100 ошибок Go

и как их избежать

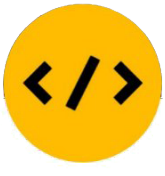
Тейва Харшани

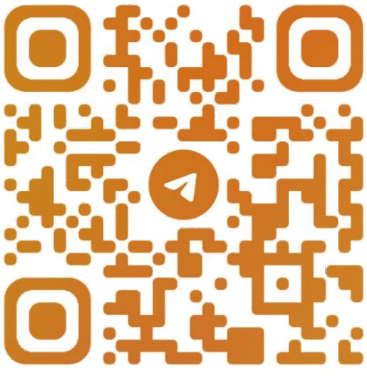
MANNING

100 ошибок Go

И КАК ИХ ИЗБЕЖАТЬ

ТЕЙВА ХАРШАНИ





ПИТЕР®

Санкт-Петербург • Москва • Минск

2024

ББК 32.973.2-018.1
УДК 004.43
Х22

Харшани Тейва

Х22 100 ошибок Go и как их избежать. — СПб.: Питер, 2024. — 480 с.: ил. — (Серия «Для профессионалов»).

ISBN 978-5-4461-2058-1

Лучший способ улучшить код — понять и исправить ошибки, сделанные при его написании. В этой уникальной книге проанализированы 100 типичных ошибок и неэффективных приемов в Go-приложениях.

Вы научитесь писать идиоматичный и выразительный код на Go, разберете десятки интересных примеров и сценариев и поймете, как обнаружить ошибки и потенциальные ошибки в своих приложениях. Чтобы вам было удобнее работать с книгой, автор разделил методы предотвращения ошибок на несколько категорий, начиная от типов данных и работы со строками и заканчивая конкурентным программированием и тестированием.

Для опытных Go-разработчиков, хорошо знакомых с синтаксисом языка.

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

ББК 32.973.2-018.1
УДК 004.43

ISBN 978-1617299599 англ.
ISBN 978-5-4461-2058-1

Краткое содержание

Оглавление

Дэйву Харшани: продолжай оставаться тем, кто ты есть, братик.
Твой потолок — звезды.

Милой Мелиссе.

Предисловие

В 2019 году я во второй раз начал профессионально заниматься работой на Go в качестве основного языка программирования. Тогда я заметил некоторые закономерности, связанные с ошибками написания кода на Go. Я подумал, что обобщение информации о таких частых ошибках было бы полезно для разработчиков.

В своем блоге я сделал пост «10 самых распространенных ошибок, с которыми я сталкивался в проектах на Go» («The Top 10 Most Common Mistakes I've Seen in Go Projects»). Пост стал популярным: его прочитали более 100 000 человек, он был выбран новостным бюллетенем Golang Weekly как один из лучших за 2019 год. Мне льстили положительные отзывы, которые я получал от сообщества Go.

Я понял, что обсуждение типичных ошибок — это мощный инструмент разработки. Сопровождаемый конкретными примерами, он поможет им эффективно осваивать новые навыки, облегчать запоминание как контекста, в котором эти ошибки встречаются, так и способов, позволяющих их избегать.

Около года я собирал примеры типичных ошибок: из профессиональных проектов других разработчиков, из репозиториев опенсорсных программ, из книг, блогов, исследований и обсуждений в сообществе Go. Могу сказать, что я и сам был «достойным источником информации» в плане подобных ошибок.

К концу 2020 года размер моей коллекции ошибок достиг 100 штук, и это показалось мне подходящим, чтобы предложить идею публикации какому-либо издательству. В результате я связался с Manning, которое считал высококлассным

издательством, публиковавшим качественные книги, — для меня оно стало идеальным партнером. Потребовалось почти два года и бесчисленное количество итераций, чтобы четко сформулировать суть каждой из 100 ошибок вместе с релевантными примерами и несколькими решениями, где контекст — это ключевой фактор.

Очень надеюсь, что моя книга поможет вам избежать этих распространенных ошибок и улучшить владение языком Go.

Благодарности

Хочу выразить свою признательность многим людям. Моим родителям — за то, что поддержали меня в тот момент, когда во время учебы я ощутил себя так, как будто нахожусь в ситуации полного провала. Моему дяде Жан-Полю Демону (Jean-Paul Demont) за то, что помог увидеть свет в конце туннеля. Пьеру Готье (Pierre Gautier) за то, что был замечательным вдохновителем и помог мне поверить в себя. Дэмиену Шамбону (Damien Chambon) за то, что заставлял меня постоянно поднимать планку и подталкивал меня к лучшему. Лорану Бернару (Laurent Bernard) за то, что был образцом для подражания и привел меня к осознанию того, что навыки социального общения очень важны. Валентину Делепласу (Valentin Deleplace) за последовательность и логичность его исключительно полезных отзывов. Дугу Раддеру (Doug Rudder) за то, что обучил меня тонкому искусству передачи идей в письменной форме. Тиффани Тейлор (Tiffany Taylor) и Кэти Теннант (Katie Tennant) за высококачественное редактирование и корректуру текста, а также Тиму ван Дерзену (Tim van Deurzen) за глубину и качество профессионального рецензирования.

Хочу также поблагодарить Клару Шамбон (Clara Chambon) — мою любимую маленькую крестницу, Виржини Шамбон (Virginie Chambon) — милейшего человека на свете, всю семью Харшани, Афродити Катику (Afroditi Katika), Серхио Гарсеза (Sergio Garcez) и Каспера Бентсена (Kasper Bentsen) — замечательных инженеров-разработчиков, а также все сообщество Go.

Наконец, я хотел бы поблагодарить своих рецензентов: Адама Ванадамайкена (Adam Wanadamaiken), Алессандро Кампейса (Alessandro Campeis), Аллена Гуча (Allen Gooch), Андреса Сакко (Andres Sacco), Анупама Сенгупту (Anupam Sengupta), Борко Джурковича (Borko Djurkovic), Брэда Хоррокса (Brad Horrocks), Камала Какара (Camal Cakar), Чарльза М. Шелтона (Charles

M. Shelton), Криса Аллана (Chris Allan), Клиффорда Тербера (Clifford Thurber), Козимо Дамиано Прете (Cosimo Damiano Prete), Дэвида Кронкайта (David Cronkite), Дэвида Джейкобса (David Jacobs), Дэвида Моравека (David Moravec), Фрэнсиса Сеташа (Francis Setash), Джанлуиджи Спаньоло (Gianluigi Spagnuolo), Джузеппе Максиа (Giuseppe Maxia), Хироюки Мушу (Hiroyuki Musha), Джеймса Бишопа (James Bishop), Джерома Майера (Jerome Meyer), Джоэля Холмса (Joel Holmes), Джонатана Р. Чоута (Jonathan R. Choate), Йорта Роденбурга (Jort Rodenburg), Кита Кима (Keith Kim), Кевина Ляо (Kevin Liao), Лева Вайде (Lev Veyde), Мартина Денерта (Martin Dehnert), Мэтта Велке (Matt Welke), Нираджа Шаха (Neeraj Shah), Оскара Утбулта (Oscar Utbult), Пейти Ли (Peiti Li), Филиппа Джанертка (Philipp Janertq), Роберта Веннера (Robert Wenner), Райана Барроуска (Ryan Burrowsq), Райана Хубера (Ryan Huber), Санкета Найка (Sanket Naik), Сатадру Ройя (Satadru Roy), Шона Д. Вика (Shon D. Vick), Тада Майера (Thad Meyer) и Вадима Туркова. Все ваши предложения и замечания помогли сделать эту книгу лучше.

Об этой книге

Книга «100 ошибок Go и как их избежать» содержит описание 100 распространенных ошибок, которые допускают Go-разработчики. Она в значительной степени сосредоточена на самом языке и его стандартной библиотеке, а не на внешних библиотеках или фреймворках. Обсуждения большинства ошибок сопровождаются конкретными примерами, иллюстрирующими те обстоятельства, когда такие ошибки могут совершаться. Эта книга — не какая-то догма. Каждое предлагаемое решение детализировано в той мере, чтобы передать контекст.

Для кого эта книга

Эта книга предназначена для разработчиков, уже знакомых с языком Go. В ней не рассматриваются его основные понятия — синтаксис или ключевые слова. Предполагается, что вы уже занимались реальным проектом на Go. Но прежде чем углубляться в большинство конкретных тем, удостоверимся, что некоторые базовые вещи понимаются ясно и четко.

Структура книги

Книга состоит из 12 глав:

Глава 1 «Go: просто научиться, но сложно освоить» объясняет, почему, несмотря на то что Go считается простым языком, его нелегко освоить досконально. В ней также приведены типы ошибок, которые мы рассмотрим в книге.

Глава 2 «Организация кода и проекта» содержит описание распространенных ошибок, которые могут помешать организовать программный код чистым, идиоматичным, удобным для дальнейшей обработки и поддержки образом.

В главе 3 «Типы данных» обсуждаются ошибки, связанные с основными типами, срезами и картами.

В главе 4 «Управляющие структуры» исследуются распространенные ошибки, связанные с циклами и другими управляющими структурами.

В главе 5 «Строки» рассматривается принцип представления строк и связанные с ним распространенные ошибки, приводящие к неточности или неэффективности кода.

В главе 6 «Функции и методы» обсуждаются распространенные проблемы, связанные с функциями и методами, такие как выбор типа получателя и предотвращение распространенных ошибок отложенного выполнения (defer).

В главе 7 «Обработка ошибок» рассматривается идиоматическая и точная обработка ошибок в Go.

В главе 8 «Конкурентность: основы» представлены основные концепции конкурентности. Мы разберем, почему конкурентность не всегда быстрее, в чем различия между конкурентностью и параллелизмом, а также обсудим типы рабочей нагрузки.

В главе 9 «Конкурентность: практика» рассмотрены примеры ошибок, связанных с конкурентностью при использовании каналов, горутин и других примитивов Go.

Глава 10 «Стандартная библиотека» содержит описание распространенных ошибок, допускаемых при использовании стандартной библиотеки с HTTP, JSON или (например) time API.

В главе 11 «Тестирование» обсуждаются ошибки, которые делают тестирование и бенчмаркинг менее универсальными, эффективными и точными.

Глава 12 «Оптимизация» завершает книгу. В ней исследуются способы того, как оптимизировать приложение для повышения его производительности, — от понимания основ функционирования центрального процессора до конкретных тем, связанных с Go.

О коде в книге

Книга содержит множество примеров исходного кода как в нумерованных листингах, так и в тексте. В обоих случаях исходный код форматируется `моноширинным шрифтом`, в отличие от обычного текста. Иногда для кода также применяется

жирный шрифт, чтобы выделить фрагменты, изменившиеся по сравнению с предыдущими шагами, — например, при добавлении новой функциональности в существующую строку кода.

Во многих случаях оригинальная версия исходного кода переформатируется; добавляются разрывы строк и измененные отступы, чтобы код помещался на странице. Иногда даже этого оказывается недостаточно и в листинги включаются маркеры продолжения строк (➥). Также из исходного кода часто удаляются комментарии, если код описывается в тексте.

Исполняемые фрагменты кода можно загрузить из версии liveBook (электронной) по адресу https://livebook.manning.com/book/100-go-mistakes-how-to-avoid-them. Полный код примеров книги доступен для загрузки на сайте Manning по адресу https://www.manning.com/books/100-go-mistakes-how-to-avoid-them и GitHub https://github.com/teivah/100-go-mistakes.

Форум liveBook

Приобретая книгу «100 ошибок Go и как их избежать», вы получаете бесплатный доступ к закрытому веб-форуму издательства Manning (на английском языке), на котором можно оставлять комментарии о книге, задавать технические вопросы и получать помощь от автора и других пользователей. Чтобы получить доступ к форуму, откройте страницу https://livebook.manning.com/book/100-go-mistakes-how-to-avoid-them/discussion. Информацию о форумах Manning и правилах поведения на них см. на https://livebook.manning.com/#!/discussion.

В рамках своих обязательств перед читателями издательство Manning предоставляет ресурс для содержательного общения читателей и авторов. Эти обязательства не подразумевают конкретную степень участия автора, которое остается добровольным (и неоплачиваемым). Задавайте автору хорошие вопросы, чтобы он не терял интереса к происходящему! Форум и архивы обсуждений доступны на веб-сайте издательства, пока книга продолжает издаваться.

Об авторе

ТЕЙВА ХАРШАНИ — старший инженер-программист в Docker. Работал в области страхования, транспорта и в отраслях, где критически важна безопасность, например в управлении воздушным движением. Увлечен языком Go и тем, как разрабатывать и реализовывать на нем надежные приложения.

Иллюстрация на обложке

На обложке книги — рисунок под названием «Femme de Buccari en Croatie» («Женщина из Бакара, Хорватия»).

Иллюстрация взята из вышедшего в 1797 году каталога национальных костюмов, составленного Жаком Грассе де Сен-Савьером. Каждая иллюстрация этого каталога тщательно прорисована и раскрашена от руки. В прежние времена по одежде человека можно было легко определить, где он живет и какова его профессия или положение. Manning отдает дань изобретательности и инициативности компьютерных технологий, используя для своих изданий обложки, демонстрирующие богатое вековое разнообразие региональных культур, оживающее на изображениях из собраний, подобных этому.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

1 Go: просто научиться, но сложно освоить

В этой главе:

- ✓ Что делает Go эффективным, масштабируемым и производительным языком
- ✓ Почему языку Go просто научиться, но овладеть им по-настоящему сложно
- ✓ Общее описание распространенных типов ошибок, допускаемых разработчиками

Ошибаться свойственно всем. Как сказал Альберт Эйнштейн:

Тот, кто никогда не совершал ошибок, тот никогда не пробовал что-то новое.

В конце концов, важно не количество совершенных ошибок, а наша способность учиться на них. Это утверждение относится и к программированию. Мастерство, которое мы приобретаем, — это не волшебство. Мы делаем множество ошибок и учимся на них. Это основная мысль книги. Мы рассмотрим и изучим 100 распространенных ошибок, которые допускаются во многих сферах использования языка Go, и это поможет вам стать более опытным программистом.

В главе 1 мы кратко расскажем, почему Go с годами стал одним из основных и стандартных инструментов работы. Мы обсудим, почему, несмотря на то что Go считается простым в изучении, овладение его нюансами может быть весьма сложным. Наконец, познакомимся с основными понятиями из этой книги.

1.1. GO: ОСНОВНЫЕ МОМЕНТЫ

Если вы читаете нашу книгу, то, скорее всего, уже «подсели» на Go. Поэтому в этом разделе будет только краткий обзор, призванный напомнить, что делает Go таким мощным языком.

Отрасль разработки программного обеспечения (ПО) за последние десятилетия значительно изменилась. Большинство современных систем больше не создается одним человеком. Все они — результат работы команд, состоящих из многих программистов, а иногда даже из сотен, если не тысяч. Написанный программный код должен быть читабельным, выразительным, удобным в сопровождении, чтобы обеспечивать надежную работу системы на протяжении многих лет. С другой стороны, в нашем быстро меняющемся мире максимальное повышение гибкости и сокращение времени выхода на рынок очень важны для большинства компаний. Программирование тоже должно следовать этой тенденции, поэтому компании стремятся к тому, чтобы программисты работали максимально продуктивно при чтении, написании и сопровождении кода.

В ответ на эти вызовы и требования в 2007 году компания Google создала язык Go. С тех пор многие организации приняли его для использования в различных областях программирования: в API, автоматизации, базах данных, интерфейсах командной строки и т. д. Сегодня многие считают Go одним из основных языков для разработки облачных систем.

Что касается функциональности, то в Go нет наследования типов, исключений, макросов, частичных функций, поддержки ленивых вычислений или неизменяемости, перегрузки операторов, сопоставления шаблонов и т. д. Почему? Вот что об этом говорит официальный FAQ по Go (https://go.dev/doc/faq):

> *Почему в Go нет какой-то функции X? Ваша любимая функция может отсутствовать, поскольку не вписывается в логику или структуру языка, влияет на скорость компиляции или ясность дизайна кода либо просто потому, что сделала бы фундаментальную модель системы слишком сложной.*

Оценка качества языка программирования на основании количества функций в нем, вероятно, некорректна. По крайней мере для Go эта метрика не главная.

При оценке адекватности использования языка в масштабе какой-то организации используют несколько важных характеристик. К ним относятся:

- *Стабильность*. Несмотря на то что в Go вносятся частые изменения (направленные на улучшение самого языка и устранение уязвимостей с точки зрения безопасности), он остается достаточно стабильным языком. Некоторые считают это качество одной из лучших особенностей языка.
- *Выразительность*. Мы можем определить выразительность языка по тому, насколько написание и чтение кода отвечает представлениям о естественности и интуитивной понятности. Уменьшенное количество ключевых слов и ограниченные способы решения общих проблем делают Go выразительным языком для больших кодовых баз.
- *Компиляция*. Что может быть более раздражающим для разработчиков, чем долгое ожидание сборки для тестирования приложения? Стремление к быстрой компиляции всегда было сознательной целью разработчиков языка. А это основа высокой производительности.
- *Безопасность*. Go — надежный язык со статической типизацией. Следовательно, у него есть строгие правила времени компиляции, которые в большинстве случаев обеспечивают безопасность типов.

Go был создан с нуля с очень полезными функциями: с примитивами конкурентности, горутинами и каналами. Ему особо не нужно полагаться на внешние библиотеки для создания эффективных конкурентных приложений. Наблюдение за тем, насколько важна конкурентность в наши дни, также показывает, почему Go сейчас самый подходящий язык и будет оставаться им в обозримом будущем.

Некоторые считают Go простым языком, и отчасти это правда. Например, новичок может разобраться с его основными возможностями менее чем за один день. Возникает вопрос: зачем же изучать книгу, посвященную систематизации ошибок в Go, если он так прост?

1.2. ПРОСТО НЕ ОЗНАЧАЕТ ЛЕГКО

Между понятиями «просто» и «легко» есть тонкая разница. «Простой» применительно к технологии означает несложный для изучения или понимания. «Легкость» означает возможность добиваться чего угодно без особых усилий. Go прост в изучении, но не всегда легок в освоении.

Возьмем, к примеру, конкурентность. В 2019 году было опубликовано исследование, посвященное ошибкам конкурентности: «Понимание реальных ошибок

конкурентности в Go»[1]. Это исследование было первым систематическим анализом ошибок конкурентности. Оно опиралось на данные нескольких популярных репозиториев Go — Docker, gRPC и Kubernetes. Один из самых важных выводов заключается в том, что большинство блокирующих ошибок вызвано неточным использованием парадигмы передачи сообщений (message passing) по каналам, несмотря на убеждение, что передача сообщений легче обрабатывается и менее подвержена ошибкам, чем разделяемая память.

Какой должна быть реакция на такой вывод? Должны ли мы считать, что разработчики языка ошибались насчет передачи сообщений? Должны ли мы пересмотреть использование конкурентности в нашем проекте? Конечно нет.

Это не вопрос противопоставления передачи сообщений разделяемой памяти и выявления из них «победителя». Но разработчики Go должны хорошо понимать, как использовать конкурентность, каково ее влияние на современные процессоры, когда следует предпочесть один подход другому и как избежать при этом попадания в типичные ловушки. Этот пример подчеркивает, что хотя каналы и горутины могут быть простыми для изучения, на практике это совсем не просто.

Понятие «просто не значит легко» можно обобщить на многие аспекты Go, а не только на конкурентность. И чтобы стать опытными Go-разработчиками, нужно хорошо разбираться во всех его аспектах. А это требует времени, усилий и ошибок.

Цель книги — помочь ускорить наш путь к мастерству, рассмотрев 100 ошибок в Go.

1.3. 100 ОШИБОК В GO

Почему следует прочитать эту книгу? Почему бы вместо этого не углубить знания с помощью «обычной» книги, которая достаточно подробно рассматривает разные темы?

В статье, опубликованной в 2011 году, нейробиологи доказали, что столкновение с ошибками — это лучшие моменты для развития способностей нашего мозга[2].

---

[1] T. Tu, X. Liu, et al. (с соавторами), *Understanding Real-World Concurrency Bugs in Go*, работа была представлена на ASPLOS 2019, April 13–17, 2019.

[2] J. S. Moser, H. S. Schroder, с соавторами, “Mind Your Errors: Evidence for a Neural Mechanism Linking Growth Mindset to Adaptive Posterror Adjustments,” Psychological Science, vol. 22, no. 12, pp. 1484–1489, Dec. 2011.

Все мы проходили через процесс обучения на какой-то ошибке, вспоминая этот случай через месяцы или даже годы, когда с ним был связан какой-то контекст. В статье Джанет Меткалф (Janet Metcalfe) говорится, что это происходит потому, что ошибки оказывают стимулирующее воздействие[1]. Суть в том, что мы можем помнить не только саму ошибку, но и ее контекст. И поэтому обучение на ошибках так эффективно.

Чтобы усилить этот эффект, в книге каждая рассматриваемая типичная ошибка подкреплена примерами из реальной практики. Эта книга не только о теории, она поможет избежать ошибок и принимать взвешенные, осознанные решения.

Скажи мне, и я забуду. Научи меня, и я запомню. Вовлеки меня, и я научусь.

Неизвестный автор

Здесь представлены семь основных категорий ошибок, которые можно классифицировать как:

- баги;
- излишнюю сложность;
- плохую читаемость;
- неоптимальную или неидиоматическую организацию;
- отсутствие удобства в API;
- неоптимизированный код;
- недостаточную производительность.

