте параметр обобщенного типа. Если вы не хотите предоставлять значения по умолчанию, но хотите быть уверены, что все свойства будут предоставлены, то создайте вспомогательную функцию.

Обсуждение

Контекстный API React позволяет обмениваться данными на глобальном уровне. Чтобы использовать его, вам понадобятся:

- *поставщики* — передают данные в поддереву;
- *потребители* — это компоненты, которые *потребляют* переданные данные в свойствах рендеринга.

Благодаря типизации React вы можете использовать контекст, не выполняя ничего другого бóльшую часть времени. Все делается с помощью выведения типов и обобщенных типов.

Сначала мы создаем контекст. Здесь мы хотим хранить глобальные настройки приложения, такие как тема и язык приложения, а также глобальное состояние. При создании контекста React мы хотим передать свойства по умолчанию:

```
import React from "react";

const AppContext = React.createContext({
  authenticated: true,
  lang: "en",
  theme: "dark",
});
```

Таким образом, все, что вам нужно сделать с точки зрения типов, уже сделано за вас. Вы имеете три свойства: `authenticated`, `lang` и `theme`; у них есть типы `boolean` и `string`. Типизация React использует эту информацию, чтобы предоставить вам правильные типы при их применении.

Далее компоненту, расположенному выше в дереве компонентов, необходимо предоставить контекст, например корневой компонент приложения. Этот поставщик передает значения, которые вы установили, всем нижестоящим потребителям:

```
function App() {
  return (
    <AppContext.Provider
      value={{
        authenticated: true,
        lang: "de",
        theme: "light",
      }}
    >
      <Header />
    </AppContext.Provider>
  );
}
```

Теперь каждый компонент внутри дерева может использовать этот контекст. Вы получаете ошибки типа, когда забываете свойство или используете неправильный тип:

```
function App() {
  // Свойство 'theme' отсутствует в типе '{ lang: string; }', но требуется
  // в типе '{ lang: string; theme: string; authenticated: boolean }'.(2741)
  return (
    <AppContext.Provider
      value={{
        lang: "de",
      }}
    >
      <Header />
    </AppContext.Provider>
  );
}
```

Теперь используем наше глобальное состояние. Потребление контекста может осуществляться с помощью свойств рендеринга. Вы можете деструктурировать свои свойства рендеринга настолько глубоко, насколько захотите, чтобы получить только те из них, с которыми хотите работать:

```
function Header() {
  return (
    <AppContext.Consumer>
      ({ { authenticated } }) => {
        if (authenticated) {
          return <h1>Logged in!</h1>;
        }
        return <h1>You need to sign in</h1>;
      }
    </AppContext.Consumer>
  );
}
```

Другой способ использовать контекст — применить соответствующий пользовательский хук `useContext`:

```
function Header() {
  const { authenticated } = useContext(AppContext);
  if (authenticated) {
    return <h1>Logged in!</h1>;
  }
  return <h1>You need to sign in</h1>;
}
```

Ранее мы определили наши свойства с помощью правильных типов, поэтому `authenticated` на данный момент имеет тип `boolean`. Опять же, нам не пришлось ничего делать, чтобы обеспечить дополнительную безопасность типов.

Весь предыдущий пример работает лучше всего, если у нас есть свойства и значения по умолчанию. Иногда у вас нет значений по умолчанию или вам нужно иметь больше вариантов при выборе свойств, которые вы хотите задать.

Вместо того чтобы выводить все из значений по умолчанию, мы явно аннотируем параметр обобщенного типа, но не с помощью полного типа, а используя `Partial`.

Мы создаем тип для свойств контекста:

```
type ContextProps = {
  authenticated: boolean;
  lang: string;
  theme: string;
};
```

И инициализируем новый контекст:

```
const AppContext = React.createContext<Partial<ContextProps>>({});
```

Изменение семантики свойств контекста по умолчанию имеет некоторые побочные эффекты и для ваших компонентов. Теперь вам не нужно указывать каждое значение; пустой объект контекста может сделать то же самое! Все ваши свойства необязательны:

```
function App() {
  return (
    <AppContext.Provider
      value={{
        authenticated: true,
      }}
    >
      <Header />
    </AppContext.Provider>
  );
}
```

Помимо прочего, это означает еще и то, что нужно проверять, определено ли каждое свойство. Это не изменит код, в котором вы используете логические значения, но для каждого другого свойства потребуется выполнить еще одну проверку на `undefined`:

```
function Header() {
  const { authenticated, lang } = useContext(AppContext);
  if (authenticated && lang) {
    return <>
      <h1>Logged in!</h1>
      <p>Your language setting is set to {lang}</p>
    </> ;
  }
  return <h1>You need to sign in (or don't you have a language setting?)</h1>;
}
```

Если вы не можете указать значения по умолчанию и хотите быть уверены, что все свойства будут предоставлены поставщиком контекста, то можете помочь себе, используя вспомогательную функцию. В данном случае мы хотим, чтобы в результате явного создания универсального экземпляра был предоставлен не только тип, но и правильные средства защиты типов, что позволит при потреблении контекста правильно установить все возможные неопределенные значения:

```
function createContext<Props extends {}>() { ❶
  const ctx = React.createContext<Props | undefined>(undefined); ❷
  function useInnerCtx() { ❸
    const c = useContext(ctx);
    if (c === undefined) ❹
      throw new Error("Context must be consumed within a Provider");
    return c; ❺
  }
  return [useInnerCtx, ctx.Provider as React.Provider<Props>] as const; ❻
}
```

Что происходит в `createContext`?

- ❶ Мы создаем функцию, у которой нет аргументов, кроме параметров обобщенного типа. Не имея связи с параметрами функции, мы не можем создать экземпляр `Props` с помощью вывода. Это значит: для того чтобы `createContext` предоставлял правильные типы, нам нужно явно создать его экземпляр.
- ❷ Мы создаем контекст, который допускает использование `Props` или `undefined`. Добавив `undefined` к типу, мы можем передать `undefined` в качестве значения. Никаких значений по умолчанию!
- ❸ Внутри `createContext` мы создаем пользовательский хук. Он оборачивает `useContext`, используя только что созданный контекст `ctx`.
- ❹ Затем мы выполняем защиту типа, проверяя, не содержит ли возвращаемый `Props` значение `undefined`. Помните, что при вызове `createContext` мы создаем экземпляр параметра обобщенного типа с `Props | undefined`. В этой строке `undefined` снова удаляется из типа объединения.
- ❺ Здесь `c` — это свойства.
- ❻ Мы утверждаем, что `ctx.Provider` не принимает значений `undefined`. Мы вызываем `as const`, чтобы вернуть `[useInnerContext, ctx.Provider]` в качестве типа кортежа.

Используйте `createContext` аналогично `React.createContext`:

```
const [useAppContext, AppContextProvider] = createContext<ContextProps>();
```

При использовании `AppContextProvider` нам необходимо указать все значения:

```
function App() {
  return (
    <AppContextProvider
      value={{ lang: "en", theme: "dark", authenticated: true }}
    >
      <Header />
    </AppContextProvider>
  );
}

function Header() {
  // использование Context не сильно меняется
  const { authenticated } = useAppContext();
  if (authenticated) {
    return <h1>Logged in!</h1>;
  }
  return <h1>You need to sign in</h1>;
}
```

В зависимости от вашего сценария использования вы можете получить точные типы, не неся избыточных накладных расходов.

10.6. Типизация компонентов высшего порядка

Задача

Вы пишете *компоненты высшего порядка*, чтобы заранее задать определенные свойства для других компонентов, но не знаете, как их типизировать.

Решение

Используйте тип `React.ComponentType<P>` из `@types/react`, чтобы определить компонент, который расширяет ваши предустановленные атрибуты.

Обсуждение

На React оказывает влияние функциональное программирование, которое мы видим в том, как разрабатываются компоненты (с помощью функций), собираются (с помощью композиции) и обновляются (однонаправленный поток данных без сохранения состояния). Методам и парадигмам функционального программирования не потребовалось много времени, чтобы найти свое применение в разработке React. Одним из таких методов являются компоненты высшего порядка, которые созданы под влиянием *функций высшего порядка*.

Функции высшего порядка принимают один или несколько параметров, возвращая новую функцию. Иногда эти параметры используются для предварительного заполнения некоторых других параметров, как мы видим, например, во всех рецептах главы 7, посвященных каррированию. Компоненты высшего порядка ведут себя аналогично: они берут один или несколько компонентов и возвращают себе другой компонент. Обычно их создают для предварительного заполнения определенных свойств, когда нужно быть уверенными, что эти свойства не будут изменены в дальнейшем.

Представьте компонент общего назначения `Card`, который принимает `title` и `content` в виде строк:

```
type CardProps = {
  title: string;
  content: string;
};

function Card({ title, content }: CardProps) {
  return (
    <>
      <h2>{title}</h2>
      <div>{content}</div>
    </>
  );
}
```

Вы используете эту карточку для отображения определенных событий, таких как предупреждения, информационные всплывающие окна и сообщения об ошибках. Самая простая информационная карточка имеет заголовок "Info":

```
<Card title="Info" content="Your task has been processed" />;
```

Можно было бы задать подмножество свойств Card, чтобы разрешить использовать только определенное подмножество строк для title, но вместе с тем вы хотите иметь возможность использовать Card как можно чаще. Поэтому вы создаете новый компонент, который присваивает title значение "Info" и позволяет устанавливать другие свойства:

```
const Info = withInjectedProps({ title: "Info" }, Card);

// Это должно работать
<Info content="Your task has been processed" />;

// Это должно вызвать ошибку
<Info content="Your task has been processed" title="Warning" />;
```

Другими словами, вы *внедряете* (inject) подмножество свойств, а оставшиеся устанавливаете у вновь созданного компонента. Функция withInjectedProps пишется легко:

```
function withInjectedProps(injected, Component) {
  return function (props) {
    const newProps = { ...injected, ...props };
    return <Component {...newProps} />;
  };
}
```

Она принимает свойства injected и Component в качестве параметров, возвращает новый компонент функции, который принимает в качестве параметров оставшиеся свойства, и создает экземпляр исходного компонента с объединенными свойствами.

Как же типизировать withInjectedProps? Посмотрим на результат и узнаем, что находится внутри:

```
function withInjectedProps<T extends {}, U extends T>( ❶
  injected: T,
  Component: React.ComponentType<U> ❷
) {
  return function (props: Omit<U, keyof T>) { ❸
    const newProps = { ...injected, ...props } as U; ❹
    return <Component {...newProps} />;
  };
}
```


В данном коде происходит следующее.

- ❶ Нам нужно определить два параметра обобщенного типа. `T` относится к свойствам, которые мы уже внедрили; он расширяется от `{}`, позволяя нам убедиться в том, что мы передаем только объекты. `U` — параметр обобщенного типа для всех свойств `Component`. *U расширяет T*, то есть является подмножеством `T`. Это значит, `U` имеет больше свойств, чем `T`, но должен содержать то, что уже определено в `T`.
- ❷ Мы определяем, что `Component` должен быть типа `React.ComponentType<U>`. Это определение включает в себя компоненты классов, а также компоненты функций и говорит о том, что `props` будут установлены в `U`. Благодаря взаимосвязи `T` и `U` и способа, с помощью которого мы определили параметры `withInjectedProps`, мы гарантируем, что все, что будет передано для `Component`, определяет подмножество свойств `Component` с `injected`. Если мы допустим опечатку, то быстро получим первое сообщение об ошибке!
- ❸ Компонент функции, который будет возвращен, использует оставшиеся свойства. Благодаря `Omit<U, keyof T>` мы гарантируем, что не разрешим повторно задавать предварительно заполненные атрибуты.
- ❹ Типы объединения `T` и `Omit<U, keyof T>` должны снова привести к `U`, но параметры обобщенного типа могут быть явно созданы с помощью чего-то другого, поэтому могут снова не соответствовать `Component`. Утверждение типа помогает гарантировать, что свойства действительно соответствуют нашим требованиям.

Вот и все! Благодаря этим новым типам мы получаем правильное автозаполнение и ошибки:

```
const Info = withInjectedProps({ title: "Info" }, Card);

<Info content="Your task has been processed" />;
<Info content="Your task has been processed" title="Warning" />;
//                                     ^
// Type '{ content: string; title: string; }' is not assignable
// to type 'IntrinsicAttributes & Omit<CardProps, "title">'.
// Property 'title' does not exist on type
// 'IntrinsicAttributes & Omit<CardProps, "title">'.(2322)
```

Функция `withInjectedProps` настолько гибкая, что мы можем создавать функции высшего порядка, которые, в свою очередь, создают компоненты высшего порядка для различных ситуаций, например `withTitle`, предназначенный для предварительного заполнения атрибутов `title` типа `string`:

```
function withTitle<U extends { title: string }>(
  title: string,
```

```

Component: React.ComponentType<U>
) {
  return withInjectedProps({ title }, Component);
}

```

Ваше мастерство в функциональном программировании не знает границ.

10.7. Типизация обратных вызовов в синтетической системе событий React

Задача

Вы хотите получить наилучшую типизацию для всех событий браузера в React и использовать систему типов, чтобы ограничить ваши обратные вызовы совместимыми элементами.

Решение

Используйте типы событий `@types/react` и специализируйтесь на компонентах с помощью параметров обобщенного типа.

Обсуждение

Веб-приложения оживают благодаря взаимодействию с пользователем. Каждое взаимодействие вызывает событие. События — ключевой момент, и типизация React в TypeScript имеет отличную поддержку событий, но она требует, чтобы вы не использовали собственные события из `lib.dom.d.ts`. Если вы это сделаете, то React выдаст ошибки:

```

type WithChildren<T = {}> = T & { children?: React.ReactNode };

type ButtonProps = {
  onClick: (event: MouseEvent) => void;
} & WithChildren;

function Button({ onClick, children }: ButtonProps) {
  return <button onClick={onClick}>{children}</button>;
//           ^
// Type '(event: MouseEvent) => void' is not assignable to
// type 'MouseEventHandler<HTMLButtonElement>'.
// Types of parameters 'event' and 'event' are incompatible.
// Type 'MouseEvent<HTMLButtonElement, MouseEvent>' is missing the following
// properties from type 'MouseEvent': offsetX, offsetY, x, y,
// and 14 more.(2322)
}

```

React использует собственную систему событий, которые далее называются *синтетическими событиями*. Это кросс-браузерные обертки вокруг нативного события браузера, с тем же интерфейсом, что и у нативного аналога, но выровненные в целях совместимости. Изменение типа с `@types/react` делает ваши обратные вызовы снова совместимыми:

```
import React from "react";

type WithChildren<T = {}> = T & { children?: React.ReactNode };

type ButtonProps = {
  onClick: (event: React.MouseEvent) => void;
} & WithChildren;

function Button({ onClick, children }: ButtonProps) {
  return <button onClick={onClick}>{children}</button>;
}
```

Браузерные `MouseEvent` и `React.MouseEvent` достаточно сильно различаются для системы *структурных* типов TypeScript; это значит, в синтетических аналогах отсутствуют некоторые свойства. Из предыдущего сообщения об ошибке видно, что у оригинального `MouseEvent` на 18 свойств больше, чем у `React.MouseEvent`, некоторые из них могут быть важны, например координаты и смещения, которые пригодятся, если, допустим, вы хотите рисовать на холсте в браузере.

Если вы хотите получить доступ к свойствам исходного события, то можете использовать свойство `nativeEvent`:

```
function handleClick(event: React.MouseEvent) {
  console.log(event.nativeEvent.offsetX, event.nativeEvent.offsetY);
}

const btn = <Button onClick={handleClick}>Hello</Button>;
```

Поддерживаются следующие события: `AnimationEvent`, `ChangeEvent`, `ClipboardEvent`, `CompositionEvent`, `DragEvent`, `FocusEvent`, `FormEvent`, `KeyboardEvent`, `MouseEvent`, `PointerEvent`, `TouchEvent`, `TransitionEvent` и `WheelEvent`, а также `SyntheticEvent` для всех остальных событий.

До сих пор мы применяли правильные типы, чтобы убедиться, что у нас нет ошибок компилятора. Это достаточно просто. Но мы используем TypeScript не только для того, чтобы выполнить процедуры применения типов, не вызывающие нареканий у компилятора, но и для предотвращения ситуаций, которые могут вызвать проблемы.

Снова вспомним о кнопке. Или о ссылке (элемент `a`). Предполагается, что эти элементы нужно нажимать, таково их предназначение. Но в браузере события нажатия может получить любой элемент. Ничто не мешает вам добавить `onClick` к `div` — элементу, который имеет наименьшее семантическое значение из всех

элементов, и ни одна вспомогательная технология не скажет вам, что `div` может получать событие `MouseEvent`, если вы не добавите к нему множество атрибутов.

Разве не было бы хорошо, если бы мы могли запретить нашим коллегам (и нам самим) использовать определенные обработчики событий для *неправильных* элементов? `React.MouseEvent` — это обобщенный тип, который принимает совместимые элементы в качестве своего первого типа. Для этого параметра задано значение `Element`, которое является базовым типом для всех элементов в браузере. Но вы можете определить меньший набор совместимых элементов, указав подтип этого обобщенного параметра:

```
type WithChildren<T = {}> = T & { children?: React.ReactNode };

// Button сопоставляется с HTMLButtonElement
type ButtonProps = {
  onClick: (event: React.MouseEvent<HTMLButtonElement>) => void;
} & WithChildren;

function Button({ onClick, children }: ButtonProps) {
  return <button onClick={onClick}>{children}</button>;
}

// handleClick принимает события от HTMLButtonElement или HTMLAnchorElement
function handleClick(
  event: React.MouseEvent<HTMLButtonElement | HTMLAnchorElement>
) {
  console.log(event.currentTarget.tagName);
}

let button = <Button onClick={handleClick}>Works</Button>;
let link = <a href="/" onClick={handleClick}>Works</a>;

let broken = <div onClick={handleClick}>Does not work</div>;
//           ^
// Type '(event: MouseEvent<HTMLButtonElement | HTMLAnchorElement,
// MouseEvent>) => void' is not assignable to type
// 'MouseEventHandler<HTMLDivElement>'.
// Types of parameters 'event' and 'event' are incompatible.
// Type 'MouseEvent<HTMLDivElement, MouseEvent>' is not assignable to
// type 'MouseEvent<HTMLButtonElement | HTMLAnchorElement, MouseEvent>'.
// Type 'HTMLDivElement' is not assignable to type #
// 'HTMLButtonElement | HTMLAnchorElement'.
```

В одних областях типы React предоставляют вам больше возможностей, но в других им не хватает функций. Например, нативное для браузера событие `InputEvent` не поддерживается в `@types/react`. Система синтетических событий задумывалась как кросс-браузерное решение, а в некоторых совместимых с React браузерах по-прежнему отсутствует реализация `InputEvent`. Пока они не обновятся, вы можете безопасно использовать базовое событие `SyntheticEvent`:

```
function onInput(event: React.SyntheticEvent) {
  event.preventDefault();
  // сделайте что-нибудь
}

const inp = <input type="text" onInput={onInput} />;
```

Теперь вы обеспечиваете *хоть какую-то* безопасность типов.

10.8. Типизация полиморфных компонентов

Задача

Вы создаете прокси-компонент (см. рецепт 10.1), который должен вести себя как один из множества различных HTML-элементов. Трудно подобрать правильные варианты типа.

Решение

Утверждайте перенаправленные свойства как `any` или используйте JSX-фабрику `React.createElement` напрямую.

Обсуждение

В React распространен паттерн определения полиморфных (или `as`) компонентов, которые определяют поведение, но могут действовать как разные элементы. Представьте кнопку с призывом к действию (call to action, CTA), которая может служить ссылкой на сайт или являться обычной HTML-кнопкой. Если вы хотите стилизовать их одинаково, то они обязаны вести себя одинаково, но в зависимости от контекста должны иметь правильный HTML-элемент для верного действия.



Выбор правильного элемента — важный фактор доступности. Элементы `a` и кнопки представляют собой то, что пользователи могут нажимать, но семантика элемента `a` кардинально отличается от семантики кнопки `button`. Название элемента `a` — это сокращение от `anchor`, и он должен содержать ссылку (`href`) на место назначения. Кнопку можно нажать, но это действие обычно выполняется с помощью сценария JavaScript. Оба элемента могут выглядеть одинаково, но действовать по-разному. К тому же они и объявляются по-разному с помощью вспомогательных технологий, таких как программы чтения с экрана. Подумайте о своих пользователях и выберите правильный элемент для верной цели.

Идея состоит в том, что в вашем компоненте есть свойство `as`, которое выбирает тип элемента. В зависимости от типа элемента `as` вы можете перенаправлять свойства,

соответствующие типу элемента. Конечно, вы можете комбинировать этот паттерн со всем тем, что видели в рецепте 10.1:

```
<Cta as="a" href="https://typescript-cookbook.com">
  Hey hey
</Cta>

<Cta as="button" type="button" onClick={(e) => { /* do something */ }}>
  My my
</Cta>
```

При использовании TypeScript вы должны быть уверены, что получите возможность автозаполнения для правильных свойств и ошибки для неправильных. Если вы добавляете ссылку href на кнопку button, то TypeScript должен выдать правильные волнистые линии:

```
// Type '{ children: string; as: "button"; type: "button"; href: string; }'
// is not assignable to type 'IntrinsicAttributes & { as: "button"; } &
// ClassAttributes<HTMLButtonElement> &
// ButtonHTMLAttributes<HTMLButtonElement> & { ...; }'.
// Property 'href' does not exist on type ... (2322)
//                                     v
<Cta as="button" type="button" href="" ref={e1 => e1?.id}>
  My my
</Cta>
```

Попробуем типизировать Cta. Сначала разработаем компонент вообще без типов. В JavaScript все выглядит не слишком сложно:

```
function Cta({ as: Component, ...props }) {
  return <Component {...props} />;
}
```

Мы извлекаем свойство as и переименовываем его в Component. Это механизм деструктуризации из JavaScript, который синтаксически похож на аннотацию TypeScript, но работает с деструктурированными свойствами, а не с самим объектом (где вам понадобится аннотация типа). Мы переименуем его в компонент, используя верхний регистр, чтобы можно было создать его экземпляр через JSX. Остальные свойства будут собраны в ...props и распределены при создании компонента. Обратите внимание: вы можете распределять дочерние компоненты с помощью ...props, что является небольшим приятным побочным эффектом JSX.

Когда мы хотим типизировать Cta, то создаем тип CtaProps, который работает либо с элементами "a", либо с элементами "button", а остальные свойства берет из JSX.IntrinsicElements, аналогично тому, что мы видели в рецепте 10.1:

```
type CtaElements = "a" | "button";

type CtaProps<T extends CtaElements> = {
  as: T;
} & JSX.IntrinsicElements[T];
```

Подключая наши типы к `Cta`, мы видим, что сигнатура функции работает очень хорошо лишь с несколькими дополнительными аннотациями. Но при создании экземпляра компонента мы получаем довольно сложную ошибку, которая сообщает нам о том, что многое идет не так:

```
function Cta<T extends CtaElements>({
  as: Component,
  ...props
}): CtaProps<T> {
  return <Component {...props} />;
//           ^
// Type 'Omit<CtaProps<T>, "as" | "children"> & { children: ReactNode; }'
// is not assignable to type 'IntrinsicAttributes &
// LibraryManagedAttributes<T, ClassAttributes<HTMLAnchorElement> &
// AnchorHTMLAttributes<HTMLAnchorElement> & ClassAttributes<...> &
// ButtonHTMLAttributes<...>'
// Type 'Omit<CtaProps<T>, "as" | "children"> & { children: ReactNode; }' is not
// assignable to type
// 'LibraryManagedAttributes<T, ClassAttributes<HTMLAnchorElement>
// & AnchorHTMLAttributes<HTMLAnchorElement> & ClassAttributes<...>
// & ButtonHTMLAttributes<...>'.(2322)
}
```

Итак, откуда же взялось это сообщение? Чтобы TypeScript корректно работал с JSX, нам нужно использовать определения типов в глобальном пространстве имен `JSX`. Если оно находится в области видимости, то TypeScript знает, какие элементы, не являющиеся компонентами, могут быть созданы и какие атрибуты они могут принимать. Это `JSX.IntrinsicElements`, которые мы используем в этом примере и в рецепте 10.1.