Далее я дам краткое описание каждой категории ошибок.

1.3.1. Баги

Первый и, возможно, самый очевидный тип — это ошибки в исходном коде. В 2020 году исследование, проведенное Synopsys, оценило стоимость багов в ПО только в США более чем в 2 триллиона долларов[2].

---

[1] J. Metcalfe, “Learning from Errors,” Annual Review of Psychology, vol. 68, pp. 465–489, Jan. 2017.

[2] Synopsys, “The Cost of Poor Software Quality in the US: A 2020 Report.” 2020. https://news.synopsys.com/2021-01-06-Synopsys-Sponsored-CISQ-Research-Estimates-Cost-of-Poor-Software-Quality-in-the-US-2-08-Trillion-in-2020.

Баги могут приводить и к трагическим последствиям. Вспомним случай с аппаратом для лучевой терапии Therac-25 производства компании Atomic Energy of Canada Limited (AECL). Из-за состояния гонки машина дала своим пациентам дозы облучения, которые в сотни раз превышали ожидаемые, что привело к смерти трех пациентов. Этот пример показывает, что баги могут повлечь за собой не только денежные потери. И мы, как разработчики, должны помнить, насколько важна наша работа.

Я рассмотрю множество случаев, которые могут привести к различным багам, включая гонки данных, утечки, логические ошибки и др. Хотя точные тесты и должны обнаруживать такие ошибки как можно раньше, иногда мы можем пропускать их из-за различных факторов, например из-за нехватки времени или их сложности. И разработчику важно убедиться, что для устранения таких багов сделано все возможное.

1.3.2. Излишняя сложность

Следующая категория ошибок связана с излишней сложностью. Значительная часть сложности ПО вызвана тем, что разработчики стремятся думать о своем воображаемом будущем. Вместо того чтобы решать конкретные задачи прямо сейчас, может возникнуть соблазн создать «эволюционирующее» ПО, которое будет пригодным для любого будущего варианта использования. В большинстве случаев это приводит к тому, что объем недостатков превышает число преимуществ, что делает код сложным для понимания и анализа.

Возвращаясь к Go, можно вспомнить множество примеров того, как у разработчиков возникает соблазн разработать абстрактные функции для будущего, например интерфейсы или дженерики. В этой книге обсуждаются примеры, когда следует проявлять особую осторожность, чтобы не переусложнить код.

1.3.3. Плохая читаемость

Как написал Роберт Мартин (Robert Martin) в книге «Clean Code: A Handbook of Agile Software Craftsmanship»[1], соотношение времени, затрачиваемого на чтение и написание кода, значительно превышает 10 : 1. Большинство из нас начинали программировать в собственных проектах, где удобочитаемость не так важна. Но сегодняшняя разработка ПО — это программирование во временно́м

[1] Мартин Р. «Чистый код: создание, анализ и рефакторинг». Санкт-Петербург, издательство «Питер».

измерении: нужно убедиться, что с приложением все еще можно работать и поддерживать его спустя месяцы, годы или, возможно, даже десятилетия после релиза.

При программировании на Go можно наделать много ошибок, которые затруднят читаемость кода. Среди таких ошибок может быть и вложенный код, и представления типов данных, а иногда и использование неименованных результирующих параметров. На протяжении этой книги мы будем учиться писать читаемый код и заботиться о его будущих читателях (в частности, о себе).

1.3.4. Неоптимальная или неидиоматическая организация

Другой тип ошибки — это неоптимальная или неидиоматическая организация кода и проекта. Такие проблемы могут затруднить анализ и дальнейшую поддержку проекта. В этой книге рассмотрены некоторые из распространенных ошибок такого рода. Например, мы увидим, как структурировать проект и обращаться с пакетами утилит или функциями инициализации. Рассмотрение этих ошибок поможет организовать код и проекты более эффективно и идиоматично.

1.3.5. Отсутствие удобства в API

Распространенные ошибки, снижающие удобство API для наших потребителей, — это еще один тип. Если API неудобен для пользователя, он будет менее выразительным и, следовательно, более трудным для понимания и более подверженным дальнейшим ошибкам.

Такие ошибки встречаются во многих ситуациях и могут заключаться в чрезмерном использовании типа `any`, в использовании неправильных порождающих паттернов при работе с опциями или в слепом применении стандартных методов объектно-ориентированного программирования, что влияет на удобство использования API. Мы рассмотрим распространенные ошибки, мешающие передавать в распоряжение наших пользователей удобные для них API.

1.3.6. Неоптимизированный код

Код, оптимизированный в недостаточной степени, — еще один тип ошибок разработчиков. Их можно сделать по разным причинам, например из-за непонимания особенностей языка или даже из-за отсутствия фундаментальных знаний.

Недостаточная производительность — одно из наиболее очевидных последствий этой ошибки, но не единственное.

Оптимизация кода полезна и для точности. Например, в этой книге представлены некоторые распространенные методы, обеспечивающие высокую точность операций с плавающей точкой. Мы также рассмотрим множество случаев, которые могут негативно сказаться на производительности кода, например, из-за недостаточного распараллеливания задач, незнания того, как уменьшать использование ресурсов памяти, или влияния выравнивания данных. Поговорим о вопросах оптимизации под разными углами.

1.3.7. Недостаточная производительность

В большинстве случаев мы задаемся вопросом: какой язык лучше всего выбрать для конкретного нового проекта? Ответ: тот, с которым мы работаем наиболее продуктивно. Для достижения мастерства очень важно знать, как работает язык, и использовать его по максимуму.

Мы рассмотрим конкретные примеры, которые помогут стать продуктивными при работе на Go. Например, написание эффективных тестов для обеспечения работоспособности кода, использование стандартной библиотеки для повышения эффективности, а также извлечение максимальной пользы из инструментов профилирования и линтеров. Пришло время разобраться в этих 100 распространенных ошибках Go!

ИТОГИ

- Go — это современный язык программирования, который позволяет повысить производительность разработчиков, что сегодня крайне важно для большинства компаний.
- Go прост в изучении, но нелегок в освоении. Поэтому важно углубить свои знания, чтобы использовать его наиболее эффективно.
- Обучение на разборе ошибок и на конкретных примерах — это мощный способ овладеть языком. Книга на примерах разбора 100 распространенных ошибок ускорит путь к профессиональному мастерству.

2 Организация кода и проекта

В этой главе:

- ✓ Идиоматическая организация кода
- ✓ Эффективная работа с абстракциями: интерфейсы и дженерики
- ✓ Как структурировать проект: лучшие практики

Сделать текст кода в Go чистым, идиоматичным и удобным для сопровождения — непростая задача. Чтобы понять суть лучших практик, связанных с написанием кода и организацией проекта, потребуется накопить определенный опыт и набить шишки. Каких ловушек следует избегать (например, затенения переменных и злоупотребления вложенным кодом)? Как структурировать пакеты? Когда и где использовать интерфейсы или дженерики, функции инициализации и пакеты утилит? Рассмотрим распространенные ошибки в организации кода.

2.1. ОШИБКА #1: НЕПРЕДНАМЕРЕННО ЗАТЕНЯТЬ ПЕРЕМЕННЫЕ

Область видимости переменной — это те места кода, в которых можно ссылаться на эту переменную, другими словами, та часть приложения, где действует привязка имени. В Go имя переменной, уже объявленное во внешней области

видимости, может быть повторно объявлено во внутренней области видимости. Такая ситуация называется затенением переменной и может приводить к распространенным ошибкам.

В примере ниже показан непреднамеренный побочный эффект из-за наличия затененной переменной. В этом фрагменте кода HTTP-клиент создается двумя разными способами, в зависимости от булева значения `tracing`:

```
var client *http.Client  <-- Объявляется переменная client
if tracing {
    client, err := createClientWithTracing()  <-- Создается HTTP-клиент со включенной трассировкой. (Переменная client затенена в этом блоке)
    if err != nil {
        return err
    }
    log.Println(client)
} else {
    client, err := createDefaultClient()  <-- Создается HTTP-клиент по умолчанию. (Переменная client также затенена в этом блоке)
    if err != nil {
        return err
    }
    log.Println(client)
}
// Использование переменной client
```

В этом примере в самом начале объявляется переменная `client`. Затем мы используем краткий оператор присваивания переменной (`:=`) в обоих внутренних блоках, чтобы присвоить результат вызова функции внутренним переменным `client`, а не внешней переменной `client`. В результате оказывается, что внешняя переменная всегда равна нулю.

> **ПРИМЕЧАНИЕ** Этот код компилируется, поскольку внутренние переменные `client` используются в вызовах логирования. В противном случае появлялись бы ошибки компиляции: `client declared and not used`.

Как обеспечить присвоение значения именно исходной переменной `client`? Есть два варианта.

```
var client *http.Client
if tracing {
    c, err := createClientWithTracing()  <-- Создается временная переменная c
    if err != nil {
        return err
    }
    client = c  <-- Переменной client присваивается значение этой временной переменной
} else {
    // Та же логика
}
```

Здесь мы присваиваем результат временной переменной `c`, область видимости которой находится только в пределах блока `if`. Затем присваиваем его обратно переменной `client`. То же делаем для блока `else`.

Во втором варианте используется оператор присваивания (`=`) во внутренних блоках для непосредственного присвоения результатов функции переменной `client`. Но для этого нужно создать переменную `error`, поскольку оператор присваивания работает только в том случае, если имя переменной уже было объявлено. Например:

```
var client *http.Client
var err error  <-- Объявляется переменная err
if tracing {
    client, err = createClientWithTracing()  <-- Используется оператор присваивания, чтобы напрямую присвоить переменной client значение, возвращаемое *http.Client
    if err != nil {
        return err
    }
} else {
    // Та же логика
}
```

Чтобы не присваивать значение временной переменной, мы можем напрямую присвоить результат переменной `client`.

Оба способа вполне допустимы. Основное различие между ними заключается в том, что во втором варианте мы выполняем только одно присваивание, что можно считать более легким для чтения. Кроме того, со вторым вариантом можно объединить и реализовать обработку ошибок вне блоков операторов `if/else`, как показано в следующем примере:

```
if tracing {
    client, err = createClientWithTracing()
} else {
    client, err = createDefaultClient()
}
if err != nil {
    // Типичная обработка ошибок
}
```

Затенение переменной происходит, когда ее имя повторно объявляется во внутренней области видимости, но мы видели, что эта практика чревата ошибками. Установка правила, запрещающего затененные переменные, зависит от личного вкуса. Иногда бывает удобно повторно использовать существующее имя, например `err`, для обозначения тех переменных, которые так или иначе связаны с ошибками. Но следует быть начеку, потому что теперь мы знаем, что можем столкнуться со сценарием, когда код компилируется, но переменная на самом

деле получает значение, отличающееся от ожидаемого. Позже в этой главе мы рассмотрим, как обнаруживать затененные переменные.

В следующем разделе показано, почему важно не злоупотреблять вложенным кодом.

2.2. ОШИБКА #2: ЛИШНИЙ ВЛОЖЕННЫЙ КОД

Ментальная модель, относящаяся к конкретному программному продукту, представляет собой внутреннее мысленное представление о том, как ведет себя система. При программировании нужно придерживаться таких ментальных моделей (например, общих взаимодействий в коде и реализациях функций). Код считается удобочитаемым по множеству критериев: использование имен/названий, согласованность, соответствующее форматирование и т. д. Читабельный код требует меньше когнитивных усилий для понимания его соответствия ментальной модели, поэтому его легче читать и сопровождать.

Важнейший аспект удобочитаемости — это фактор количества вложенных уровней. Предположим, что мы работаем над новым проектом и нужно понять, что делает следующая функция `join`:

```
func join(s1, s2 string, max int) (string, error) {
    if s1 == "" {
        return "", errors.New("s1 is empty")
    } else {
        if s2 == "" {
            return "", errors.New("s2 is empty")
        } else {
            concat, err := concatenate(s1, s2)
            if err != nil {
                return "", err
            } else {
                if len(concat) > max {
                    return concat[:max], nil
                } else {
                    return concat, nil
                }
            }
        }
    }
}

func concatenate(s1 string, s2 string) (string, error) {
    // ...
}
```

Вызывает функцию concatenate для выполнения определенной конкатенации, но может возвращать ошибки

Эта функция `join` объединяет две строки и возвращает подстроку, если длина больше максимальной. Кроме того, она обрабатывает проверки `s1` и `s2` и проверяет, возвращает ли вызов `concatenate` ошибку.

С точки зрения реализации функциональности все сделано правильно. Но выстраивание ментальной модели, охватывающей все различные случаи, скорее всего, будет непростой задачей. Почему? Из-за количества вложенных уровней.

Посмотрим на код, выполняющий ту же функцию, но реализованный по-другому:

```
func join(s1, s2 string, max int) (string, error) {
    if s1 == "" {
        return "", errors.New("s1 is empty")
    }
    if s2 == "" {
        return "", errors.New("s2 is empty")
    }
    concat, err := concatenate(s1, s2)
    if err != nil {
        return "", err
    }
    if len(concat) > max {
        return concat[:max], nil
    }
    return concat, nil
}

func concatenate(s1 string, s2 string) (string, error) {
    // ...
}
```

Вы, наверное, заметили, что выстраивание ментальной модели в этой новой версии кода требует меньше когнитивного напряжения, хотя код выполняет то же самое, что и раньше. Здесь есть только два вложенных уровня. Как упомянул Мэт Райер (Mat Ryer), эксперт, участвующий в дискуссии подкаста Go Time (https://medium.com/@matryer/line-of-sight-in-code-186dd7cdea88):

> *Выровняйте «счастливый путь» (happy path) по левому краю — так вы сможете быстро просмотреть, что происходит ниже на каком-то одном уровне и увидеть, что на нем ожидаемо выполняется.*

В первой версии выполнения этого упражнения было сложно определить, что из ожидаемого выполняется, из-за вложенных операторов `if/else`. И наоборот, вторая версия требует просмотра вниз первого уровня, чтобы увидеть поток

выполняемых действий, и второго уровня, чтобы увидеть, как обрабатываются пограничные случаи, как показано на рис. 2.1.

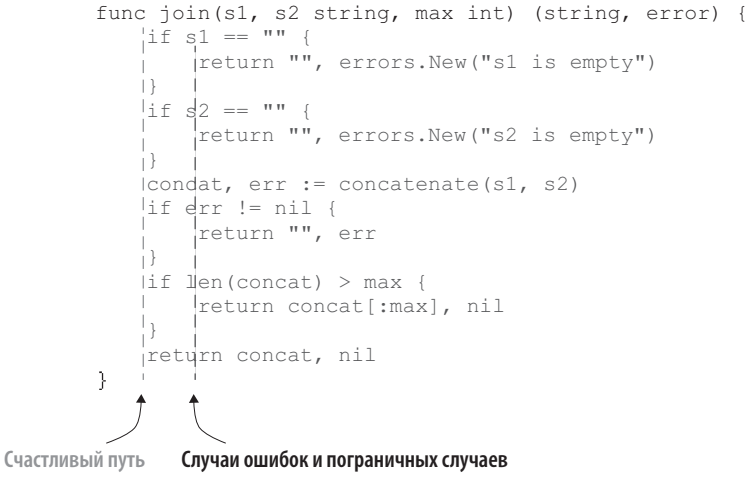


Рис. 2.1. Чтобы понять, что входит в ожидаемый поток выполняемых действий, нужно просмотреть столбец «счастливого пути»

Как правило, чем больше вложенных уровней требует функция, тем сложнее ее читать и понимать. Рассмотрим несколько различных применений этого правила, чтобы оптимизировать код для удобства чтения:

- Когда происходит возврат из блока `if`, следует во всех случаях опускать блок `else`. Например, мы не должны писать:

```
if foo() {
    // ...
        return true
} else {
    // ...
}
```

Вместо этого следует опустить блок `else`, как показано здесь:

```
if foo() {
    // ...
    return true
}
// ...
```

Во второй версии этого фрагмента код, находившийся в блоке `else`, перемещается на верхний уровень, что упрощает его чтение.

- Можно следовать этой логике в случае с путем, не являющимся «счастливым»:

```
if s != "" {
    // ...
} else {
    return errors.New("empty string")
}
```

Здесь пустая переменная `s` определяет путь, не являющимся «счастливым». Поэтому нужно изменить это условие так:

```
if s == "" {  <-- Изменение условия в if
    return errors.New("empty string")
}
// ...
```

Эту версию кода читать легче, потому что она показывает «счастливый» путь на левом краю и уменьшает количество блоков.

Написание читаемого кода — важная задача для каждого разработчика. Стремление уменьшить количество вложенных блоков, выравнивание счастливого пути по левому краю и возврат как можно раньше — это конкретные средства для улучшения читабельности кода.

Далее обсудим типичные ошибки в проектах Go, связанные с неправильным использованием функции инициализации.

2.3. ОШИБКА #3: НЕПРАВИЛЬНО ИСПОЛЬЗОВАТЬ ФУНКЦИЮ ИНИЦИАЛИЗАЦИИ

Иногда в приложениях Go неправильно используются функции инициализации. Потенциальные последствия — трудности в отслеживании и обработке ошибок или сложный в понимании код. Освежим наше представление о том, что такое функция инициализации, а затем рассмотрим, когда ее использование уместно.

2.3.1. Концепция

Функция инициализации (`init`) — это функция, используемая для инициализации состояния приложения. Она не имеет аргументов и не возвращает результата

(функция `func()`). Когда пакет инициализируется, оцениваются все объявления констант и переменных в пакете. Затем выполняются функции инициализации. Вот пример инициализации пакета `main`:

```
package main

import "fmt"

var a = func() int {
    fmt.Println("var")  <-- Исполняется в первую очередь
    return 0
}()

func init() {
    fmt.Println("init")  <-- Исполняется во вторую очередь
}

func main() {
    fmt.Println("main")  <-- Исполняется в последнюю очередь
}
```

Исполнение кода этого примера выведет следующее:

```
var
init
main
```

Функция `init` выполняется при инициализации пакета. В следующем примере мы определяем два пакета — `main` и `redis`, где `main` зависит от `redis`. Сначала main.go из основного пакета:

```
package main

import (
    "fmt"

    "redis"
)

func init() {
    // ...
}

func main() {
    err := redis.Store("foo", "bar")  <-- Указание на зависимость от пакета redis
    // ...
}
```

А затем redis.go из пакета redis:

```
package redis

// imports

func init() {
    // ...
}

func Store(key, value string) error {
    // ...
}
```

Поскольку `main` зависит от `redis`, сначала выполняется функция инициализации в пакете `redis`, затем — в основном пакете, а затем сама функция `main`. На рис. 2.2 показана эта последовательность.

Мы можем определить несколько функций инициализации `init` для каждого пакета. В таком случае последовательность выполнения функции инициализации внутри пакета задается алфавитным порядком исходных файлов. Например, если пакет содержит файл a.go и файл b.go и в обоих содержится функция инициализации, то первой выполняется та из них, что находится в a.go.

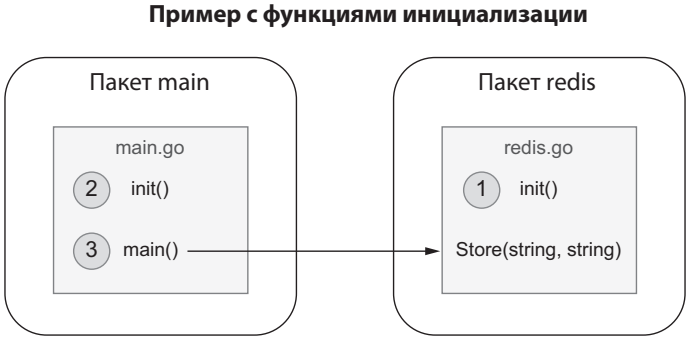


Рис. 2.2. Сначала выполняется функция инициализации init из пакета redis, затем функция инициализации init из пакета main и, наконец, сама функция main

Не следует слишком сильно полагаться на такой порядок выполнения функций инициализации внутри пакета — это может быть опасно, ведь исходные файлы могут быть переименованы и это может повлиять на порядок выполнения функций `init`.

Мы также можем определить несколько функций `init` в одном исходном файле. Например, такой код вполне допустим:

```
package main

import "fmt"

func init() {  <-- Первая функция init
    fmt.Println("init 1")
}
func init() {  <-- Вторая функция init
    fmt.Println("init 2")
}

func main() {
}
```

Первая выполненная функция `init` является первой в исходном порядке. Вот вывод этого кода:

```
init 1
init 2
```

Мы также можем использовать функции инициализации `init` для реализации побочных эффектов. В следующем примере мы определяем пакет `main`, который не имеет сильной зависимости от `foo` (например, нет прямого использования публичной функции — public function). Но в примере требуется, чтобы пакет `foo` был инициализирован. Мы можем сделать это, используя оператор `_`:

```
package main

import (
    "fmt"
    _ "foo"  <-- Импортом foo достигается побочный эффект
)

func main() {
    // ...
}
```

В этом случае пакет `foo` инициализируется перед `main`. Следовательно, функции инициализации (`init`) в `foo` выполняются.

Другой аспект функции `init` в том, что ее нельзя вызвать напрямую, как в следующем примере:

```
package main

func init() {}

func main() {
    init().   ◄──
}
```

Этот код выдаст ошибку компиляции:

```
$ go build .
./main.go:6:2: undefined: init
```

Теперь, когда мы освежили в памяти представление о работе функций `init`, посмотрим, когда следует их использовать.

2.3.2. Когда использовать функции init

Рассмотрим пример уместного использования: удержание пула соединений с базой данных. В функции `init` открывается база данных с помощью `sql.Open`. Мы задаем эту базу данных как глобальную переменную, которую позже могут использовать другие функции:

```
var db *sql.DB

func init() {
    dataSourceName :=
        os.Getenv("MYSQL_DATA_SOURCE_NAME")  ◄──  Переменная среды
    d, err := sql.Open("mysql", dataSourceName)
    if err != nil {
        log.Panic(err)
    }
    err = d.Ping()
    if err != nil {
        log.Panic(err)
    }
    db = d    ◄──  Определяет связь DB с глобальной переменной db
}
```

Мы открываем базу данных, проверяем, можем ли ее пропинговать, а затем связываем ее с глобальной переменной. Что можно сказать о такой реализации? Опишем три ее основных недостатка.

Прежде всего, обработка ошибок в функции инициализации носит ограниченный характер. Действительно, поскольку функция инициализации не выдает сообщения об ошибках, единственный способ сообщить о возможной ошибке — вызвать прерывание по `panic`, что приведет к остановке выполнения

приложения. В нашем примере остановить приложение можно в любом случае, если не удается открыть базу данных. Но решение о такой остановке не обязательно должно приниматься самим пакетом. Возможно, вызывающая сторона предпочла бы реализовать повторную попытку или использовать резервный механизм. В этом случае открытие базы данных в функции инициализации не позволяет клиентским пакетам реализовать свою логику обработки ошибок.

Другой важный недостаток связан с тестированием. Если мы добавим в этот файл тесты, функция инициализации будет выполняться перед запуском тестовых случаев, что не обязательно будет тем, что нужно (например, если добавить юнит-тесты в служебную функцию, которая не требует создания такой связи). Поэтому функция `init` в этом примере усложняет написание юнит-тестов.

Последний недостаток заключается в том, что в примере требуется присвоить пул соединений базы данных глобальной переменной. Глобальные переменные имеют ряд серьезных недостатков, например:

- Внутри пакета глобальные переменные могут изменяться любыми функциями.
- Юнит-тесты могут быть более сложными, поскольку функция, зависящая от глобальной переменной, больше не будет изолирована.

В большинстве случаев следует инкапсулировать переменную, а не сохранять ее глобальной.

По этим причинам предыдущую инициализацию, скорее всего, лучше будет обрабатывать как часть простой старой функции, например:

```
func createClient(dsn string) (*sql.DB, error)
    db, err := sql.Open("mysql", dsn)
    if err != nil {
        return nil, err
    }
    if err = db.Ping(); err != nil {
        return nil, err
    }
    return db, nil
}
```

Принимается имя источника данных и возвращается *sql.DB и ошибка

Возвращается ошибка

Используя эту функцию, мы устранили основные недостатки, о которых говорили ранее, следующим образом:

- Ответственность за обработку ошибок возлагается на вызывающую функцию.

- Появляется возможность создать интеграционный тест для проверки, работает ли эта функция.
- Пул соединений/связей инкапсулирован внутри этой функции.

Нужно ли любой ценой избегать функций инициализации? Не совсем. Есть случаи, когда эти функции могут быть полезны. Например, официальный блог Go (http://mng.bz/PW6w) использует функцию инициализации для настройки статической конфигурации HTTP:

```
func init() {
    redirect := func(w http.ResponseWriter, r *http.Request) {
        http.Redirect(w, r, "/", http.StatusFound)
    }
    http.HandleFunc("/blog", redirect)
    http.HandleFunc("/blog/", redirect)

    static := http.FileServer(http.Dir("static"))
    http.Handle("/favicon.ico", static)
    http.Handle("/fonts.css", static)
    http.Handle("/fonts/", static)
    http.Handle("/lib/godoc/", http.StripPrefix("/lib/godoc/",
        http.HandlerFunc(staticHandler)))
}
```

В этом примере функция инициализации не может стать причиной сбоя (`http.HandleFunc` может вызвать `panic`, но только если обработчик равен `nil`, чего в данном случае нет). При этом нет необходимости создавать какие-либо глобальные переменные, и функция не повлияет на возможные юнит-тесты. Таким образом, этот фрагмент кода представляет собой хороший пример того, где функции инициализации могут оказаться полезны. Подводя итог, мы увидели, что функции инициализации могут привести к некоторым проблемам:

- Они могут ограничивать возможности по обработке ошибок.
- Они могут усложнить реализацию тестов (например, понадобится устанавливать внешнюю зависимость, которая в рамках юнит-тестов может и не потребоваться).
- Если инициализация требует, чтобы мы определили какое-то состояние, то это нужно будет сделать через использование глобальных переменных.

Использовать функции инициализации нужно очень внимательно. Но они могут быть полезны в некоторых ситуациях, например при определении статической конфигурации, как мы увидели в этом разделе. В противном случае, как и просто в большинстве случаев, инициализацию следует обрабатывать с помощью специальных функций.

2.4. ОШИБКА #4: ЗЛОУПОТРЕБЛЯТЬ ГЕТТЕРАМИ И СЕТТЕРАМИ

Инкапсуляция данных в программировании означает сокрытие значений или состояния объекта. Геттеры и сеттеры — это средства для включения инкапсуляции путем предоставления экспортированных методов поверх неэкспортированных полей объектов.

В Go нет автоматической поддержки геттеров и сеттеров, как в других языках. Не считается обязательным или идиоматичным использование геттеров и сеттеров для доступа к полям структуры (`struct`). Например, стандартная библиотека реализует структуры, где некоторые поля доступны напрямую, как структура `time.Timer`:

```
timer := time.NewTimer(time.Second)
<-timer.C.    ◄——   C — это поле <– chan Time
```

Мы могли бы даже модифицировать `C` напрямую, хотя это и не рекомендуется (больше не будем получать события). Но этот пример показывает, что стандартная библиотека Go не требует использования геттеров и/или сеттеров, даже когда не надо изменять поле.

С другой стороны, использование геттеров и сеттеров дает некоторые преимущества:

- Они инкапсулируют поведение, связанное с получением данных какого-то поля или присвоением ему значения, что позволяет добавлять новые функции позднее (например, проверку поля, возврат вычисленного значения или обертывание доступа к полю вокруг мьютекса).
- Они скрывают внутреннее представление, давая больше гибкости в определении того, что мы раскрываем.
- Они дают точку перехвата при отладке, когда свойство изменяется во время исполнения, что упрощает отладку.

Если мы сталкиваемся с такими случаями или предвидим возможный вариант использования, гарантируя прямую совместимость, использование геттеров и сеттеров может принести некоторую пользу. Например, если мы используем их с полем `Balance`, мы должны следовать вот этим соглашениям о наименованиях:

- Метод геттера должен называться `Balance` (а не `GetBalance`).
- Метод сеттера должен называться `SetBalance`.

Пример:

```
currentBalance := customer.Balance().  ◄——  Геттер
if currentBalance < 0 {
    customer.SetBalance(0).  ◄——  Сеттер
}
```

Не следует перегружать код геттерами и сеттерами в структурах, если они не приносят никакой пользы. Будьте прагматиками и ищите баланс между эффективностью и соблюдением идиом, которые в других парадигмах программирования иногда считаются непререкаемыми.

Помните, что Go — уникальный язык, созданный исходя из целей достижения многих характеристик, включая простоту. Но если возникнет потребность в геттерах и сеттерах или эта потребность предвидится в будущем, гарантируя при этом «совместимость вперед», в их использовании нет ничего плохого.

Далее обсудим проблему злоупотребления интерфейсами.

2.5. ОШИБКА #5: ЗАГРЯЗНЯТЬ ИНТЕРФЕЙСЫ

Интерфейсы — это один из краеугольных камней языка Go при разработке и структурировании кода. Но как и со многими другими инструментами или концепциями, излишнее их использование становится недостатком. Загрязнение интерфейса (interface pollution) — это перегруз кода ненужными абстракциями, затрудняющими понимание. Это распространенная ошибка разработчиков, переходящих на Go с других языков и имеющих другие привычки. Прежде чем углубиться в тему, освежим знания об интерфейсах в Go. Затем посмотрим, когда использование интерфейсов уместно, а когда это становится загрязнением кода.

2.5.1. Концепции

Интерфейс предоставляет способ задать поведение объекта. Мы используем интерфейсы для создания общих абстракций, которые могут быть реализованы несколькими объектами. Интерфейсы в Go реализуются неявно. В языке нет явного ключевого слова (например, `implements`), которое бы показывало, что объект X реализует интерфейс Y.

Чтобы понять, что делает интерфейсы такими мощными инструментами, рассмотрим два популярных интерфейса из стандартной библиотеки: `io.Reader` и `io.Writer`. Пакет `io` предоставляет абстракции для примитивов ввода/вывода.

Среди этих абстракций `io.Reader` относится к чтению данных из источника данных, а `io.Writer` — к записи данных в нужное место, как показано на рис. 2.3.

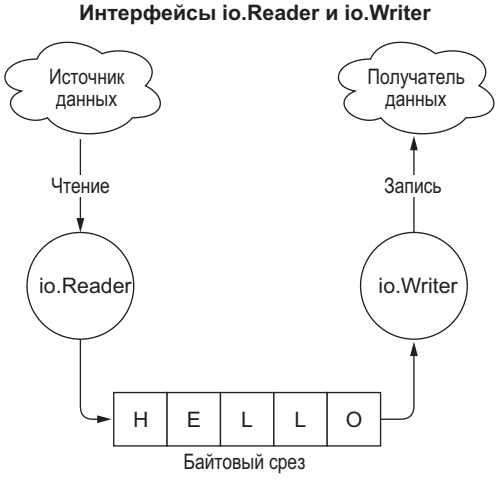


Рис. 2.3. io.Reader читает из источника данных и заполняет байтовый срез, а io.Writer записывает из байтового среза в нужное место

`io.Reader` содержит в себе только метод `Read`:

```
type Reader interface {
    Read(p []byte) (n int, err error)
}
```

Пользовательские реализации интерфейса `io.Reader` должны принимать байтовый срез, заполняя его своими данными и возвращая либо количество прочитанных байтов, либо ошибку.

Со своей стороны, `io.Writer` определяет единственный метод — `Write`:

```
type Writer interface {
    Write(p []byte) (n int, err error)
}
```

Пользовательские реализации `io.Writer` должны записывать данные, поступающие из среза, в их получатель и возвращать либо количество записанных байтов, либо ошибку. Поэтому оба интерфейса представляют собой фундаментальные абстракции:

- `io.Reader` считывает данные из источника.
- `io.Writer` записывает данные в их получатель.

В чем смысл наличия этих двух интерфейсов в языке? Какой смысл в создании этих абстракций?

Предположим, нужно реализовать функцию, которая должна копировать содержимое одного файла в другой. Можно создать специальную функцию, которая бы в качестве входных данных принимала два файла `*os.Files`. Или создать более общую функцию, используя абстракции `io.Reader` и `io.Writer`:

```
func copySourceToDest(source io.Reader, dest io.Writer) error {
    // ...
}
```

Эта функция будет работать с параметрами `*os.File` (поскольку `*os.File` реализует как `io.Reader`, так и `io.Writer`) и любым другим типом, реализующим эти интерфейсы. Например, мы могли бы создать свой собственный `io.Writer`, который пишет в базу данных, и код остался бы прежним. Это увеличивает универсальность функции, и следовательно, возможность ее повторного использования.

Кроме того, написание юнит-теста для этой функции проще, потому что вместо обработки файлов можно использовать пакеты строк и байтов, которые предоставляют полезные реализации:

```
func TestCopySourceToDest(t *testing.T) {
    const input = "foo"
    source := strings.NewReader(input)   <-- Создает io.Reader
    dest := bytes.NewBuffer(make([]byte, 0))   <-- Создает io.Writer

    err := copySourceToDest(source, dest)   <-- Вызывает copySourceToDest
    if err != nil {                             из *strings.Reader и *bytes.Buffer
        t.FailNow()
    }

    got := dest.String()
    if got != input {
        t.Errorf("expected: %s, got: %s", input, got)
    }
}
```

В этом примере источником (`source`) является `*strings.Reader`, а назначением (`dest`) — `*bytes.Buffer`. Мы тестируем поведение функции `copySourceToDest` без создания каких-либо файлов.

При проектировании интерфейсов помните о степени детализации (то есть сколько методов содержится в интерфейсе). Известная среди Go-разработчиков присказка (https://www.youtube.com/watch?v=PAAkCSZUG1c&t=318s) говорит, насколько большим должен быть интерфейс:

Чем больше интерфейс, тем слабее абстракция.

Роб Пайк (Rob Pike)

Добавление методов к интерфейсу может снизить возможности по его повторному использованию.

`io.Reader` и `io.Writer` — мощные абстракции, поскольку их невозможно сделать еще проще. Кроме того, можно комбинировать детализированные интерфейсы для создания абстракций более высокого уровня. Так обстоит дело с `io.ReadWriter`, который сочетает в себе функции чтения и записи:

```
type ReadWriter interface {
    Reader
    Writer
}
```

> **ПРИМЕЧАНИЕ** Как сказал Эйнштейн, «все нужно делать как можно проще, но не проще этого». Применительно к интерфейсам это означает, что поиск идеальной детализации интерфейса не обязательно должен быть простым процессом.

Рассмотрим распространенные случаи, когда использование интерфейсов уместно.

2.5.2. Когда использовать интерфейсы

Когда следует создавать интерфейсы в Go? Рассмотрим три конкретных сценария, когда считается, что интерфейсы могут быть полезны. Обратите внимание, что цель состоит не в том, чтобы дать исчерпывающие рекомендации: чем больше примеров я бы добавил, тем в большей степени они зависели бы от контекста. Но эти три случая дают общее представление о вопросе:

- Общее поведение.
- Снижение связанности.
- Ограничение поведения.

Общее поведение

Первый вариант, который мы обсудим, — это использование интерфейсов, когда несколько типов реализуют общее поведение. Тогда можно заключить

это поведение внутрь какого-то интерфейса. В стандартной библиотеке много таких примеров. Например, сортировка какой-либо коллекции может быть разложена на три действия:

- Получение данных о количестве элементов в коллекции.
- Сообщение о том, должен ли один элемент быть размещен перед другим.
- Перестановка двух элементов.

В пакет `sort` добавляется следующий интерфейс:

```
type Interface interface {
    Len() int  ◄——  Число элементов
    Less(i, j int) bool   ◄——  Сравнение двух элементов
    Swap(i, j int)  ◄——  Перестановка двух элементов
}
```

Этот интерфейс имеет большой потенциал для переиспользования, поскольку включает в себя общее поведение для сортировки любой проиндексированной коллекции.

Можно найти десятки реализаций пакета `sort`. Если в какой-то момент мы имеем дело, например, с набором целых чисел и хотим его отсортировать, будет ли нас интересовать то, как это может быть реализовано? Важен ли алгоритм сортировки: сортировка слиянием или быстрая сортировка? Во многих случаях это неважно. От способа сортировки можно абстрагироваться, и здесь мы зависим только от `sort.Interface`.

Нахождение правильной абстракции для факторизации поведения также может принести много пользы. Так, в пакете `sort` предоставляются служебные функции, которые используют `sort.Interface`: например, проверка того, была ли коллекция уже отсортирована. Например,

```
func IsSorted(data Interface) bool {
    n := data.Len()
    for i := n – 1; i > 0; i-- {
        if data.Less(i, i-1) {
            return false
        }
    }
    return true
}
```

`sort.Interface` — правильный уровень абстракции, и это делает его очень ценным.

Рассмотрим следующий случай, когда полезно использование интерфейсов.

Снижение связанности (decoupling)

Еще один важный сценарий — отделение кода от его реализации. Если мы полагаемся на абстракцию вместо конкретной реализации, сама реализация может быть заменена на другую без необходимости менять код. Это и есть принцип подстановки Лисков (буква L в принципах SOLID Роберта Мартина).

Одно из преимуществ снижения связанности может относиться к юнит-тестам. Предположим, мы хотим реализовать метод `CreateNewCustomer`, который создает нового потребителя и сохраняет его. Мы решили полагаться непосредственно на конкретную реализацию (скажем, на структуру `mysql.Store`):

```
type CustomerService struct {
    store mysql.Store    <-- Зависит от конкретного способа реализации
}

func (cs CustomerService) CreateNewCustomer(id string) error {
    customer := Customer{id: id}
    return cs.store.StoreCustomer(customer)
}
```

А что будет, если мы захотим протестировать этот метод? Поскольку `customerService` использует реальную реализацию для хранения `Customer`, нужно протестировать его с помощью интеграционных тестов, что требует запуска экземпляра MySQL (если только мы не используем альтернативный метод `go-sqlmock`, но эта тема выходит за рамки данного раздела). Хотя интеграционные тесты полезны, это не всегда то, что мы хотим делать. Для большей гибкости нужно отвязать `CustomerService` от фактической реализации. Сделать это можно через интерфейс:

```
type customerStorer interface {    <-- Создается абстракция хранилища
    StoreCustomer(Customer) error
}

type CustomerService struct {
    storer customerStorer    <-- Отвязывает CustomerService от фактической реализации
}

func (cs CustomerService) CreateNewCustomer(id string) error {
    customer := Customer{id: id}
    return cs.storer.StoreCustomer(customer)
}
```

Сохранение созданного потребителя в базе теперь осуществляется через интерфейс, что дает бóльшую гибкость в тестировании метода. Например, мы можем:

- использовать конкретную реализацию в интеграционных тестах;

- применять в юнит-тестах имитации (моки) или любые другие тестовые дублеры;
- делать и то и другое.

Обсудим третий сценарий: ограничение поведения.

Ограничение поведения

Третий сценарий на первый взгляд может показаться контринтуитивным. Речь идет об ограничении типа определенным поведением. Представим, что мы реализуем пользовательский конфигурационный пакет для работы с динамической конфигурацией. Мы создаем специальный контейнер для конфигураций `int` с помощью структуры `IntConfig`, в которой определены два метода: `Get` и `Set`. Вот как будет выглядеть такой код:

```
type IntConfig struct {
    // ...
}

func (c *IntConfig) Get() int {
    // Получить конфигурацию
}

func (c *IntConfig) Set(value int) {
    // Обновить конфигурацию
}
```

Теперь предположим, что мы получили `IntConfig`, который содержит в себе определенную конфигурацию, например какое-то пороговое значение. Но в нашем коде нас интересует только получение значения этой конфигурации, и мы хотим предотвратить его обновление. Как мы можем обеспечить, чтобы семантически эта конфигурация была доступна только для чтения, если мы не хотим изменять пакет конфигурации? Ответ: создав абстракцию, которая ограничивает поведение только получением значения конфигурации:

```
type intConfigGetter interface {
    Get() int
}
```

Тогда в коде можно указать только `intConfigGetter` вместо конкретной реализации:

```
type Foo struct {
    threshold intConfigGetter
}
```

```
func NewFoo(threshold intConfigGetter) Foo {  ◄——  Вводится геттер конфигурации
    return Foo{threshold: threshold}
}

func (f Foo) Bar() {
    threshold := f.threshold.Get()  ◄——  Чтение конфигурации
// ...
}
```

В этом примере геттер конфигурации внедряется в фабричный метод `NewFoo`. Он не влияет на потребителя этой функции, поскольку он по-прежнему может передавать структуру `IntConfig` по мере реализации `intConfigGetter`. Затем в методе `Bar` можно только прочитать конфигурацию, но не изменить ее. Поэтому мы также можем использовать интерфейсы, чтобы ограничить тип определенным поведением, например, если нужно соблюсти семантику.

Мы рассмотрели три возможных сценария использования интерфейсов, в которых они считаются полезными: выделение общего поведения, некоторое снижение связанности и ограничение типа определенным поведением. Это не исчерпывающий список, но он дает общее представление о том, когда интерфейсы в Go полезны.

Закончим этот раздел обсуждением проблем загрязнения интерфейса.

2.5.3. Загрязнение интерфейса

Злоупотребление интерфейсами в проектах на Go — частое явление. Возможно, у разработчика, который этим грешит, был опыт работы с C# или с Java и он счел естественным создавать интерфейсы, а не конкретные типы. Но в Go все должно работать не так.

Интерфейсы полезны для создания абстракций. И главное предостережение при знакомстве программиста с абстракциями — это помнить, что абстракции *нужно открывать, а не создавать*. Это означает, что мы не должны начинать создавать абстракции в коде, если для этого нет веской причины. Нужно не конструировать интерфейсы, а ждать возникновения конкретной потребности в них. Иными словами, создавайте интерфейс только тогда, когда он действительно нужен, а не тогда, когда возникает лишь ощущение, что он может понадобиться.

В чем основная проблема, связанная с чрезмерным использованием интерфейсов? Они делают поток кода менее ясным и более сложным. Добавление бесполезного косвенного уровня не приносит никакой пользы, а лишь создает бесполезную абстракцию, затрудняющую чтение, понимание и осмысление

кода. Если нет веской причины для добавления интерфейса и неясно, как этот интерфейс делает код лучше, нужно поставить под сомнение цель создания такого интерфейса. Почему бы не вызвать реализацию какого-либо действия напрямую?