Один из типов, который тоже необходимо определить, — это `LibraryManagedAttributes`. Он используется для предоставления атрибутов, которые определяются либо самим фреймворком (например, `key`), либо с помощью таких средств, как `defaultProps`:

```
export interface Props {
  name: string;
}

function Greet({ name }: Props) {
  return <div>Hello {name.toUpperCase()}!</div>;
}
// Переходит в LibraryManagedAttributes
Greet.defaultProps = { name: "world" };

// Проверки типов! Утверждения типов не нужны!
let el = <Greet key={1} />;
```

Типизация в React решает проблему `LibraryManagedAttributes` с помощью условного типа. И как вы увидите в рецепте 12.7, при вычислении условные типы

не будут расширены всеми возможными вариантами типов объединения. Это означает, что TypeScript не сможет проверить, подходят ли ваши типы для компонентов, поскольку не сможет оценить `LibraryManagedAttributes`.

Один из вариантов решения этой проблемы — утверждение свойств для `any`:

```
function Cta<T extends CtaElements>({
  as: Component,
  ...props
}: CtaProps<T>) {
  return <Component {...(props as any)} />;
}
```

Код работает, но является признаком *небезопасной* операции, которая не должна быть *небезопасной*. Другой способ — использовать в этом случае не `JSX`, а `JSX`-фабрику `React.createElement`.

Каждый вызов `JSX` — синтаксическое дополнение к вызову фабрики `JSX`:

```
<h1 className="headline">Hello World</h1>
```

```
// будет преобразован в
React.createElement("h1", { className: "headline" }, ["Hello World"]);
```

Если вы используете вложенные компоненты, то третий параметр `createElement` будет содержать вложенные вызовы фабричных функций. `JSX`-фабрику `React.createElement` вызывать гораздо проще, чем `JSX`, и TypeScript не будет обращаться к глобальному пространству имен `JSX` при создании новых элементов. Похоже, это идеальный обходной путь.

`JSX`-фабрике `React.createElement` нужны три аргумента: компонент, свойства `props` и дочерние компоненты `children`. На данный момент мы использовали `props` для всех дочерних компонентов, но для `React.createElement` нам нужно указать их явно. Это также означает, что нам нужно явно определить дочерние компоненты `children`.

Для этого мы создаем вспомогательный тип `WithChildren<T>`. Он использует существующий тип и добавляет необязательные дочерние элементы в виде `React.ReactNode`:

```
type WithChildren<T = {}> = T & { children?: React.ReactNode };
```

`WithChildren` очень гибок. Мы можем обернуть в него любой тип нашего свойства:

```
type CtaProps<T extends CtaElements> = WithChildren<{
  as: T;
}> & JSX.IntrinsicElements[T];
```


Или мы можем создать тип объединения:

```
type CtaProps<T extends CtaElements> = {  
  as: T;  
} & JSX.IntrinsicElements[T] & WithChildren;
```

Тип `T` по умолчанию имеет значение `{}`, поэтому становится универсальным для использования. Это значительно упрощает добавление дочерних элементов `children` всякий раз, когда они вам нужны. В качестве следующего шага мы де-структурируем дочерние элементы `children` из `props` и передаем все аргументы в `React.createElement`:

```
function Cta<T extends CtaElements>({  
  as: Component,  
  children,  
  ...props  
}: CtaProps<T>) {  
  return React.createElement(Component, props, children);  
}
```

И при этом наш полиморфный компонент принимает правильные параметры, не вызывая ошибок.

ГЛАВА 11

Классы

Когда в 2012 году был впервые выпущен TypeScript, экосистема языка JavaScript и его возможности были несопоставимы с тем, что мы имеем сегодня. TypeScript представил обширный функционал в виде не только системы типов, но и синтаксиса, обогатив существующий язык возможностями абстрагирования частей вашего кода от модулей, пространств имен и типов.

Одной из таких возможностей стали классы — основной элемент объектно-ориентированного программирования. Изначально классы TypeScript были созданы под большим влиянием C#, что неудивительно, если вы знаете разработчиков обоих языков программирования¹. Но они также разработаны на основе концепций из забытых предложений ECMAScript 4.

Со временем JavaScript получил большую часть языковых возможностей, впервые реализованных в TypeScript и других языках; классы, а также приватные поля, статические блоки и декораторы теперь являются частью стандарта ECMAScript и доступны для языковых сред выполнения в браузере и на сервере.

Таким образом, TypeScript находится в выгодном положении между инновациями, которые были привнесены в язык на ранних этапах его развития, и стандартами, которые команда TypeScript рассматривает в качестве основы для всех будущих функций системы типов. Хотя оригинальный дизайн языка TypeScript близок к тому, что в итоге получилось у JavaScript, но есть несколько различий, о которых стоит упомянуть.

В этой главе мы обсудим поведение классов в TypeScript и JavaScript, имеющиеся у нас возможности для самовыражения, а также различия между стандартным и оригинальным дизайном. Мы рассмотрим ключевые слова, типы и обобщенные типы, а также исследуем способы, позволяющие определять, что TypeScript добавляет в JavaScript, а что JavaScript привносит в него.

¹ C# и TypeScript созданы компанией Microsoft, и Андерс Хейлсберг принимал активное участие в разработке обоих языков программирования.

11.1. Выбор правильного модификатора видимости

Задача

В TypeScript есть два варианта видимости свойств и доступа к ним: в одном используется специальный синтаксис ключевых слов (`public`, `protected`, `private`), а в другом — обычный синтаксис JavaScript, когда свойства начинаются с символа решетки. Какой из них вам следует выбрать?

Решение

Отдавайте предпочтение собственному синтаксису JavaScript, поскольку он имеет некоторые последствия во время выполнения, которые лучше не упускать. Если вы используете сложную настройку, которая содержит различные модификаторы видимости, то остановитесь на модификаторах TypeScript. Они никуда не денутся.

Обсуждение

Классы TypeScript существуют довольно давно, и, хотя они испытывают огромное влияние классов ECMAScript, которые появились спустя несколько лет после них, команда TypeScript решила внедрить функции, которые в то время были полезны и популярны в традиционном объектно-ориентированном программировании на основе классов.

Одной из таких функций являются *модификаторы видимости свойств*, также называемые *модификаторами доступа*. Это специальные ключевые слова, которые вы можете поместить перед элементами — свойствами и методами, чтобы сообщить компилятору, как их можно увидеть и получить к ним доступ из других частей вашего программного обеспечения.



Все модификаторы видимости, как и приватные поля JavaScript, работают как с методами, так и со свойствами.

По умолчанию используется модификатор видимости `public`, который можно задать явно или просто опустить:

```
class Person {
  public name; // модификатор public необязателен
  constructor(name: string) {
    this.name = name;
  }
}

const myName = new Person("Stefan").name; // работает
```

Другим модификатором является `protected`, ограничивающий видимость классами и подклассами:

```
class Person {
    protected name;
    constructor(name: string) {
        this.name = name;
    }
    getName() {
        // доступ есть
        return this.name;
    }
}

const myName = new Person("Stefan").name;
//                                     ^
// Property 'name' is private and only accessible within
// class 'Person'.(2341)

class Teacher extends Person {
    constructor(name: string) {
        super(name);
    }

    getFullName() {
        // доступ есть
        return `Professor ${this.name}`;
    }
}
```

Модификатор доступа `protected` можно перезаписать в производных классах, чтобы сделать его `public`. Кроме того, он запрещает доступ к членам из ссылок на классы, которые не принадлежат к тому же подклассу. Так что этот код работает:

```
class Player extends Person {
    constructor(name: string) {
        super(name);
    }
    pair(p: Player) {
        // работает
        return `Pairing ${this.name} with ${p.name}`;
    }
}
```

Но пока он работает, использование базового класса или другого подкласса не принесет результата:

```
class Player extends Person {
    constructor(name: string) {
        super(name);
    }
}
```

```
pair(p: Person) {
    return `Pairing ${this.name} with ${p.name}`;
    //                                     ^
    // Property 'name' is protected and only accessible through an
    // instance of class 'Player'. This is an instance of
    // class 'Person'.(2446)
}
}
```

Последний модификатор видимости — `private`, который разрешает доступ только в пределах одного класса:

```
class Person {
    private name;
    constructor(name: string) {
        this.name = name;
    }
}

const myName = new Person("Stefan").name;
//                                     ^
// Property 'name' is protected and only accessible within
// class 'Person' and its subclasses.(2445)

class Teacher extends Person {
    constructor(name: string) {
        super(name);
    }

    getFullName() {
        return `Professor ${this.name}`;
        //                                     ^
        // Property 'name' is private and only accessible
        // within class 'Person'.(2341)
    }
}
```

Модификаторы видимости можно использовать и в конструкторах как быстрый способ определения свойств и их инициализации:

```
class Category {
    constructor(
        public title: string,
        public id: number,
        private reference: bigint
    ) {}
}

// транспируется в

class Category {
    constructor(title, id, reference) {
```

```

    this.title = title;
    this.id = id;
    this.reference = reference;
  }
}

```

Учитывая все описанные здесь возможности, следует отметить, что модификаторы видимости в TypeScript — это аннотации времени компиляции, которые удаляются после этапа компиляции. Часто удаляются целые объявления свойств, если они инициализируются не в описании класса, а в конструкторе, как мы видели в последнем примере.

Кроме того, модификаторы видимости действительны только при проверке во время компиляции. Это значит, в дальнейшем свойство `private` в TypeScript будет полностью доступно в JavaScript. Таким образом, вы можете обойти проверку доступа `private`, присвоив своим экземплярам статус `as any`, или получить к ним прямой доступ после компиляции вашего кода. Они также являются *перечислимыми*, а это означает, что их имена и значения становятся видимыми при сериализации с помощью `JSON.stringify` или `Object.getOwnPropertyNames`. Короче говоря, в момент выхода за границы системы типов они ведут себя как обычные члены класса JavaScript.



Помимо модификаторов видимости, к свойствам класса можно добавлять модификаторы только для чтения `readonly`.

Ограниченный доступ к свойствам — это возможность, которую целесообразно использовать не только в системе типов, поэтому ECMAScript принял аналогичную концепцию, называемую *приватными полями*, для обычных классов JavaScript.

Вместо модификатора видимости приватные поля фактически вводят новый синтаксис в виде знака решетки (*хеша*) перед именем члена.



Введение нового синтаксиса для приватных полей привело к бурным дебатам в сообществе по поводу удобства и эстетичности знака решетки. Некоторые участники даже назвали его отвратительным. Если это дополнение вас тоже раздражает, то, возможно, стоит подумать о знаке решетки как небольшом ограждении, которое вы ставите перед элементами, к которым не хотите давать общий доступ. Такой подход может сделать синтаксис со знаком решетки более приятным для глаз.

Знак решетки становится частью имени объекта, а это означает, что доступ к нему тоже должен осуществляться с помощью знака перед ним:

```

class Person {
  #name: string;
}

```

```
constructor(name: string) {
    this.#name = name;
}

// Мы можем использовать геттеры!
get name(): string {
    return this.#name.toUpperCase();
}
}

const me = new Person("Stefan");
console.log(me.#name);
//           ^
// Property '#name' is not accessible outside
// class 'Person' because it has a private identifier.(18013)

console.log(me.name); // работает
```

Приватные поля полностью соответствуют JavaScript; компилятор TypeScript ничего не удалит, и они сохраняют свою функциональность, скрывая информацию внутри класса даже после этапа компиляции. Результат компиляции, в которой использовалась последняя версия ECMAScript в качестве целевой, выглядит почти идентично версии TypeScript, только без аннотаций типов:

```
class Person {
    #name;

    constructor(name) {
        this.#name = name;
    }

    get name() {
        return this.#name.toUpperCase();
    }
}
```

К приватным полям нельзя получить доступ во время выполнения кода, и они не поддаются перечислению, то есть информация об их содержимом не будет раскрыта никаким образом.

Проблема в том, что в настоящее время в TypeScript существуют как приватные модификаторы видимости, так и приватные поля. Модификаторы видимости были всегда и отличаются большим разнообразием в сочетании с членами `protected`. С другой стороны, приватные поля настолько близки к JavaScript, насколько это возможно, и, учитывая цель TypeScript создать «синтаксис JavaScript для типов», в значительной степени достигают цели, когда речь заходит о долгосрочных планах развития языка. Итак, какой из них вам следует выбрать?

Независимо от того, какой модификатор вы выберете, они оба выполняют свою задачу: сообщают вам во время компиляции, когда есть доступ к свойству там, где

его не должно быть. Это первая обратная связь, информирующая вас о том, что что-то может быть не так, и это то, к чему следует стремиться при использовании TypeScript. Так что, если вам нужно скрыть информацию от посторонних глаз, каждый инструмент справится со своей задачей.

Но если вы посмотрите более широко, то все опять же зависит от ваших настроек. Если вы уже настроили проект с тщательно разработанными правилами видимости, то, возможно, не сможете сразу перенести их в нативную версию JavaScript. Кроме того, отсутствие видимости `protected` в JavaScript может повлечь проблемы с достижением ваших целей. Нет необходимости что-либо менять, если то, что у вас есть, уже работает.

Если у вас возникнут проблемы с видимостью во время выполнения, показывающей элементы, которые вы хотите скрыть: то есть если вы предполагаете, что другие люди будут использовать ваш код в качестве библиотеки и не должны иметь доступ ко всей внутренней информации, то приватные поля — идеальный вариант. Они хорошо поддерживаются в браузерах и других средах выполнения языка, а TypeScript поставляется с полифилами для более старых платформ.

11.2. Явное переопределение методов

Задача

В вашей иерархии классов вы расширяете базовые классы и переопределяете конкретные методы в подклассах. При рефакторинге базового класса вы можете в конечном счете использовать старые неиспользуемые методы, поскольку ничто не говорит вам о том, что базовый класс изменился.

Решение

Установите флаг `noImplicitOverride` и используйте ключевое слово `override`, чтобы дать сигнал о переопределении.

Обсуждение

Вы хотите рисовать фигуры на HTML-холсте. Ваше программное обеспечение может использовать набор точек с координатами x и y и, основываясь на определенной функции рендеринга, нарисовать многоугольники, прямоугольники или другие элементы.

Вы решили использовать иерархию классов, в которой базовый класс `Shape` принимает произвольный список элементов `Point` и проводит линии между ними. Этот класс обеспечивает обслуживание своей структуры с помощью сеттеров и геттеров, а также реализует саму функцию рендеринга `render`:


```
type Point = {
  x: number;
  y: number;
};

class Shape {
  points: Point[];
  fillStyle: string = "white";
  lineWidth: number = 10;

  constructor(points: Point[]) {
    this.points = points;
  }

  set fill(style: string) {
    this.fillStyle = style;
  }

  set width(width: number) {
    this.lineWidth = width;
  }

  render(ctx: CanvasRenderingContext2D) {
    if (this.points.length) {
      ctx.fillStyle = this.fillStyle;
      ctx.lineWidth = this.lineWidth;
      ctx.beginPath();
      let point = this.points[0];
      ctx.moveTo(point.x, point.y);
      for (let i = 1; i < this.points.length; i++) {
        point = this.points[i];
        ctx.lineTo(point.x, point.y);
      }
      ctx.closePath();
      ctx.stroke();
    }
  }
}
```

Чтобы использовать его, создайте 2D-контекст из элемента HTML-холста, добавьте новый экземпляр класса `Shape` и передайте контекст в функцию `render`:

```
const canvas = document.getElementsByTagName("canvas")[0];
const ctx = canvas?.getContext("2d");

const shape = new Shape([
  { x: 50, y: 140 },
  { x: 150, y: 60 },
  { x: 250, y: 140 },
]);
shape.fill = "red";
```

```
shape.width = 20;

shape.render(ctx);
```

Теперь мы хотим использовать созданный базовый класс и добавлять подклассы для конкретных фигур, например прямоугольников. Мы сохраняем методы обслуживания и, в частности, переопределяем `constructor`, а также метод рендеринга `render`:

```
class Rectangle extends Shape {
  constructor(points: Point[]) {
    if (points.length !== 2) {
      throw Error(`Wrong number of points, expected 2, got ${points.length}`);
    }
    super(points);
  }

  render(ctx: CanvasRenderingContext2D) {
    ctx.fillStyle = this.fillStyle;
    ctx.lineWidth = this.lineWidth;
    let a = this.points[0];
    let b = this.points[1];
    ctx.strokeRect(a.x, a.y, b.x - a.x, b.y - a.y);
  }
}
```

Использование `Rectangle` в значительной степени аналогично:

```
const rectangle = new Rectangle([
  {x: 130, y: 190},
  {x: 170, y: 250}
]);
rectangle.render(ctx);
```

По мере развития нашего программного обеспечения мы неизбежно меняем классы, методы и функции, и кто-то в нашей кодовой базе переименует метод `render` в `draw`:

```
class Shape {
  // см. ниже

  draw(ctx: CanvasRenderingContext2D) {
    if (this.points.length) {
      ctx.fillStyle = this.fillStyle;
      ctx.lineWidth = this.lineWidth;
      ctx.beginPath();
      let point = this.points[0];
      ctx.moveTo(point.x, point.y);
      for (let i = 1; i < this.points.length; i++) {
        point = this.points[i];
```

```
        ctx.lineTo(point.x, point.y);
    }
    ctx.closePath();
    ctx.stroke();
}
}
```

Само по себе это не проблема. Но возможен вариант, когда нигде в нашем коде мы не используем метод `render` из `Rectangle`, вероятно, потому, что опубликовали это программное обеспечение в виде библиотеки и не применяли ее в наших тестах. В таком случае ничто не говорит нам о том, что метод `render` в `Rectangle` все еще существует, без какой-либо связи с исходным классом.

Именно поэтому TypeScript позволяет вам аннотировать методы, которые вы хотите переопределить с помощью ключевого слова `override`. Это синтаксическое расширение TypeScript, которое будет удалено, как только TypeScript перенесет ваш код в JavaScript.

Когда метод помечен ключевым словом `override`, TypeScript проверит, что метод с таким же именем и сигнатурой существует в базовом классе. Если вы переименуете `render` в `draw`, то TypeScript сообщит вам, что метод `render` не был объявлен в базовом классе `Shape`:

```
class Rectangle extends Shape {
    // см. ниже

    override render(ctx: CanvasRenderingContext2D) {
//      ^
// This member cannot have an 'override' modifier because it
// is not declared in the base class 'Shape'.(4113)
        ctx.fillStyle = this.fillStyle;
        ctx.lineWidth = this.lineWidth;
        let a = this.points[0];
        let b = this.points[1];
        ctx.strokeRect(a.x, a.y, b.x - a.x, b.y - a.y);
    }
}
```

Это сообщение об ошибке — отличная гарантия того, что переименования и рефакторинги не нарушат ваши существующие контракты.



Конструктор можно рассматривать как переопределенный метод, однако его семантика отличается и обрабатывается с помощью других правил (например, необходимо убедиться, что вы вызываете `super` при создании экземпляра подкласса).

Установив флажок `noImplicitOverrides` в файле `tsconfig.json`, вы можете дополнительно убедиться в том, что вам нужно пометить функции ключевым словом `override`. В противном случае TypeScript выдаст еще одну ошибку:

```
class Rectangle extends Shape {
  // см. ниже

  draw(ctx: CanvasRenderingContext2D) {
    // ^
    // This member must have an 'override' modifier because it
    // overrides a member in the base class 'Shape'.(4114)
    ctx.fillStyle = this.fillStyle;
    ctx.lineWidth = this.lineWidth;
    let a = this.points[0];
    let b = this.points[1];
    ctx.strokeRect(a.x, a.y, b.x - a.x, b.y - a.y);
  }
}
```



Такие приемы, как реализация интерфейсов, которые определяют базовую форму класса, позволяют предотвратить возникновение подобных проблем. Итак, полезно использовать ключевое слово `override` и `noImplicitOverrides` в качестве дополнительных средств защиты при создании иерархий классов.

Если ваше программное обеспечение должно задействовать иерархии классов, то применять `override` в сочетании с `noImplicitAny` — хороший способ гарантировать, что вы ничего не забыли. Иерархии классов, как и любые другие, имеют тенденцию со временем усложняться, поэтому используйте любые возможные меры предосторожности.

11.3. Описание конструкторов и прототипов

Задача

Вы хотите динамически создавать экземпляры подклассов определенного абстрактного класса, но TypeScript не позволяет создавать экземпляры абстрактных классов.

Решение

Опишите свои классы с помощью паттерна *интерфейса конструктора*.

Обсуждение

Если вы используете иерархии классов в TypeScript, то структурные особенности TypeScript иногда мешают вам. Для примера рассмотрим следующую иерархию

классов, в которой мы хотим отфильтровать набор элементов на основе различных правил:

```
abstract class FilterItem {
  constructor(private property: string) {};
  someFunction() { /* ... */ };
  abstract filter(): void;
}

class AFilter extends FilterItem {
  filter() { /* ... */ }
}

class BFilter extends FilterItem {
  filter() { /* ... */ }
}
```

Абстрактный класс `FilterItem` должен быть реализован другими классами. В этом примере `AFilter` и `BFilter`, являющиеся конкретизациями `FilterItem`, служат основой для фильтров:

```
const some: FilterItem = new AFilter('afilter'); // ok
```

Ситуация становится интереснее, если мы не работаем с экземплярами сразу. Допустим, мы хотим создавать новые фильтры на основе токена, полученного в результате AJAX-вызова. Чтобы упростить выбор фильтра, мы сохраняем все возможные фильтры в карте:

```
declare const filterMap: Map<string, typeof FilterItem>;

filterMap.set('number', AFilter);
filterMap.set('stuff', BFilter);
```

В качестве обобщенного типа карты используется `string` (для токена из бэкенда) и все, что дополняет сигнатуру типа `FilterItem`. Мы используем ключевое слово `typeof`, чтобы иметь возможность добавлять в карту классы, а не объекты. Мы хотим создать их экземпляры позже.

Пока все работает так, как и следовало ожидать. Проблема возникает, когда мы хотим получить класс из карты и создать с его помощью новый объект:

```
let obj: FilterItem;
// получить конструктор
const ctor = filterMap.get('number');

if(typeof ctor !== 'undefined') {
  obj = new ctor();
  //      ^
  // cannot create an object of an abstract class
}
```

Это проблема! На данный момент TypeScript знает только о том, что мы получаем `FilterItem` обратно, и мы не можем создать экземпляр `FilterItem`. Абстрактные классы смешивают информацию о типе (*пространство имен типов*) с фактической реализацией (*пространство имен значений*). В качестве первого шага просто рассмотрим типы: что мы ожидаем получить обратно от `filterMap`? Создадим интерфейс (или псевдоним типа), который определяет, как должна выглядеть *структура* `FilterItem`:

```
interface IFilter {
  new(property: string): IFilter;
  someFunction(): void;
  filter(): void;
}

declare const filterMap: Map<string, IFilter>;
```

Обратите внимание на ключевое слово `new`. Это способ, с помощью которого TypeScript определяет сигнатуру типа функции-конструктора. Если мы заменим абстрактный класс реальным интерфейсом, то появится множество ошибок. Где бы мы ни поместили команду `implements IFilter`, ни одна реализация, похоже, не удовлетворяет нашему контракту:

```
abstract class FilterItem implements IFilter { /* ... */ }
// ^
// Class 'FilterItem' incorrectly implements interface 'IFilter'.
// Type 'FilterItem' provides no match for the signature
// 'new (property: string): IFilter'.

filterMap.set('number', AFilter);
//           ^
// Argument of type 'typeof AFilter' is not assignable
// to parameter of type 'IFilter'. Type 'typeof AFilter' is missing
// the following properties from type 'IFilter': someFunction, filter
```

Что здесь происходит? Похоже, что ни реализация, ни сам класс не могут получить все свойства и функции, которые мы определили в объявлении нашего интерфейса. Почему?

Классы JavaScript — нечто особенное: у них есть не один тип, который мы могли бы легко определить, а два: тип статической части и тип экземпляра. Возможно, будет понятнее, если мы преобразуем наш класс в то, чем он был до ES6, — в функцию-конструктор и прототип:

```
function AFilter(property) { // это часть статической стороны
  this.property = property; // это часть стороны экземпляра
}

// функция стороны экземпляра
AFilter.prototype.filter = function() { /* ... */ }

// не часть нашего примера, но на статической стороне
AFilter.something = function () { /* ... */ }
```

Один тип для создания объекта. Один тип для самого объекта. Поэтому разделим его и создадим для него два объявления типов:

```
interface FilterConstructor {
  new (property: string): IFilter;
}

interface IFilter {
  someFunction(): void;
  filter(): void;
}
```

Первый тип, `FilterConstructor`, представляет собой *интерфейс конструктора*. Здесь представлены все статические свойства и сама функция-конструктор. Она возвращает экземпляр: `IFilter`. Он содержит информацию о типе на стороне экземпляра. Это все функции, которые мы объявляем.

Благодаря разделению на части наши последующие типизации также становятся намного понятнее:

```
declare const filterMap: Map<string, FilterConstructor>; /* 1 */

filterMap.set('number', AFilter);
filterMap.set('stuff', BFilter);

let obj: IFilter; /* 2 */
const ctor = filterMap.get('number');
if(typeof ctor !== 'undefined') {
  obj = new ctor('a');
}
```

1. Мы добавляем в нашу карту экземпляры типа `FilterConstructor`. Это означает, что мы можем добавлять только те классы, которые создают нужные объекты.
2. В итоге мы хотим получить экземпляр `IFilter`. Именно его возвращает функция-конструктор при вызове с помощью `new`.

Наш код снова компилируется, и мы получаем все необходимые функции автозавершения и инструменты. Вдобавок (и это очень хорошо) мы не можем добавлять абстрактные классы в карту, поскольку они не создают корректных экземпляров:

```
filterMap.set('notworking', FilterItem);
//           ^
// Cannot assign an abstract constructor type to a
// non-abstract constructor type.
```

Паттерн интерфейса конструктора используется во всем TypeScript и стандартной библиотеке. Чтобы получить представление о нем, взгляните на интерфейс `ObjectConstructor` из `lib.es5.d.ts`.

11.4. Использование обобщенных типов в классах

Задача

Обобщенные типы TypeScript во многом рассчитаны на то, чтобы их можно было часто выводить, но в классах это не всегда работает.

Решение

Явно аннотируйте обобщенные типы при создании экземпляра, если не можете вывести их из своих параметров; в противном случае по умолчанию они будут иметь значение `unknown` и принимать широкий диапазон значений. Используйте обобщенные ограничения и параметры по умолчанию, чтобы обеспечить дополнительную безопасность.

Обсуждение

Классы позволяют использовать обобщенные типы. Вместо того чтобы добавлять параметры обобщенного типа только в функции, мы можем добавлять их в классы. Параметры обобщенного типа для методов класса действительны только в области видимости функции, а параметры обобщенного типа для классов — для всего класса.

Создадим коллекцию, простую обертку вокруг массива с ограниченным набором удобных функций. Мы можем добавить `T` в определение класса `Collection` и повторно использовать этот параметр типа во всем классе:

```
class Collection<T> {
  items: T[];
  constructor() {
    this.items = [];
  }

  add(item: T) {
    this.items.push(item);
  }

  contains(item: T): boolean {
    return this.items.includes(item);
  }
}
```

Благодаря этому мы можем явно заменить `T` на аннотацию обобщенного типа, например, разрешить коллекцию, состоящую только из чисел или строк:


```
const numbers = new Collection<number>();
numbers.add(1);
numbers.add(2);
```

```
const strings = new Collection<string>();
strings.add("Hello");
strings.add("World");
```

От нас, разработчиков, не требуется явно аннотировать параметры обобщенных типов. TypeScript обычно пытается вывести обобщенные типы на основе их использования. Если мы *забудем* добавить параметр обобщенного типа, то TypeScript вернется к значению `unknown`, что позволит нам добавить все:

```
const unknowns = new Collection();
unknowns.add(1);
unknowns.add("World");
```

На секунду остановимся на этом моменте. TypeScript очень честен с нами. В момент создания нового экземпляра `Collection` мы не знаем, какой тип у наших элементов. `unknown` — наиболее точное описание состояния коллекции. И у этого есть все отрицательные стороны: мы можем добавлять что угодно, и нам нужно выполнять проверку типа каждый раз, когда мы извлекаем значение. TypeScript делает единственное, что возможно на данный момент, но мы могли бы захотеть добиться большего. Конкретный тип для `T` обязателен для правильной работы `Collection`.

Посмотрим, можем ли мы использовать выведение типов. В TypeScript для классов оно работает так же, как и для функций. Если есть параметр определенного типа, то TypeScript примет этот тип и заменит параметр обобщенного типа. Классы спроектированы таким образом, чтобы сохранять состояние, а оно изменяется на протяжении всего их использования. Состояние также определяет наш параметр обобщенного типа `T`. Чтобы правильно вывести тип `T`, нужно запросить параметр в конструкторе, который может быть начальным значением:

```
class Collection<T> {
  items: T[];
  constructor(initial: T) {
    this.items = [initial];
  }

  add(item: T) {
    this.items.push(item);
  }

  contains(item: T): boolean {
    return this.items.includes(item);
  }
}
```

```
// T – это число!
const numbersInf = new Collection(0);
numbersInf.add(1);
```

Код работает, но оставляет желать лучшего для дизайна нашего API-интерфейса. Что делать, если у нас нет начальных значений? У других классов могут быть параметры, которые можно использовать для вывода типа, однако для коллекции различных элементов это не имеет большого смысла.

Для `Collection` абсолютно необходимо указывать тип с помощью аннотации. Остается только убедиться, что мы не забыли ее добавить. Для этого мы можем воспользоваться обобщенными параметрами TypeScript по умолчанию и нижним типом `never`:

```
class Collection<T = never> {
  items: T[];
  constructor() {
    this.items = [];
  }

  add(item: T) {
    this.items.push(item);
  }

  contains(item: T): boolean {
    return this.items.includes(item);
  }
}
```

Мы установили для параметра обобщенного типа `T` значение по умолчанию `never`, что позволяет нашему классу демонстрировать очень интересное поведение. Тип `T` по-прежнему может быть явно заменен на любой тип с помощью аннотации; эта схема работает так же, как и раньше, но в тот момент, когда мы забываем аннотацию, тип не становится `unknown`, это `never`. То есть ни одно значение несовместимо с нашей коллекцией, что приводит ко множеству ошибок в тот момент, когда мы пытаемся что-то добавить:

```
const nevers = new Collection();
nevers.add(1);
//      ^
// Argument of type 'number' is not assignable
// to parameter of type 'never'.(2345)
nevers.add("World");
//      ^
// Argument of type 'string' is not assignable
// to parameter of type 'never'.(2345)
```

Этот запасной вариант делает использование наших обобщенных классов гораздо более безопасным.

11.5. Решение об использовании классов или пространства имен

Задача

TypeScript предлагает множество вариантов синтаксиса для объектно-ориентированных концепций, таких как пространства имен или статические и абстрактные классы. Этим возможностей нет в JavaScript, что же вам делать?

Решение

Придерживайтесь объявлений пространства имен для дополнительных объявлений типов, избегайте абстрактных классов, когда это возможно, и отдавайте предпочтение модулям ECMAScript вместо статических классов.

Обсуждение

Мы видим, что разработчики, много работавшие с традиционными объектно-ориентированными языками программирования, такими как Java или C#, стремятся поместить все в класс. В Java у вас нет других вариантов, поскольку классы — это единственный способ структурировать код. В JavaScript (и, соответственно, в TypeScript) есть множество других возможностей сделать то, что вы хотите, не выполняя какие-либо дополнительные действия. Одна из них — статические классы или классы со статическими методами:

```
// Environment.ts

export default class Environment {
  private static variableList: string[] = []
  static variables(): string[] { /* ... */ }
  static setVariable(key: string, value: any): void { /* ... */ }
  static getValue(key: string): unknown { /* ... */ }
}
```

```
// Использование в другом файле
import * as Environment from "./Environment";
```

```
console.log(Environment.variables());
```

Этот код работает и даже (без аннотаций типа) является допустимым JavaScript-кодом, однако в нем содержится слишком много действий для чего-то, что может быть просто обычными, скучными функциями:

```
// Environment.ts
const variableList: string = []
```

```

export function variables(): string[] { /* ... */ }
export function setVariable(key: string, value: any): void { /* ... */ }
export function getValue(key: string): unknown { /* ... */ }

// Использование в другом файле
import * as Environment from "./Environment";

console.log(Environment.variables());

```

Интерфейс для ваших пользователей точно такой же. Вы можете получить доступ к переменным в области видимости модуля так же, как получаете доступ к статическим свойствам класса, но при этом они автоматически попадают в область видимости модуля. Вы решаете, что экспортировать и сделать видимым, а не модификаторы полей TypeScript. Кроме того, вы не создаете экземпляр `Environment`, который ничего не делает.

Даже реализация упрощается. Посмотрите на версию `variables()` для класса:

```

export default class Environment {
  private static variableList: string[] = [];
  static variables(): string[] {
    return this.variableList;
  }
}

```

Она отличается от версии модуля:

```

const variableList: string = []

export function variables(): string[] {
  return variableList;
}

```

Отсутствие `this` означает, что нужно меньше думать. Плюс вы получаете дополнительное преимущество: вашим сборщикам будет легче заниматься встряской дерева вызова, так что в итоге вы получите только то, что действительно используете:

```

// В пакет попадают
// только переменные function и variableList
import { variables } from "./Environment";

console.log(variables());

```

Вот почему правильный модуль всегда предпочтительнее класса со статическими полями и методами. Это всего лишь дополнительный шаблон, не приносящий никакой пользы.

Как и в случае со статическими классами, разработчики, знакомые с Java или C#, используют пространства имен — функцию, которую TypeScript ввел в целях орга-

низации кода задолго до стандартизации модулей ECMAScript. Они позволяли разделять объекты по файлам, объединяя их с помощью ссылочных маркеров:

```
// файл users/models.ts
namespace Users {
  export interface Person {
    name: string;
    age: number;
  }
}

// файл users/controller.ts

/// <reference path="./models.ts" />
namespace Users {
  export function updateUser(p: Person) {
    // сделайте остальное
  }
}
```

В те времена в TypeScript даже была функция компоновки. Она и сейчас должна работать. Но, как уже отмечалось, это было до того, как в ECMAScript появились модули. Теперь благодаря им у нас есть способ, позволяющий организовать и структурировать код, совместимый с остальной экосистемой JavaScript. И это плюс.

Так зачем нужны пространства имен? Они по-прежнему актуальны, если вы хотите расширить определения из сторонних зависимостей, например тех, которые живут внутри узловых модулей. Допустим, вы хотите расширить глобальное пространство имен JSX и убедиться, что элементы `img` содержат замещающие тексты:

```
declare namespace JSX {
  interface IntrinsicElements {
    "img": HTMLAttributes & {
      alt: string;
      src: string;
      loading?: 'lazy' | 'eager' | 'auto';
    }
  }
}
```

Или вы хотите написать сложные определения типов в окружающих модулях. А что еще? Больше от этого нет никакой пользы.

Пространства имен оборачивают ваши определения в объект, что выглядит примерно так:

```
export namespace Users {
  type User = {
    name: string;
    age: number;
  };
}
```

```

export function createUser(name: string, age: number): User {
  return { name, age };
}
}

```

Он создает нечто очень замысловатое:

```

export var Users;
(function (Users) {
  function createUser(name, age) {
    return {
      name, age
    };
  }
  Users.createUser = createUser;
})(Users || (Users = {}));

```

Этот код не только добавляет лишние элементы, но и мешает вашим сборщикам правильно встряхивать дерево! Вдобавок для того, чтобы их использовать, нужно написать немного больше слов:

```

import * as Users from "./users";

Users.Users.createUser("Stefan", "39");

```

Отказ от использования пространств имен значительно упрощает задачу. Придерживайтесь того, что предлагает JavaScript. Отказавшись от пространств имен за пределами файлов объявлений, вы сделаете код понятным, простым и аккуратным.

И последнее, но не менее важное: существуют абстрактные классы. Это способ структурировать более сложную иерархию классов, при котором вы заранее определяете поведение, но фактическую реализацию некоторых функций выполняют классы, которые являются *расширением* вашего абстрактного класса:

```

abstract class Lifeform {
  age: number;
  constructor(age: number) {
    this.age = age;
  }

  abstract move(): string;
}

class Human extends Lifeform {
  move() {
    return "Walking, mostly...";
  }
}

```

Все подклассы `Lifeform` должны реализовать метод `move`. Это концепция, которая существует практически в каждом языке программирования, основанном на классах. Проблема в том, что JavaScript традиционно не основан на них. Например,

абстрактный класс, подобный приведенному ниже, генерирует допустимый класс JavaScript, но не может быть создан в TypeScript:

```
abstract class Lifeform {
  age: number;
  constructor(age: number) {
    this.age = age;
  }
}

const lifeform = new Lifeform(20);
//           ^
// Cannot create an instance of an abstract class.(2511)
```

Это может привести к нежелательным ситуациям, если вы пишете обычный JavaScript, но используете TypeScript для предоставления информации в виде неявной документации, например, если определение функции выглядит следующим образом:

```
declare function moveLifeform(lifeform: Lifeform);
```

- Вы или ваши пользователи могут воспринять это как приглашение передать объект `Lifeform` для перемещения в `moveLifeform`. Внутри него вызывается функция `lifeform.move()`.
- `Lifeform` может быть создан в JavaScript, поскольку это допустимый класс.
- Метод `move` не существует в `Lifeform`, что нарушает работу вашего приложения!

Все это происходит из-за ложного чувства безопасности. На самом деле вам нужно поместить некую предопределенную реализацию в цепочку прототипов и иметь контракт, который сообщает вам, чего ожидать:

```
interface Lifeform {
  move(): string;
}

class BasicLifeForm {
  age: number;
  constructor(age: number) {
    this.age = age;
  }
}

class Human extends BasicLifeForm implements Lifeform {
  move() {
    return "Walking";
  }
}
```

В момент просмотра `Lifeform` вы можете увидеть интерфейс и все, что для него требуется, но вы редко сталкиваетесь с ситуацией, когда случайно создаете экземпляр неправильного класса.

Если учесть все сказанное о том, когда *не следует* использовать классы и пространства имен, то в каких случаях их надо применять? Каждый раз, когда вам нужны несколько экземпляров одного и того же объекта, функциональность которого напрямую зависит от его внутреннего состояния.

11.6. Написание статических классов

Задача

Объектно-ориентированное программирование на основе классов научило вас использовать статические классы для определенных функций, но вам интересно, как эти принципы поддерживаются в TypeScript.

Решение

Традиционных статических классов в TypeScript не существует, но есть статические модификаторы для членов класса, которые используются для нескольких целей.

Обсуждение

Статические классы — это классы, которые нельзя создать в виде конкретных объектов. Их назначение — содержать методы и другие члены, которые создаются один раз и остаются неизменными при обращении к ним из разных точек кода. Статические классы необходимы для языков программирования, в которых в качестве средства абстракции используются только классы. Это такие языки, как Java или C#. В JavaScript, а впоследствии и в TypeScript существует гораздо больше способов реализации.

В TypeScript мы не можем объявлять классы `static`, но мы можем определять члены `static` для классов. Поведение будет такое, какого следовало ожидать: метод или свойство не является частью объекта, но доступ к ним можно получить из самого класса.

Как мы видели в рецепте 11.5, классы, содержащие только статические члены, являются антипаттерном в TypeScript. Существуют функции, и вы можете сохранять состояние для каждого модуля. Обычно лучше всего использовать комбинацию экспортируемых функций и относящихся к модулю записей:

```
// Антипаттерн
export default class Environment {
  private static variableList: string[] = []
  static variables(): string[] { /* ... */ }
```



```
    static setVariable(key: string, value: any): void { /* ... */ }
    static getValue(key: string): unknown { /* ... */ }
}
```

```
// Улучшено: функции и переменные в рамках модуля
const variableList: string = []
```

```
export function variables(): string[] { /* ... */ }
export function setVariable(key: string, value: any): void { /* ... */ }
export function getValue(key: string): unknown { /* ... */ }
```

Но частям `static` класса все равно найдется применение. В рецепте 11.3 мы установили, что класс состоит из статических и динамических членов.

Элемент `constructor` — часть статических свойств класса, а свойства и методы — часть динамических функций класса. С помощью ключевого слова `static` мы можем добавить в эти статические объекты дополнительные элементы.

Представим класс `Point`, который описывает точку в двумерном пространстве. У нее есть координаты `x` и `y`, и мы создадим метод, который вычислит расстояние между этой точкой и другой:

```
class Point {
  x: number;
  y: number;

  constructor(x: number, y: number) {
    this.x = x;
    this.y = y;
  }

  distanceTo(point: Point): number {
    const dx = this.x - point.x;
    const dy = this.y - point.y;
    return Math.sqrt(dx * dx + dy * dy);
  }
}

const a = new Point(0, 0);
const b = new Point(1, 5);

const distance = a.distanceTo(b);
```

Это хорошее поведение, но API может показаться немного странным, если мы выберем начальную и конечную точки, тем более что расстояние будет одинаковым независимо от того, какая из них первая. Статический метод на `Point` устраняет порядок, и мы получаем метод `distance`, который принимает два аргумента:

```
class Point {
  x: number;
  y: number;
```

```

constructor(x: number, y: number) {
  this.x = x;
  this.y = y;
}

distanceTo(point: Point): number {
  const dx = this.x - point.x;
  const dy = this.y - point.y;
  return Math.sqrt(dx * dx + dy * dy);
}

static distance(p1: Point, p2: Point): number {
  return p1.distanceTo(p2);
}
}

const a = new Point(0, 0);
const b = new Point(1, 5);

const distance = Point.distance(a, b);

```

Аналогичная версия с использованием паттерна «функция-конструктор/прототип» (constructor function/prototype), который использовался в классах JavaScript до появления классов ECMAScript, выглядела бы следующим образом:

```

function Point(x, y) {
  this.x = x;
  this.y = y;
}

Point.prototype.distanceTo = function(p) {
  const dx = this.x - p.x;
  const dy = this.y - p.y;
  return Math.sqrt(dx * dx + dy * dy);
}

Point.distance = function(a, b) {
  return a.distanceTo(b);
}

```

Как и в рецепте 11.3, вы можете увидеть, какие части являются статическими, а какие — динамическими. Все, что есть в *прототипе*, относится к динамическим частям. Все остальное *статично*.

Но классы — это не только синтаксический сахар для паттерна «функция-конструктор/прототип». Добавив приватные поля, которые отсутствуют в обычных объектах, мы можем сделать что-то, что действительно связано с классами и их экземплярами.

Допустим, мы хотим скрыть метод `distanceTo`, поскольку он может привести к путанице, и предпочли бы, чтобы пользователи вместо него применяли статический

метод. В таком случае простой модификатор `private` перед `distanceTo` сделает его недоступным снаружи, но сохранит его доступность внутри статических членов:

```
class Point {
  x: number;
  y: number;

  constructor(x: number, y: number) {
    this.x = x;
    this.y = y;
  }

  #distanceTo(point: Point): number {
    const dx = this.x - point.x;
    const dy = this.y - point.y;
    return Math.sqrt(dx * dx + dy * dy);
  }

  static distance(p1: Point, p2: Point): number {
    return p1.#distanceTo(p2);
  }
}
```

Видимость тоже меняется. Допустим, у нас есть класс, который представляет определенную задачу `Task` в системе, и мы хотим ограничить количество существующих задач.

Мы используем статическое приватное поле `nextId`, которое начинается с `0`, и оно увеличивается по мере создания каждого экземпляра `Task`. Если мы достигаем `100`, то код выдает ошибку:

```
class Task {
  static #nextId = 0;
  #id: number;

  constructor() {
    if (Task.#nextId > 99) {
      throw "Max number of tasks reached";
    }
    this.#id = Task.#nextId++;
  }
}
```

Если мы хотим ограничить количество экземпляров динамическим значением из серверной части, то можем использовать `static` — блок создания экземпляров, который получает эти данные и соответствующим образом обновляет статические приватные поля:

```
type Config = {
  instances: number;
};
```

```
class Task {
  static #nextId = 0;
  static #maxInstances: number;
  #id: number;

  static {
    fetch("/available-slots")
      .then((res) => res.json())
      .then((result: Config) => {
        Task.#maxInstances = result.instances;
      });
  }

  constructor() {
    if (Task.#nextId > Task.#maxInstances) {
      throw "Max number of tasks reached";
    }
    this.#id = Task.#nextId++;
  }
}
```

За исключением полей в экземплярах, TypeScript на момент написания книги не проверяет, созданы ли экземпляры статических полей. Если мы, например, асинхронно загружаем количество доступных слотов из серверной части, то у нас есть определенный промежуток времени, в течение которого мы можем создавать экземпляры, но нет возможности проверить, достигли ли мы своего максимума.

Таким образом, даже если в TypeScript нет конструкции статического класса, а классы только со статическими элементами считаются антипаттерном, то во многих ситуациях статические члены могут быть весьма полезными.

11.7. Работа со строгой инициализацией свойств

Задача

Классы сохраняют состояние, но ничто не сообщает вам, инициализируется ли оно.

Решение

Активируйте строгую инициализацию свойств, установив в вашем `tsconfig` значение `strictPropertyInitialization` равным `true`.

Обсуждение

Классы можно рассматривать как шаблоны кода для создания объектов. Вы определяете свойства и методы, и только при создании экземпляра присваиваются фактические значения. Классы TypeScript берут базовые классы JavaScript

и расширяют их, добавляя дополнительный синтаксис для определения типов. Например, TypeScript позволяет определять свойства экземпляра в виде типа или интерфейса:

```
type State = "active" | "inactive";

class Account {
  id: number;
  userName: string;
  state: State;
  orders: number[];
}
```

Однако это обозначение определяет только форму: оно пока не устанавливает никаких конкретных значений. При транспиляции в обычный JavaScript все эти свойства стираются; они существуют только в *пространстве имен типов*.

Такой способ записи, вероятно, очень удобен для чтения и дает разработчику представление о том, какие свойства следует ожидать. Но нет никакой гарантии, что эти свойства действительно существуют. Если мы их не инициализируем, то все будет либо отсутствовать, либо быть `undefined`.

В TypeScript есть меры предосторожности на этот случай. Если установить в файле `tsconfig.json` флаг `strictPropertyInitialization` в значение `true`, то TypeScript убедится, что все свойства, которые вы ожидаете, действительно инициализируются при создании нового объекта из вашего класса.



`strictPropertyInitialization` — часть режима `strict` TypeScript. Если в вашем `tsconfig` для параметра `strict` установлено значение `true` (а это необходимо), то вы активируете и строгую инициализацию свойств.

Как только эта функция будет активирована, TypeScript поприветствует вас множеством красных волнистых линий:

```
class Account {
  id: number;
  // ^ Property 'id' has no initializer and is
  // not definitely assigned in the constructor.(2564)
  userName: string;
  // ^ Property 'userName' has no initializer and is
  // not definitely assigned in the constructor.(2564)
  state: State;
  // ^ Property 'state' has no initializer and is
  // not definitely assigned in the constructor.(2564)
  orders: number[];
  // ^ Property 'orders' has no initializer and is
  // not definitely assigned in the constructor.(2564)
}
```

Прекрасно! Теперь нам предстоит убедиться, что каждое свойство получит значение. Сделать это можно разными способами. Учитывая пример с `Account`, мы можем определить некоторые ограничения или правила, если домен нашего приложения позволяет нам это сделать.

- Необходимо задать `id` и `userName`; они управляют связью с нашей серверной частью и необходимы для отображения.
- Должно быть задано и `state`, но по умолчанию оно имеет значение `active`. Обычно учетные записи в нашем программном обеспечении активны, если только не установлены намеренно в состоянии `inactive`.
- `orders` — это массивы, содержащие идентификаторы заказов, но что, если мы ничего не заказывали? Пустой массив работает так же хорошо, или, возможно, мы установили, что `orders` еще не определены.

Учитывая эти ограничения, мы уже можем исключить две ошибки. Мы устанавливаем для `state` значение по умолчанию `active` и делаем `orders` необязательными. Вдобавок есть возможность установить для `orders` тип `number[] | undefined`, который является тем же самым, что и необязательный:

```
class Account {
  id: number; // все еще ошибки
  userName: string; // все еще ошибки
  state: State = "active"; // ok
  orders?: number[]; // ok
}
```

Два других свойства по-прежнему выдают ошибки. Добавляя `constructor` и инициализируя эти свойства, мы исключаем и другие ошибки:

```
class Account {
  id: number;
  userName: string;
  state: State = "active";
  orders?: number[];

  constructor(userName: string, id: number) {
    this.userName = userName;
    this.id = id;
  }
}
```

Настоящий класс TypeScript готов! Кроме того, TypeScript допускает использование сокращения конструктора, когда вы можете преобразовать параметры конструктора в свойства класса с тем же именем и значением, добавив модификатор видимости, например `public`, `private` или `protected`. Это удобная функция, которая позволяет избавиться от большого количества шаблонного кода. Важно, чтобы вы не определяли одно и то же свойство в структуре класса:

```
class Account {
  state: State = "active";
  orders?: number[];

  constructor(public userName: string, public id: number) {}
}
```

Если вы посмотрите на класс прямо сейчас, то увидите, что мы полагаемся только на возможности TypeScript. Транспирированный класс, его эквивалент в JavaScript, выглядит совсем иначе:

```
class Account {
  constructor(userName, id) {
    this.userName = userName;
    this.id = id;
    this.state = "active";
  }
}
```

Все находится в `constructor`, поскольку он определяет экземпляр.