> **ПРИМЕЧАНИЕ** При вызове метода через интерфейс мы можем столкнуться с оверхедом производительности. Требуется поиск в структуре данных хеш-таблицы, чтобы найти конкретный тип, на который указывает интерфейс. Но это не проблема во многих контекстах, поскольку оверхед минимален.

Следует быть очень осторожными при создании абстракций в коде: их следует обнаруживать, а не создавать. Для разработчиков характерно чрезмерно усложнять код в попытках угадать идеальный уровень абстракции. Этого следует избегать, поскольку в большинстве случаев в результате код «загрязняется» ненужными абстракциями и становится сложным для чтения.

> *Не программируйте интерфейсы, открывайте их.*
>
> *Роб Пайк*

Не будем пытаться решить проблемы абстрактно, будем решать только то, что нужно сейчас. И последнее, но не менее важное: если вы не понимаете, как какой-то интерфейс улучшает код, то следует подумать о его удалении для упрощения кода.

В следующем разделе продолжим эту тему и рассмотрим связанную с интерфейсами распространенную ошибку: создание интерфейсов на стороне производителя (producer).

2.6. ОШИБКА #6: ИНТЕРФЕЙСЫ НА СТОРОНЕ ПРОИЗВОДИТЕЛЯ

В предыдущем разделе мы поговорили о том, когда использование интерфейсов оправданно. Но Go-разработчики часто неправильно понимают другой вопрос: где должен жить интерфейс?

Прежде чем углубиться в эту тему, удостоверимся, что термины этого раздела вам понятны:

- *Сторона производителя (Producer)* — интерфейс, определенный в том же пакете, что и конкретная реализация (рис. 2.4).
- *Сторона потребителя (Consumer)* — интерфейс, определенный во внешнем пакете, где он используется (рис. 2.5).

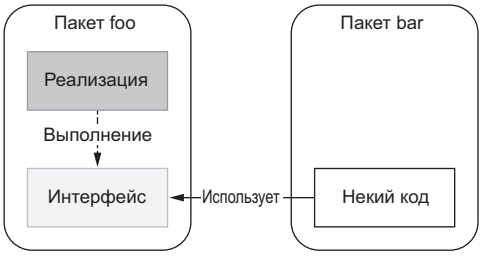


Рис. 2.4. Интерфейс определяется вместе с конкретной реализацией пакета

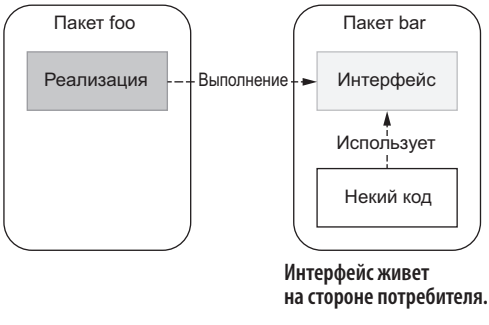


Рис. 2.5. Интерфейс определяется там, где он и используется

Часто можно увидеть, как разработчики создают интерфейсы на стороне производителя наряду с конкретной реализацией. Этот программный дизайн привычен для разработчиков, имеющих опыт работы с C # или с Java. Но в Go в большинстве случаев так делать не следует.

Обсудим пример: создадим специальный пакет для хранения и извлечения данных о потребителях. Мы решаем, что все вызовы в том же пакете должны проходить через следующий интерфейс:

```
package store

type CustomerStorage interface {
    StoreCustomer(customer Customer) error
    GetCustomer(id string) (Customer, error)
    UpdateCustomer(customer Customer) error
```

```
    GetAllCustomers() ([]Customer, error)
    GetCustomersWithoutContract() ([]Customer, error)
    GetCustomersWithNegativeBalance() ([]Customer, error)
}
```

Можно подумать, что есть веские причины для создания этого интерфейса и предоставления доступа к нему на стороне производителя. Возможно, это хороший способ отвязать код потребителя от фактической реализации. Или, возможно, мы стараемся предвидеть, что это поможет потребителям в создании тестовых дублеров. Какой бы ни была причина, в Go это не лучшая практика.

Интерфейсы в Go реализованы неявно, что обычно меняет правила игры по сравнению с языками с явной реализацией. В большинстве случаев подход, которому стоит следовать, аналогичен тому, что мы описали в предыдущем разделе: *абстракции следует открывать, а не создавать*. Это означает, что производитель не должен навязывать определенную абстракцию всем потребителям. Вместо этого потребитель должен решить, нужна ли ему какая-либо форма абстракции, а затем определить наилучший уровень абстракции для своих нужд.

В предыдущем примере один из потребителей не будет заинтересован в отвязывании своего кода. Возможно, другой потребитель захочет отвязать свой код, но его интересует только метод `GetAllCustomers`. Тогда он может создать интерфейс только одним методом, ссылаясь на структуру `Customer` из внешнего пакета:

```
package client

type customersGetter interface {
    GetAllCustomers() ([]store.Customer, error)
}
```

Исходя из организации пакетов, результат этого показан на рис. 2.6. Несколько замечаний:

- Поскольку интерфейс `customersGetter` используется только в пакете `client`, он может остаться неэкспортированным.
- На рисунке это выглядит как циклические зависимости. Но зависимости от `store` к `client` нет, поскольку интерфейс реализован неявно. Поэтому такой подход не всегда возможен в языках с явной реализацией.

Суть состоит в том, что пакет `client` теперь может определить для своих нужд наиболее точную абстракцию (в этом примере есть только один метод). Это связано с концепцией принципа разделения интерфейса (I — ISP — в SOLID),

которая гласит, что ни один потребитель не должен зависеть от методов, которые он не использует. И в этом случае лучший подход — разместить конкретную реализацию на стороне производителя, дать к ней доступ и позволить потребителю решить, как ее использовать и нужна ли вообще здесь абстракция.

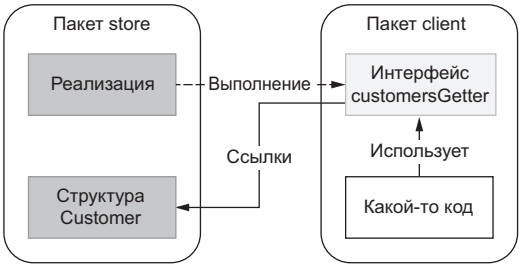


Рис. 2.6. Пакет client определяет необходимую ему абстракцию, создавая собственный интерфейс

Для полноты изложения отметим, что подход интерфейсов на стороне производителя иногда используется в стандартной библиотеке. Например, пакет `encoding` определяет интерфейсы, реализованные другими субпакетами, такими как `encoding/json` или `encoding/binary`. Является ли пакет `encoding` неверным с этой точки зрения? Точно нет. В этом случае абстракции, определенные в пакете `encoding`, используются во всей стандартной библиотеке, и разработчики языка знали, что предварительное создание этих абстракций полезно. Мы вернулись к обсуждению предыдущего раздела: не создавайте абстракцию, если вы просто думаете, что она может быть полезна в будущем, или не можете доказать, что она будет действительно нужна.

В большинстве случаев интерфейс должен жить на стороне потребителя. Но в определенных контекстах (например, когда мы твердо знаем, а не просто предвидим, что абстракция будет полезна для потребителей) можно сделать его на стороне производителя. В этом случае мы должны стремиться к тому, чтобы она была минимальной, что увеличивало бы потенциал ее переиспользования и делало ее легко компонуемой.

Продолжим обсуждение интерфейсов в контексте сигнатур функций.

2.7. ОШИБКА #7: ВОЗВРАТ ИНТЕРФЕЙСОВ

При разработке сигнатуры функции может потребоваться вернуть либо интерфейс, либо конкретную реализацию. Разберемся, почему возврат интерфейса во многих случаях считается в Go плохой практикой.

Мы только что объяснили, почему интерфейсы вообще могут жить на стороне потребителя. На рис. 2.7 показано, что произойдет с точки зрения зависимостей, если функция вернет интерфейс вместо структуры. Это приводит к проблемам.

Рассмотрим два пакета:

- `client`, который содержит интерфейс `Store`.
- `store`, который содержит реализацию `Store`.

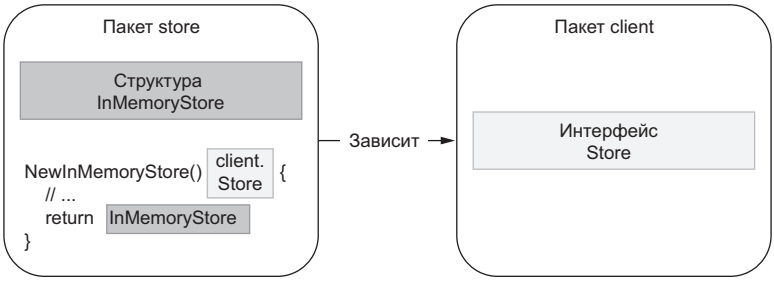


Рис. 2.7. Есть зависимость пакета store от пакета client

В пакете `store` мы определяем структуру `InMemoryStore`, реализующую интерфейс `Store`. Мы также создаем функцию `NewInMemoryStore` для возврата интерфейса `Store`. При таком дизайне есть зависимость пакета реализации от пакета потребителя, и это может показаться странным.

Например, пакет `client` больше не может вызывать функцию `NewInMemoryStore`, в противном случае возникла бы циклическая зависимость. Возможным решением может быть вызов этой функции из другого пакета и внедрение реализации `Store` в `client`. Однако обязанность сделать это означает, что такой дизайн должен быть оспорен.

Что произойдет, если структуру `InMemoryStore` будет использовать другой потребитель? Тогда, возможно, захотелось бы переместить интерфейс `Store` в другой пакет или обратно в пакет реализации, но мы уже обсуждали, почему во многих случаях это плохая идея. Похоже, что этот код с душком.

Возврат интерфейса, как правило, ограничивает гибкость, поскольку мы заставляем всех потребителей использовать один конкретный тип абстракции.

В большинстве случаев можно черпать вдохновение из закона Постеля (Postel's law, https://datatracker.ietf.org/doc/html/rfc761):

> *Будьте консервативны в том, что вы делаете, и либеральны в том, что принимаете от других.*
>
> *Transmission Control Protocol (TCP, протокол управления передачей)*

Если применить эту идиому к Go, то это будет означать:

- возврат структур вместо интерфейсов;
- допущение использования интерфейсов, если это возможно.

Конечно, есть и исключения. Разработчики знают, что правила никогда не выполняются на 100 %. Самое важное из них касается типа `error` — интерфейса, возвращаемого многими функциями. Можно изучить и другое исключение в стандартной библиотеке с пакетом `io`:

```
func LimitReader(r Reader, n int64) Reader {
    return &LimitedReader{r, n}
}
```

В этом примере функция возвращает экспортированную структуру `io.LimitedReader`. Но сигнатура функции — это интерфейс, `io.Reader`. В чем причина нарушения правила, которое мы обсуждали до сих пор? `io.Reader` — это предварительная абстракция. Это не тот уровень, который определяется клиентами, а навязываемый разработчиками языка, которые заранее знали, что этот уровень абстракции будет полезен (например, с точки зрения возможности переиспользования и компоновки).

В большинстве случаев возвращать лучше не интерфейсы, а конкретные реализации. В противном случае дизайн будет усложненным из-за зависимостей пакетов, а гибкость — ограниченной, поскольку всем клиентам придется использовать одну и ту же абстракцию. Этот вывод аналогичен предыдущим разделам: если мы четко знаем (а не просто предполагаем), что абстракция будет полезна для потребителей, то можем подумать о возврате интерфейса. В противном случае мы не должны навязывать использование абстракций; необходимость их использования должна быть «обнаружена» клиентами. Если клиенту по какой-либо причине нужно абстрагировать реализацию, он все равно сможет сделать это на клиентской стороне.

В следующем разделе обсудим распространенную ошибку, связанную с использованием any.

2.8. ОШИБКА #8: ANY НЕ ГОВОРИТ НИ О ЧЕМ

В Go тип интерфейса, который определяет нулевые методы, известен как пустой интерфейс, `interface{}`. В Go 1.18 предварительно объявленный тип `any` стал чем-то вроде псевдонима для пустого интерфейса, поэтому во всех случаях `interface{}` может быть заменен на `any`. Во многих случаях `any` можно считать чрезмерным обобщением, и как считает Роб Пайк, `any` не передает никаких смыслов (https://www.youtube.com/watch?v=PAAkCSZUG1c&t=7m36s). Напомним о некоторых основных понятиях, а затем обсудим потенциальные проблемы с `any`.

Тип `any` может содержать любой тип значения:

```
func main() {
    var i any

    i = 42          <-- Тип int
    i = "foo"       <-- Тип string
    i = struct {    <-- Структура
        s string
    }{
        s: "bar",
    }
    i = f           <-- Функция

    _ = I           <-- Присвоение значения пустому идентификатору, чтобы пример мог скомпилироваться
}

func f() {}
```

При присвоении значению типа `any` мы теряем всю информацию о типе, что требует подтверждения типа (type assertion), чтобы получить что-либо полезное из переменной `i`, как в предыдущем примере. Посмотрим другой пример, где использование `any` не совсем точно. В нем мы реализуем структуру `Store` и скелет двух методов, `Get` и `Set`. Мы используем эти методы для хранения различных типов структур, `Customer` и `Contract`:

```
package store

type Customer struct{
    // какой-то код
}
type Contract struct{
```

```
    // какой-то код
}

type Store struct{}

func (s *Store) Get(id string) (any, error) {   ◄—— Возвращает any
    // ...
}

func (s *Store) Set(id string, v any) error {   ◄—— Принимает any
    // ...
}
```

Хотя в коде `Store` нет ничего ошибочного с точки зрения компиляции, следует остановиться и подумать о сигнатурах методов. Поскольку мы принимаем и возвращаем аргументы `any`, методам не хватает выразительности. Если другим разработчикам потребуется использовать структуру `Store`, им придется покопаться в документации или в коде, чтобы понять, как использовать эти методы. Следовательно, принятие или возврат типа `any` не передает значимой информации. Поскольку во время компиляции нет защиты, ничто не мешает вызывающей функции вызвать эти методы с любым типом данных, например `int`:

```
s := store.Store{}
s.Set("foo", 42)
```

Используя `any`, мы теряем некоторые преимущества Go как языка со статической типизацией. Следует избегать типа `any` и делать сигнатуры максимально явными. Что касается нашего примера, это может означать дублирование методов `Get` и `Set` для каждого типа:

```
func (s *Store) GetContract(id string) (Contract, error) {
    // ...
}

func (s *Store) SetContract(id string, contract Contract) error {
    // ...
}

func (s *Store) GetCustomer(id string) (Customer, error) {
    // ...
}

func (s *Store) SetCustomer(id string, customer Customer) error {
    // ...
}
```

Здесь методы достаточно выразительны, что снижает риск непонимания. Наличие большего количества методов не всегда проблема, поскольку клиенты также могут создавать свои собственные абстракции с помощью какого-либо

интерфейса. Например, если клиента интересуют только методы `Contract`, он может написать что-то вроде этого:

```
type ContractStorer interface {
    GetContract(id string) (store.Contract, error)
    SetContract(id string, contract store.Contract) error
}
```

В каких случаи `any` полезен? Посмотрим на стандартную библиотеку и два примера, где функции или методы принимают аргументы `any`. Первый пример находится в пакете `encoding/json`. Поскольку мы можем маршалировать любой тип, функция `Marshal` принимает аргумент `any`:

```
func Marshal(v any) ([]byte, error) {
    // ...
}
```

Другой пример можно найти в пакете `database/sql`. Если запрос параметризован (например, `SELECT * FROM FOO WHERE id = ?`), параметры могут быть любыми. Следовательно, он также использует аргументы `any`:

```
func (c *Conn) QueryContext(ctx context.Context, query string,
args ...any) (*Rows, error) {
// ...
}
```

Таким образом, `any` может быть полезен, если есть реальная необходимость принять или вернуть любой возможный тип (например, когда дело доходит до маршалинга или форматирования). В общем, мы должны любой ценой избегать чрезмерного обобщения своего кода. Возможно, иногда небольшое дублирование кода будет приветствоваться, если это улучшает другие аспекты, например выразительность.

Обсудим другой тип абстракций: дженерики.

2.9. ОШИБКА #9: ПУТАНИЦА В ИСПОЛЬЗОВАНИИ ДЖЕНЕРИКОВ

В Go 1.18 в язык добавлены дженерики. Это позволяет писать код с типами, которые можно указать позже и создавать при необходимости. При этом может возникнуть путаница, когда использовать дженерики. Мы опишем общую концепцию дженериков в Go, а затем покажем, когда их полезно использовать, а когда нет.

2.9.1. Концепция

Следующая функция извлекает все ключи из типа `map[string]int`:

```
func getKeys(m map[string]int) []string {
    var keys []string
    for k := range m {
        keys = append(keys, k)
    }
    return keys
}
```

Что, если мы захотим использовать аналогичную функцию для другого типа карт, например для `map[int]string`? До появления дженериков у разработчиков было несколько вариантов: использование генерации кода, отражение или дублирование кода. Например, мы могли бы написать две функции, по одной для каждого типа карт, или даже попытаться расширить возможности `getKeys`, чтобы он принимал разные типы карт:

```
func getKeys(m any) ([]any, error)
    switch t := m.(type) {
    default:
        return nil, fmt.Errorf("unknown type: %T", t)
    case map[string]int:
        var keys []any
        for k := range t {
            keys = append(keys, k)
        }
        return keys, nil
    case map[int]string:
        // Скопировать логику извлечения
    }
}
```

Принимает и возвращает аргументы типа any

Обрабатывает ошибки времени выполнения, если тип еще не реализован

В этом примере несколько проблем. Прежде всего, увеличивается шаблонный код. Если нужно добавить еще какой-то случай, придется дублировать цикл `range`. При этом функция теперь принимает тип `any`, это означает, что мы теряем некоторые преимущества Go как типизированного языка. Проверка того, поддерживается ли этот тип, проводится во время выполнения программы, а не во время ее компиляции. Поэтому нужно вернуть ошибку, если представленный тип неизвестен. Наконец, поскольку тип ключа может быть либо `int`, либо `string`, мы обязаны вернуть срез типа `any`, чтобы выделить типы ключей. Такой подход увеличивает нагрузку на вызывающую функцию, поскольку клиенту может также потребоваться выполнить проверку типа ключей или дополнительное преобразование. Благодаря дженерикам можно сделать рефакторинг этого кода, используя параметры типа.

Параметры типа — это общие типы, которые можно использовать с функциями и типами. Например, аргументом следующей функции является параметр типа:

```
func foo[T any](t T) {  ◄—— T — это параметр типа
    // ...
}
```

При вызове `foo` мы передаем туда аргумент типа `any`. Передача аргумента типа называется *инстанцированием* (*instantiation*), и эта работа выполняется во время компиляции. Это позволяет сохранить безопасность типов как часть основных возможностей языка и избежать оверхеда во время выполнения.

Вернемся к функции `getKeys` и воспользуемся параметрами типа для написания универсальной версии, которая будет принимать карты любого типа:

```
func getKeys[K comparable, V any](m map[K]V) []K {  ◄—— Ключи — типа comparable,
    var keys []K  ◄—— Создается срез keys                а V — типа any
    for k := range m {
        keys = append(keys, k)
    }
    return keys
}
```

Для обработки карты мы определяем два вида параметров типа. Прежде всего значения могут быть типа `any`: `V any`. Однако в Go ключи карты не могут быть типа `any`. Например, мы не можем использовать срезы:

```
var m map[[]byte]int
```

Этот код приводит к ошибке компиляции: `invalid map key type []byte`. Чтобы не принимать любой тип ключа, мы обязаны ограничить аргументы типа, чтобы тип ключа соответствовал определенным требованиям. Здесь требуется, чтобы тип ключа был сравним (можно использовать `==` или `!=`). Следовательно, мы определили `K` как `comparable`, а не как `any`.

Ограничение аргументов типа для соответствия определенным требованиям называется *constraint* (слово и означает *ограничение*) — это тип интерфейса, который может содержать:

- набор поведения (методов);
- произвольные типы.

Последнее рассмотрим на конкретном примере. Допустим, мы не хотим принимать какой-либо тип `comparable` для типа ключа `map`. Например, мы

хотим ограничить его типами `int` или `string`. Мы можем определить ограничение так:

```
type customConstraint interface {
    ~int | ~string
}

func getKeys[K customConstraint,
        V any](m map[K]V) []K {
    // Та же реализация
}
```

Определяется пользовательский тип, который ограничивает возможные типы значениями int и string

Изменяется параметр типа K на тип customConstraint

Для начала мы определяем интерфейс `customConstraint`, чтобы ограничить допустимые типы значениями `int` или `string`, используя оператор объединения `|` (мы обсудим использование `~` немного позже). `K` теперь является `customConstraint` вместо `comparable`, как было раньше.

Сигнатура `getKeys` гарантирует, что мы можем вызывать его с картой любого типа данных, но тип ключа должен быть `int` или `string`. Например, на вызывающей стороне:

```
m = map[string]int{
    "one": 1,
    "two": 2,
    "three": 3,
}
keys := getKeys(m)
```

В Go подразумевается, что `getKeys` вызывается с аргументом типа `string`. Предыдущий вызов эквивалентен вот этому:

```
keys := getKeys[string](m)
```

До сих пор мы обсуждали примеры использования дженериков для функций. Но мы также можем использовать дженерики со структурами данных. Например, можно создать связанный список, содержащий значения любого типа. Для этого мы напишем метод `Add`, который добавляет узел:

```
type Node[T any] struct {
    Val T
    next *Node[T]
}

func (n *Node[T]) Add(next *Node[T]) {
    n.next = next
}
```

Используется параметр типа

Инстанцируется функция, принимающая аргумент типа

~int и int

В чем разница между ограничениями аргументов типа (constraint), использующими ~int и int? Использование int ограничивает тип только этим типом, тогда как ~int ограничивает все типы, базовым типом которых является int. Для иллюстрации давайте представим constraint, в котором мы хотели бы ограничить тип любым типом int, реализующим метод String()string:

```
type customConstraint interface {
    ~int
    String() string
}
```

Использование этого constraint'а ограничивает аргументы типа пользовательскими типами. Например,

```
type customInt int

func (i customInt) String() string {
    return strconv.Itoa(int(i))
}
```

Поскольку customInt представляет собой тип int и реализует метод String() string, тип customInt удовлетворяет заданному ограничению (constraint). Но если мы изменим его, чтобы он содержал int вместо ~int, использование customInt приведет к ошибке компиляции, поскольку тип int не выполняет String()string.

В этом примере мы используем параметры типа для определения T и используем оба поля в Node. Что касается метода, инстанцируется принимающая функция. Действительно, поскольку Node является универсальным, он также должен следовать заданному параметру типа.

И последнее, что следует отметить в отношении параметров типа: они не могут использоваться с аргументами метода, а только с аргументами функции или получателями методов. Например, следующий метод не скомпилируется:

```
type Foo struct {}

func (Foo) bar[T any](t T) {}

./main.go:29:15: methods cannot have type parameters
```

Если мы хотим использовать дженерики с методами, получатель должен быть параметром типа.

Рассмотрим конкретные случаи, когда использовать дженерики.

2.9.2. Общие случаи использования и злоупотребления

Когда полезно использовать дженерики? Обсудим несколько распространенных случаев, когда это рекомендуется:

- *Структуры данных.* Мы можем использовать дженерики, чтобы выделить тип элемента, например, если реализуем двоичное дерево, связанный список или кучу.
- *Функции, работающие со срезами, картами и каналами любого типа.* Например, функция объединения двух каналов будет работать с любым типом канала. Следовательно, можно использовать параметры типа, чтобы определить тип канала:

```
func merge[T any](ch1, ch2 <-chan T) <-chan T {
    // ...
}
```

- *Факторизация поведения вместо типов.* Пакет `sort`, например, содержит интерфейс `sort.Interface`, включающий в себя три метода:

```
type Interface interface {
    Len() int
    Less(i, j int) bool
    Swap(i, j int
}
```

Этот интерфейс используется различными функциями: `sort.Ints` или `sort.Float64s`. Используя параметры типа, можно выделить действие по сортировке (например, определив структуру, содержащую срез, и функцию сравнения):

```
type SliceFn[T any] struct {   <-- Используется параметр типа
    S       []T
    Compare func(T, T) bool    <-- Сравниваются два элемента T
}

func (s SliceFn[T]) Len() int           { return len(s.S) }
func (s SliceFn[T]) Less(i, j int) bool { return s.Compare(s.S[i], s.S[j]) }
func (s SliceFn[T]) Swap(i, j int)      { s.S[i], s.S[j] = s.S[j], s.S[i] }
```

Поскольку структура `SliceFn` реализует `sort.Interface`, можно отсортировать предоставленный срез с помощью функции `sort.Sort(sort.Interface)`:

```
s := SliceFn[int]{
    S: []int{3, 2, 1},
    Compare: func(a, b int) bool {
        return a < b
    },
}
sort.Sort(s)
fmt.Println(s.S)
```

```
[1 2 3]
```

Здесь факторизация действия позволяет избежать создания для каждого типа своей функции.

А когда использовать дженерики не рекомендуется?

- *При вызове метода с аргументом типа*. Рассмотрим функцию, которая получает на входе `io.Writer` и вызывает метод `Write`:

  ```
  func foo[T io.Writer](w T) {
      b := getBytes()
      _, _ = w.Write(b)
  }
  ```

  В этом случае использование дженериков не принесет коду никакой пользы. Нужно напрямую сделать значение аргумента `w` равным `io.Writer`.

- *Когда это делает код более сложным*. Дженерики никогда не бывают обязательными, и разработчики Go прекрасно жили без них более десяти лет. Если мы используем дженерики — универсальные функции или структуры — и обнаруживаем, что это не делает код более понятным, то следует пересмотреть свое решение для конкретного случая.

Хотя дженерики и могут быть полезны, будьте осторожны при их использовании. Принцип здесь такой же, как и при использовании интерфейсов. Дженерики вводят некоторую форму абстракции, а нам нужно помнить, что ненужные абстракции только усложняют работу.

Не будем загрязнять код ненужными абстракциями и сосредоточимся на решении конкретных задач. Это означает, что использовать параметры типа не нужно, если на то нет оснований. Подождите, когда надо будет писать шаблонный код, и только тогда рассмотрите возможность использования дженериков.

Далее обсудим возможные проблемы при использовании встраивания типов.

2.10. ОШИБКА #10: НЕ ЗНАТЬ О ВОЗМОЖНЫХ ПРОБЛЕМАХ СО ВСТРАИВАНИЕМ ТИПОВ

При создании структуры Go позволяет встраивать типы. Но иногда это может привести к неожиданному поведению, если мы не понимаем всех последствий такого встраивания. Поговорим, как встраивать типы, что это дает и какие могут быть проблемы.

В Go поле структуры называется встроенным, если оно объявлено без имени. Например,

```
type Foo struct {
    Bar  <-- Встроенное поле
}

type Bar struct {
    Baz int
}
```

В структуре `Foo` тип `Bar` объявлен без связанного имени, следовательно, это встроенное поле.

Мы используем встраивание для продвижения (*promote*) полей и методов встроенного типа. Поскольку `Bar` содержит поле `Baz`, это поле продвигается в `Foo` (рис. 2.8). Таким образом, `Baz` становится доступным из `Foo`:

```
foo := Foo{}
foo.Baz = 42
```

Доступ к `Baz` возможен по двум разным путям: либо по продвигаемому через `Foo.Baz`, либо по номинальному через `Bar`, `Foo.Bar.Baz`. Оба относятся к одному и тому же полю.

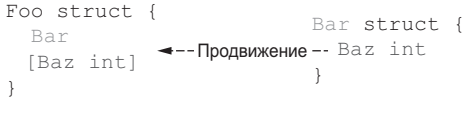


Рис. 2.8. baz продвинут, а поэтому доступен напрямую из S

Теперь, когда мы вспомнили, что такое встроенные типы, давайте рассмотрим пример неправильного их использования. Ниже мы реализуем `struct`, который хранит некоторые данные в памяти, и хотим защитить его от конкурентного доступа с помощью мьютекса:

```
type InMem struct {
    sync.Mutex  ◄——  Встроенное поле
    m map[string]int
}
func New() *InMem {
    return &InMem{m: make(map[string]int)}
}
```

Интерфейсы и встраивание

Встраивание также используется внутри интерфейсов для объединения интерфейса с другими. В следующем примере `io.ReadWriter` состоит из `io.Reader` и `io.Writer`:

```
type ReadWriter interface {
    Reader
    Writer
}
```

То, что описано в этом разделе, относится только ко встроенным полям в структурах.

Мы решили сделать карту неэкспортируемой, чтобы клиенты не могли взаимодействовать с ней напрямую, а только через экспортированные методы. Между тем поле мьютекс встроено. Поэтому мы можем реализовать метод `Get` так:

```
func (i *InMem) Get(key string) (int, bool) {
    i.Lock()  ◄——  Прямой доступ к методу Lock
    v, contains := i.m[key]
    i.Unlock()  ◄——  То же самое относительно метода Unlock
    return v, contains
}
```

Поскольку этот мьютекс встроен, мы можем напрямую обращаться к методам `Lock` и `Unlock` из получателя `i`.

Мы уже говорили, что это пример неправильного использования встраивания типов. Почему так? Поскольку `sync.Mutex` — это встроенный тип, методы `Lock` и `Unlock` будут продвигаться. Поэтому оба метода станут видимыми для внешних потребителей, использующих `InMem`:

```
m := inmem.New()
m.Lock() // ??
```

Такое продвижение, вероятно, — нежелательный эффект. В большинстве случаев мьютекс — это то, что мы хотим инкапсулировать в структуру и сделать невидимым для внешних клиентов. Поэтому здесь не следует делать его встроенным полем:

```
type InMem struct {
    mu sync.Mutex  <-- Указывает, что поле sync.Mutex не является встроенным
    m map[string]int
}
```

Поскольку мьютекс не встроен и не экспортируется, доступ к нему внешних потребителей закрыт.

Рассмотрим другой пример, где встраивание можно считать правильным подходом.

Нужно написать собственный логгер, содержащий `io.WriteCloser` и делающий доступными два метода: `Write` и `Close`. Если бы `io.WriteCloser` не был встроенным, код был бы таким:

```
type Logger struct {
    writeCloser io.WriteCloser
}

func (l Logger) Write(p []byte) (int, error) {
    return l.writeCloser.Write(p)  <-- Перенаправляет вызов на writeCloser
}

func (l Logger) Close() error {
    return l.writeCloser.Close()  <-- Перенаправляет вызов на writeCloser
}

func main() {
    l := Logger{writeCloser: os.Stdout}
    _, _ = l.Write([]byte("foo"))
    _ = l.Close()
}
```

`Logger` должен был предоставить доступ как к методу `Write`, так и к методу `Close`, которые бы *только* перенаправляли вызов на `io.WriteCloser`. Но если теперь сделать поле встроенным, то можно удалить эти методы перенаправления:

```
type Logger struct {
    io.WriteCloser  <-- io.Writer делается встроенным
```

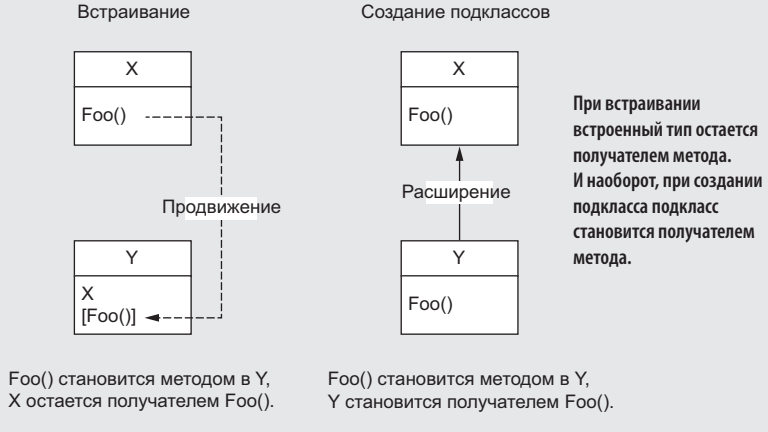
```
}

func main() {
    l := Logger{WriteCloser: os.Stdout}
    _, _ = l.Write([]byte("foo"))
    _ = l.Close()
}
```

Логгер остается для клиентов тем же самым, с двумя экспортированными методами `Write` и `Close`. Но при этом становится возможным не расписывать эти дополнительные методы для простой переадресации вызова. Кроме того, продвижение `Write` и `Close` означает, что `Logger` удовлетворяет интерфейсу `io.Write-Closer`.

Встраивание и создание подклассов в ООП

Отличие встраивания от создания подклассов в ООП иногда может сбивать с толку. Основное отличие между ними связано с идентификацией получателя метода. Посмотрим на следующую иллюстрацию. В левой части представлен тип X, встроенный в Y, тогда как в правой части Y расширяет X.



При встраивании X остается получателем `Foo`. Но при создании подклассов получатель `Foo` становится подклассом — Y. Встраивание связано со структурированием, а не с наследованием.

Какой следует сделать вывод о встраивании типов? Прежде всего оно редко бывает по-настоящему нужно, а это значит, что независимо от сути конкретной

задачи мы, скорее всего, сможем решить ее и без применения встраивания типов. В основном оно используется для удобства: для продвижения поведения.

Если же мы все-таки решаем использовать встраивание типов, то нужно помнить о двух основных ограничениях:

- Не следует его использовать исключительно как синтаксический сахар — для упрощения доступа к полю (например, `Foo.Baz()` вместо `Foo.Bar.Baz()`). Если это единственная причина, то вместо встраивания внутреннего типа лучше использовать поле.
- Оно не должно продвигать данные (поля) или поведение (методы), которые мы хотим скрыть от посторонних глаз: например, если оно позволяет клиентам получить доступ к поведению блокировки, которое должно оставаться приватным для структуры.

ПРИМЕЧАНИЕ Кто-то может возразить, что использование встраивания типов приводит к дополнительным усилиям в сопровождении в контексте экспортируемых структур. И правда, встраивание типа внутрь экспортируемой структуры означает некоторую осторожность по мере использования типа. Например, если мы добавляем во внутренний тип новый метод, то важно убедиться, что он не нарушает ограничения этого типа. Чтобы избежать дополнительных усилий, имеет смысл запретить встраивание типов в публичные структуры.

Осознанное использование встраивания типов с учетом этих ограничений поможет избежать шаблонного кода с дополнительными методами перенаправления. Но важно понимать, что мы не делаем это исключительно в «косметических» целях и не продвигаем элементы, которые должны оставаться скрытыми.

В следующем разделе обсудим общие паттерны для работы с опциональными конфигурациями.

2.11. ОШИБКА #11: НЕ ИСПОЛЬЗОВАТЬ ПАТТЕРН ФУНКЦИОНАЛЬНЫХ ОПЦИЙ

При разработке API может возникнуть вопрос: как быть с опциональными конфигурациями? Эффективное решение этой проблемы может улучшить удобство нашего API. В этом разделе рассматриваются конкретный пример и способы обработки опциональных конфигураций.

Предположим, нужно разработать библиотеку, в которой будет реализована функция для создания HTTP-сервера. Аргументами этой функции будут разные входные данные: адрес и порт. Вот так выглядит скелет функции:

```
func NewServer(addr string, port int) (*http.Server, error) {
    // ...
}
```

Клиенты нашей библиотеки начали пользоваться этой функцией, и до поры до времени они довольны. Но в какой-то момент они начинают жаловаться, что функция несколько ограниченна, в частности, не имеет других входных параметров (например, тайм-аут записи и контекст подключения). Мы отвечаем, что добавление новых параметров функции приводит к проблемам совместимости, вынуждая клиентов изменять способ вызова `NewServer`. Между тем мы хотели бы расширить логику управления портами так (рис. 2.9):

- Если порт не указан, то используется заданный по умолчанию порт.
- Если параметр порта отрицателен, то возвращается ошибка.
- Если параметр порта равен 0, то используется случайный порт.
- В остальных случаях функция использует порт, заданный клиентом.

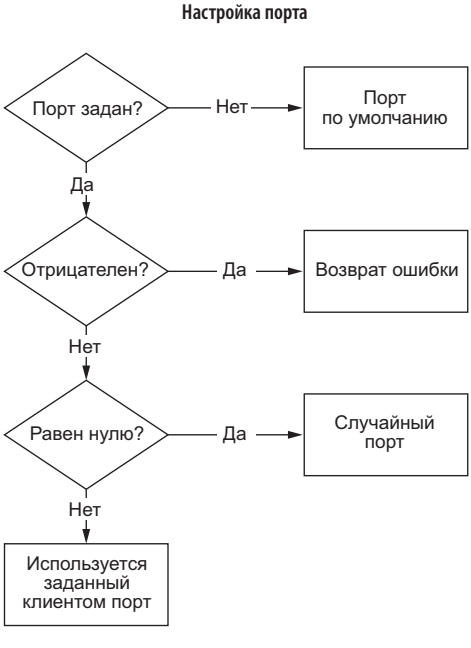


Рис. 2.9. Логика, связанная с опциями значения параметра порта

Как мы можем реализовать эту функцию удобным для API способом? Посмотрим на варианты.

2.11.1. Структура Config

Поскольку Go не поддерживает необязательные параметры в сигнатурах функций, первый возможный подход — использовать структуру конфигурации для передачи того, что обязательно, а что — опционально. Например, обязательные параметры могут задаваться как параметры функции, тогда как опциональные могут обрабатываться в структуре `Config`:

```
type Config struct {
    Port        int
}

func NewServer(addr string, cfg Config) {
}
```

Такое решение устраняет проблему совместимости. Если мы добавим новые опции, они не будут ломаться на стороне клиента. Но этот подход не решает задачу по управлению портами в соответствии с требованиями. Следует иметь в виду, что если поле структуры не задано, его нужно инициализировать нулевым значением:

- 0 для целочисленного параметра;
- для типа с плавающей точкой;
- " " для строки;
- Nil — для срезов, карт, каналов, указателей, интерфейсов и функций.

Поэтому в следующем примере обе структуры эквивалентны друг другу:

```
c1 := httplib.Config{
    Port: 0,  <-- Придает порту начальное значение, равное нулю
}
c2 := httplib.Config{
              <-- Порт отсутствует, поэтому его начальное значение делается равным нулю
}
```

В нашем случае нужно найти способ, как отличить порт, значение которого намеренно установлено как 0, от отсутствующего порта. Одним из вариантов станет обработка всех параметров структуры конфигурации как указателей:

```
type Config struct {
    Port        *int
}
```

С помощью целочисленного указателя мы семантически сможем выделить разницу между значением 0 и отсутствующим значением (нулевым указателем — nil).

Это рабочий вариант, но у него есть несколько недостатков. Прежде всего клиентам может быть неудобно задавать целочисленный указатель. Им придется создать переменную, а затем передавать указатель:

```
port := 0
config := httplib.Config{
    Port: &port,  ◄—— Задается целочисленный указатель
}
```

Ничего страшного, но в целом API становится менее удобным в использовании. Кроме того, чем больше опций мы добавляем, тем сложнее становится код.

Вторым недостатком является то, что клиент, использующий нашу библиотеку с конфигурацией по умолчанию, должен будет передавать пустую структуру:

```
httplib.NewServer("localhost", httplib.Config{})
```

Код выглядит так себе. Те, кто будут его читать, должны будут понять смысл этой магической структуры.

Другой вариант — использовать классический паттерн Строитель, как представлено в следующем разделе.

2.11.2. Паттерн Строитель

Первоначально входивший в состав паттернов проектирования «Банды четырех», *Строитель* обеспечивает гибкое решение различных проблем создания объектов. Конструирование `Config` отделено от самой структуры. Для этого требуется дополнительная структура `ConfigBuilder`, которая получает методы для конфигурирования и создания файла `Config`.

Рассмотрим конкретный пример и то, как он поможет в разработке удобного API, отвечающего всем требованиям, включая управление портами:

```
type Config struct {  ◄—— Структура Config
    Port int
}
```

```
type ConfigBuilder struct {
    port *int
}

func (b *ConfigBuilder) Port(
    port int) *ConfigBuilder {
    b.port = &port
    return b
}

func (b *ConfigBuilder) Build() (Config, error) {
    cfg := Config{}

    if b.port == nil {
        cfg.Port = defaultHTTPPort
    } else {
        if *b.port == 0 {
            cfg.Port = randomPort()
    } else if *b.port < 0 {
        return Config{}, errors.New("port should be positive")
    } else {
        cfg.Port = *b.port
    }
    }
    return cfg, nil
}

func NewServer(addr string, config Config) (*http.Server, error) {
    // ...
}
```

Структура Config builder, содержащая опциональный порт

Открытый (public) метод для настройки порта

Метод Build для создания структуры config

Основная логика, связанная с управлением портами

Структура `ConfigBuilder` содержит конфигурацию клиента. Она предоставляет метод `Port` для настройки порта. Обычно такой метод конфигурирования возвращает сам Строитель, чтобы мы могли использовать цепочку методов (например, `builder.Foo("foo").Bar("bar")`). Он также предоставляет метод `Build`, который содержит логику инициализации значения порта (независимо от того, был ли указатель нулевым — `nil` и т. д.) и возвращает созданную структуру `Config`.

> **ПРИМЕЧАНИЕ** Реализация Строителя возможна различными способами — какого-то единственно верного нет. Например, некоторые предпочитают подход, при котором логика для определения окончательного значения порта находится внутри метода `Port`, а не `Build`. Цель этого раздела — представить обзор Строителя, а не рассмотреть все варианты реализации.

Затем клиент будет использовать созданный на основе Строителя API (мы предполагаем, что поместили код в пакет `httplib`):

```
builder := httplib.ConfigBuilder{}  ◄—— Создается Строитель config
builder.Port(8080)  ◄—— Задается порт
cfg, err := builder.Build()  ◄—— Создается структура config
if err != nil {
    return err

}
server, err := httplib.NewServer("localhost", cfg)  ◄—— Передается структура config
if err != nil {
    return err
}
```

Сначала клиент создает `ConfigBuilder` и использует его для настройки опционного поля, например порта. Затем вызывает метод `Build` и проверяет наличие ошибок. Если все в порядке, то конфигурация передается в `NewServer`.