Хотя сокращения и синтаксис классов в TypeScript кажутся привлекательными, но будьте осторожны с тем, как сильно вы опираетесь на них. В последние годы TypeScript изменился, став в основном синтаксическим расширением для типов поверх обычного JavaScript. Но функции классов, существующие уже много лет, по-прежнему доступны и добавляют в ваш код семантику, отличную от той, которую вы ожидали. Если вы предпочитаете, чтобы ваш код был «JavaScript с типами», то будьте осторожны, когда решитесь углубиться в возможности классов TypeScript.

Строгая инициализация свойств учитывает сложные сценарии, такие как установка свойства в функции, вызываемой через `constructor`. Она учитывает, что асинхронный класс может оставить ваш класс с потенциально неинициализированным состоянием. Допустим, вы просто хотите инициализировать свой класс с помощью свойства `id` и получить свойство `userName` из серверной части. Если вы выполните асинхронный вызов в конструкторе и установите свойство `userName` после завершения вызова `fetch`, то все равно получите ошибки строгой инициализации свойств:

```
type User = {
  id: number;
  userName: string;
};

class Account {
  userName: string;
  // ^ Property 'userName' has no initializer and is
  // not definitely assigned in the constructor.(2564)
  state: State = "active";
  orders?: number[];
```

```

constructor(public id: number) {
  fetch(`/api/getName?id=${id}`)
    .then((res) => res.json())
    .then((data: User) => (this.userName = data.userName ?? "not-found"));
}
}

```

И это правильно! Ничто не говорит вам о том, что вызов `fetch` будет успешным, и даже если вы отловите ошибки в `catch` и убедитесь, что свойство будет инициализировано с помощью резервного значения, в течение определенного периода времени ваш объект будет находиться в неинициализированном состоянии свойства `userName`.

Обойти это ограничение можно с помощью нескольких способов. Один из хороших паттернов — это статическая фабричная функция, работающая асинхронно, когда вы сначала получаете данные, а затем вызываете конструктор, который ожидает оба свойства:

```

class Account {
  state: State = "active";
  orders?: number[];
  constructor(public id: number, public userName: string) {}

  static async create(id: number) {
    const user: User = await fetch(`/api/getName?id=${id}`).then((res) =>
      res.json()
    );
    return new Account(id, user.userName);
  }
}

```

Это позволяет создавать экземпляры обоих объектов в неасинхронном контексте, если у вас есть доступ к обоим свойствам, или в асинхронном, если вы можете получить доступ только к `id`. Мы меняем ответственность и полностью удаляем асинхронность из конструктора.

Другой способ состоит в том, чтобы просто игнорировать неинициализированное состояние. Что делать, если состояние `userName` совершенно неважно для вашего приложения и вы хотите обращаться к нему только при необходимости? Используйте *утверждение определения присваивания* (восклицательный знак), чтобы сообщить TypeScript, что будете рассматривать это свойство как инициализированное:

```

class Account {
  userName!: string;
  state: State = "active";
  orders?: number[];

  constructor(public id: number) {
    fetch(`/api/getName?id=${id}`)

```



```

    .then((res) => res.json())
    .then((data: User) => (this.userName = data.userName));
  }
}

```

Теперь ответственность лежит на вас, и благодаря восклицательному знаку вы можете использовать синтаксис, характерный для TypeScript, который можно квалифицировать как небезопасную операцию, в том числе ошибки времени выполнения.

11.8. Работа с типами `this` в классах

Задача

Вы расширяете возможности базовых классов, чтобы повторно использовать функциональность, и ваши методы имеют сигнатуры, которые ссылаются на экземпляр того же класса. Вы хотите быть уверены, что никакие другие подклассы не смешиваются в ваших интерфейсах, но не хотите переопределять методы только для того, чтобы изменить тип.

Решение

Используйте тип `this` вместо фактического типа класса.

Обсуждение

В этом примере мы хотим с помощью классов смоделировать различные роли пользователей программного обеспечения для доски объявлений. Мы начнем с общего класса `User`, который идентифицируется по ID пользователя и имеет возможность открывать потоки:

```

class User {
  #id: number;
  static #nextThreadId: number;

  constructor(id: number) {
    this.#id = id;
  }

  equals(user: User): boolean {
    return this.#id === user.#id;
  }

  async openThread(title: string, content: string): Promise<number> {
    const threadId = User.#nextThreadId++;
  }
}

```

```

    await fetch("/createThread", {
      method: "POST",
      body: JSON.stringify({
        content,
        title,
        threadId,
      }),
    });
    return threadId;
  }
}

```

Этот класс содержит также метод `equals`. Где-то в нашей кодовой базе нам нужно убедиться, что две ссылки на пользователей совпадают, а поскольку мы идентифицируем пользователей по их ID, то можем легко сравнить числа.

`User` — это базовый класс всех пользователей, так что если мы добавим роли с дополнительными привилегиями, то можем легко наследовать от него. Например, `Admin` имеет возможность закрывать потоки, а также хранит набор иных привилегий, которые мы можем использовать в других методах.



В сообществе программистов ведется много споров о том, является ли наследование методом, который лучше игнорировать, поскольку его преимущества вряд ли перевешивают его недостатки. Тем не менее в некоторых частях JavaScript наследование используется, например, в веб-компонентах.

Мы наследуем от `User`, поэтому нам не нужно писать еще один метод `openThread` и мы можем повторно использовать тот же метод `equals`, так как все администраторы являются и пользователями:

```

class Admin extends User {
  #privileges: string[];
  constructor(id: number, privileges: string[] = []) {
    super(id);
    this.#privileges = privileges;
  }

  async closeThread(threadId: number) {
    await fetch("/closeThread", {
      method: "POST",
      body: "" + threadId,
    });
  }
}

```

После настройки классов мы можем создавать новые объекты типа `User` и `Admin`, добавляя экземпляры нужных классов. Кроме того, мы можем вызвать метод `equals`, чтобы сравнить, могут ли два пользователя быть одинаковыми:

```
const user = new User(1);
const admin = new Admin(2);

console.log(user.equals(admin));
console.log(admin.equals(user));
```

Однако вызывает беспокойство одно обстоятельство: направление сравнения. Конечно, сравнение двух чисел — коммутативный процесс, поэтому не должно иметь значения, сравниваем ли мы `user` с `admin`, но если мы подумаем об окружающих классах и подтипах, то увидим, что есть возможности для улучшения:

- можно проверить, равны ли `user` и `admin`, поскольку это может привести к получению привилегий;
- вряд ли мы хотим, чтобы `admin` был равен `user`, так как более широкий супертип содержит меньше информации;
- если у нас есть другой подкласс `Moderator`, смежный с `Admin`, то мы определенно не хотим иметь возможность сравнивать их, поскольку у них нет общих свойств за пределами базового класса.

Тем не менее в том виде, в котором `equals` разрабатывается сейчас, все сравнения будут работать. Мы можем обойти это ограничение, изменив тип элемента, который хотим сравнить. Сначала мы аннотировали входной параметр как `User`, но в действительности хотим сравнить его с *другим экземпляром того же типа*. Для этого существует специальный тип, который называется `this`:

```
class User {
  // ...

  equals(user: this): boolean {
    return this.#id === user.#id;
  }
}
```

Это отличается от стираемого параметра `this`, знакомого нам по функциям, с которыми мы познакомились в рецепте 2.7. Тип параметра `this` позволяет задать конкретный тип для глобальной переменной `this` в области видимости функции. Тип `this` является ссылкой на класс, в котором находится метод, и меняется в зависимости от реализации. Таким образом, если мы аннотируем `user` с помощью `this` в `User`, то он станет `Admin` в классе, который наследуется от `User`, или `Moderator`, и т. д. При этом `admin.equals` ожидает, что будет сравниваться с другим классом `Admin`; в противном случае мы получим сообщение об ошибке:

```
console.log(admin.equals(user));
//                               ^
// Argument of type 'User' is not assignable to parameter of type 'Admin'.
```

Другой способ по-прежнему работает. Поскольку `Admin` содержит все свойства `User` (это же подкласс), то мы можем легко сравнить `user.equals(admin)`.

Эти типы можно использовать и в качестве типов возвращаемых параметров. Взгляните на этот `OptionBuilder`, который реализует *паттерн постановщика*:

```
class OptionBuilder<T = string | number | boolean> {
  #options: Map<string, T> = new Map();
  constructor() {}

  add(name: string, value: T): OptionBuilder<T> {
    this.#options.set(name, value);
    return this;
  }

  has(name: string) {
    return this.#options.has(name);
  }

  build() {
    return Object.fromEntries(this.#options);
  }
}
```

Это легкая обертка вокруг `Map`, которая позволяет нам задавать пары «ключ — значение». Она имеет цепочечный интерфейс; это значит, после каждого вызова `add` мы возвращаем текущий экземпляр, что позволяет нам выполнять вызовы `add` один за другим. Обратите внимание, что мы аннотировали возвращаемый тип с помощью `OptionBuilder<T>`:

```
const options = new OptionBuilder()
  .add("deflate", true)
  .add("compressionFactor", 10)
  .build();
```

Теперь мы создаем `StringOptionBuilder`, который является потомком `OptionBuilder` и устанавливает тип возможных элементов в `string`. Кроме того, мы добавляем метод `safeAdd`, который проверяет, не установлено ли уже определенное значение перед записью, поэтому мы не переопределяем предыдущие настройки:

```
class StringOptionBuilder extends OptionBuilder<string> {
  safeAdd(name: string, value: string) {
    if (!this.has(name)) {
      this.add(name, value);
    }
    return this;
  }
}
```

Начиная использовать новый конструктор, мы видим, что не можем разумно использовать `safeAdd`, если `add` задействуется в качестве первого шага:

```
const languages = new StringOptionBuilder()
  .add("en", "English")
```

```

    .safeAdd("de", "Deutsch")
// ^
// Property 'safeAdd' does not exist on type 'OptionBuilder<string>'.(2339)
    .safeAdd("de", "German")
    .build();

```

TypeScript сообщает нам, что `safeAdd` не существует для типа `OptionBuilder<string>`. Куда же подевалась эта функция? Проблема в том, что `add` имеет очень широкую аннотацию. Конечно, `StringOptionBuilder` — подтип `OptionBuilder<string>`, но при наличии аннотации мы теряем информацию о более узком типе. Как можно решить эту проблему? Использовать `this` в качестве типа возвращаемого значения:

```

class OptionBuilder<T = string | number | boolean> {
  // ...

  add(name: string, value: T): this {
    this.#options.set(name, value);
    return this;
  }
}

```

Наблюдается тот же эффект, что и в предыдущем примере. В `OptionBuilder<T>` тип `this` становится `OptionBuilder<T>`, а в `StringBuilder` — `StringBuilder`. Если вы возвращаете `this` и не указываете аннотацию возвращаемого типа, то `this` станет *предположительным* типом возвращаемого значения. Поэтому использование типа `this` в явном виде зависит от ваших предпочтений (см. рецепт 2.1).

11.9. Декораторы

Задача

Вы хотите регистрировать выполнение ваших методов, чтобы получить данные телеметрии, но вручную добавлять журналы к каждому методу очень сложно.

Решение

Напишите декоратор методов класса `log` для аннотирования ваших методов.

Обсуждение

Паттерн проектирования *декоратор* был описан в известной книге *Design Patterns: Elements of Reusable Object-Oriented Software*¹ Эриха Гаммы и др. и описывает прием, который позволяет *декорировать* классы и методы, чтобы можно было динамически добавлять или перезаписывать определенное поведение.

¹ Гамма Э. и др. Паттерны объектно-ориентированного проектирования. — СПб.: Питер, 2021.

То, что начиналось как естественный паттерн проектирования объектно-ориентированного программирования, стало настолько популярным, что языки программирования, в которых есть объектно-ориентированные аспекты, добавили декораторы в качестве языковой функции со специальным синтаксисом. Вы можете ознакомиться с его разновидностями в Java (так называемые *аннотации*) или C# (так называемые *атрибуты*), а также в JavaScript.

Предложение ECMAScript по декораторам рассматривалось очень долгое время, но в 2022 году достигло стадии 3 (готовности к реализации). И поскольку все функции также достигли этой стадии, TypeScript является одним из первых инструментов, который принял новую спецификацию.



Декораторы уже давно существуют в TypeScript под флагом компилятора `experimentalDecorators`. В TypeScript 5.0 полностью реализован собственный вариант декоратора ECMAScript, который доступен без флага. Фактическая реализация ECMAScript кардинально отличается от первоначального проекта, и если вы разработали декораторы до TypeScript 5.0, то они не будут работать с новой спецификацией. Обратите внимание, что установленный флаг `experimentalDecorators` отключает встроенные декораторы ECMAScript. Кроме того, `lib.decorators.d.ts` содержит всю информацию о типах для собственных декораторов ECMAScript, а типы в `lib.decorators.legacy.d.ts` содержат сведения о старых типах. Убедитесь, что ваши настройки верны и что вы не используете типы из неправильного файла определений.

Декораторы позволяют обрабатывать практически все элементы класса. В этом примере мы хотим начать с декоратора методов, который позволит регистрировать выполнение вызовов методов.

Декораторы описываются как функции со *значением* и *контекстом*, которые зависят от типа элемента класса, который вы хотите обработать. Эти функции-декораторы возвращают другую функцию, которая будет выполнена перед вашим собственным методом (или перед инициализацией поля, вызовом метода доступа и т. д.).

Простой декоратор журнала `log` для методов может выглядеть так:

```
function log(value: Function, context: ClassMethodDecoratorContext) {
    return function (this: any, ...args: any[]) {
        console.log(`calling ${context.name.toString()}`);
        return value.call(this, ...args);
    };
}

class Toggler {
    #toggled = false;
```

```

@log
toggle() {
  this.#toggled = !this.#toggled;
}
}

```

```

const toggler = new Toggler();
toggler.toggle();

```

Функция `log` соответствует типу `ClassMethodDecorator`, определенному в исходном предложении декоратора (<https://oreil.ly/76JuE>):

```

type ClassMethodDecorator = (value: Function, context: {
  kind: "method";
  name: string | symbol;
  access: { get(): unknown };
  static: boolean;
  private: boolean;
  addInitializer(initializer: () => void): void;
}) => Function | void;

```

Доступно множество типов контекстов декораторов. Файл `lib.decorator.d.ts` определяет следующие декораторы:

```

type ClassMemberDecoratorContext =
  | ClassMethodDecoratorContext
  | ClassGetterDecoratorContext
  | ClassSetterDecoratorContext
  | ClassFieldDecoratorContext
  | ClassAccessorDecoratorContext
  ;

/**
 * The decorator context types provided to any decorator.
 */
type DecoratorContext =
  | ClassDecoratorContext
  | ClassMemberDecoratorContext
  ;

```

Из названий вы можете точно определить, для какой части класса они предназначены.

Обратите внимание: мы еще не написали подробные типы. Мы используем большое количество `any` в основном потому, что типы могут стать очень сложными. Если мы захотим добавить типы для всех параметров, то нам придется прибегнуть к большому количеству обобщенных типов:

```

function log<This, Args extends any[], Return>(
  value: (this: This, ...args: Args) => Return,

```

```

    context: ClassMethodDecoratorContext
  ): (this: This, ...args: Args) => Return {
    return function (this: This, ...args: Args) {
      console.log(`calling ${context.name.toString()}`);
      return value.call(this, ...args);
    };
  }
}

```

Параметры обобщенного типа необходимы для описания метода, который мы передаем. Мы хотим перехватить следующие типы.

- `this` — это параметр обобщенного типа для типа параметра `this` (см. рецепт 2.7). Нам нужно задать `this`, поскольку декораторы запускаются в контексте экземпляра объекта.
- Тогда у нас будут аргументы метода в виде `Args`. Как вы узнали в рецепте 2.4, аргументы метода или функции можно описать в виде кортежа.
- И последнее, но не менее важное: тип возвращаемого параметра. Метод должен возвращать значение определенного типа, и мы хотим это указать.

Используя все три метода, мы можем описать методы ввода и вывода способом, который является наиболее универсальным для всех классов. Мы можем использовать обобщенные ограничения, чтобы убедиться, что наш декоратор работает только в определенных случаях, но для `log` хотим иметь возможность регистрировать каждый вызов метода.



На момент написания книги декораторы ECMAScript в TypeScript были довольно новыми. Со временем типы совершенствуются, поэтому информация о типах, которую вы получаете, уже может улучшиться.

Помимо вышесказанного, мы хотим зарегистрировать поля нашего класса и их начальное значение до вызова метода `constructor`:

```

class Toggler {
  @logField #toggled = false;

  @log
  toggle() {
    this.#toggled = !this.#toggled;
  }
}

```

Для этого мы создаем еще один декоратор `logField`, который работает с `ClassFieldDecoratorContext`. В предложении по декоратору (<https://oreil.ly/76JuE>) описывается декоратор для полей классов следующим образом:

```

type ClassFieldDecorator = (value: undefined, context: {
  kind: "field";
  name: string | symbol;

```



```

    access: { get(): unknown, set(value: unknown): void };
    static: boolean;
    private: boolean;
  }) => (initialValue: unknown) => unknown | void;

```

Обратите внимание, что `value` определено как `undefined`. Начальное значение передается в метод замены:

```

type FieldDecoratorFn = (val: any) => any;

function logField<Val>(
  value: undefined,
  context: ClassFieldDecoratorContext
): FieldDecoratorFn {
  return function (initialValue: Val): Val {
    console.log(`Initializing ${context.name.toString()} to ${initialValue}`);
    return initialValue;
  };
}

```

Есть одна вещь, которая кажется странной. Зачем нужны разные декораторы для разных типов членов? Разве декоратор журналов `log` не должен справляться с ними всеми? Он вызывается в определенном *контексте декоратора*, и мы можем определить нужный контекст через свойство `kind` (паттерн, который мы рассматривали в рецепте 3.2). Так что нет ничего проще, чем написать функцию `log`, которая выполняет разные вызовы декоратора в зависимости от контекста, верно?

И да и нет. Конечно, лучше всего иметь функцию-обертку, которая правильно ветвится, но определения типов, как мы уже видели, довольно сложны. Найти *одну* сигнатуру функции, которая смогла бы обрабатывать их все, практически невозможно, не устанавливая везде значения по умолчанию `any`. И помните: нужны правильные типы сигнатур функций; в противном случае декораторы не будут работать с членами класса.

Множество различных сигнатур функций приводят к *перегрузкам функций*. Поэтому вместо того, чтобы искать одну сигнатуру функции для всех возможных декораторов, мы создаем перегрузки для *декораторов полей*, *декораторов методов* и т. д. Мы можем вводить их так же, как вводили бы отдельные декораторы. Сигнатура функции для реализации принимает значение `any` и объединяет все необходимые типы контекста декоратора, чтобы впоследствии мы могли выполнить надлежащую проверку на распознавание типов:

```

function log<This, Args extends any[], Return>(
  value: (this: This, ...args: Args) => Return,
  context: ClassMethodDecoratorContext
): (this: This, ...args: Args) => Return;
function log<Val>(
  value: Val,
  context: ClassFieldDecoratorContext
): FieldDecoratorFn;

```

```
function log(
  value: any,
  context: ClassMethodDecoratorContext | ClassFieldDecoratorContext
) {
  if (context.kind === "method") {
    return logMethod(value, context);
  } else {
    return logField(value, context);
  }
}
```

Вместо того чтобы иметь дело со всем реальным кодом в ветвях `if`, лучше вызывать исходные методы. Если вы не хотите, чтобы ваши функции `logMethod` или `logField` были доступны, то можете поместить их в модуль и экспортировать только `log`.



Существует множество различных типов декораторов, и все они имеют поля, которые немного отличаются друг от друга. Определения типов в `lib.decorators.d.ts` превосходны, но если вам нужно немного больше информации, то ознакомьтесь с оригинальным предложением по декораторам в TC39 (<https://oreil.ly/76JuE>). Оно содержит не только подробные сведения обо всех типах декораторов, но и определения типов TypeScript, которые дополняют картину.

И последнее, что мы хотим сделать, — адаптировать `logMethod` для ведения журнала как *до*, так и *после* вызова. Для обычных методов это такое же простое действие, как временное сохранение возвращаемого значения:

```
function log<This, Args extends any[], Return>(
  value: (this: This, ...args: Args) => Return,
  context: ClassMethodDecoratorContext
) {
  return function (this: This, ...args: Args) {
    console.log(`calling ${context.name.toString()}`);
    const val = value.call(this, ...args);
    console.log(`called ${context.name.toString()}: ${val}`);
    return val;
  };
}
```

Но в случае асинхронных методов все немного интереснее. Вызов асинхронного метода приводит к получению `Promise`. Сам `Promise` может быть уже выполнен или отложен на потом. Это означает, что если мы будем придерживаться предыдущей реализации, то *вызванное* сообщение журнала может появиться до того, как метод действительно выдаст значение.

В качестве обходного решения нужно выстроить цепочку сообщений журнала после того, как `Promise` выдаст результат. Для этого следует проверить, действительно ли метод является `Promise`. Промисы JavaScript интересны вот чем: все, что

от них нужно ожидать, — наличие метода `then`. Проверить это можно с помощью вспомогательного метода:

```
function isPromise(val: any): val is Promise<unknown> {
  return (
    typeof val === "object" &&
    val &&
    "then" in val &&
    typeof val.then === "function"
  );
}
```

И после этого мы решаем, вводить журналирование в систему напрямую или отложить, в зависимости от того, есть ли у нас `Promise`:

```
function logMethod<This, Args extends any[], Return>(
  value: (this: This, ...args: Args) => Return,
  context: ClassMethodDecoratorContext
): (this: This, ...args: Args) => Return {
  return function (this: This, ...args: Args) {
    console.log(`calling ${context.name.toString()}`);
    const val = value.call(this, ...args);
    if (isPromise(val)) {
      val.then((p: unknown) => {
        console.log(`called ${context.name.toString()}: ${p}`);
        return p;
      });
    } else {
      console.log(`called ${context.name.toString()}: ${val}`);
    }

    return val;
  };
}
```

Декораторы могут быть очень сложными, но в конечном счете это полезный инструмент, позволяющий модифицировать классы в JavaScript и TypeScript различными способами.

ГЛАВА 12

Стратегии разработки типов

Все рецепты до сих пор были посвящены конкретным аспектам языка программирования TypeScript и его системы типов. В главах 2 и 3 вы узнали об эффективном использовании базовых типов, в главе 4 — о том, как с помощью обобщенных типов сделать код более пригодным для повторного использования, в главе 5 — о том, как, используя условные типы, создать расширенные типы для очень сложных ситуаций, в главе 6 — о литеральных типах строковых шаблонов, а в главе 7 — о вариативных типах кортежей.

В главе 8 мы создали коллекцию вспомогательных типов, а в главе 9 обсудили ограничения стандартной библиотеки. В главе 10 вы узнали о способах работы с JSX как с языковым расширением, а в главе 11 — как и когда использовать классы. Во всех рецептах подробно рассматривались плюсы и минусы каждого подхода, что позволит вам принимать правильные решения в любой ситуации, создавать более совершенные типы, улучшать надежность программ и делать процесс разработки стабильным.

Это уже немало! Однако по-прежнему не хватает последнего элемента, который собирает всю полученную информацию воедино: как подходить к решению задач нового типа? С чего начать? На что нужно обратить внимание?

Ответы на эти вопросы и составляют содержание данной главы. Здесь вы узнаете о концепции *простых в сопровождении типов* (low maintenance types). Мы рассмотрим процесс, который позволит вам начать с простых типов и постепенно совершенствовать их. Вы узнаете о секретных возможностях интерактивной среды TypeScript Playground (<https://www.typescriptlang.org/play>) и о том, как работать с библиотеками, облегчающими проверку. Вы найдете руководства, которые помогут вам принимать трудные решения, и узнаете о том, как устранять наиболее частые, но труднопреодолимые ошибки несоответствия типов, которые обязательно встретятся вам при работе с TypeScript.

Предыдущая часть книги помогла вам пройти путь от новичка до подмастерья. А следующие рецепты позволят вам стать экспертом. Добро пожаловать в последнюю главу!

12.1. Написание простых в сопровождении типов

Задача

Каждый раз, когда меняется модель, вам приходится изменять десятки типов по всей кодовой базе. Это утомительно, и к тому же легко что-то упустить.

Решение

Выводите типы из других, уже используемых, и создавайте простые в сопровождении типы.

Обсуждение

На протяжении всей книги мы тратили много времени, создавая типы на основе других типов. Когда мы можем вывести тип из чего-то существующего, это означает, что мы тратим меньше времени на написание типа и адаптацию информации о нем и больше времени уделяем исправлению ошибок в JavaScript.

TypeScript — это слой метаданных поверх JavaScript. Наша цель — по-прежнему писать JavaScript, но сделать его максимально надежным и простым: инструментарий помогает вам оставаться продуктивным и не мешает работать.

Именно так я обычно пишу код TypeScript: пишу обычный код JavaScript, а там, где TypeScript нуждается в дополнительной информации, добавляю несколько аннотаций. Одно условие: я не хочу заниматься сопровождением типов. Я предпочитаю создавать типы, которые могут обновляться сами при изменении их зависимостей или окружения. Я называю такой подход *созданием простых в сопровождении типов*.

Он состоит из трех частей:

- 1) смоделируйте свои данные или выводите данные из существующих моделей;
- 2) определите производные (сопоставленные типы, частичные элементы и т. д.);
- 3) определите поведение с помощью условных типов.

Рассмотрим краткую и неполную функцию `copy`. Я хочу скопировать файлы из одного каталога в другой. Чтобы облегчить себе жизнь, я создал набор параметров по умолчанию, чтобы мне не приходилось слишком часто повторять одни и те же действия:

```
const defaultOptions = {
  from: "./src",
  to: "./dest",
};
```

```
function copy(options) {  
  // Давайте объединим параметры и параметры по умолчанию  
  const allOptions = { ...defaultOptions, ...options};  
  
  // todo: реализация остального  
}
```

Это паттерн, который вы часто можете встретить в JavaScript. Сразу бросается в глаза, что в TypeScript отсутствует *некоторая* информация о типе. В частности, в данный момент параметр `options` функции `copy` имеет тип `any`. Так что добавим тип для него!

Я могу создавать типы явным образом:

```
type Options = {  
  from: string;  
  to: string;  
};  
  
const defaultOptions: Options = {  
  from: "./src",  
  to: "./dest",  
};  
  
type PartialOptions = {  
  from?: string;  
  to?: string;  
};  
  
function copy(options: PartialOptions) {  
  // Давайте объединим параметры и параметры по умолчанию  
  const allOptions = { ...defaultOptions, ...options};  
  
  // todo: реализация остального  
}
```

Это разумный подход. Вы думаете о типах, потом назначаете их, а затем получаете привычные для вас обратную связь от редактора и проверку типов. А если что-то изменится? Предположим, мы добавим еще одно поле в `Options`; нам пришлось бы адаптировать наш код трижды:

```
type Options = {  
  from: string;  
  to: string;  
  overwrite: boolean; // добавлено  
};  
  
const defaultOptions: Options = {  
  from: "./src",
```

```

    to: "./dest",
    overwrite: true, // добавлено
  };

type PartialOptions = {
  from?: string;
  to?: string;
  overwrite?: boolean; // добавлено
};

```

Но зачем? Информация уже есть! В `defaultOptions` мы сообщаем TypeScript, что именно нам нужно. Проведем оптимизацию.

1. Удалим тип `PartialOptions` и используем служебный тип `Partial<T>`, чтобы получить тот же эффект. Вы, наверное, уже догадались об этом.
2. Используем оператор `typeof` в TypeScript, чтобы создать новый тип прямо в процессе работы:

```

const defaultOptions = {
  from: "./src",
  to: "./dest",
  overwrite: true,
};

function copy(options: Partial<typeof defaultOptions>) {
  // Давайте объединим параметры и параметры по умолчанию
  const allOptions = { ...defaultOptions, ...options };

  // todo: реализация остального
}

```

Вот и все. Просто укажите, где нужно сообщить TypeScript, что вы ищете.

- Если мы добавим новые поля, то нам вообще не придется ничего сопровождать.
- Если мы переименуем поле, то получим *только* интересующую нас информацию: все сценарии использования `copy`, когда нам приходится менять параметры, которые мы передаем функции.
- У нас есть единственный источник истины: реальный объект `defaultOptions`. Это объект, который имеет значение, поскольку это единственная информация, которая у нас есть во время выполнения.

И код становится более лаконичным, а TypeScript — менее назойливым и более соответствующим тому, как мы пишем JavaScript.

Другой пример, уже хорошо известный вам: магазин игрушек, который появился в рецепте 3.1 и с которым мы работали в рецептах 4.5 и 5.3. Пересмотрите все три элемента и подумайте, как изменить только модель, чтобы обновить все остальные типы.

12.2. Поэтапное уточнение типов

Задача

Вашему API нужны сложные типы, использующие такие расширенные возможности, как условные и литеральные типы строковых шаблонов. Вы не знаете, с чего начать.

Решение

Совершенствуйте свои типы поэтапно. Начните с базовых примитивных и объектных типов, подмножеств, добавьте обобщенные типы, а затем переходите к более сложным типам. Процесс, описанный в этом уроке, поможет вам в создании типов. Кроме того, это хороший способ подвести итог всему, чему вы научились.

Обсуждение

Взгляните на следующий пример:

```
app.get("/api/users/:userID", function (req, res) {
  if (req.method === "POST") {
    res.status(20).send({
      message: "Got you, user " + req.params.userID,
    });
  }
});
```

У нас есть сервер типа Express (<https://expressjs.com/>), который позволяет определить маршрут (или путь) и выполняет обратный вызов при запросе URL.

Обратный вызов принимает два аргумента.

- *Объект запроса (request)*. Здесь мы получаем информацию об используемом методе HTTP (<https://oreil.ly/zcoUS>) — например, GET, POST, PUT, DELETE — и дополнительных параметрах, которые вводятся. В этом примере `userID` должен быть сопоставлен с параметром `userID`, который, по сути, содержит идентификатор пользователя!
- *Объект ответа (response или reply)*. Здесь мы хотим подготовить правильный ответ от сервера клиенту. Нам нужно отправить правильные коды состояния (метод `status`) и выходные данные в формате JSON по сети.

Код, представленный в этом примере, сильно упрощен, но дает хорошее представление о наших действиях. Кроме того, предыдущий пример полон ошибок! Взгляните:


```
app.get("/api/users/:userID", function (req, res) {
  if (req.method === "POST") { /* Error 1 */
    res.status(20).send({ /* Error 2 */
      message: "Welcome, user " + req.params.userID /* Error 3 */,
    });
  }
});
```

Три строчки кода реализации и три ошибки? Что произошло?

1. Первая ошибка связана с нюансами. Мы сообщаем приложению, что хотим прослушивать запросы GET (отсюда `app.get`), но делаем что-то только в том случае, если метод запроса — POST. В данный момент в приложении метод `req.method` не может быть POST. Таким образом, мы никогда не отправим никакой ответ, что может привести к непредвиденным задержкам.
2. Мы явно отправляем код состояния, и это прекрасно! Однако `20` не является допустимым кодом состояния. Клиенты могут не понять, что здесь происходит.
3. Это ответ, который мы хотим отправить обратно. Мы получили доступ к проанализированным аргументам, но обнаружили опечатку. Это `userID`, а не `userId`. Все пользователи получили бы приветствие со словами «Добро пожаловать, пользователь undefined!» Вы точно видели нечто подобное.

Решение подобных проблем — главная цель TypeScript. Он хочет понять ваш код JavaScript лучше, чем вы сами. И там, где TypeScript не может понять, что вы имеете в виду, вы можете ему помочь, предоставив дополнительную информацию о типах. Проблема в том, что зачастую бывает сложно начать добавлять типы. Возможно, вы размышляете о самых загадочных крайних случаях, но не знаете, как к ним подступиться.

Я хочу предложить процесс, который может помочь вам начать работу, а также покажет, на чем стоит остановиться. Вы можете поэтапно повышать эффективность своих типов. Каждое усовершенствование улучшает ситуацию, и вы можете повышать безопасность типов в течение длительного периода времени.

Этап 1. Базовая типизация

Мы начнем с некой базовой информации о типе. У нас есть объект `app`, который указывает на функцию `get`. Функция `get` принимает параметр `path`, который является строкой, и функцию обратного вызова:

```
const app = {
  get /* post, put, delete, ... to come! */,
};

function get(path: string, callback: CallbackFn) {
  // будет реализовано --> сейчас не важно
}
```

`CallbackFn` — это тип функции, которая имеет тип возвращаемого значения `void` и принимает два аргумента:

- `req`, который имеет тип `ServerRequest`;
- `reply`, который имеет тип `ServerReply`.

```
type CallbackFn = (req: ServerRequest, reply: ServerReply) => void;
```

`ServerRequest` — довольно сложный объект в большинстве фреймворков. Мы делаем упрощенную версию в целях демонстрации. Мы передаем строку `method` для HTTP-метода "GET", "POST", "PUT", "DELETE" и т. д. и запись `params`. Записи — это объекты, которые связывают набор ключей с набором свойств. На данный момент мы хотим разрешить сопоставление каждого строкового ключа со строковым свойством. Мы проведем рефакторинг этого объекта позже:

```
type ServerRequest = {
  method: string;
  params: Record<string, string>;
};
```

Для `ServerReply` мы предоставляем несколько функций, зная, что у реального объекта `ServerReply` их гораздо больше. Функция `send` принимает необязательный аргумент `obj` с данными, которые мы хотим отправить. У нас есть возможность задать код состояния с помощью функции `status`, используя текущий интерфейс (fluent interface)¹:

```
type ServerReply = {
  send: (obj?: any) => void;
  status: (statusCode: number) => ServerReply;
};
```

Используя несколько базовых составных типов и простой примитивный тип для путей, мы уже значительно повысили безопасность типов в нашем проекте. Мы можем исключить пару ошибок:

```
app.get("/api/users/:userID", function(req, res) {
  if(req.method === 2) {
    // ^ This condition will always return 'false' since the types
    //   'string' and 'number' have no overlap.(2367)

    res.status("200").send()
    //   ^
    // Argument of type 'string' is not assignable to
    // parameter of type 'number'.(2345)
  }
});
```

Это замечательно, но нам предстоит еще многое. Мы по-прежнему можем отправлять неправильные коды статуса (возможно любое число) и не иметь представ-

¹ Текущие интерфейсы позволяют создавать цепочки операций, возвращая экземпляр при каждом вызове метода.

ления о возможных методах HTTP (возможна любая строка). Поэтому уточним наши типы.

Этап 2. Подмножество примитивных типов

Вы можете рассматривать примитивные типы как набор всех возможных значений определенной категории. Например, `string` содержит все возможные строки, которые могут быть выражены в JavaScript, `number` — все возможные числа с плавающей точкой с двойной точностью, а `boolean` — все возможные логические значения, которые являются `true` и `false`.

TypeScript позволяет преобразовать эти наборы в более мелкие подмножества. Например, мы можем создать тип `Methods`, который будет содержать все возможные строки, которые мы можем получить для HTTP-методов:

```
type Methods = "GET" | "POST" | "PUT" | "DELETE";

type ServerRequest = {
  method: Methods;
  params: Record<string, string>;
};
```

`Methods` — это меньший набор из большего набора `string`. Кроме того, `Methods` — это еще и тип объединения литеральных типов, наименьшая единица из данного набора. Литеральная строка. Литеральное число. Здесь нет никакой двусмысленности: это просто "GET". Вы объединяете эти литералы с другими литеральными типами, создавая подмножество любых более крупных типов, которые у вас есть. К тому же вы можете создать подмножество с литеральными типами как `string`, так и `number`, или с различными типами составных объектов. Существует множество возможностей комбинировать и объединять литеральные типы.

Это немедленно повлияет на обратный вызов нашего сервера. Внезапно мы можем различать эти четыре метода (или больше, если необходимо) и использовать все возможности кода. TypeScript будет помогать нам.

Еще на одну категорию ошибок стало меньше. Теперь мы точно знаем, какие возможные HTTP-методы доступны. То же самое мы можем сделать и для кодов состояния HTTP, определив подмножество допустимых чисел, которые может принимать `statusCode`:

```
type StatusCode =
  100 | 101 | 102 | 200 | 201 | 202 | 203 | 204 | 205 |
  206 | 207 | 208 | 226 | 300 | 301 | 302 | 303 | 304 |
  305 | 306 | 307 | 308 | 400 | 401 | 402 | 403 | 404 |
  405 | 406 | 407 | 408 | 409 | 410 | 411 | 412 | 413 |
  414 | 415 | 416 | 417 | 418 | 420 | 422 | 423 | 424 |
  425 | 426 | 428 | 429 | 431 | 444 | 449 | 450 | 451 |
  499 | 500 | 501 | 502 | 503 | 504 | 505 | 506 | 507 |
  508 | 509 | 510 | 511 | 598 | 599;
```

```
type ServerReply = {
  send: (obj?: any) => void;
  status: (statusCode: StatusCode) => ServerReply;
};
```

Тип `StatusCode` снова является типом объединения. И таким образом мы исключаем еще одну категорию ошибок. Внезапно код, подобный этому, дает сбой:

```
app.get("/api/user/:userID", (req, res) => {
  if(req.method === "POS") {
    // ^ This condition will always return 'false' since
    // the types 'Methods' and '"POS"' have no overlap.(2367)
    res.status(20)
    // ^
    // Argument of type '20' is not assignable to parameter of
    // type 'StatusCode'.(2345)
  }
})
```

И наше программное обеспечение становится намного безопаснее. Но мы можем пойти дальше!

Этап 3. Добавление обобщенных типов

Когда мы определяем маршрут с помощью `app.get`, то неявно знаем, что единственным возможным HTTP-методом является "GET". Но, учитывая наши типы определения, нам все равно приходится проверять все возможные части объединения.

Тип для `CallbackFn` правильный, так как мы могли бы определить функции обратного вызова для всех возможных HTTP-методов. Но если мы явно вызываем `app.get`, то было бы неплохо обойтись без нескольких дополнительных этапов, которые необходимы только для соблюдения типизации.

В этом могут помочь обобщенные типы TypeScript. Мы хотим определить `ServerRequest` таким образом, чтобы можно было указать часть `Methods` вместо всего набора. Для этого мы используем обобщенный синтаксис, в котором можно определять параметры так же, как это делается в функциях:

```
type ServerRequest<Met extends Methods> = {
  method: Met;
  params: Record<string, string>;
};
```

В этом коде происходит следующее:

- `ServerRequest` становится обобщенным типом, на что указывают угловые скобки;
- мы определяем обобщенный параметр `Met`, который является подмножеством типа `Methods`;

- мы используем этот параметр в качестве универсальной переменной для определения метода.

Благодаря этому изменению мы можем указывать различные варианты `ServerRequest`, не используя дублирования:

```
type OnlyGET = ServerRequest<"GET">;
type OnlyPOST = ServerRequest<"POST">;
type POSTorPUT = ServerRequest<"POST" | "PUT">;
```

Мы изменили интерфейс `ServerRequest`, поэтому нам придется изменить и все остальные типы, которые используют `ServerRequest`, такие как `CallbackFn` и функция `get`:

```
type CallbackFn<Met extends Methods> = (
  req: ServerRequest<Met>,
  reply: ServerReply
) => void;

function get(path: string, callback: CallbackFn<"GET">) {
  // будет реализовано
}
```

С помощью функции `get` мы передаем фактический аргумент нашему обобщенному типу. Нам известно, что это будет не просто подмножество `Methods`; мы точно знаем, с каким подмножеством мы имеем дело.

Теперь, когда мы используем `app.get`, у нас есть только одно возможное значение для `req.method`:

```
app.get("/api/users/:userID", function (req, res) {
  req.method; // может быть только GET
});
```

Этот подход гарантирует, что при создании обратного вызова `app.get` мы не будем предполагать, что доступны такие HTTP-методы, как "POST" или подобные ему. На данный момент мы точно знаем, с чем имеем дело, поэтому отразим это в типах.

Мы уже сделали многое, желая убедиться, что `request.method` достаточно типизирован и представляет реальное положение дел. Одним из приятных преимуществ подмножества типа объединения `Methods` является то, что мы можем создать универсальную функцию обратного вызова *вне* `app.get`, которая будет типобезопасной:

```
const handler: CallbackFn<"PUT" | "POST"> = function(res, req) {
  res.method // может быть "POST" или "PUT"
};

const handlerForAllMethods: CallbackFn<Methods> = function(res, req) {
  res.method // могут быть все методы
};
```

```
app.get("/api", handler);
//           ^
// Argument of type 'CallbackFn<"POST" | "PUT">' is not
// assignable to parameter of type 'CallbackFn<"GET">'.

app.get("/api", handlerForAllMethods); // Это работает
```

Этап 4. Расширенные типы для проверки типов

Чего мы еще не касались, так это ввода объекта `params`. На данный момент мы получаем запись, которая позволяет получить доступ к каждому ключу типа `string`. Теперь наша задача — сделать это действие более конкретным!

Для этого мы добавляем еще одну обобщенную переменную, одну для методов и одну для возможных ключей в нашей `Record`:

```
type ServerRequest<Met extends Methods, Par extends string = string> = {
  method: Met;
  params: Record<Par, string>;
};
```

Переменная обобщенного типа `Par` может быть подмножеством типа `string`, а значением по умолчанию является любая строка. Благодаря этому мы можем сообщить `ServerRequest`, какие ключи ожидаем:

```
// request.method = "GET"
// request.params = {
//   userID: string
// }
type WithUserID = ServerRequest<"GET", "userID">;
```

Добавим новый аргумент в нашу функцию `get` и тип `CallbackFn`, чтобы можно было установить запрашиваемые параметры:

```
function get<Par extends string = string>(
  path: string,
  callback: CallbackFn<"GET", Par>
) {
  // будет реализовано
}

const app = {
  get /* post, put, delete, ... to come! */,
};

type CallbackFn<Met extends Methods, Par extends string> = (
  req: ServerRequest<Met, Par>,
  reply: ServerReply
) => void;
```

Если мы не задаем `Par` явно, то тип будет работать привычным для нас образом, поскольку по умолчанию `Par` имеет значение `string`. Но если мы установим его, то у нас сразу же появится правильное определение для объекта `req.params`:

```
app.get<"userID">("/api/users/:userID", function (req, res) {
  req.params.userID; // Работает!!
  req.params.anythingElse; // Не работает!
});
```

Это замечательно! Однако можно улучшить одну маленькую деталь. Мы по-прежнему можем передавать *любую* строку в аргументе `path` в `app.get`. Разве не было бы лучше, если бы мы могли отразить `Par` и там? Мы можем! Именно здесь мы можем использовать *типы литералов шаблонов строк* (см. главу 6).

Создадим тип с именем `IncludesRouteParams`, чтобы убедиться, что `Par` правильно включен в Express-стиль добавления двоеточия перед именем параметра:

```
type IncludesRouteParams<Par extends string> =
  | `${string}/${Par}`
  | `${string}/${Par}/${string}`;
```

Обобщенный тип `IncludesRouteParams` принимает один аргумент, который является подмножеством `string`. Он создает тип объединения из двух литералов шаблона.

- Первый литерал шаблона начинается с *любой строки*, затем содержит символ `/`, за которым следует символ `:`, а за ним — имя параметра. Такая схема гарантирует, что мы перехватим все случаи, когда параметр находится в конце строки маршрута.
- Второй литерал шаблона начинается с *любой строки*, за которой следует тот же шаблон из `/`, `:` и имени параметра. Затем идет еще один символ `/`, а за ним — *любая строка*. Эта ветвь типа объединения гарантирует, что мы перехватим все случаи, когда параметр находится где-то внутри маршрута.

Вот как `IncludesRouteParams` с именем параметра `userID` ведет себя в разных тестовых примерах:

```
const a: IncludesRouteParams<"userID"> = "/api/user/:userID"; // работает
const b: IncludesRouteParams<"userID"> = "/api/user/:userID/orders"; // работает
const c: IncludesRouteParams<"userID"> = "/api/user/:userId"; // ломает код
const d: IncludesRouteParams<"userID"> = "/api/user"; // ломает код
const e: IncludesRouteParams<"userID"> = "/api/user/:userIDAndmore"; // ломает код
```

Добавим наш новый вспомогательный тип в объявление функции `get`:

```
function get<Par extends string = string>(
  path: IncludesRouteParams<Par>,
  callback: CallbackFn<"GET", Par>
) {
  // будет реализовано
}
```

```
app.get<"userID">(
  "/api/users/:userID",
  function (req, res) {
    req.params.userID; // Да!
  }
);
```

Отлично! Мы получаем еще один механизм безопасности, который гарантирует, что мы не упустим возможности добавить параметры в фактический маршрут.

Этап 5. Блокировка литеральных типов

Но я все еще недоволен. Как только маршруты усложняются, становится очевидно, что этот подход чреват проблемами.

- Первая проблема состоит в том, что нам нужно явно указывать параметры в параметре обобщенного типа. Мы должны привязать `Par` к `"userID"`, хотя все равно указали бы его в аргументе `path` функции. Это не похоже на JavaScript!
- Такой подход позволяет работать только с одним параметром маршрута. В тот момент, когда мы добавляем тип объединения, например `"userID" | "orderId"`, проверка на отказоустойчивость выполняется только при наличии *одного* из этих аргументов. Именно так работают наборы. Это может быть один или другой аргумент.

Должен быть более удачный способ. И он есть. Иначе этот рецепт закончился бы на очень горькой ноте.

Изменим порядок! Вместо того чтобы определять параметры маршрута в переменной обобщенного типа, мы извлекаем переменные из `path`, передаваемого в качестве первого аргумента `app.get`:

```
function get<Path extends string = string>(
  path: Path,
  callback: CallbackFn<"GET", ParseRouteParams<Path>>
) {
  // будет реализовано
}
```

Мы удаляем обобщенный тип `Par` и добавляем `Path`, который может быть подмножеством любого типа `string`. Устанавливая `path` в этот обобщенный тип `Path`, в момент передачи параметра в `get` мы перехватываем его строковый литеральный тип. Мы передаем `Path` новому обобщенному типу `ParseRouteParams`, который еще не создали.

Поработаем над `ParseRouteParams`. Здесь мы снова меняем порядок событий. Мы могли бы передавать запрошенные параметры маршрута в обобщенный тип, чтобы убедиться, что с путем все в порядке. Но вместо этого мы передаем путь маршрута и извлекаем возможные параметры маршрута. Для этого нам нужно создать условный тип.

Этап 6. Добавление условных типов

Условные типы синтаксически аналогичны тернарному оператору в JavaScript. Вы проверяете условие, и если оно выполняется, то возвращаете ветвь А, в противном случае возвращаете ветвь Б. Например:

```
type ParseRouteParams<Route> =
  Route extends `${string}/${infer P}`
    ? P
    : never;
```

Здесь мы проверяем, является ли `Route` подмножеством каждого пути, который заканчивается параметром в конце Express-стиля (с предшествующим `"/:"`). Если да, то мы извлекаем эту строку, то есть записываем ее содержимое в новую переменную. Если условие выполнено, то мы возвращаем только что извлеченную строку; в противном случае возвращаем `never`, как при отсутствии параметров маршрута.

Если мы попробуем выполнить данные действия, то получим что-то вроде этого:

```
type Params = ParseRouteParams<"/api/user/:userID">; // Params – это "userID"
```

```
type NoParams = ParseRouteParams<"/api/user">; // NoParams – never: нет параметров!
```

Это уже намного лучше, чем то, что мы делали раньше. Теперь мы хотим перехватить все остальные возможные параметры. Для этого нужно добавить еще одно условие:

```
type ParseRouteParams<Route> = Route extends `${string}/${infer P}/${infer R}`
  ? P | ParseRouteParams<`${R}`>
  : Route extends `${string}/${infer P}`
    ? P
    : never;
```

Теперь наш условный тип работает следующим образом.