Такой подход делает управление портами удобнее. Не требуется передавать целочисленный указатель, так как метод `Port` принимает целое число. Но все еще нужно передать структуру `Config`, которая может быть пустой, если клиент хочет использовать конфигурацию по умолчанию:

```
server, err := httplib.NewServer("localhost", nil)
```

Другой недостаток в некоторых ситуациях связан с обработкой ошибок. В языках программирования, где генерируются исключения, методы Строителя, например `Port`, могут генерировать исключения, если входные данные неверны. Если мы хотим сохранить возможность связывать вызовы в цепочки, функция не может возвращать ошибку. Поэтому приходится откладывать проверку в методе `Build`. Если клиент может передавать несколько параметров, но нужно обработать именно тот случай, когда порт недействителен, это усложняет обработку ошибок.

Рассмотрим другой подход, называемый паттерном функциональных опций, который опирается на вариативные аргументы.

2.11.3. Паттерн функциональных опций

Последний подход, который мы обсудим, — это *паттерн функциональных опций* (рис. 2.10). Есть разные реализации, отличающиеся друг от друга небольшими вариациями, но основная идея в следующем:

Неэкспортированная структура содержит конфигурацию: `options`.

Каждая из ее опций представляет собой функцию, которая возвращает ошибку одного и того же типа: `type Option func(options *options)`. Например, `WithPort`

имеет аргумент типа `int`, значение которого соответствует порту, и возвращает тип `Option`, представляющий способ обновления структуры `options`.

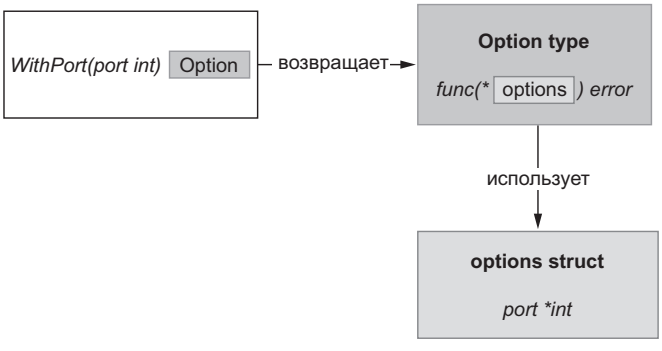


Рис. 2.10. Опция WithPort обновляет окончательную структуру options

Вот реализация структуры `options`, типа `Option` и опции `With-Port` на Go:

```
type options struct {
    port *int
}

type Option func(options *options) error

func WithPort(port int) Option {
    return func(options *options) error {
        if port < 0 {
            return errors.New("port should be positive")
        }
        options.port = &port
        return nil
    }
}
```

Конфигурация структуры

Представление типа функции, который обновляет структуру конфигурации

Функция конфигурации, которая обновляет порт

Здесь `WithPort` возвращает замыкание. *Замыкание (closure)* — это анонимная функция, которая ссылается на переменные вне своего тела, в данном случае на переменную `port`. Замыкание учитывает тип `Option` и реализует логику проверки порта. Каждое поле конфигурации требует создания публичной функции (которую принято начинать с префикса `With`), содержащей аналогичную логику: проверка входных данных при необходимости и обновление структуры конфигурации.

Посмотрим на последнюю часть на стороне провайдера, а именно — как реализован `NewServer`. Мы будем передавать параметры как вариативные аргументы.

Следовательно, нужно перебрать все эти параметры, чтобы видоизменить структуру конфигурации `options`:

```
func NewServer(addr string, opts ...Option) (
    *http.Server, error) {
    var options options
    for _, opt := range opts {
        err := opt(&options)
        if err != nil {
            return nil, err
        }
    }
    // На этом этапе строится структура options struct, которая содержит config
    // Таким образом, мы можем реализовать нашу логику, связанную
    // с конфигурацией порта var port int
    var port int
    if options.port == nil {
        port = defaultHTTPPort
    } else {
        if *options.port == 0 {
            port = randomPort()
        } else {
            port = *options.port
        }
    }
    // ...
}
```

Принимает вариативные аргументы Option

Создается пустая структура options

Производится перебор всех входных опций

Происходит вызов каждой опции, что приводит к изменению общей структуры options

Начнем с создания пустой структуры `options`. Затем мы перебираем каждый аргумент `Option` и выполняем их для видоизменения структуры `options` (имейте в виду, что тип `Option` — это функция). Как только структура `options` создана, можно реализовать окончательную логику управления портами.

Поскольку `NewServer` принимает вариативные аргументы `Option`, клиент может вызывать этот API, передавая несколько параметров после обязательного аргумента адреса. Например,

```
server, err := httplib.NewServer("localhost",
        httplib.WithPort(8080),
        httplib.WithTimeout(time.Second))
```

Но если ему нужна конфигурация по умолчанию, не нужно задавать аргумент (например, пустую структуру, как в предыдущих случаях). Вызов со стороны клиента теперь может выглядеть так:

```
server, err := httplib.NewServer("localhost")
```

Это паттерн функциональных опций. Он предоставляет удобный и дружественный к API способ обработки опций. Хотя Строитель может быть допустимым вариантом, у него есть некоторые незначительные недостатки, которые делают паттерн функциональных опций идиоматическим способом решения этой проблемы в Go. Этот паттерн используется в разных библиотеках Go, например gRPC.

В следующем разделе поговорим о неорганизованности проекта.

2.12. ОШИБКА #12: НЕОРГАНИЗОВАННОСТЬ ПРОЕКТА

Организация проекта на Go — непростая задача. Поскольку Go предоставляет большую свободу при создании пакетов и модулей, не так просто выделить действительно лучшие практики. В этом разделе обсудим распространенный способ структурирования проекта, а затем несколько практических способов улучшения его организации.

2.12.1. Структура проекта

В языке Go нет строгого соглашения о структурировании проекта. Но с годами появился один макет проекта (https://github.com/golang-standards/project-layout).

Если проект достаточно мал (всего несколько файлов) или если компания уже создала свой стандарт, возможно, этот макет проекта не стоит использовать или переходить на него. В других случаях советуем рассмотреть такой вариант. Основные каталоги в проекте:

- /cmd — основные исходные файлы. Файл main.go приложения `foo` должен находиться в /cmd/foo/main.go.
- /internal — закрытый (private) код: мы не хотим, чтобы другие импортировали его для своих приложений или библиотек.
- /pkg — общедоступный (public) код, который мы хотим предоставлять другим в пользование.
- /test — дополнительные внешние тесты и тестовые данные. Юнит-тесты в Go находятся в том же пакете, что и исходные файлы. Но, например, общедоступные тесты API или интеграционные тесты должны находиться в /test.

- /configs — файлы конфигурации.
- /docs — проектные и пользовательские документы.
- /examples — примеры для нашего приложения и/или общедоступной библиотеки.
- /api — файлы контрактов API (Swagger, Protocol Buffers и т. д.).
- /web — ресурсы, относящиеся к веб-приложению (статические файлы и т. д.).
- /build — файлы упаковки и непрерывной интеграции (CI).
- /scripts — скрипты для анализа, установки и т. д.
- /vendor — зависимости приложений (например, зависимости модулей Go).

В этом списке нет каталога /src, как в некоторых других языках. Причина в том, что /src слишком общий, поэтому в этом макете отдается предпочтение /cmd, /internal или /pkg.

> **ПРИМЕЧАНИЕ** В 2021 году Расс Кокс (Russ Cox), один из основных мейнтейнеров Go, раскритиковал этот макет. В основном проект существует в рамках организации GitHub golang-standards, хоть и не является официальным стандартом. Имейте в виду, что в отношении структуры проекта нет обязательных соглашений. Этот макет может быть полезен вам или нет, но нерешительность — единственное неправильное решение. Поэтому согласуйте макет для поддержания единообразия в своей компании, чтобы разработчики не тратили время на переход от одного репозитория к другому.

Теперь поговорим, как организовать в Go основную логику репозитория.

2.12.2. Организация пакета

В Go нет концепции «подпакетов», но можно организовывать пакеты в подкаталогах. Если посмотреть на стандартную библиотеку, то сетевой каталог организован так:

```
/net
    /http
        client.go
        ...
    /smtp
        auth.go
        ...
    addrselect.go
    ...
```

`net` действует и как пакет, и как каталог, содержащий другие пакеты. Но `net/http` не наследуется из `net` и не имеет особых прав доступа к пакету `net`. Элементы внутри `net/http` могут видеть только экспортированные элементы `net`. Основное преимущество подкаталогов состоит в том, что пакеты хранятся там, где у них максимальная связность (cohesion) с другими элементами.

Что касается общей организации, то на этот счет есть разные мнения. Например, нужно ли организовать приложение, отталкиваясь от контекста или от уровней? Это зависит от ваших предпочтений. Можно использовать группировку кода по контексту (например, контекст потребителя, контекст контракта и т. д.) или следовать принципам гексагональной архитектуры и группировке по техническому уровню. Если решение соответствует варианту использования, оно не может быть неправильным, пока мы действуем последовательно.

В отношении пакетов есть несколько лучших практик, которым будет полезно следовать. Так, рекомендуется избегать преждевременной упаковки, поскольку это может привести к чрезмерному усложнению проекта. Иногда лучше использовать простую организацию и развивать проект, сохраняя четкое понимание того, что в нем содержится, чем заставлять себя создавать идеальную структуру с самого начала.

Детализация (гранулярность) — еще одна важная вещь, которую следует учитывать. Избегайте десятков «нанопакетов», содержащих только один или два файла. Если же подобное происходит, то, вероятно, потому, что пропущены некоторые логические связи между этими пакетами. Это затрудняет понимание проекта теми, кто будет читать код. И наоборот, избегайте огромных пакетов, за содержимым которых теряется смысл того, почему этот пакет назван именно так.

К выбору названий пакетов подходите с осторожностью. Придумывать имена сложно. Чтобы помочь клиентам понять проект Go, называйте пакеты, отталкиваясь от их возможностей, а не от их содержимого. Название должно быть осмысленным. Имя пакета должно быть коротким, лаконичным, выразительным, а еще состоять из одного слова в нижнем регистре.

Что касается вопроса о том, что следует экспортировать, то тут правило простое. Сводите к минимуму то, что должно быть экспортировано, чтобы уменьшить связанность (coupling) между пакетами и скрывать ненужные экспортируемые элементы. Если нет уверенности, надо ли экспортировать какой-то элемент или нет, по умолчанию его не нужно экспортировать. Позже, если обнаружится, что экспортировать его все же нужно, можно изменить код. Помните и о некоторых исключениях, например о создании экспортируемых полей, чтобы структуру можно было демаршалировать с помощью `encoding/json`.

Организовывать проект непросто, но соблюдение приведенных правил упростит его поддержку. Помните, что согласованность также жизненно важна для облегчения сопровождения. Так что важно удостовериться, что в кодовой базе все консистентно.

Рассмотрим пакеты утилит.

2.13. ОШИБКА #13: СОЗДАВАТЬ ПАКЕТЫ УТИЛИТ

Поговорим о весьма распространенном неудачном приеме: создании общих пакетов, таких как `utils`, `common` и `base`. Рассмотрим проблемы, связанные с таким подходом, и узнаем, как улучшить организацию кода.

Рассмотрим пример, вдохновленный официальным блогом Go. Речь идет о реализации заданной структуры данных (карты, где значение игнорируется). Идиоматический способ сделать это в Go — обработка через тип `map[K]struct{}` с параметром `K`, который может быть любым типом, допустимым в `map`, в качестве ключа, в то время как значение является типом `struct{}`. Карта с типом значения `struct{}` передает, что нас не интересует само значение. Представим два метода в пакете `util`:

```
package util

func NewStringSet(...string) map[string]struct{} {    <-- Создается набор строк

    // ...
}
func SortStringSet(map[string]struct{}) []string {    <-- Возвращается отсортированный список ключей
    // ...
}
```

Клиент будет использовать этот пакет вот так:

```
set := util.NewStringSet("c", "a", "b")
fmt.Println(util.SortStringSet(set))
```

Проблема в том, что название пакета `util` бессмысленно. Мы могли бы назвать его `common`, `shared` или `base`, но эти названия тоже бессмысленны и не дают никакого представления о том, что делает этот пакет.

Вместо «пакет утилит» (utility package) лучше придумать более выразительное имя, например `stringset` («набор строк»):

```
package stringset

func New(...string) map[string]struct{} { ... }
func Sort(map[string]struct{}) []string { ... }
```

В этом примере мы удалили суффиксы для `NewStringSet` и `SortStringSet`, которые соответственно стали `New` и `Sort`. На стороне клиента это теперь выглядит так:

```
set := stringset.New("c", "a", "b")
fmt.Println(stringset.Sort(set))
```

> **ПРИМЕЧАНИЕ** В предыдущем разделе я затронул идею «нанопакетов». Создание в приложении десятков нанопакетов может усложнить отслеживание того, что стоит за выполнением кода. Но сама идея использования нанопакетов не всегда плохая. Если небольшая группа кода имеет высокую внутреннюю связность и не относится к чему-либо еще, приемлемо выделить ее в отдельный пакет. Строгого правила для этого нет, и задача часто состоит в том, чтобы найти баланс.

Можно пойти еще дальше. Вместо создания служебных функций можно создать специфический тип и предоставить `Sort` как метод так:

```
package stringset

type Set map[string]struct{}
func New(...string) Set { ... }
func (s Set) Sort() []string { ... }
```

Это изменение делает клиент еще более простым. На пакет `stringset` будет только одна ссылка:

```
set := stringset.New("c", "a", "b")
fmt.Println(set.Sort())
```

Небольшой рефакторинг убирает бессмысленное имя пакета и дает выразительный API. Как упомянул Дейв Чейни (член команды проекта Go), мы достаточно часто находим служебные пакеты, которые управляют стандартными возможностями. Например, если мы решим иметь клиентский и серверный пакеты, куда нужно поместить общие типы? В этом случае одним из возможных решений будет объединение клиента, сервера и общего кода в единый пакет.

Именование пакетов — важная часть общей конструкции приложения, к этому следует подходить с вниманием и осторожностью. Как правило, создание общих пакетов без осмысленных имен — плохая практика, к ней можно отнести служебные пакеты с именами вроде `utils`, `common` или `base`. Именуйте пакеты, отталкиваясь от того, какие действия он производит, а не от того, что в нем содержится. Это будет эффективным способом повысить его выразительность.

В следующем разделе обсудим пакеты и коллизии имен.

2.14. ОШИБКА #14: ИГНОРИРОВАТЬ КОЛЛИЗИИ ИМЕН ПАКЕТОВ

Коллизии имен пакетов возникают, когда переменная имеет такое же имя, как и у существующего пакета, что мешает его переиспользованию. Рассмотрим пример с библиотекой, открывающей клиент `Redis`:

```
package redis

type Client struct { ... }

func NewClient() *Client { ... }

func (c *Client) Get(key string) (string, error) { ... }
```

Перейдем на сторону клиента. Несмотря на существование имени пакета `redis`, в Go вполне допустимо создать и переменную с именем `redis`:

```
redis := redis.NewClient()  <-- Вызывается NewClient из пакета redis
v, err := redis.Get("foo")  <-- Используется переменная с именем redis
```

Здесь происходит коллизия имени переменной `redis` с именем пакета `redis`. Хотя такое использование наименований и разрешено, его следует избегать. Во всей области действия переменной `redis` пакет `redis` будет недоступен.

Предположим, что какой-то квалификатор ссылается как на переменную, так и на имя пакета внутри всей функции. В этом случае тот, кто читает код, может и не понять, на что он ссылается. Как избежать такой коллизии? Первый вариант — использовать другое имя переменной. Например:

```
redisClient := redis.NewClient()
v, err := redisClient.Get("foo")
```

Это, пожалуй, самый простой способ. Но если по какой-то причине нужно оставить `redis` в качестве имени нашей переменной, можно поиграть с импортом пакетов. Применяя импорт пакетов, можно использовать псевдоним, чтобы изменить квалификатор для ссылки на пакет `redis`. Например:

```
import redisapi "mylib/redis"  <-- Для пакета redis создается псевдоним

// ...

redis := redisapi.NewClient()  <-- Указывается на доступ к пакету redis через псевдоним redisapi
v, err := redis.Get("foo")
```

Для импорта использовался псевдоним `redisapi`, ссылающийся на пакет `redis`, чтобы сохранить имя переменной `redis`.

> **ПРИМЕЧАНИЕ** Один из вариантов — использование точечного импорта для доступа ко всем общедоступным элементам пакета без ссылки на квалификатор пакета. Но такой подход приводит к общему возрастанию путаницы, и в большинстве случаев его следует избегать.

Следует избегать коллизий имен переменных и встроенных функций. Например, мы могли бы сделать что-то вроде такого:

```
copy := copyFile(src, dst)  <-- Происходит коллизия переменной copy
                                со встроенной функцией копирования
```

Но тогда встроенная функция `copy` будет недоступна, пока есть переменная `copy`. Поэтому нужно стремиться предотвращать коллизии имен переменных, чтобы избегать двусмысленности. Если все же по какой-то причине коллизия происходит, найдите другое имя, которое будет нести нужный смысл, либо используйте псевдоним импорта.

В следующем разделе рассмотрим распространенную ошибку, связанную с документацией.

2.15. ОШИБКА #15: НЕ ПИСАТЬ ДОКУМЕНТАЦИЮ ПО КОДУ

Создание соответствующей документации — важная часть написания кода. Она упрощает клиентам использование API, а также помогает в поддержке

и сопровождении проекта. Некоторые правила Go помогут сделать код идиоматичным.

Прежде всего, каждый экспортируемый элемент должен быть задокументирован. Будь то структура, интерфейс, функция или что-то еще, если элемент экспортируется, он должен быть задокументирован. Принято добавлять комментарии, начиная с имени экспортируемого элемента. Например:

```
// Customer — это представление потребителя.
type Customer struct{}
// ID возвращает идентификатор потребителя.
func (c Customer) ID() string { return "" }
```

По правилам, каждый комментарий должен быть полным предложением, заканчивающимся точкой. Также имейте в виду, что когда мы описываем функцию (или метод), мы должны указывать то, *что* функция должна делать, а не то, *как* она это делает. Это относится и к ядру функции, и к комментариям, но не к документации. В идеале документация должна содержать достаточно информации, чтобы клиенту не нужно было каждый раз изучать код, чтобы понять, как использовать экспортируемый элемент.

Устаревшие элементы

Экспортированный элемент можно объявить устаревшим с помощью комментария `// Deprecated`:

```
// ComputePath возвращает быстрейший путь между двумя точками.
// Deprecated: Эта функция использует устаревший способ расчета быстрейшего
// пути. Вместо нее используйте ComputeFastestPath.
func ComputePath() {}
```

Тогда, если разработчик использует функцию `ComputePath`, он должен получить соответствующее предупреждение. (Большинство IDE обрабатывают комментарии `// Deprecated:`.)

Когда дело доходит до документирования переменной или константы, нас может заинтересовать передача связанных с ними двух аспектов: назначения и содержания. Первый должен быть отражен в документации, что будет полезно для внешних клиентов. Последний не обязательно делать публичным. Например:

```
// DefaultPermission разрешение по умолчанию, используемое движком магазина.
const DefaultPermission = 0o644 // Необходим доступ на чтение и запись.
```

Эта константа представляет собой разрешение по умолчанию. Документация передает ее назначение, тогда как комментарий рядом с константой описывает ее фактическое содержание (доступы для чтения и записи).

Чтобы помочь клиентам и мейнтейнерам понять объем пакета, нужно документировать каждый пакет. По соглашению комментарий начинается с `// Package`, за которым следует имя пакета:

```
// Пакет math предоставляет основные константы и математические функции.
//
// Этот пакет не гарантирует битовую идентичность результатов
// в разных архитектурах.
package math
```

Первая строка комментария к пакету должна быть краткой, поскольку она появится в пакете (см. пример на рис. 2.11). В последующих строках можно изложить остальную информацию.

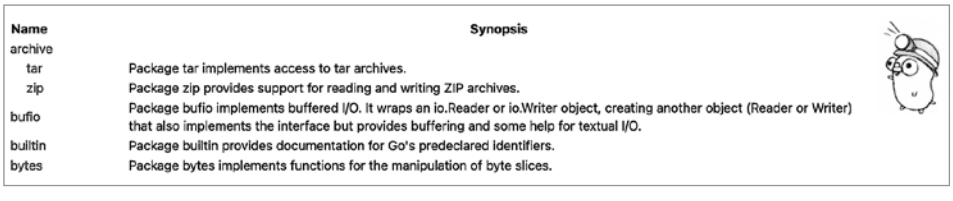


Рис. 2.11. Пример сгенерированной документации стандартной библиотеки Go

Документирование пакета можно выполнить в любом из файлов Go, здесь нет никаких правил. Обычно документацию по пакету помещают в соответствующий файл с тем же именем, что и сам пакет, или в какой-то определенный файл, например doc.go.

И последнее, что следует сказать о документации пакетов: комментарии, не примыкающие к объявлению, опускаются. Например, следующий комментарий об авторских правах не будет виден в создаваемой документации:

```
// Copyright 2009 The Go Authors. All rights reserved.
// Use of this source code is governed by a BSD-style
// license that can be found in the LICENSE file.
// Пакет math предоставляет основные константы и математические функции.
//   ◄—— Пустая строка. Предыдущие комментарии не будут включены в документацию
// Этот пакет не гарантирует битовую идентичность результатов
// в разных архитектурах.
package math
```

Каждый экспортируемый элемент должен быть задокументирован. Документирование кода не должно быть чем-то ограничено. Пользуйтесь всеми возможностями, чтобы убедиться, что оно поможет клиентам и мейнтейнерам понять назначение кода.