1. В первом условии мы проверяем, есть ли где-то в середине параметр маршрута. Если да, то мы извлекаем и параметр маршрута, и все остальное, что следует после него. Мы возвращаем найденный параметр маршрута `P` в типе объединения, где рекурсивно вызываем тот же обобщенный тип, что и для остальных `R`. Например, если мы передаем в `ParseRouteParams` маршрут `"/api/users/:userID/orders/:orderID"`, то выводим `"userID"` в `P` и `"orders/:orderID"` в `R`. Мы вызываем тот же тип с помощью `R`.
2. Здесь вступает в силу второе условие. Мы проверяем, есть ли тип в конце. Так происходит в случае с `"orders/:orderID"`. Мы извлекаем `"orderID"` и возвращаем этот литеральный тип.
3. Если параметров маршрута больше не осталось, то мы возвращаем `never`:

```
// Params – это "userID"
type Params = ParseRouteParams<"/api/user/:userID">;
```

```
// MoreParams – это "userID" | "orderID"
type MoreParams = ParseRouteParams<"/api/user/:userID/orders/:orderID">;
```

Применим этот новый тип и посмотрим, как будет выглядеть наше окончательное использование `app.get`:

```
app.get("/api/users/:userID/orders/:orderId", function (req, res) {
  req.params.userID; // Работает
  req.params.orderID; // Также доступно
});
```

Вот и все! Подведем итоги. Типы, которые мы только что создали для одной функции `app.get`, позволяют исключить множество возможных ошибок:

- мы можем передавать в `res.status()` только правильные числовые коды состояния;
- метод `req.method` — одна из четырех возможных строк, и, используя `app.get`, мы знаем, что это может быть только "GET";
- мы можем проанализировать параметры маршрута и убедиться, что в параметрах обратного вызова нет опечаток.

Если мы посмотрим на пример, приведенный в начале этого рецепта, то получим следующие сообщения об ошибках:

```
app.get("/api/users/:userID", function(req, res) {
  if (req.method === "POST") {
//   ^ This condition will always return 'false' since
//   the types 'Methods' and '"POST"' have no overlap.(2367)
    res.status(20).send({
//           ^
// Argument of type '20' is not assignable to parameter of
// type 'StatusCode'.(2345)
      message: "Welcome, user " + req.params.userID
//                                     ^
//   Property 'userID' does not exist on type
//   '{ userID: string; }'. Did you mean 'userID'?
    });
  }
});
```

И все это еще до запуска кода! Серверы в стиле Express — прекрасный пример динамической природы JavaScript. В зависимости от метода, который вы вызываете, и строки, которую передаете в качестве первого аргумента, внутри обратного вызова многое меняется. Возьмите другой пример — и все ваши типы будут выглядеть совершенно по-другому.

Самое замечательное в этом подходе то, что каждый этап повышал безопасность типа.

1. Вы можете ограничиться базовыми типами и получить от этого больше пользы, чем от полного отсутствия типов.

2. Подмножество помогает избавиться от опечаток за счет уменьшения количества допустимых значений.
3. Обобщенные типы помогают адаптировать поведение к конкретному сценарию использования.
4. Расширенные типы, такие как литеральные типы строковых шаблонов, делают ваше приложение более естественным в мире строгой типизации.
5. Блокировка обобщенных типов позволяет работать с литералами в JavaScript и рассматривать их как типы.
6. Условные типы делают ваши типы такими же гибкими, как и ваш код JavaScript.

Что лучше всего? После того как вы добавите свои типы, пользователи будут писать обычный код JavaScript и по-прежнему получать всю информацию о типах. Это выгодно для всех.

12.3. Проверка контрактов с помощью `satisfies`

Задача

Вы хотите работать с литеральными типами, но вам нужна проверка типа аннотации, позволяющая убедиться, что вы выполняете условия контракта.

Решение

Используйте оператор `satisfies`, чтобы выполнить проверку типов, подобную аннотации, сохраняя при этом литеральные типы.

Обсуждение

Сопоставленные типы великолепны, поскольку обеспечивают гибкость объектных структур, которой славится JavaScript. Но их использование чревато некоторыми важными последствиями для системы типов. Возьмем следующий пример из универсальной библиотеки обмена сообщениями, которая принимает «определение канала», где может быть определено несколько токенов канала:

```
type Messages =  
  | "CHANNEL_OPEN"  
  | "CHANNEL_CLOSE"  
  | "CHANNEL_FAIL "  
  | "MESSAGE_CHANNEL_OPEN"  
  | "MESSAGE_CHANNEL_CLOSE"  
  | "MESSAGE_CHANNEL_FAIL";
```

```
type ChannelDefinition = {
  [key: string]: {
    open: Messages;
    close: Messages;
    fail: Messages;
  };
};
```

Ключи из этого объекта определения канала являются такими, какими их хочет видеть пользователь. Следовательно, это определение канала допустимо:

```
const impl: ChannelDefinition = {
  test: {
    open: 'CHANNEL_OPEN',
    close: 'CHANNEL_CLOSE',
    fail: 'CHANNEL_FAIL'
  },
  message: {
    open: 'MESSAGE_CHANNEL_OPEN',
    close: 'MESSAGE_CHANNEL_CLOSE',
    fail: 'MESSAGE_CHANNEL_FAIL'
  }
}
```

Однако возникает проблема, когда мы хотим получить доступ к ключам, которые определили так гибко. Допустим, у нас есть функция, открывающая канал. Мы передаем весь объект определения канала целиком, а также канал, который хотим открыть:

```
function openChannel(
  def: ChannelDefinition,
  channel: keyof ChannelDefinition
) {
  // будет реализовано
}
```

Так что же представляют собой ключи `ChannelDefinition`? По сути, это каждый ключ: `[key: string]`. Поэтому в тот момент, когда мы назначаем определенный тип, TypeScript обрабатывает `impl` как этот конкретный тип, игнорируя фактическую реализацию. Контракт выполнен. Двигаемся дальше. Этот код позволяет передавать неправильные ключи:

```
// Проходит, хотя "message" не является частью impl
openChannel(impl, "message");
```

Поэтому нас больше интересует фактическая реализация, а не тип, который мы присваиваем константе. Это означает, что мы должны избавиться от типа `ChannelDefinition` и убедиться, что мы заботимся о фактическом типе.

Начнем с того, что функция `openChannel` должна принимать любой объект, который является подтипом `ChannelDefinition`, но работать с конкретным подтипом:

```
function openChannel<
  T extends ChannelDefinition
>(def: T, channel: keyof T) {
  // будет реализовано
}
```

TypeScript теперь работает на двух уровнях.

- Он проверяет, действительно ли `T` расширяет `ChannelDefinition`. Если да, то мы работаем с типом `T`.
- Все параметры нашей функции типизированы с помощью обобщенного типа `T`. Это также означает, что мы получаем *реальные* ключи `T` с помощью `keyof T`.

Чтобы извлечь из этого пользу, мы должны избавиться от определения типа для `impl`. Явное определение типа переопределяет все фактические типы. С того момента, как мы явно указываем тип, TypeScript рассматривает его как `ChannelDefinition`, а не как фактический базовый подтип. Кроме того, нам нужно установить контекст `const`, чтобы мы могли преобразовывать все строки к их единому типу (и таким образом обеспечить совместимость с `Messages`):

```
const impl = {
  test: {
    open: "CHANNEL_OPEN",
    close: "CHANNEL_CLOSE",
    fail: "CHANNEL_FAIL",
  },
  message: {
    open: "MESSAGE_CHANNEL_OPEN",
    close: "MESSAGE_CHANNEL_CLOSE",
    fail: "MESSAGE_CHANNEL_FAIL",
  },
} as const;
```

Без контекста `const` выводимый тип `impl` выглядит так:

```
/// typeof impl
{
  test: {
    open: string;
    close: string;
    fail: string;
  };
};
```

```

message: {
  open: string;
  close: string;
  fail: string;
};
}

```

С контекстом `const` фактический тип `impl` выглядит так:

```

// type of impl
{
  test: {
    readonly open: "CHANNEL_OPEN";
    readonly close: "CHANNEL_CLOSE";
    readonly fail: "CHANNEL_FAIL";
  };
  message: {
    readonly open: "MESSAGE_CHANNEL_OPEN";
    readonly close: "MESSAGE_CHANNEL_CLOSE";
    readonly fail: "MESSAGE_CHANNEL_FAIL";
  };
}

```

Контекст `const` позволяет выполнять условия контракта, заключенного с помощью `ChannelDefinition`. Теперь `openChannel` работает корректно:

```

openChannel(impl, "message"); // удовлетворяет контракт
openChannel(impl, "message");
//           ^
// Argument of type '"message"' is not assignable to parameter
// of type '"test" | "message"'.(2345)

```

Этот код работает, но есть один нюанс. Единственный момент, когда мы можем проверить, действительно ли `impl` является допустимым подтипом `ChannelDefinition`, — это когда мы его используем. Иногда мы хотим заранее сделать аннотацию, чтобы выявить потенциальные нарушения в нашем контракте. Мы хотим проверить, *удовлетворяет* ли эта конкретная реализация условиям контракта.

К счастью, для этого есть ключевое слово. Мы можем определить объекты и выполнить проверку типа, чтобы убедиться, что эта реализация удовлетворяет типу, но TypeScript будет рассматривать ее как литеральный тип:

```

const impl = {
  test: {
    open: "CHANNEL_OPEN",
    close: "CHANNEL_CLOSE",
    fail: "CHANNEL_FAIL",
  },
  message: {
    open: "MESSAGE_CHANNEL_OPEN",
    close: "MESSAGE_CHANNEL_CLOSE",
  },
};

```

```
    fail: "MESSAGE_CHANNEL_FAIL",
  },
} satisfies ChannelDefinition;

function openChannel<T extends ChannelDefinition>(
  def: T,
  channel: keyof T
) {
  // будет реализовано
}
```

Благодаря этому мы можем быть уверены в том, что выполняем контракты, но при этом получаем те же преимущества, что и при использовании контекста `const`. Единственное различие — что для полей не устанавливается значение `readonly`. Но TypeScript принимает литеральный тип всего, поэтому нет возможности установить для полей какое-либо другое значение после удовлетворительной проверки соответствия типа:

```
impl.test.close = "CHANEL_CLOSE_MESSAGE";
//           ^
// Type '"CHANEL_CLOSE_MESSAGE"' is not assignable
// to type '"CHANNEL_CLOSE"'.(2322)
```

Таким образом, мы в двойном выигрыше: мы выполняем правильную проверку типов во время написания аннотаций, а также получаем возможность использования суженных типов для конкретных ситуаций.

12.4. Тестирование сложных типов

Задача

Вы написали тщательно продуманные и сложные типы и хотите убедиться, что они ведут себя правильно.

Решение

Некоторые общеизвестные вспомогательные типы работают как тестовый фреймворк. Протестируйте свои типы!

Обсуждение

В динамически типизированных языках программирования люди всегда обсуждают вопрос о том, нужны ли типы, если можно иметь подходящий набор тестов. По крайней мере, так утверждает одна сторона; другая считает, что зачем так много тестировать, когда есть типы? Ответ, вероятно, находится где-то посередине.

Действительно, типы могут быть полезны во множестве тестовых сценариев. Чем является результат: числом или объектом с определенными свойствами определенных типов? Мы можем легко проверить это с помощью типов. Выдает ли функция правильные результаты? Соответствуют ли эти значения ожиданиям разработчика? Тесты помогут выяснить и это.

На протяжении всей книги вы узнавали много нового об очень сложных типах. С помощью условных типов вы открыли возможности метапрограммирования в TypeScript, где могли создавать новые типы на основе определенных свойств предыдущих типов. Эффективные, полные по Тьюрингу и расширенные. В связи с этим возникает вопрос: как гарантировать, что эти сложные типы действительно будут делать то, что должны? Может быть, стоит *протестировать типы*?

Это действительно возможно. В сообществе известно несколько вспомогательных типов, которые могут служить своего рода платформой для тестирования. Следующие типы взяты из отличного репозитория Type Challenges (<https://tsch.js.org/>), который позволяет максимально эффективно протестировать навыки работы с системой типов TypeScript. Они решают очень сложные задачи: одни из них имеют отношение к реальным сценариям использования, а другие служат просто для развлечения.

Их библиотека тестирования начинается с нескольких типов, которые предполагают наличие истинных или ложных значений. Они довольно просты. Используя обобщенные и литеральные типы, мы можем проверить, является ли это логическое значение истинным или ложным:

```
export type Expect<T extends true> = T;
export type ExpectTrue<T extends true> = T;
export type ExpectFalse<T extends false> = T;
export type IsTrue<T extends true> = T;
export type IsFalse<T extends false> = T;
```

Сами по себе они делают не так уж много, но работают просто потрясающе при использовании в сочетании с `Equal<X, Y>` и `NotEqual<X, Y>`, которые возвращают либо `true`, либо `false`:

```
export type Equal<X, Y> =
  (<T>() => T extends X ? 1 : 2) extends
  (<T>() => T extends Y ? 1 : 2) ? true : false;
export type NotEqual<X, Y> = true extends Equal<X, Y> ? false : true;
```

Функция `Equal<X, Y>` интересна тем, что создает обобщенные функции и проверяет их на соответствие обоим типам, которые следует сравнивать друг с другом. Разрешение для каждого условного типа отсутствует, поэтому TypeScript сравнивает оба условных типа и проверяет наличие совместимости. Это этап логики

использования условных типов в TypeScript, который мастерски объяснен Алексом Чашиным на Stack Overflow (<https://oreil.ly/ywWd4>).

Следующий пакет позволяет проверить, является ли тип `any`:

```
export type IsAny<T> = 0 extends 1 & T ? true : false;
export type NotAny<T> = true extends IsAny<T> ? false : true;
```

Это простой условный тип, который проверяет соответствие `0` с `1 & T`, что должно сужаться до `1` или `never`, а это всегда приводит к ветви `false` условного типа, за исключением тех случаев, когда мы пересекаемся с `any`. Пересечение с `any` всегда является `any`, а с `0` — подмножеством `any`.

Следующий пакет — новая интерпретация `Remap` и `DeepRemap`, которые мы рассматривали в рецепте 8.3, а также `Like` как способ сравнения типов, которые одинаковы по структуре, но не по конструкции:

```
export type Debug<T> = { [K in keyof T]: T[K] };
export type MergeInsertions<T> = T extends object
  ? { [K in keyof T]: MergeInsertions<T[K]> }
  : T;
```

```
export type Alike<X, Y> = Equal<MergeInsertions<X>, MergeInsertions<Y>>;
```

Предыдущая проверка `Equal` теоретически должна быть в состоянии понимать, что `{ x: number, y: string }` равно `{ x: number } & { y: string }`, но элементы реализации средства проверки типов TypeScript не воспринимают их как равные. Именно здесь и можно использовать `Like`.

Последний пакет файла тестирования вызовов типа выполняет две задачи:

- проверяет подмножества с помощью простого условного типа;
- проверяет, может ли созданный вами кортеж рассматриваться как допустимый аргумент для функции:

```
export type ExpectExtends<VALUE, EXPECTED> = EXPECTED extends VALUE
  ? true
  : false;
export type ExpectValidArgs<
  FUNC extends (...args: any[]) => any,
  ARGS extends any[]
> = ARGS extends Parameters<FUNC> ? true : false;
```

Иметь небольшую библиотеку вспомогательных типов, подобную этой, для тестирования и отладки типов действительно полезно, когда ваши типы усложняются. Добавьте их в свои файлы определения глобальных типов (см. рецепт 9.7) и используйте их.

12.5. Проверка типов данных во время выполнения с помощью Zod

Задача

Вы используете данные из внешних источников и не можете быть уверены в их правильности.

Решение

Определите схемы с помощью библиотеки `Zod` и используйте ее для проверки данных из внешних источников.

Обсуждение

Поздравляем! Мы почти добрались до конца книги. Если вы читали ее от начала и до конца, то уже знаете, что система типов TypeScript преследует несколько целей. Прежде всего она хочет предоставить вам отличный инструментарий, чтобы вы могли продуктивно работать при разработке приложений. Кроме того, она стремится удовлетворить потребности всех фреймворков JavaScript и сделать их использование интересным и простым. Она рассматривается как дополнение к JavaScript, как синтаксис для статических типов. Есть также некоторые нецелевые задачи или компромиссы. Система типов TypeScript предпочитает продуктивность корректности, позволяет разработчикам приспосабливать правила к своим потребностям и не претендует на доказуемую надежность.

В рецепте 3.9 мы говорили о том, что можно повлиять на TypeScript, если считать, что типы должны быть какими-то другими, используя *утверждения типов*, а в рецепте 9.2 обсудили, как сделать *небезопасные операции* более надежными и легко выявляемыми. Система типов TypeScript работает только во время компиляции, поэтому все меры безопасности исчезают, как только вы запускаете JavaScript в выбранной вами среде выполнения.

Обычно достаточно проверки типов во время компиляции. Пока вы находитесь *внутри приложения* и сами пишете свои типы, пусть TypeScript проверяет, что все в порядке и код готов к работе. Однако в JavaScript-приложениях вы имеете дело со многими вещами, которые находятся вне вашего контроля: например, с пользовательским вводом или сторонним API, к которому нужно получить доступ и обработать. В процессе разработки неизбежно наступает момент, когда нужно покинуть пределы хорошо типизированного приложения и иметь дело с данными, которым вы не можете доверять.

В процессе разработки взаимодействие с внешними источниками или пользовательским вводом данных может быть достаточно эффективным. Однако убедиться в том, что используемые данные остаются неизменными при запуске в производственной среде, можно, приложив дополнительные усилия. Возможно, вы захотите проверить, что ваши данные соответствуют определенной схеме.

К счастью, существуют библиотеки, которые справляются с подобными задачами. Одной из библиотек, которая приобрела популярность в последние годы, является Zod (<https://zod.dev/>). Она предназначена для TypeScript и гарантирует не только достоверность используемых вами данных и их соответствие вашим ожиданиям, но и то, что вы получаете типы TypeScript, которые можно использовать во всей вашей программе. Zod — своеобразный страж между внешним миром, который вы не контролируете, и внутренним, в котором все хорошо типизировано и проверено на соответствие типам.

Представьте API, предоставляющий данные для типа `Person`, с которым мы сталкивались на протяжении всей книги. У `Person` есть имя и возраст, профессия, которая необязательна, а также статус: в нашей системе он может быть активным, неактивным или только зарегистрированным пользователем, ожидающим подтверждения.

API упаковывает пару объектов `Person` в массив, содержащийся внутри типа `Result`. Короче говоря, это пример классического типа ответа для HTTP-вызовов:

```
type Person = {
  name: string;
  age: number;
  profession?: string | undefined;
  status: "active" | "inactive" | "registered";
};

type Results = {
  entries: Person[]
};
```

Вы знаете, как типизировать такие модели. К настоящему времени вы уже мастерски распознаете и применяете как синтаксис, так и шаблоны. Мы хотим иметь один и тот же тип, но во время выполнения для данных, находящихся вне нашего контроля, используем Zod. И написание того же типа в JavaScript (пространство имен значений) выглядит очень знакомо:

```
import { z } from "zod";

const Person = z.object({
  name: z.string(),
  age: z.number().min(0).max(150),
  profession: z.string().optional(),
```

```
status: z.union([
  z.literal("active"),
  z.literal("inactive"),
  z.literal("registered"),
]),
});

const Results = z.object({
  entries: z.array(Person),
});
```

Как видите, мы используем JavaScript и добавляем имена в пространство имен *значений*, а не в пространство имен *типов* (см. рецепт 2.9), но инструменты, которые мы получаем благодаря текучему интерфейсу Zod, очень хорошо знакомы нам, разработчикам TypeScript. Мы определяем объекты, строки, числа и массивы. Мы можем определять еще и типы объединения, и литералы. Здесь представлены все строительные блоки для определения моделей, и мы можем вкладывать типы, как видим, определив сначала `Person` и повторно используя его в `Results`.

Текущий интерфейс позволяет сделать некоторые свойства необязательными. Все это нам знакомо по TypeScript. Кроме того, можно задать правила проверки. Мы можем сказать, что возраст должен быть больше или равен 0 и меньше 100. Это то, что мы не можем адекватно выполнять в системе типов.

Эти объекты не являются типами, которые мы можем использовать так, как если бы это были типы TypeScript. Они представляют собой *схемы*, ожидающие данные, которые могут проанализировать и проверить. Zod изначально написан на TypeScript, поэтому у нас есть вспомогательные типы, которые позволяют перейти от пространства значений к пространству типов. С помощью `z.infer` (тип, а не функция) мы можем извлечь тип, который определили с помощью функций схемы Zod:

```
type PersonType = z.infer<typeof Person>;
type ResultType = z.infer<typeof Results>;
```

Как же применить проверочные приемы Zod? Поговорим о функции `fetchData`, которая вызывает API, получающий записи типа `ResultType`. Мы просто не знаем, соответствуют ли полученные значения типам, которые мы определили. Поэтому, получив данные в формате `json`, мы используем схему `Results` для анализа полученных данных. Если этот процесс завершен успешно, то мы получаем данные, имеющие тип `ResultType`:

```
type ResultType = z.infer<typeof Results>;

async function fetchData(): Promise<ResultType> {
  const data = await fetch("/api/persons").then((res) => res.json());
  return Results.parse(data);
}
```

Обратите внимание, что мы уже использовали первые меры защиты при определении интерфейса функции. `Promise<ResultType>` основан на том, что мы получаем из `z.infer`.

`Results.parse(data)` имеет выведенный тип, но без имени. Система структурных типов гарантирует, что мы возвращаем правильные данные. Могут возникать ошибки, и мы можем отследить их с помощью соответствующих методов `Promise.catch` или блоков `try-catch`.

Вот пример с использованием `try-catch`:

```
fetchData()
  .then((res) => {
    // сделать что-то с результатами
  })
  .catch((e) => {
    // потенциальная ошибка zod!
  });

// или
try {
  const res = await fetchData();
  // сделать что-то с результатами
} catch (e) {
  // потенциальная zod!
}
```

Мы можем гарантировать, что продолжим работу только при наличии корректных данных, но не обязаны проверять ошибки. Если мы хотим гарантировать, что сначала посмотрим на результат синтаксического анализа, прежде чем продолжить работу с программой, то можем использовать `safeParse`:

```
async function fetchData(): Promise<ResultType> {
  const data = await fetch("/api/persons").then((res) => res.json());
  const results = Results.safeParse(data);
  if (results.success) {
    return results.data;
  } else {
    // В зависимости от вашего приложения вам может потребоваться
    // более сложный способ обработки ошибок, чем возврат
    // пустого результата.
    return { entries: [] };
  }
}
```

Благодаря этому коду Zod становится ценным ресурсом, если вам нужно использовать внешние данные. Кроме того, вы можете адаптироваться к изменениям в API. Предположим, что ваша программа может работать только с активными и неактивными состояниями `Person`; она не знает, как работать с `registered`.

Легко применить преобразование, при котором, основываясь на полученных данных, вы измените состояние "registered" на "active":

```
const Person = z.object({
  name: z.string(),
  age: z.number().min(0).max(150),
  profession: z.string().optional(),
  status: z
    .union([
      z.literal("active"),
      z.literal("inactive"),
      z.literal("registered"),
    ])
    .transform((val) => {
      if (val === "registered") {
        return "active";
      }
      return val;
    })
});
```

В этом случае вы работаете с двумя разными типами: *входной* представляет собой то, что дает вам API, а *выходной* — это данные, которые вы получаете после синтаксического анализа. К счастью, можно получить оба типа из соответствующих вспомогательных типов Zod — `z.input` и `z.output`:

```
type PersonTypeIn = z.input<typeof Person>;
/*
type PersonTypeIn = {
  name: string;
  age: number;
  profession?: string | undefined;
  status: "active" | "inactive" | "registered";
};
*/

type PersonTypeOut = z.output<typeof Person>;
/*
type PersonTypeOut = {
  name: string;
  age: number;
  profession?: string | undefined;
  status: "active" | "inactive";
};
*/
```

Типизация Zod достаточно умна, чтобы понять, что вы удалили один из трех литералов из `status`. Поэтому никаких неожиданностей нет, и вы действительно имеете дело с ожидаемыми данными.

API Zod элегантен, прост в использовании и тесно связан с возможностями TypeScript. Для данных на границах, которые вы не можете контролировать, где вам нужно использовать ожидаемый вид данных, предоставленный сторонними организациями, Zod — это спасение, не требующее от вас слишком большого объема работы. Однако за это приходится платить: проверка во время выполнения требует времени. Чем крупнее набор данных, тем больше времени нужно для его обработки. Кроме того, размер пакета составляет 12 Кбайт — это очень много. Будьте уверены, что вам нужна такая проверка для данных на ваших границах.

Если данные, которые вы запрашиваете, поступают от другой команды вашей компании, может быть, от человека, сидящего рядом с вами, то никакая библиотека, даже Zod, не сравнится с общением друг с другом и совместной работой во имя достижения одних и тех же целей. Использование типов — это способ управления сотрудничеством, а не средство избавиться от него.

12.6. Обход ограничений доступа к индексу

Задача

При обращении к свойству объекта с помощью индексного доступа TypeScript сообщает, что тип, который вы хотите присвоить, не может быть присвоен `never`.

Решение

TypeScript ищет «наименьший общий знаменатель» из возможных значений. Используйте обобщенный тип для блокировки определенных ключей, чтобы TypeScript не предполагал, что правило должно применяться ко всем ключам.

Обсуждение

Иногда при написании TypeScript действия, которые вы обычно выполняете на JavaScript, работают немного по-другому и приводят к странным и загадочным ситуациям. Порой вы просто хотите присвоить значение свойству объекта с помощью индексного доступа и получаете ошибку, подобную этой:

```
"Type 'string | number' is not assignable to type 'never'. Type 'string' is not assignable to type 'never'.(2322)."
```

(«Тип `'string | number'` не может быть присвоен типу `'never'`. Тип `'string'` не может быть присвоен типу `'never'`. (2322)».)

В этом нет ничего необычного, просто «неожиданные пересечения типов» заставляют вас уделять системе типов немного больше внимания.

Рассмотрим пример. Мы создаем функцию, которая позволяет преобразовать объект `anotherPerson` в объект `person`, указав ключ. И `person`, и `anotherPerson` имеют один и тот же тип `Person`, но TypeScript выдает ошибки:

```
let person = {
  name: "Stefan",
  age: 39,
};

type Person = typeof person;

let anotherPerson: Person = {
  name: "Not Stefan",
  age: 20,
};

function update(key: keyof Person) {
  person[key] = anotherPerson[key];
  //^ Type 'string | number' is not assignable to type 'never'.
  // Type 'string' is not assignable to type 'never'.(2322)
}

update("age");
```

В TypeScript сложно отследить присвоение свойств с помощью оператора доступа к индексу. Даже если вы используете `keyof Person`, чтобы сузить список всех возможных ключей доступа, то возможные значения, которые могут быть присвоены, — это `string` или `number` (для имени и возраста соответственно). Это нормально, если у вас есть индексный доступ с правой стороны оператора (чтение), но процесс усложняется при наличии индексного доступа с левой стороны оператора (запись).

TypeScript не может гарантировать, что передаваемое вами значение на самом деле является правильным. Посмотрите на сигнатуру этой функции:

```
function updateAmbiguous(key: keyof Person, value: Person[keyof Person]) {
  //...
}

updateAmbiguous("age", "Stefan");
```

Ничто не мешает мне добавить к каждому ключу ошибочно введенное значение. За исключением TypeScript, который выдает ошибку. Но почему TypeScript сообщает, что тип `never`?

Чтобы разрешить *некоторые* присваивания, TypeScript идет на компромисс. Вместо того чтобы вообще не разрешать *никаких* присваиваний с правой стороны, TypeScript ищет «наименьший общий знаменатель» из возможных значений, например:


```

type Switch = {
  address: number,
  on: 0 | 1
};

declare const switcher: Switch;
declare const key: keyof Switch;

```

Здесь оба ключа являются подмножествами `number`. `address` — это весь набор чисел; с другой стороны — это либо `0`, либо `1`. Абсолютно точно можно задать `0` или `1` для обоих полей! И это то же самое, что вы получаете с помощью TypeScript:

```

switcher[key] = 1; // Это работает
switcher[key] = 2; // Ошибка
// ^ Type '2' is not assignable to type '0 | 1'.(2322)

```

TypeScript получает возможные присваиваемые значения, выполняя *пересечение типов* всех типов свойств. В случае с `Switch` это `number & (0 | 1)`, которое сводится к `0 | 1`. В случае всех свойств `Person` это `string & number`, которые не пересекаются; поэтому это `never`. А вот и виновник проблемы!

Один из способов обойти это ограничение (и это для вашего же блага) — использовать обобщенные типы. Вместо того чтобы разрешить доступ ко всем значениям `keyof Person`, нужно *привязать* определенное подмножество `keyof Person` к обобщенной переменной:

```

function update<K extends keyof Person>(key: K) {
  person[key] = anotherPerson[key]; // работает
}

update("age");

```

Когда я обновляю `update("age")`, `K` привязан к литеральному типу `"age"`. Никакой двусмысленности!

Теоретически существует лазейка, поскольку мы могли бы создать экземпляр `update` с гораздо более широким общим значением:

```

update<"age" | "name">("age");

```

На данный момент команда TypeScript разрешает такие действия. См. также комментарий Андерса Хейлсберга (<https://oreil.ly/0Fetp>). Обратите внимание, что он просит показать примеры использования такого сценария, который прекрасно описывает, как работает команда TypeScript. Первоначальное присваивание с помощью индексного доступа с правой стороны имеет такую высокую вероятность ошибок, что эти присваивания дают вам достаточно гарантий, пока вы не сделаете это очень целенаправленно. Это позволяет исключить целые классы ошибок, не создавая лишних проблем.

12.7. Принятие решения о том, использовать ли перегрузку функций или условные типы

Задача

Используя условные типы, вы получаете больше возможностей для определения сигнатур функций, чем раньше. Вы задаетесь вопросом, нужны ли вам еще перегрузки функций или они устарели.

Решение

Перегрузки функций увеличивают читабельность и предоставляют более простой способ определения ожиданий от вашего типа, чем условные типы. Используйте их, когда этого требует ситуация.

Обсуждение

Появление таких возможностей системы типов, как условные или вариативные типы кортежей, привело к тому, что один из методов описания интерфейса функции — перегрузка функций — отошел на второй план. И на то есть веские причины. Обе возможности были реализованы в целях устранения недостатков, связанных с обычными перегрузками функций.

Смотрите пример конкатенации непосредственно в примечаниях к выпуску TypeScript 4.0. Это функция конкатенации массивов `concat`:

```
function concat(arr1, arr2) {
  return [...arr1, ...arr2];
}
```

Правильная типизация такой функции, при которой были бы учтены все возможные крайние случаи, привела бы к большому количеству перегрузок:

```
// 7 перегрузок для пустого второго массива
function concat(arr1: [], arr2: []): [];
function concat<A>(arr1: [A], arr2: []): [A];
function concat<A, B>(arr1: [A, B], arr2: []): [A, B];
function concat<A, B, C>(arr1: [A, B, C], arr2: []): [A, B, C];
function concat<A, B, C, D>(arr1: [A, B, C, D], arr2: []): [A, B, C, D];
function concat<A, B, C, D, E>(
  arr1: [A, B, C, D, E],
  arr2: []
): [A, B, C, D, E];
function concat<A, B, C, D, E, F>(
  arr1: [A, B, C, D, E, F],
```

```

    arr2: []
  ): [A, B, C, D, E, F];
  // Еще 7 для arr2, имеющего один элемент
  function concat<A2>(arr1: [], arr2: [A2]): [A2];
  function concat<A1, A2>(arr1: [A1], arr2: [A2]): [A1, A2];
  function concat<A1, B1, A2>(arr1: [A1, B1], arr2: [A2]): [A1, B1, A2];
  function concat<A1, B1, C1, A2>(
    arr1: [A1, B1, C1],
    arr2: [A2]
  ): [A1, B1, C1, A2];
  function concat<A1, B1, C1, D1, A2>(
    arr1: [A1, B1, C1, D1],
    arr2: [A2]
  ): [A1, B1, C1, D1, A2];
  function concat<A1, B1, C1, D1, E1, A2>(
    arr1: [A1, B1, C1, D1, E1],
    arr2: [A2]
  ): [A1, B1, C1, D1, E1, A2];
  function concat<A1, B1, C1, D1, E1, F1, A2>(
    arr1: [A1, B1, C1, D1, E1, F1],
    arr2: [A2]
  ): [A1, B1, C1, D1, E1, F1, A2];
  // и т. д. и т. п.

```

И этот код учитывает только массивы, содержащие до шести элементов. Вариативные типы кортежей очень помогают в таких ситуациях:

```

type Arr = readonly any[];

function concat<T extends Arr, U extends Arr>(arr1: T, arr2: U): [...T, ...U] {
  return [...arr1, ...arr2];
}

```

Новая сигнатура функции требует гораздо меньше усилий для проведения синтаксического анализа и очень четко определяет, какие типы она ожидает получить в качестве аргументов и что возвращает. Возвращаемое значение также соответствует типу возвращаемого значения. Никаких дополнительных утверждений: TypeScript может убедиться, что вы возвращаете правильное значение.

Аналогичная ситуация с условными типами. Этот пример очень похож на рецепт 5.1. Представьте программное обеспечение, которое извлекает заказы на основе идентификатора клиента, артикула или заказа. Возможно, вы захотите создать что-то вроде этого:

```

function fetchOrder(customer: Customer): Order[]
function fetchOrder(product: Product): Order[]
function fetchOrder(orderId: number): Order
// реализация
function fetchOrder(param: any): Order | Order[] {
  //...
}

```

Но это только половина дела. Что делать, если в итоге вы столкнулись с неоднозначными типами, когда не знаете точно, получите *только* Customer или *только* Product? Вам нужно учесть все возможные комбинации:

```
function fetchOrder(customer: Customer): Order[]
function fetchOrder(product: Product): Order[]
function fetchOrder(orderId: number): Order
function fetchOrder(param: Customer | Product): Order[]
function fetchOrder(param: Customer | number): Order | Order[]
function fetchOrder(param: number | Product): Order | Order[]
// реализация
function fetchOrder(param: any): Order | Order[] {
  //...
}
```

Добавьте больше возможностей, и в итоге получите еще больше комбинаций. В данном случае использование условных типов может значительно сократить сигнатуру вашей функции:

```
type FetchParams = number | Customer | Product;

type FetchReturn<T> = T extends Customer
  ? Order[]
  : T extends Product
  ? Order[]
  : T extends number
  ? Order
  : never;

function fetchOrder<T extends FetchParams>(params: T): FetchReturn<T> {
  //...
}
```

Условные типы распределяют объединение, поэтому FetchReturn возвращает объединение возвращаемых типов.

Таким образом, есть веская причина использовать эти методы вместо того, чтобы перегружать слишком много функций. Итак, вернемся к вопросу: нужны ли нам еще перегрузки функций?

Да, нужны.

Различные формы функций

Перегрузка функций может оказаться полезной в том случае, если у вас есть разные списки аргументов для вариантов функций. Это означает, что не только сами аргументы (параметры) могут иметь некоторое разнообразие (в этом плане отлично работают условные типы и вариативные кортежи), но и количество и расположение аргументов.

Представьте функцию поиска, которую можно вызвать двумя разными способами:

- используя поисковый запрос — он возвращает `Promise`, которого вы можете дождаться;
- используя поисковый запрос и обратный вызов — в этом сценарии функция ничего не возвращает.

Это *можно* сделать с помощью условных типов, но придется очень потрудиться:

```
// => (1)
type SearchArguments =
  // Первый список аргументов: query и callback
  | [query: string, callback: (results: unknown[]) => void]
  // Второй список аргументов: только query
  | [query: string];

// Условный тип, выбирающий либо void, либо Promise
// в зависимости от ввода => (2)
type ReturnSearch<T> = T extends [query: string]
  ? Promise<Array<unknown>>
  : void;

// фактическая функция => (3)
declare function search<T extends SearchArguments>(...args: T): ReturnSearch<T>;
// z типа void
const z = search("omikron", (res) => {});

// y типа Promise<unknown>
const y = search("omikron");
```

Мы проделали следующие действия.

1. Мы определили наш список аргументов, используя типы кортежей. Начиная с версии TypeScript 4.0 мы можем называть поля кортежей так же, как и объекты. Мы создаем тип объединения, поскольку у нас есть два разных варианта сигнатуры функции.
2. Тип `ReturnSearch` выбирает тип возвращаемого значения на основе варианта списка аргументов. Если это просто строка, то возвращается `Promise`. В противном случае возвращается `void`.
3. Мы добавляем наши типы, ограничивая обобщенную переменную `SearchArguments`, чтобы можно было правильно выбрать тип возвращаемого значения.

Это очень много кода! И он содержит массу сложных функций, которые мы так любим видеть в списке функций TypeScript: условные типы, обобщенные типы, обобщенные ограничения, кортежные типы, типы объединения! Мы получаем

неплохое автозаполнение, но оно далеко не так понятно, как в случае простой перегрузки функции:

```
function search(query: string): Promise<unknown[]>;
function search(query: string, callback: (result: unknown[]) => void): void;
// Это реализация, она касается только вас
function search(
  query: string,
  callback?: (result: unknown[]) => void
): void | Promise<unknown> {
  // Реализация
}
```

Мы используем типы объединения только в части реализации. Остальное очень четко и понятно. Мы знаем наши аргументы и знаем, чего ожидать в ответ. Никаких сложностей, только простые типы. Лучшая часть перегрузки функций состоит в том, что *фактическая* реализация не загрязняет пространство типов. Вы можете использовать `any` и просто не обращать на это внимания.

Точные аргументы

Еще одна ситуация, когда перегрузка функций может упростить работу, — это когда вам нужны точные аргументы и их сопоставление. Рассмотрим функцию, которая применяет событие к обработчику события. Например, у нас есть событие `MouseEvent`, и мы хотим вызвать с его помощью обработчик `MouseEvent`. То же самое возможно для событий клавиатуры и т. д. Если мы используем условные типы и типы объединения для сопоставления события и обработчика, то в итоге можем получить что-то вроде этого:

```
// Все возможные обработчики событий
type Handler =
  | MouseEventHandler<HTMLElement>
  | KeyboardEventHandler<HTMLElement>;

// Сопоставить обработчик с событием
type Ev<T> = T extends MouseEventHandler<infer R>
  ? MouseEvent<R>
  : T extends KeyboardEventHandler<infer R>
  ? KeyboardEvent<R>
  : never;

// Создать
function apply<T extends Handler>(handler: T, ev: Ev<T>): void {
  handler(ev as any); // Здесь нам нужно утверждение
}
```

На первый взгляд, все выглядит нормально. Хотя процесс может показаться немного трудоемким, если учесть все варианты, которые вам нужно отслеживать.

Но есть и более серьезная проблема. Способ, с помощью которого TypeScript обрабатывает все возможные варианты события, приводит к *неожиданному пересечению*, как мы видим в рецепте 12.6. Это означает, что в теле функции TypeScript не может определить, какой тип обработчика вы передаете. Следовательно, он не может определить и то, какой тип события мы получаем. Итак, TypeScript сообщает, что событие может быть событием как мыши, так и клавиатуры. Вам необходимо передать обработчики, которые могут работать с обоими событиями, а это не то поведение функции, которое нам нужно.

Фактическое сообщение об ошибке выглядит так:

```
TS 2345: Argument of type KeyboardEvent<HTMLButtonElement> |
MouseEvent<HTMLButtonElement, MouseEvent> is not assignable to parameter of type
MouseEvent<HTMLButtonElement, MouseEvent> & KeyboardEvent<HTMLButtonElement>.
```

(TS 2345: Аргумент типа `KeyboardEvent<HTMLButtonElement> | MouseEvent<HTMLButtonElement, MouseEvent>` не может быть назначен параметру типа `MouseEvent<HTMLButtonElement, MouseEvent>` и `KeyboardEvent<HTMLButtonElement>`.)

Вот почему нам нужно использовать утверждение типа `as any`, чтобы можно было фактически вызвать обработчик события.

Сигнатура функции работает во многих сценариях:

```
declare const mouseHandler: MouseEventHandler<HTMLButtonElement>;
declare const mouseEv: MouseEvent<HTMLButtonElement>;
declare const keyboardHandler: KeyboardEventHandler<HTMLButtonElement>;
declare const keyboardEv: KeyboardEvent<HTMLButtonElement>;

apply(mouseHandler, mouseEv); // работает
apply(keyboardHandler, keyboardEv); // работает
apply(mouseHandler, keyboardEv); // ломается, как и должно!
//           ^
// Argument of type 'KeyboardEvent<HTMLButtonElement>' is not assignable
// to parameter of type 'MouseEvent<HTMLButtonElement, MouseEvent>'
```

Но как только возникает двусмысленность, все работает не так, как должно:

```
declare const mouseOrKeyboardHandler:
  MouseEventHandler<HTMLButtonElement> |
  KeyboardEventHandler<HTMLButtonElement>;

// Так принято, но может вызывать проблемы!
apply(mouseOrKeyboardHandler, mouseEv);
```

Когда `mouseOrKeyboardHandler` является обработчиком клавиатуры, мы не можем адекватно передать событие мыши. А это именно то, о чем пыталась сообщить ошибка TS2345, появившаяся ранее!

Мы просто переместили проблему в другое место и скрыли ее с помощью утверждения `as any`.

Явные и точные сигнатуры функций упрощают *все*. Отображение и сигнатуры типов становятся более понятными, и нет необходимости использовать условные типы или типы объединения:

```
// Перегрузка 1: MouseEventHandler и MouseEvent
function apply(
  handler: MouseEventHandler<HTMLButtonElement>,
  ev: MouseEvent<HTMLButtonElement>
): void;
// Перегрузка 2: KeyboardEventHandler и KeyboardEvent
function apply(
  handler: KeyboardEventHandler<HTMLButtonElement>,
  ev: KeyboardEvent<HTMLButtonElement>
): void;
// Реализация. Откат к any. Это не тип!
// TypeScript не будет проверять наличие этой строки
// и она не будет отображаться в автозаполнении.
// Это только для того, чтобы вы могли реализовать свои идеи.
function apply(handler: any, ev: any): void {
  handler(ev);
}
```

Перегрузки функций помогают обработать все возможные сценарии. Мы гарантируем отсутствие неоднозначных типов:

```
apply(mouseHandler, mouseEv); // работает!
apply(keyboardHandler, keyboardEv); // работает!
apply(mouseHandler, keyboardEv); // ломается, как и должно!
// ^ No overload matches this call.
apply(mouseOrKeyboardHandler, mouseEv); // ломается, как и должно
// ^
```

Для реализации можно даже использовать `any`. Вы можете быть уверены, что не столкнетесь с ситуацией, которая подразумевает двусмысленность, поэтому можете использовать этот беспечный тип и не беспокоиться.

Тело универсальной функции

И последнее, но не менее важное, о чем нужно упомянуть, — это сочетание условных типов и перегрузок функций. Вспомните пример из рецепта 5.1: мы видели, что условные типы усложняют тело функции при сопоставлении значений с соответствующими обобщенными типами возвращаемых данных. Перемещение условного типа в перегрузку функции и использование очень широкой сигнатуры функции для реализации помогает как пользователям функции, так и ее разработчикам:

```
function createLabel<T extends number | string | StringLabel | NumberLabel>(
  input: T
): GetLabel<T>;
```



```
function createLabel(
  input: number | string | StringLabel | NumberLabel
): NumberLabel | StringLabel {
  if (typeof input === "number") {
    return { id: input };
  } else if (typeof input === "string") {
    return { name: input };
  } else if ("id" in input) {
    return { id: input.id };
  } else {
    return { name: input.name };
  }
}
```

Перегрузки функций все еще очень полезны и для многих сценариев являются оптимальным решением. Их легче читать, проще писать, и во многих случаях они более точны, чем то, что мы получаем с помощью других средств.

Но это не означает «или-или». Вы можете сочетать условные типы и перегрузки функций, если это необходимо для вашего сценария.

12.8. Именованние обобщенных типов

Задача

T и U не несут никакой информации о параметрах обобщенных типов.

Решение

Следуйте шаблону именованния.

Обсуждение

Обобщенные типы TypeScript — возможно, одна из самых эффективных функций языка. Они открывают путь к собственному языку метапрограммирования TypeScript, который позволяет очень гибко и динамично создавать типы. Он близок к тому, чтобы стать собственным функциональным языком программирования.

Особенно с появлением *строковых литеральных типов* и *рекурсивных условных типов* в самых последних версиях TypeScript можно создавать типы, которые делают удивительные вещи. Этот тип из рецепта 12.2 анализирует Express-стиль из информации о маршруте и извлекает объект со всеми его параметрами:

```
type ParseRouteParameters<T> =
  T extends `${string}/${infer U}/${infer R}` ?
    { [P in U | keyof ParseRouteParameters<`/${R}`>]: string } :
  T extends `${string}/${infer U}` ?
    { [P in U]: string } : {}
```

```

type X = ParseRouteParameters<"/api/:what/:is/notyou/:happening">
// type X = {
//   what: string,
//   is: string,
//   happening: string,
// }

```

Определяя обобщенный тип, мы также задаем его параметры. Они могут быть конкретным типом (или, правильнее сказать, подтипом):

```
type Foo<T extends string> = ...
```

Они могут иметь значения по умолчанию:

```
type Foo<T extends string = "hello"> = ...
```

А при использовании значений по умолчанию важен *порядок*. Это лишь одно из многих сходств с обычными функциями JavaScript! Раз уж мы заговорили о функциях, то почему мы используем однобуквенные имена для параметров обобщенного типа?

Большинство параметров обобщенных типов начинаются с буквы *T*. Последующие параметры идут по алфавиту (*U*, *V*, *W*) или являются аббревиатурами, как, например, *K* для обозначения *key*. Однако это может привести к появлению нечитаемых типов. Выражение `Extract<T, U>` не помогает понять, извлекается ли *T* из *U* или наоборот.

Прояснит ситуацию немного более подробная информация:

```
type Extract<From, Union> = ...
```

Теперь мы знаем, что хотим извлечь из первого параметра все, что может быть присвоено `Union`. Кроме того, мы понимаем, что хотим иметь тип `Union`.

Это своего рода документация, и наши параметры типа могут иметь говорящие сами за себя имена, как и в случае с обычными функциями. Используйте схему именования, например описанную ниже.

- Имена всех параметров типов начинаются с заглавной буквы, как и имена всех остальных типов!
- Используйте отдельные буквы только в том случае, если их применение полностью понятно. Например, `ParseRouteParams` может иметь только один аргумент — маршрут.
- Не сокращайте до *T* (это слишком... обобщенно!), а используйте что-то, что проясняет название. Например, в `ParseRouteParams<R>` элемент *R* означает `Route`.
- Редко используйте отдельные буквы; применяйте короткие слова или сокращения: `Elem` для обозначения `Element`, `Route` можно использовать как есть.

- Используйте префиксы, чтобы отличать их от встроенных типов. Например, `Element` является встроенным типом, поэтому используйте `GElement` (или `Elem`).
- Используйте префиксы, чтобы сделать общие имена более понятными: например, `URLObj` понятнее, чем `Obj`.
- Те же шаблоны применимы к выведенным типам внутри обобщенного типа.

Еще раз рассмотрим `ParseRouteParams` и присвоим более четкие имена:

```
type ParseRouteParams<Route> =
  Route extends `${string}/${infer Param}/${infer Rest}` ?
    { [Entry in Param | keyof ParseRouteParameters<`/${Rest}`>]: string } :
  Route extends `${string}/${infer Param}` ?
    { [Entry in Param]: string } : {}
```

Становится гораздо понятнее, для чего предназначен каждый тип. Кроме того, мы видим, что нам нужно перебрать все элементы `Entry` в `Param`, даже если `Param` — просто набор одного типа.

Пожалуй, код стал гораздо более читабельным, чем раньше!

Обратите внимание: отличить параметры типа от реальных типов практически невозможно. Есть еще одна схема, которую активно пропагандирует Мэтт Покок (<https://oreil.ly/Y1i-Q>), — использование префикса `T`:

```
type ParseRouteParameters<TRoute> =
  Route extends `${string}/${infer TParam}/${infer TRest}` ?
    { [TEntry in TParam | keyof ParseRouteParameters<`/${TRest}`>]: string } :
  Route extends `${string}/${infer TParam}` ?
    { [TEntry in TParam]: string } : {}
```

Эта схема близка к венгерской нотации (<https://oreil.ly/c23gW>) для типов.

Какой бы вариант вы ни использовали, сделать так, чтобы имена обобщенных типов и их параметров были понятны вам и вашим коллегам, в TypeScript так же важно, как и в других языках программирования.

12.9. Прототипирование в TypeScript Playground

Задача

Ваш проект настолько объемен, что вам трудно исправлять ошибки в типизации.

Решение

Перенесите свои типы в TypeScript Playground и разрабатывайте их изолированно.

Обсуждение

Интерактивная среда TypeScript Playground (<https://www.typescriptlang.org/play>) (рис. 12.1) — веб-приложение, работающее с TypeScript с момента его первого выпуска и демонстрирующее, как синтаксис TypeScript компилируется в JavaScript.

Первоначально возможности TypeScript Playground были ограничены и направлены на помощь начинающим разработчикам, но в последние годы приложение превратилось в мощный инструмент онлайн-разработки, имеющий обширный арсенал возможностей и незаменимый для разработки на TypeScript. Команда TypeScript просит пользователей сообщать о проблемах, в том числе воссоздавать ошибки с помощью онлайн-редактора. Кроме того, участники команды тестируют новые и будущие функции, позволяя загружать в приложение ночную версию. Одним словом, интерактивная среда TypeScript необходима для разработки TypeScript.