В последнем разделе этой главы рассмотрим распространенную ошибку, касающуюся инструментов: неиспользование линтеров.

2.16. ОШИБКА #16: НЕ ИСПОЛЬЗОВАТЬ ЛИНТЕРЫ

Линтер — это автоматический инструмент для анализа кода и отлова ошибок в нем. В задачи этого раздела не входит предоставление исчерпывающего списка существующих линтеров, поскольку такой список очень быстро устареет. Но нужно понимать и помнить, почему линтеры важны для большинства проектов на Go.

Рассмотрим пример. В разделе, посвященном ошибке #1, мы обсуждали затенение переменных. Используя линтер `vet`, встроенный в набор инструментов Go, а также `shadow`, можно обнаружить затененные переменные:

```
package main

import "fmt"

func main() {
    i := 0
    if true {
        i := 1  <-- Затененная переменная
        fmt.Println(i)
    }
    fmt.Println(i)
}
```

Поскольку `vet` включен в бинарный пакет Go, установим сначала `shadow`, свяжем его с `vet`, а затем запустим:

```
$ go install \
  golang.org/x/tools/go/analysis/passes/shadow/cmd/shadow   <-- Установка shadow
$ go vet -vettool=$(which shadow)                           <-- Установление связи с Go vet через использование аргумента vettool
./main.go:8:3:
  declaration of "i" shadows declaration at line 6          <-- Go vet выявляет затененную переменную
```

Как мы видим, `vet` сообщает, что переменная `i` в этом примере затенена. Использование соответствующих линтеров поможет сделать код более надежным и обнаружить потенциальные ошибки.

> **ПРИМЕЧАНИЕ** Линтеры не выявляют все ошибки, описанные в этой книге. Поэтому рекомендуем продолжить чтение ;).

Еще раз подчеркиваем, что цель этого раздела — не перечисление всех доступных линтеров. Но вот список, с которым точно можно ежедневно сверяться:

- https://golang.org/cmd/vet/ — стандартный анализатор Go.
- https://github.com/kisielk/errcheck — средство проверки ошибок.
- https://github.com/fzipp/gocyclo — анализатор цикломатической сложности.
- https://github.com/jgautheron/goconst — анализатор повторяющихся строковых констант.

Помимо линтеров, используйте форматировщики кода для исправления стиля написанного кода. Вот некоторые инструменты, которые стоит попробовать:

- https://golang.org/cmd/gofmt/ — стандартный форматировщик кода Go.
- https://godoc.org/golang.org/x/tools/cmd/goimports — стандартный модуль форматирования импорта Go.

Взгляните и на golangci-lint (https://github.com/golangci/golangci-lint). Это инструмент для анализа кода, который обеспечивает видимость поверх многих полезных линтеров и форматировщиков. Он позволяет запускать линтеры параллельно для повышения скорости анализа, что весьма удобно.

Линтеры и форматировщики — это мощные способы улучшить качество и согласованность кода. Уделите время тому, чтобы понять, какие из них следует использовать, и убедитесь, что автоматизировали их выполнение (например, с помощью CI или с Git pre-commit hook).

ИТОГИ

- Избегайте затенения переменных во избежание ссылок на неправильную переменную или запутывания читателей кода.

- Избегайте использования вложенных уровней и выравнивайте «счастливый путь» по левому краю — это упрощает построение ментальной модели кода.
- При инициализации переменных помните, что функции инициализации содержат в себе ограниченные возможности по обработке ошибок, что усложняет обработку состояний и тестирование. В большинстве случаев инициализации следует обрабатывать как специальные функции.
- Принудительное использование геттеров и сеттеров не является в Go идиоматическим. Правильный подход заключается в том, чтобы быть прагматичным и находить должный баланс между эффективностью и следованием определенным идиомам.
- Абстракции следует «открывать», а не создавать. Для предотвращения излишней сложности создавайте интерфейс только тогда, когда он действительно нужен, а не тогда, когда вы лишь предполагаете, что он может понадобиться в будущем, либо если можете доказать, что абстракция допустима.
- Размещение интерфейсов на стороне потребителя позволяет избежать излишних абстракций.
- Чтобы избавиться от ограничений с точки зрения гибкости, в большинстве случаев функции должны возвращать не интерфейсы, а конкретные реализации. И наоборот, функции должны принимать интерфейсы всегда, когда это возможно.
- Используйте `any` только в том случае, если нужно принять или вернуть любой возможный тип, например `json.Marshal`. В противном случае `any` не несет значимой информации и может привести к проблемам при компиляции, позволяя вызывающей функции обращаться к методам с любым типом данных.
- Полагаясь на дженерики и параметры типа, можно избежать написания шаблонного кода для разделения элементов или поведения. Используйте параметры типа лишь тогда, когда видите конкретную необходимость в них. В противном случае они вводят ненужные абстракции и усложняют код.
- Использование встраивания типов также поможет избежать шаблонного кода. Но убедитесь, что это не приведет к проблемам с видимостью в тех случаях, когда некоторые поля должны оставаться скрытыми.
- Для подходящей обработки параметров в удобной для API манере используйте паттерн функциональных опций.
- Следование макету проекта может стать хорошим способом структурировать проект, особенно если в новом проекте вы стремитесь к соблюдению имеющихся соглашений для стандартизации.

- Именование — важнейшая часть проектирования приложений. Создание пакетов с именами `common`, `util` или `shared` не имеет ценности для читателя кода. Преобразуйте имена таких пакетов во что-то более осмысленное и конкретное.
- Чтобы избежать коллизий имен переменных и пакетов, приводящих к путанице или ошибкам, используйте уникальные имена для каждого из них. Если это невозможно, применяйте псевдоним импорта, изменяя квалификатор так, чтобы отличать имя пакета от имени переменной, или придумайте лучшие имена.
- Чтобы клиенты и мейнтейнеры проекта лучше понимали назначение кода, документируйте экспортированные элементы.
- Чтобы улучшить качество и внутреннюю согласованность кода, используйте линтеры и средства форматирования.

3 Типы данных

В этой главе:

- ✓ Типичные ошибки, связанные с основными типами
- ✓ Фундаментальные концепции срезов и карт, которые надо знать для предотвращения возможных ошибок, утечек или неточностей
- ✓ Сравнение значений

Работа с типами данных — часть стандартной работы инженеров-программистов. В этой главе рассмотрим распространенные ошибки, связанные с базовыми типами, срезами и картами. В этой главе мы не говорим о строках, им посвящена следующая глава.

3.1. ОШИБКА #17: ПУТАНИЦА С ВОСЬМЕРИЧНЫМИ ЛИТЕРАЛАМИ

Рассмотрим частую путаницу с представлением восьмеричного литерала, которая может привести к ошибкам. Как вы думаете, что выведет этот код?

```
sum := 100 + 010
fmt.Println(sum)
```

На первый взгляд можно ожидать, что в результате выполнения этого кода будет выведено `100 + 10 = 110`. Но вместо этого получается `108`. Почему?

В Go целочисленный литерал, начинающийся с 0, считается восьмеричным целым числом (то есть числом по основанию 8), поэтому 10 по основанию 8 равняется 8 по основанию 10. Таким образом, сумма в предыдущем примере равна `100 + 8 = 108`. Помнить о таком свойстве целочисленных литералов очень важно, чтобы избегать путаницы при чтении кода.

Восьмеричные целые числа полезны в разных сценариях. Допустим, мы хотим открыть файл с помощью `os.OpenFile`. Эта функция требует передачи разрешения как `uint32`. Если мы хотим соответствовать разрешению Linux, то для лучшей читаемости можем передать восьмеричное число вместо числа по основанию 10:

```
file, err := os.OpenFile("foo", os.O_RDONLY, 0644)
```

В этом примере `0644` представляет определенное разрешение Linux (чтение для всех пользователей, а запись только для текущего). Также можно добавить символ `o` (буква `o` в нижнем регистре) после нуля:

```
file, err := os.OpenFile("foo", os.O_RDONLY, 0o644)
```

Префиксы `0o` и `0` несут одно и то же значение. Но использование `0o` поможет сделать код более понятным.

> **ПРИМЕЧАНИЕ** Можно использовать символ `O` в верхнем регистре вместо символа o в нижнем регистре. Но передача `0O644` может усилить путаницу, потому что, в зависимости от шрифта, `0` может выглядеть очень похоже на `O`.

Обратите внимание на представления других целочисленных литералов:

- *двоичных* — используется префикс `0b` или `0B` (например, `0b100` равно десятичному 4);
- *шестнадцатеричных* — используется префикс `0x` или `0X` (например, `0xF` равно десятичному 15);
- *мнимых* — используется суффикс `i` (например, `3i`).

Наконец, можно использовать символ подчеркивания (_) в качестве разделителя для удобства чтения. Например, записать 1 миллиард так: `1_000_000_000`. Можно использовать символ подчеркивания с другими представлениями (например, `0b00_00_01`).

В Go есть возможность обработки двоичных, шестнадцатеричных, мнимых и восьмеричных чисел. Восьмеричные числа начинаются с 0. Чтобы улучшить читаемость и избежать потенциальных ошибок, сделайте восьмеричные числа явными, используя префикс `0o`.

В следующем разделе углубимся в различные аспекты, связанные с использованием целых чисел, и обсудим, как в Go обрабатываются переполнения.

3.2. ОШИБКА #18: ИГНОРИРОВАТЬ ЦЕЛОЧИСЛЕННЫЕ ПЕРЕПОЛНЕНИЯ

Непонимание того, как целочисленные переполнения обрабатываются в Go, может привести к критическим ошибкам. Далее в разделе углубимся в эту тему. Но сначала вспомним несколько концепций, связанных с целыми числами.

3.2.1. Концепции

В Go есть в общей сложности 10 типов целых чисел. Как показано в таблице, есть четыре целочисленных типа со знаком и четыре целочисленных типа без знака.

Целые числа со знаком	Целые числа без знака
`int8 (8 bits)`	`uint8 (8 bits)`
`int16 (16 bits)`	`uint16 (16 bits)`
`int32 (32 bits)`	`uint16 (16 bits)`
`int64 (64 bits)`	`uint64 (64 bits)`

Чаще всего используются два других целочисленных типа: `int` и `uint`. Они имеют размер, который зависит от системы: 32 бита в 32-битных системах или 64 бита в 64-битных системах.

Обратимся к вопросам переполнения. Предположим, нужно инициализировать `int32` до его максимального значения, а затем увеличить. Как будет вести себя вот этот код?

```
var counter int32 = math.MaxInt32
counter++
fmt.Printf("counter=%d\n", counter)
```

Он компилируется и не вызывает паники во время выполнения. Однако оператор `counter++` генерирует целочисленное переполнение:

```
counter=-2147483648
```

Оно возникает, когда результатом арифметической операции является значение вне диапазона, который может быть представлен заданным числом байтов. В `int32` используются 32 бита. Вот двоичное представление максимального значения `int32` (`math.MaxInt32`):

```
01111111111111111111111111111111
  |-значения 31 бита установлены в 1--|
```

Поскольку `int32` — это целое число со знаком, бит слева представляет знак целого числа: 0 — положительное, 1 — отрицательное. Если мы увеличим это целое число, не останется места для представления нового значения. Следовательно, это приводит к целочисленному переполнению. С точки зрения двоичного кода вот новое значение:

```
10000000000000000000000000000000
  |-значения 31 бита установлены в 0--|
```

Как мы видим, знаковый бит теперь равен 1, что означает отрицательное значение. Это значение — наименьшее возможное для целого числа со знаком, представленного 32 битами.

> **ПРИМЕЧАНИЕ** Наименьшее возможное отрицательное значение не равно 11111111111111111111111111111111. Большинство систем полагаются на операцию дополнения до двух для представления двоичных чисел (инвертировать каждый бит и добавить 1). Основная цель этой операции — сделать $x + (-x)$ равным 0 независимо от x.

В Go целочисленное переполнение, которое можно обнаружить во время компиляции, приводит к ошибке компиляции. Например:

```
var counter int32 = math.MaxInt32 + 1
constant 2147483648 overflows int32
```

Но во время выполнения целочисленное переполнение или антипереполнение (потеря значимости) не происходит, и это не приводит к панике приложения.

Важно помнить о таком поведении, поскольку оно может привести к скрытым ошибкам (например, увеличение значения целого числа или суммирование положительных целых чисел, которые приводят к результату с отрицательным знаком).

Прежде чем углубляться в то, как обнаруживать целочисленное переполнение с помощью обычных операций, подумаем, когда вообще об этом нужно беспокоиться. В большинстве случаев, например при обработке счетчика запросов или основных операциях сложения/умножения, волноваться не стоит, если мы используем правильный целочисленный тип. Но, например, в проектах с ограниченным объемом памяти, использующих меньшие целочисленные типы, работающих с большими числами или выполняющих операции конверсии/преобразования, может потребоваться провести проверки на предмет возможных переполнений.

> **ПРИМЕЧАНИЕ** Неудачный запуск ракеты Ariane 5 в 1996 году (https://www.bugsnag.com/blog/bug-day-ariane-5-disaster) произошел из-за переполнения, возникшего в результате преобразования 64-битного числа с плавающей точкой в 16-битное целое число со знаком.

3.2.2. Обнаружение целочисленного переполнения при инкрементировании

Чтобы обнаружить целочисленное переполнение при выполнении операции инкрементального увеличения значения переменной типа, основанного на определенном размере (`int8`, `int16`, `int32`, `int64`, `uint8`, `uint16`, `uint32` или `uint64`), можно сравнивать это значение с математическими константами. Например, в случае с `int32`:

```
func Inc32(counter int32) int32 {
   if counter == math.MaxInt32 {  ◄──  Сравнение с math.MaxInt32
      panic("int32 overflow")
   }
   return counter + 1
}
```

Эта функция проверяет, достигла ли переменная значения `math.MaxInt32`. Если да, то ее увеличение приведет к переполнению.

А что насчет типов `int` и `uint`? До версии Go 1.17 приходилось создавать эти константы вручную. Теперь же `math.MaxInt`, `math.MinInt` и `math.MaxUint`

стали частью пакета `math`. Если нужно проверить на переполнение переменную типа `int`, можно сделать это с помощью `math.MaxInt`:

```
func IncInt(counter int) int {
   if counter == math.MaxInt {
      panic("int overflow")
   }
   return counter + 1
}
```

Логика та же самая для `uint`. Можно использовать `math.MaxUint`:

```
func IncUint(counter uint) uint {
   if counter == math.MaxUint {
      panic("uint overflow")
   }
   return counter + 1
}
```

В этом разделе мы узнали, как проверять целочисленные переполнения при инкрементировании. А что насчет сложения?

3.2.3. Обнаружение целочисленного переполнения при сложении

Как обнаружить целочисленное переполнение при сложении? Ответ заключается в повторном использовании `math.MaxInt`:

```
func AddInt(a, b int) int {
   if a > math.MaxInt-b {  <-- Проверяется, не произойдет ли целочисленное переполнение
      panic("int overflow")
   }
   return a + b
}
```

В этом примере `a` и `b` — два операнда. Если `a` больше, чем `math.MaxInt – b`, операция приведет к целочисленному переполнению. Теперь рассмотрим умножение.

3.2.4. Обнаружение целочисленного переполнения при умножении

С умножением будет чуть сложнее. Следует выполнить проверки минимального целого числа, `math.MinInt`:

```
func MultiplyInt(a, b int) int {
   if a == 0 || b == 0 {
      return 0
   }
   result := a * b
   if a == 1 || b == 1 {
      return result
   }
   if a == math.MinInt || b == math.MinInt {
      panic("integer overflow")
   }
   if result/b != a {
      panic("integer overflow")
   }
   return result
}
```

Если один из операндов равен нулю, значение 0 возвращается напрямую

Проверяется, равен ли один из операндов 1

Проверяется, равен ли один из операндов math.MinInt

Проверяется, приводит ли операция умножения к целочисленному переполнению

Проверка целочисленного переполнения при умножении проводится в нескольких шагов. Сначала нужно проверить, равен ли один из операндов нулю, единице или `math.MinInt`. Затем разделить результат умножения на `b`. Если результат не будет равен исходному значению (`a`), значит, произошло целочисленное переполнение.

Таким образом, целочисленные переполнения (и антипереполнения) — «тихие» операции в Go. Чтобы обнаруживать переполнения для избегания скрытых ошибок, используйте описанные в этом разделе служебные функции. Также помните, что в Go есть пакет для работы с большими числами: `math/big`. Используйте его, если `int` недостаточно.

В следующем разделе поговорим о переменных с плавающей точкой.

3.3. ОШИБКА #19: НЕ ПОНИМАТЬ ПРОБЛЕМ, СВЯЗАННЫХ С ПЛАВАЮЩЕЙ ТОЧКОЙ

В Go есть два типа с плавающей точкой (не считая комплексных чисел): `float32` и `float64`. Концепция чисел с плавающей точкой была предложена для решения основной проблемы целочисленных типов: неспособности представлять дробные значения. Чтобы избежать неприятных сюрпризов, нужно знать, что арифметика чисел с плавающей точкой является аппроксимацией реальной арифметики. Поговорим об особенностях работы с такими аппроксимациями и способах повышения точности ее результатов. Рассмотрим пример с умножением:

```
var n float32 = 1.0001
fmt.Println(n * n)
```

Можно ожидать, что в результате будет выведено: `1.0001 * 1.0001 = 1.00020001`, не так ли? Но на большинстве процессоров x86 будет выдано значение `1.0002`. Почему? Сначала нужно понять тонкости арифметики чисел с плавающей точкой.

Возьмем в качестве примера тип `float64`. Обратите внимание, что между `math.SmallestNonzeroFloat64` (минимальное значение для типа `float64`) и `math.MaxFloat64` (максимальное значение для типа `float64`) есть бесконечное количество значений действительных чисел. С другой стороны, тип `float64` имеет конечное число битов — 64. Поскольку уместить бесконечно много чисел в конечное множество невозможно, приходится работать с приближениями. Как результат — потеря точности. Та же логика применима и к типу `float32`.

Представление чисел с плавающей точкой в Go соответствует стандарту IEEE-754, при этом некоторые биты относятся к мантиссе, а другие — к показателю степени (порядку числа, или экспоненте). *Мантисса* — это базовое значение, тогда как экспонента — это степень, в которую нужно возвести число 2, чтобы при перемножении на мантиссу получить искомое число.

В типах с плавающей точкой одинарной точности (`float32`) 8 бит отводятся на представление экспоненты (порядка числа), а 23 бита — мантиссы. В типах с плавающей точкой двойной точности (`float64`) эти значения составляют 11 и 52 бита для порядка и мантиссы соответственно. Остающийся бит предназначен для знака. Чтобы преобразовать число с плавающей точкой в десятичное представление, используется следующий расчет:

```
sign * 2^exponent * mantissa
```

На рис. 3.1 показано представление числа 1.0001 в виде числа типа `float32`. В показателе степени используется 8-битная нотация с избытком и смещением: значение показателя степени 01111111 означает 2^0, тогда как мантисса равна 1.000100016593933. (Обратите внимание, что этот раздел не предназначен для объяснения того, как работает это преобразование.) Следовательно, десятичное значение равно $1 \times 2^0 \times 1.000100016593933$. Таким образом, то, что мы храним в представлении с плавающей точкой одинарной точности, равно не 1.0001,

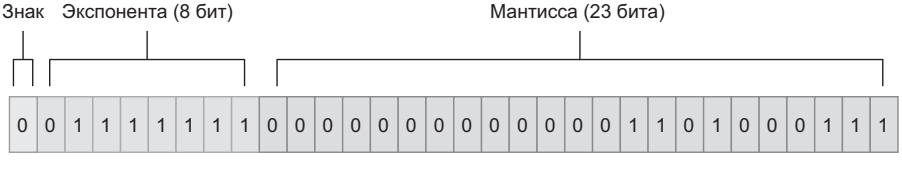


Рис. 3.1. Представление числа 1.0001 в виде типа float32

а 1.000100016593933. Недостаточная прецизионность влияет на точность хранимого значения.

Теперь мы понимаем, что и `float32`, и `float64` — это аппроксимации. Чем это чревато для разработчиков? Первый вывод относится к сравнениям. Использование оператора `==` для сравнения двух чисел с плавающей точкой может привести к неточностям. Поэтому нужно сравнивать разницу между ними, чтобы определить, меньше ли она некоторого малого значения допустимой ошибки. Например, в библиотеке тестирования `testify` (https://github.com/stretchr/testify) есть функция `InDelta`, подтверждающая, что два значения находятся друг от друга в пределах заданной дельты.

Имейте в виду, что результат вычислений с числами с плавающей точкой зависит от конкретного процессора. У большинства из них для выполнения таких вычислений есть модуль операций для работы с плавающей точкой (FPU). Но нет никакой гарантии, что результат, полученный на одном компьютере, будет таким же на другой машине с другим FPU. Сравнение двух значений с использованием дельты может быть решением для выполнения валидных тестов на разных машинах.

Виды чисел с плавающей точкой

В Go есть три специальных вида чисел с плавающей точкой:

- положительная бесконечность;
- отрицательная бесконечность;
- NaN (Not-a-Number — «не число») — для представления результата неопределенной или непредставимой операции.