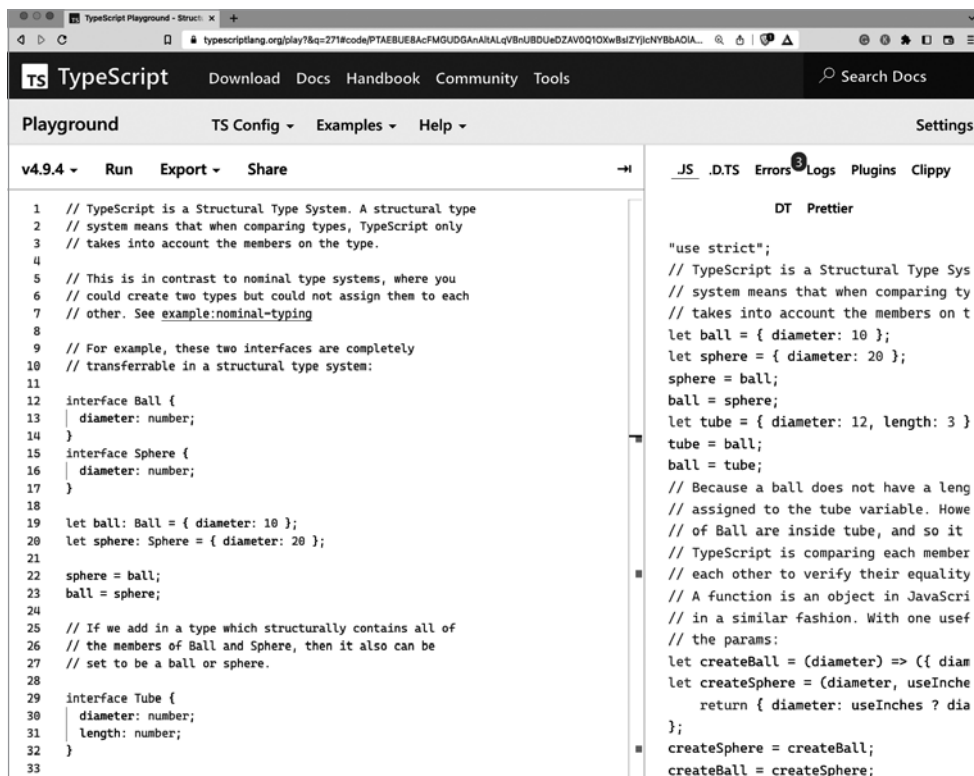


Рис. 12.1. Веб-приложение TypeScript Playground, показывающее один из встроенных примеров

Что касается ваших обычных практик разработки, то TypeScript Playground — отличный способ разрабатывать типы изолированно, независимо от вашего текущего проекта. По мере увеличения конфигураций TypeScript они становятся запутанными, и бывает трудно понять, какие типы используются в вашем реальном проекте. Если вы столкнулись со странным или неожиданным поведением ваших типов, то попробуйте воссоздать их в онлайн-редакторе, изолированно, без остальной части вашего проекта.

В TypeScript Playground нет полноценного файла `tsconfig.json`, но вы можете определить важные части конфигурации с помощью пользовательского интерфейса (рис. 12.2). В качестве альтернативы вы можете установить флаги компилятора с помощью аннотаций непосредственно в исходном коде:

```
// @strictPropertyInitialization: false
// @target: esnext
// @module: nodenext
// @lib: es2015,dom
```

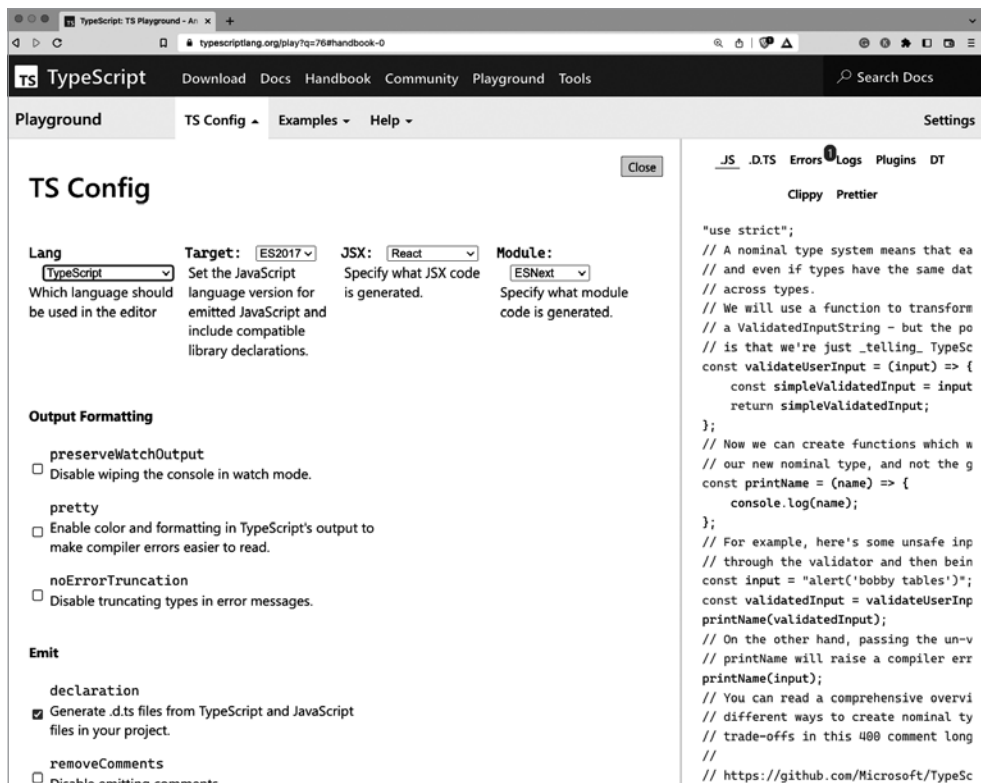


Рис. 12.2. Вместо того чтобы писать настоящий `tsconfig.json`, вы устанавливаете флаги компилятора с помощью панели TSConfig

Этот способ не слишком удобен, но очень эргономичен, так как значительно упрощает использование общих флагов компилятора.

Вы также можете компилировать код TypeScript, получать извлеченную информацию о типах, запускать небольшие фрагменты кода, чтобы посмотреть, как они себя ведут, и экспортировать все в различные места, в том числе другие популярные онлайн-редакторы и IDE.

Вы можете выбрать различные версии, чтобы убедиться, что ваша ошибка не зависит от обновления версии, и запустить различные, хорошо документированные примеры, чтобы изучить основы TypeScript, одновременно пробуя реальный исходный код.

Как отмечалось в рецепте 12.10, разработка JavaScript была бы невозможна без использования зависимостей. В TypeScript Playground можно получать информацию о типах зависимостей непосредственно из NPM. Если вы импортируете, например, React в TypeScript Playground, то веб-приложение попытается получить типы:

- 1) сначала оно просмотрит соответствующий пакет на NPM и проверит, есть ли в его содержимом определенные типы или файлы `.d.ts`;
- 2) если их нет, приложение проверит в NPM, существует ли информация о типе Definitely Typed, и скачает соответствующий пакет `@types`.

Это рекурсивный процесс: если некоторые типы требуют типов из других пакетов, то получение типов тоже будет происходить с помощью зависимостей типов. Для некоторых пакетов можно даже определить, какую версию загружать:

```
import { render } from "preact"; // types: legacy
```

Здесь для `types` установлено значение `legacy`, которое загружает соответствующую устаревшую версию из NPM.

В экосистеме есть нечто большее. Важным инструментом интерактивной среды TypeScript является *Twoslash* — формат разметки для файлов TypeScript, который позволяет выделять код, обрабатывать несколько файлов и показывать файлы, которые создает компилятор TypeScript. Он дает возможность работать с блогами и сайтами (по сути, у вас есть встроенный компилятор TypeScript для примеров кода), но отлично подходит и для создания сложных сценариев отладки.

Аннотации флагов компилятора обрабатываются Twoslash, но вдобавок вы можете получить встроенные подсказки о текущих типах, добавив маркер в комментарий прямо под именем переменной:

```
// @jsxFactory: h
import { render, h } from "preact";

function Heading() {
  return <h1>Hello</h1>
}

const elem = <Heading/>
```

```
// ^?  
// This line above triggers inline hints
```

Результат вы можете увидеть на рис. 12.3.

```
1 // @jsxFactory: h  
2 import { render, h } from "preact";  
3  
4 function Heading() {  
5 |   return <h1>Hello</h1>  
6 }  
7  
8 const elem = <Heading/>  
9 // ^? const elem: h.JSX.Element
```

Рис. 12.3. Twoslash в действии: установка флагов компилятора с помощью аннотаций

Кроме того, Twoslash является частью Bug Workbench (<https://oreil.ly/jVU3u>) — отвления интерактивной среды, предназначенного для создания и отображения сложных вариантов воспроизведения ошибок. Здесь вы можете задать несколько файлов, чтобы увидеть, как работают импорт и экспорт:

```
export const a = 2;  
  
// @filename: a.ts  
  
import { a } from "./input.js"  
console.log(a);
```

Поддержка нескольких файлов запускается с помощью первой аннотации `@filename`. Все, что находится перед этой строкой, становится файлом `input.tsx`, который, по сути, является вашей основной точкой входа.

И наконец, интерактивная среда может служить вашим демонстрационным комплексом для проведения семинаров и тренингов. Используя Twoslash, вы можете создать несколько файлов в репозитории Gist на GitHub и загрузить файлы TypeScript вместе с документацией как часть набора документов Gist docset (рис. 12.4).

TypeScript Playground — чрезвычайно мощный инструмент для обучения методом погружения. Все элементы этого веб-приложения, от простых воспроизведений кода до полноценных демонстрационных наборов, — универсальный источник знаний для разработчиков TypeScript, независимо от того, что вам нужно: исправить ошибки, опробовать что-то новое или изолированно поработать над типами. Изучив его, вы сможете легко перейти к «настоящим» IDE и инструментам.

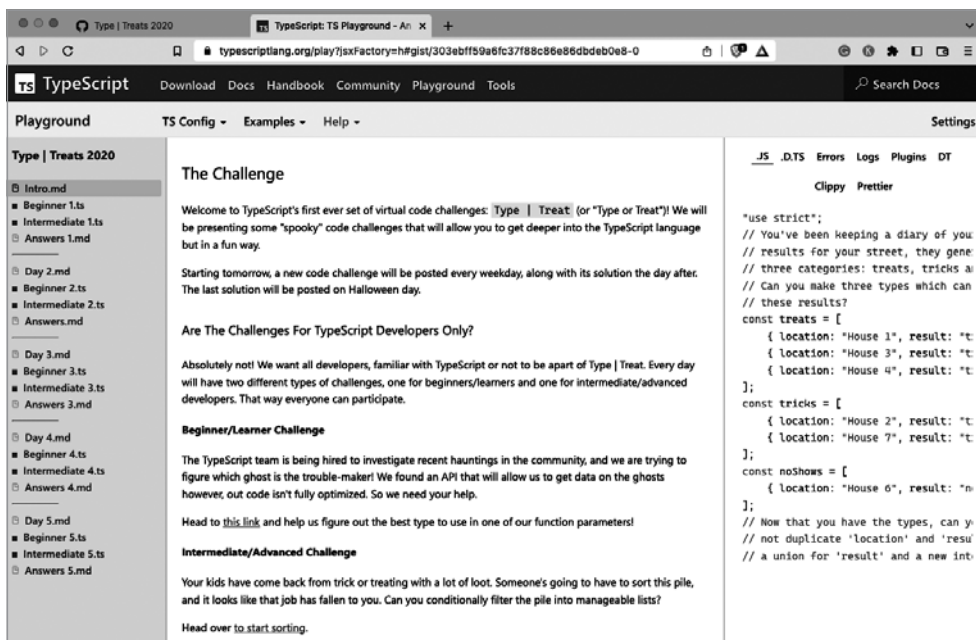


Рис. 12.4. Документация Gist docset в интерактивной среде

12.10. Предоставление нескольких версий библиотеки

Задача

Вы пишете внешние типы для библиотеки и хотите поддерживать обновления типов относительно обновлений версий библиотеки.

Решение

Используйте ссылочные директивы с тройной косой чертой, а также модули, пространства имен и интерфейсы для объединения объявлений.

Обсуждение

Программировать было бы затруднительно без внешних библиотек, которые выполняют за нас большую часть работы. Экосистема JavaScript, пожалуй, одна из самых изобильных, когда речь заходит о сторонних зависимостях, в основном

с помощью репозитория NPM (<https://npmjs.org/>). Кроме того, большинство из них имеют поддержку TypeScript либо с помощью встроенных типов, либо с помощью типов из Definitely Typed. По данным команды TypeScript, почти 80 % NPM типизировано (<https://oreil.ly/G2Ktl>). Однако по-прежнему существуют странные препятствия: например, библиотеки, написанные не на TypeScript, или устаревший код вашей компании, который все равно необходимо сделать совместимым с современным программным обеспечением.

Представьте библиотеку "lib", предоставляющую класс Connector, который вы можете использовать для работы с внутренними системами. Эта библиотека существует в нескольких версиях, и в нее постоянно добавляются новые возможности:

```
import { Connector } from "lib";

// Это существует в версии 1
const connector = new Connector();
const connection = connector.connect("127.0.0.1:4000");

connection.send("Hi!");

// Это существует в версии 2
connection.close();
```

Стоит отметить, что эту библиотеку можно использовать в нескольких проектах вашей организации, задействуя разные версии. Ваша задача — написать типы, чтобы команды получали правильную информацию об автозаполнении и типах.

В TypeScript можно предоставить несколько версий типов библиотеки, создав объявление внешнего модуля для каждой версии. Объявление внешнего модуля — это файл с расширением `.d.ts`, который предоставляет TypeScript типы для библиотеки, написанной не на TypeScript.

По умолчанию TypeScript содержит определения типов и *объединяет все*, что только можно. Если вы хотите ограничить доступ TypeScript к файлам, то обязательно используйте свойства "exclude" и "include" в файле `tsconfig.json`:

```
{
  "compilerOptions": {
    // ...
    "typeRoots": [
      "@types"
    ],
    "rootDir": "./src",
    "outDir": "dist",
  },
  "include": ["./src", "./@types"]
}
```

Мы создаем папку *там же*, где папки, которые мы добавили в `tsconfig.json`. Здесь мы создаем файл `lib.v1.d.ts`, в котором хранится основная информация о том, как создаются объекты:

```
declare module "lib" {
  export interface ConnectorConstructor {
    new (): Connector;
  }
  var Connector: ConnectorConstructor;

  export interface Connector {
    connect(stream: string): Connection;
  }

  export interface Connection {
    send(msg: string): Connection;
  }
}
```

Обратите внимание, что мы используем модули для определения имени модуля и интерфейсы для большинства наших типов. И модули, и интерфейсы открыты для объединения деклараций, то есть мы можем добавлять новые типы в разные файлы, а TypeScript объединяет их. Это очень важно, если мы хотим определить несколько версий.

Кроме того, для коннектора мы используем шаблон интерфейса конструктора (см. рецепт 11.3):

```
export interface ConnectorConstructor {
  new (): Connector;
}

var Connector: ConnectorConstructor;
```

При этом мы можем изменить сигнатуру конструктора и убедиться, что TypeScript распознает экземпляр класса.

В другом файле `lib.v2.d.ts`, расположенном рядом с `lib.v1.d.ts`, мы заново объявляем "lib" и добавляем дополнительные методы в `Connection`. Благодаря слиянию объявлений в интерфейс `Connection` добавляется метод `close`:

```
/// <reference path="lib.v1.d.ts" />

declare module "lib" {
  export interface Connection {
    close(): void;
  }
}
```

Используя директивы с тройной косой чертой, мы создаем ссылку с `lib.v2.d.ts` на `lib.v1.d.ts`, сигнализируя, что все из версии 1 должно быть добавлено в версию 2.

Все эти файлы находятся в папке `@lib`. Используя конфигурацию, которую мы объявили ранее, TypeScript не сможет их получить. Однако мы можем написать новый файл `lib.d.ts` и поместить его в `@types`, а оттуда ссылаться на ту версию, которую хотим добавить:

```
/// <reference path="../../@lib/lib.v2.d.ts" />

declare module "lib" {}
```

Простое изменение с `"../../@lib/lib.v2.d.ts"` на `"../../@lib/lib.v1.d.ts"` изменит и целевую версию, при этом мы по-прежнему будем поддерживать все версии библиотек независимо друг от друга.

Если вам интересно, то попробуйте просмотреть файлы библиотек TypeScript. В них содержится множество определений внешних типов, и там есть чему поучиться. Если вы воспользуетесь редактором для поиска ссылок, например на `Object.keys`, то увидите, что эта функция существует в нескольких местах и, исходя из конфигурации TypeScript, будет добавлен правильный файл. На рис. 12.5 показано, как Visual Studio Code отображает различные расположения файлов для `Object.keys`. TypeScript настолько гибок, что вы можете использовать те же методы для своего проекта, даже расширяя сами встроенные типы TypeScript (см. рецепт 9.7).



Рис. 12.5. Поиск ссылок на встроенные типы в Visual Studio Code показывает, как TypeScript управляет несколькими версиями ECMAScript и DOM

В заключение можно сказать, что предоставить несколько версий типов библиотеки в TypeScript можно, создав объявления внешнего модуля для каждой версии библиотеки и ссылку на соответствующее объявление в вашем коде TypeScript. Надеемся, вы сможете использовать в своем проекте менеджеры пакетов, предназначенные для управления различными версиями библиотек и соответствующими им типами, что облегчит работу с зависимостями и позволит избежать конфликтов.

12.11. Знать, когда остановиться

Задача

Писать сложные и запутанные типы — утомительно!

Решение

Не пишите сложные и запутанные типы. TypeScript развивается постепенно; используйте то, что делает вас продуктивным.

Обсуждение

Я хочу закончить эту книгу несколькими общими советами о том, как вовремя остановиться. Если вы прочитали всю книгу и дошли до этого места, значит, ознакомились с более чем сотней рецептов, содержащих множество советов по решению повседневных задач TypeScript. Настройка проекта, сложные ситуации, когда нужно найти правильный тип, или обходные пути, когда TypeScript сталкивается с ситуацией, когда он слишком строг для своего же блага, — мы рассмотрели все это.

Решения могут быть очень сложными, особенно когда мы вступаем в область условных типов и всего, что их окружает, например вспомогательных типов, вариативных типов кортежей и литеральных типов строковых шаблонов. Система типов TypeScript, несомненно, эффективна, особенно если вы понимаете, что каждое решение и функция основаны на том факте, что в их основе лежит JavaScript. Создание системы типов, предоставляющей надежные статические типы для языка программирования, который по своей сути является динамичным, — удивительное достижение. Я восхищен светлыми умами в Редмонде, которые сделали все это возможным.

Однако, несомненно, временами случаются трудности. Типы могут быть сложными для чтения или создания, и тот факт, что система типов представляет собой собственную систему метапрограммирования, полную по Тьюрингу, для которой требуются библиотеки тестирования, не помогает. А разработчики гордятся тем, что разбираются в каждом аспекте своего ремесла и инструментов, и часто предпочитают сложные типовые решения более простым типам, которые не обеспечивают такую же безопасность типов, но в конечном счете более легки для чтения и понимания.

Проект, в котором рассматриваются основные тонкости системы типов, называется Type Challenges (<https://tsch.js.org/>). Он состоит из головоломок, которые показывают, чего можно достичь с помощью системы типов. Я решаю некоторые из наиболее сложных загадок и нахожу отличные идеи о том, как лучше объяснить систему типов. И хотя головоломки отлично подходят для тренировки ума разра-

ботчика, большинство из них не позволяют получить полное понимание реальных, повседневных ситуаций.

И именно в таких ситуациях мы часто упускаем из виду замечательную возможность TypeScript, которую не так часто можно встретить в основных языках программирования: постепенное внедрение типов. Такие инструменты, как `any`, параметры обобщенных типов, утверждения типа, а также тот факт, что вы можете написать простой код JavaScript с парой комментариев, значительно снижают требования для достижения базового уровня владения языком. Последняя попытка команды TypeScript и TC39 направлена на то, чтобы еще больше снизить барьер, путем добавления аннотаций типов в JavaScript (<https://oreil.ly/yQnIO>), — предложение, которое в настоящее время обсуждается. Цель этого предложения не в том, чтобы сделать JavaScript безопасным для типов, а в том, чтобы исключить этапы компиляции, если мы хотим иметь простые и понятные аннотации типов. Движки JavaScript могут обрабатывать их как комментарии, а программы средства типов — получать реальную информацию о семантике программы.

Как разработчики, руководители проектов, инженеры и архитекторы, мы должны использовать эту возможность. Простые типы всегда лучше: их легче понять, и ими гораздо проще пользоваться.

Сайт TypeScript (<https://typescriptlang.org/>) изменил свое заявление с «JavaScript, который масштабируется» на «JavaScript с синтаксисом для типов», что должно дать вам представление о том, как использовать TypeScript в проектах: пишите код JavaScript, добавляйте аннотации там, где это необходимо, пишите простые, но полные типы и используйте TypeScript как способ, позволяющий документировать, понимать и распространять ваше программное обеспечение.

Я думаю, что TypeScript следует принципу Парето (<https://oreil.ly/smytJ>): 80 % безопасности типов обеспечивается за счет 20 % его возможностей. Это не значит, что все остальное плохо или не нужно. Вы использовали более ста рецептов этой книги, чтобы понять, в каких ситуациях действительно нужны расширенные возможности TypeScript. Это просто должно дать вам представление о том, какие области работы требуют усилий. Не прибегайте к хитростям TypeScript при каждом удобном случае. Следите за тем, чтобы неудачные типы не создавали проблем. Оценивайте усилия, необходимые для изменения типов в вашей программе, и принимайте обоснованные решения. И знайте, что в процессе доработки (см. рецепт 12.2) необходимо выполнить несколько этапов, чтобы можно было легко остановиться.

Об авторе

Стефан Баумгартнер — разработчик и архитектор, живущий в Австрии. Является автором книги *TypeScript in 50 Lessons* и ведет популярный блог о TypeScript. В свободное время он организует встречи и конференции — например, стал инициатором встречи Rust Linz и Европейской конференции по TypeScript. Также является независимым консультантом по Rust и TypeScript на сайте oida.dev. Стефан любит итальянскую кухню, бельгийское пиво и британские виниловые пластинки.

Иллюстрация на обложке

Животное на обложке — красноголовый кольчатый попугай (*Psittacula cyanocephala*). Эти птицы обитают на Индийском субконтиненте. Их часто содержат в качестве домашних питомцев. Как и остальные попугаи в неволе, красноголовые кольчатые попугаи требуют регулярного общения и социализации. По сравнению с другими попугаями они менее агрессивны, очень миролюбивы, обладают веселым нравом.

Красноголовые кольчатые попугаи диморфны, это означает, что самцы и самки имеют легко различимые черты. Окраска оперения преимущественно зеленая, вокруг шеи имеется полоса в виде «ожерелья». У самцов голова пурпурно-красного цвета и черное «ожерелье». У самок голова голубовато-серая, а перья на шее с желтым оттенком. Это птицы среднего размера, длиной от 30 до 50 сантиметров и весом около 100 граммов. Средняя продолжительность жизни красноголовых попугаев — от 15 до 20 лет.

Типичный рацион попугаев в дикой природе составляют фрукты, семена, мясистые лепестки цветов, зерновые. Известны случаи, когда эти птицы совершали набеги на сельскохозяйственные поля и сады. В неволе они здоровее, если их кормить высококачественными семенами и специальным кормом, дополняя рацион свежими фруктами и овощами (например, проростками, листовой зеленью, ягодами).

Эти птицы обычно населяют лесные массивы и лесистые районы от подножия Гималаев на юг до Шри-Ланки, включая Индию, Пакистан и Бангладеш.

Стефан Баумгартнер

Рецепты TypeScript

Перевел с английского В. Дмитрущенко

Изготовлено в России. Изготовитель: ТОО «Спринт Бук».

Место нахождения и фактический адрес: 010000, Казахстан, город Астана, район Алматы,
Проспект Рақымжан Қошқарбаев, дом 10/1, н.п. 18.

Дата изготовления: 11.2024. Наименование: книжная продукция. Срок годности: не ограничен.

Подписано в печать 15.10.24. Формат 70x100/16. Бумага офсетная. Усл. п. л. 34,830. Тираж 1000. Заказ 0000.