Согласно IEEE-754, NaN — единственное число с плавающей точкой, удовлетворяющее условию f ! = f. Вот пример, в котором эти специальные типы чисел создаются и выводятся:

```
var a float64
positiveInf := 1 / a
negativeInf := -1 / a
nan := a / a
fmt.Println(positiveInf, negativeInf, nan)

+Inf -Inf NaN
```

Мы можем проверять, является ли число с плавающей точкой бесконечностью, используя `math.IsInf`, и является ли оно NaN, используя `math.IsNaN`.

Мы увидели, что преобразования десятичных чисел в числа с плавающей точкой могут привести к потере точности. Это ошибка из-за преобразования типов. Также ошибка может накапливаться в последовательности операций с числами с плавающей точкой.

Рассмотрим пример с двумя функциями, выполняющими последовательность одних и тех же операций в разном порядке. В нашем примере `f1` начинает с инициализации `float64` значением 10 000, а затем многократно (`n` раз) прибавляет к этому результату 1.0001. И наоборот, `f2` выполняет те же операции, но в обратном порядке (прибавляя 10 000 в конце):

```
func f1(n int) float64 {
    result := 10_000.
    for i := 0; i < n; i++ {
       result += 1.0001
    }
    return result
}

func f2(n int) float64 {
    result := 0.
    for i := 0; i < n; i++ {
       result += 1.0001
    }
    return result + 10_000.
}
```

Запустим эти функции на компьютере с процессором x86. При этом мы будем задавать числу `n` разные значения.

n	Точное значение	f1	f2
10	11000.1	10010.000999999993	10010.001
1K	11000.1	11000.099999999293	11000.099999999982
1M	1.0101e+06	1.0100999999761417e+06	1.0100999999766762e+06

Обратите внимание: чем больше `n`, тем больше накапливающаяся неточность. Но мы видим и то, что точность функции `f2` лучше, чем у `f1`. Поэтому имейте в виду, что порядок вычислений с плавающей запятой может повлиять на точность результата.

При выполнении цепочки сложений и вычитаний следует сгруппировывать операции сложения или вычитания для чисел с одинаковым порядком величины, прежде чем прибавлять или вычитать те числа, значения которых сильно

отличаются. Поскольку `f2` прибавляет 10 000 в самом конце, в итоге получается более точный результат, чем у `f1`.

А что насчет умножения и деления? Представим, что хотим вычислить следующее:

```
a × (b + c)
```

Как мы знаем, результат должен быть одинаковым с

```
a × b + a × c
```

Запустим эти две формулы с `a`, имеющим порядок величины, отличный от `b` и `c`:

```
a := 100000.001
b := 1.0001
c := 1.0002

fmt.Println(a * (b + c))
fmt.Println(a*b + a*c)

200030.00200030004
200030.0020003
```

Точным результатом будет число 200030.002. Поэтому расчет по первой формуле имеет наихудшую точность. При выполнении вычислений с плавающей точкой, включающих сложение, вычитание, умножение или деление, для повышения точности нужно сначала выполнить операции умножения и деления. Иногда это может повлиять на время выполнения (в предыдущем примере требуется три операции вместо двух). В этом случае придется делать выбор между точностью и временем выполнения.

`float32` и `float64` в Go — это типы с приблизительными значениями чисел. Поэтому важно помнить о нескольких правилах:

- При сравнении двух чисел в представлении с плавающей точкой убедитесь, что разница между ними находится в допустимом диапазоне.
- При выполнении операций сложения или вычитания для достижения большей точности результата группируйте операции с числами одинакового порядка.
- Если последовательность операций требует сложения, вычитания, умножения или деления, то для повышения точности результата сначала выполните операции умножения и деления.

В следующем разделе подробно рассмотрим срезы и затронем два важных понятия: длина среза и его емкость.

3.4. ОШИБКА #20: НЕ ПОНИМАТЬ ОСОБЕННОСТЕЙ, СВЯЗАННЫХ С ДЛИНОЙ СРЕЗА И ЕГО ЕМКОСТЬЮ

Go-разработчики довольно часто путают понятия длины среза и его емкости. Усвоение этих концепций нужно для эффективной обработки основных операций — инициализации среза или добавления элементов с присоединением, копированием или нарезкой. Это непонимание может привести к неоптимальному использованию срезов или даже к утечкам памяти (о чем пойдет речь в следующих разделах).

В Go за срезом стоит массив. Это означает, что данные среза хранятся в структуре данных массива. Срез также обрабатывает логику добавления элемента, если резервный массив заполнен, или уменьшения резервного массива, если он почти пуст.

Внутри себя срез содержит указатель на резервный массив, а также длину и емкость. Длина — это количество элементов, содержащихся в срезе, тогда как емкость — это количество элементов в резервном массиве. Рассмотрим несколько примеров, чтобы было понятнее. Сначала инициализируем срез заданной длины и емкости:

```
s := make([]int, 3, 6)  ◄——  Срез длиной 3 и емкостью 6
```

Первый аргумент, задающий длину, обязателен. А вот второй аргумент, относящийся к емкости, необязателен. На рис. 3.2 показан результат выполнения этого кода в памяти.

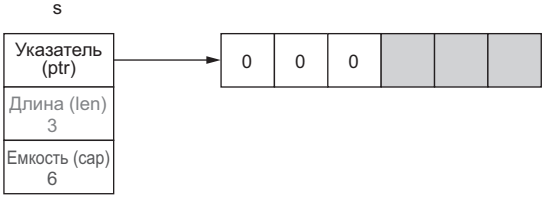


Рис. 3.2. Срез длиной 3 и емкостью 6

В данном случае `make` создает массив из шести элементов (емкость). Но поскольку длина среза была задана равной 3, то Go инициализирует только первые три

элемента. Кроме того, поскольку срез является типом `[]int`, первые три элемента инициализируются нулевым значением `int: 0`. Для элементов, обозначенных серым цветом, память зарезервирована, но они пока не используются.

Если вывести этот срез, то мы получим элементы в диапазоне его длины [0 0 0]. Если установить `s[1]` в 1, то значение второго элемента среза обновится, не влияя на длину или емкость последнего, что показано на рис. 3.3.

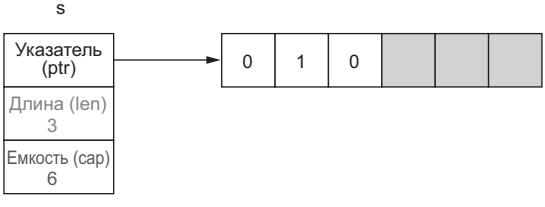


Рис. 3.3. Обновление второго элемента среза: s[1] = 1

При этом доступ к элементу за пределами диапазона длины запрещен, даже если место в памяти под него уже выделено. Например, `s[4] = 0` приведет к панике:

```
panic: runtime error: index out of range [4] with length 3
```

Как использовать оставшееся пространство среза? С помощью встроенной функции `append`:

```
s = append(s, 2)
```

Этот код добавляет к существующему срезу `s` новый элемент. Он использует первый отмеченный серым цветом элемент (место под который было выделено, но еще не использовано) для хранения значения 2, как показано на рис. 3.4.

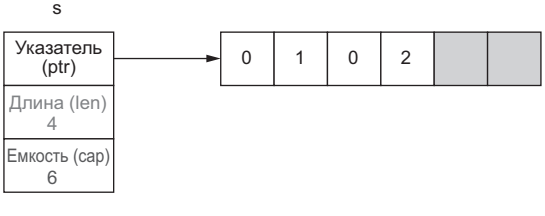


Рис. 3.4. Присоединение элемента к срезу s

Длина среза изменена с 3 на 4, поскольку теперь срез содержит четыре элемента. Что произойдет, если добавить еще три элемента (при этом резервный массив станет недостаточно большим)?

```
s = append(s, 3, 4, 5)
fmt.Println(s)
```

Если мы запустим этот код, то увидим, что срез смог справиться с нашим запросом:

```
[0 1 0 2 3 4 5]
```

Поскольку массив — это структура фиксированного размера, он может хранить новые элементы до элемента 4. Когда мы хотим вставить элемент 5, массив уже заполнен: Go внутри себя создает другой массив, удваивая его емкость и копируя в него все элементы, а затем вставляет элемент 5. Это показано на рис. 3.5.

Рис. 3.5. Поскольку исходный резервный массив заполнен, Go создает другой массив и копирует в него все элементы

> **ПРИМЕЧАНИЕ** В Go размер среза будет удваиваться до тех пор, пока он не станет содержать 1024 элемента, после чего будет увеличиваться на 25 %.

Теперь срез ссылается на новый резервный массив. А что произойдет с предыдущим массивом? Если на него больше нет ссылок, он в итоге освобождается сборщиком мусора (GC — garbage collector), если был выделен в куче. Мы подробнее обсудим память кучи при разборе ошибки #95 («не понимать различий между стеком и кучей») и рассмотрим, как работает сборщик мусора, при разборе ошибки #99 («не понимать, как работает сборщик мусора»).

Что происходит при нарезке? Нарезка — это операция, выполняемая над массивом или срезом, задающая полуоткрытый диапазон. Первый индекс включается, а второй исключается. В следующем примере показано то, что происходит при этом, а на рис. 3.6 показан результат в памяти.

```
s1 := make([]int, 3, 6)  ◄—— Срез длиной 3 и емкостью 6
s2 := s1[1:3]  ◄—— Нарезка по индексам от 1 до 3
```

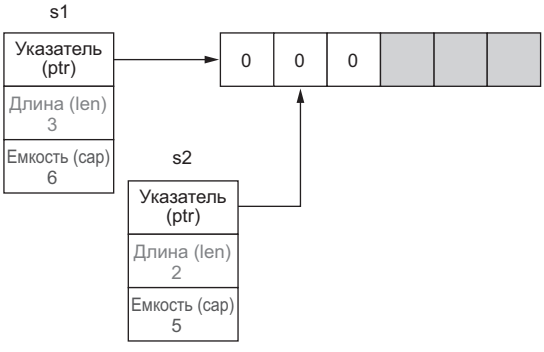


Рис. 3.6. Срезы s1 и s2 ссылаются на один и тот же резервный массив, но с разной длиной и емкостью

Первым делом создается `s1` как срез длиной 3 и емкостью 6. Когда нарезкой `s1` создается `s2`, оба среза ссылаются на один и тот же резервный массив. Однако `s2` начинается с другого индекса — с 1. Поэтому его длина и емкость (длина 2 и емкость 5) отличаются от `s1`. Если мы обновляем элемент `s1[1]` или элемент `s2[0]`, то эти изменения вносятся в один и тот же массив и, следовательно, видны в обоих срезах, как показано на рис. 3.7.

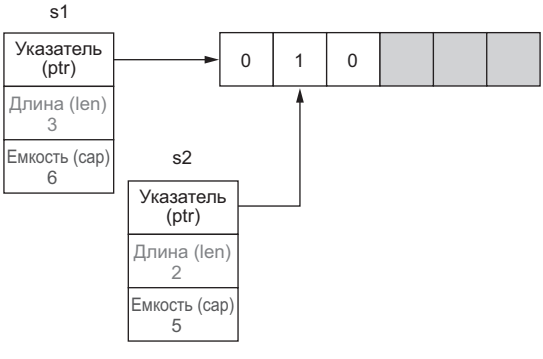


Рис. 3.7. Поскольку за s1 и s2 стоит один и тот же массив, обновление общего элемента делает изменение видимым в обоих срезах

А что произойдет, если присоединить новый элемент к (или добавить его в) `s2`? Изменяет ли следующий код также и `s1`?

```
s2 = append(s2, 2)
```

Общий резервный массив изменяется, но при этом длина изменяется только у `s2`. На рис. 3.8 показан результат добавления элемента в `s2`.

`s1` остается срезом длиной 3 и емкостью 6. Таким образом, если мы выведем `s1` и `s2`, добавленный элемент будет виден только для `s2`:

```
s1=[0 1 0], s2=[1 0 2]
```

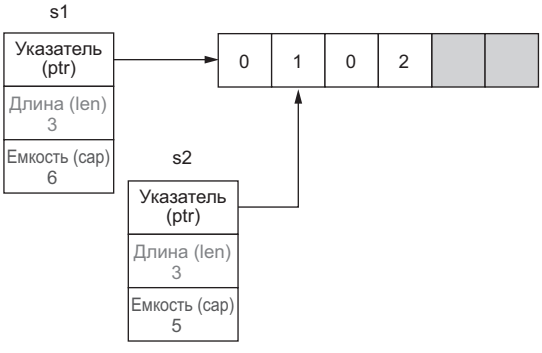


Рис. 3.8. Добавление элемента в s2

Такое поведение важно понимать, чтобы не делать неверных предположений при использовании `append`.

> **ПРИМЕЧАНИЕ** В этих примерах резервный массив внутренний и недоступен непосредственно Go-разработчику. Единственное исключение — случай, когда срез создается путем нарезки существующего массива.

И последнее, на что стоит обратить внимание: что, если мы продолжим присоединять элементы к `s2` вплоть до того момента, когда резервный массив заполнится? Каким будет состояние с точки зрения памяти? Давайте добавим еще три элемента, чтобы резервному массиву не хватило емкости:

```
s2 = append(s2, 3)
s2 = append(s2, 4)
s2 = append(s2, 5)  <-- На этом этапе резервный массив уже заполнен
```

Выполнение этого кода приводит к созданию еще одного резервного массива. На рис. 3.9 показано, как это отразится на памяти.

`s1` и `s2` теперь ссылаются на два разных массива. Поскольку `s1` по-прежнему представляет собой срез длиной 3 и емкостью 6, у него все еще есть некоторый доступный буфер, поэтому он продолжает ссылаться на исходный массив. Кроме того, новый резервный массив был создан копированием исходного, начиная с первого индекса `s2`. Вот почему новый массив начинается с 1, а не с 0.

Подводя итог, можно сказать, что *длина среза* — это количество доступных элементов в нем, тогда как *емкость среза* — это количество элементов в резервном массиве. Добавление элемента в полный срез (длина == емкость) приводит к созданию нового резервного массива с новой емкостью, копированию всех элементов из предыдущего массива и установлению указателя среза на новый массив.

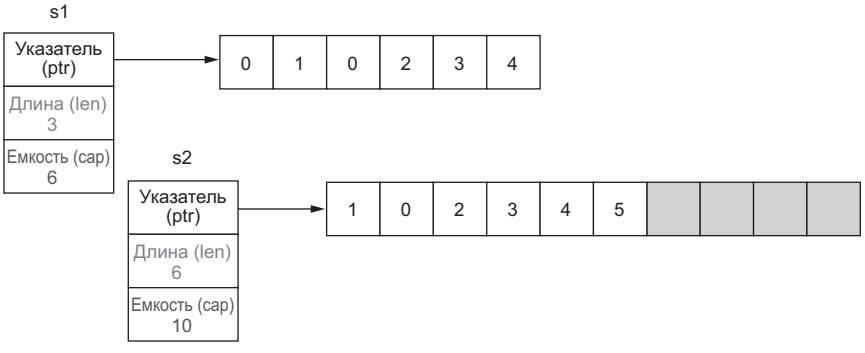


Рис. 3.9. Добавление элементов в s2 до тех пор, пока резервный массив не окажется заполненным

В следующем разделе используем понятия длины и емкости при инициализации среза.

3.5. ОШИБКА #21: НЕЭФФЕКТИВНАЯ ИНИЦИАЛИЗАЦИЯ СРЕЗА

Как было показано, при инициализации среза с помощью оператора `make` нужно указать длину и дополнительно емкость. Распространенная ошибка — забыть придать соответствующее значение обоим этим параметрам в тех случаях, когда это имеет смысл.

Предположим, нужно реализовать функцию `convert`, которая отображает срез `Foo` в срез `Bar`, и они оба будут иметь одинаковое количество элементов. Вот одна из таких реализаций, которая первым делом приходит на ум:

```
func convert(foos []Foo) []Bar {
    bars := make([]Bar, 0)  <-- Создается результирующий срез
    for _, foo := range foos {
        bars = append(bars, fooToBar(foo))  <-- Foo преобразуется в Bar, который присоединяется к срезу
    }
    return bars
}
```

Сначала мы инициализируем пустой срез элементов `Bar` с помощью `make([]Bar, 0)`. Затем используем `append` для добавления элементов `Bar`. Сначала `bars` пуст, поэтому добавление первого элемента создает резервный массив размером 1.

Каждый раз, когда резервный массив заполняется до конца, Go создает новый массив, удваивая его емкость (о чем шла речь в предыдущем разделе).

Такая логика создания нового массива, когда текущий оказывается заполненным, повторяется несколько раз, когда мы добавляем третий элемент, пятый, девятый и т. д. Предполагая, что входной срез содержит 1000 элементов, этот алгоритм требует выделения/создания десяти резервных массивов и копирования в общей сложности более 1000 элементов из одного массива в другой. Это приводит к необходимости дополнительных действий со стороны сборщика мусора по очистке памяти от всех этих временных резервных массивов.

Для улучшения производительности стоит помочь среде выполнения Go. Это можно сделать двумя способами. Первый вариант — переиспользовать тот же код, но создать в памяти срез с заданной емкостью:

```
func convert(foos []Foo) []Bar {
    n := len(foos)
    bars := make([]Bar, 0, n)

    for _, foo := range foos {
      bars = append(bars, fooToBar(foo))
    }
    return bars
}
```

Инициализация с нулевой длиной и заданной емкостью

bars обновляется, чтобы добавить новый элемент

Единственное изменение заключается в создании `bars` с емкостью, равной `n` — длине `foos`.

Go внутри себя предварительно резервирует место под массив из `n` элементов. Поэтому добавление элементов вплоть до `n` раз означает, что при каждом таком добавлении используется один и тот же резервный массив, поэтому резко сокращается количество действий по выделению места под массивы. Второй вариант — создать `bars` заданной длины:

```
func convert(foos []Foo) []Bar {
   n := len(foos)
   bars := make([]Bar, n)
   for i, foo := range foos {
      bars[i] = fooToBar(foo)
   }
   return bars
}
```

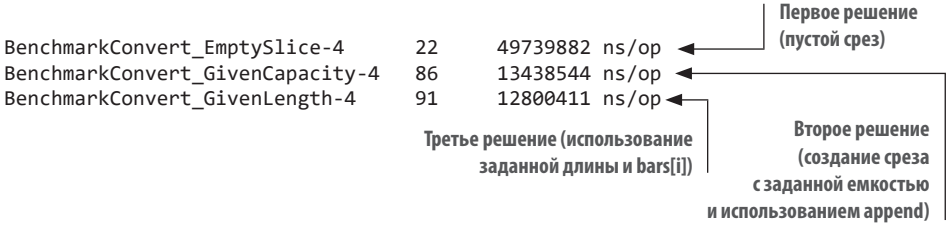
Инициализация с заданной длиной

Задание элемента i среза

Поскольку мы инициализируем срез с некоторой длиной, место для `n` элементов `Bar` уже выделено, и они все инициализированы нулевым значением.

Следовательно, чтобы установить значения элементов, используем не `append`, а `bar[i]`.

Какой из этих вариантов лучше? Выполним бенчмарк с тремя решениями и входным срезом из 1 миллиона элементов:



Как мы видим, в случае первого решения на производительность оказывается значительное влияние. Если мы продолжаем выделять массивы и копировать элементы, первый бенчмарк дает результат, который почти на 400 % медленнее, чем два других. Сравнивая второе и третье решения, можно заметить, что третье примерно на 4 % быстрее, потому что мы избегаем повторных вызовов встроенной функции `append`, которая выполняется несколько дольше по сравнению с прямым присваиванием.

Если задание емкости и использование `append` менее эффективно, чем задание длины и присвоение прямого индекса, почему этот подход используется в проектах Go? Рассмотрим пример в Pebble — в хранилище ключей и значений систем с открытым исходным кодом, разработанном Cockroach Labs (https://github.com/cockreachdb/pebble).

Функция `collectAllUserKeys` должна выполнить итерации по срезу структур, чтобы отформатировать определенный срез байтов. Результирующий срез будет в два раза длиннее входного:

```
func collectAllUserKeys(cmp Compare,
    tombstones []tombstoneWithLevel) [][]byte {
    keys := make([][]byte, 0, len(tombstones)*2)
    for _, t := range tombstones {
        keys = append(keys, t.Start.UserKey)
        keys = append(keys, t.End)
    }
    // ...
}
```

Здесь сознательный выбор заключается в использовании заданной емкости и функции `append`. Почему? Если бы мы использовали заданную длину вместо емкости, код был бы таким:

```
func collectAllUserKeys(cmp Compare,
    tombstones []tombstoneWithLevel) [][]byte {
    keys := make([][]byte, len(tombstones)*2)
    for i, t := range tombstones {
        keys[i*2] = t.Start.UserKey
        keys[i*2+1] = t.End
    }
    // ...
}
```

Посмотрите, каким сложным выглядит код для обработки индекса среза. Учитывая, что рассматриваемая функция не слишком чувствительна к производительности, было решено выбрать самый простой для чтения вариант.

Срезы и условия

А что случится, если будущая длина среза точно неизвестна? Например, что, если длина выходного среза зависит от какого-то условия?

```
func convert(foos []Foo) []Bar {
   // инициализация bars
   for _, foo := range foos {
      if something(foo) {
         // Добавление элемента к bar
      }
      }
   return bars
}
```

Добавление элемента Foo только в том случае, если выполняется определенное условие

В этом примере элемент `Foo` преобразуется в `Bar` и добавляется в срез только при определенном условии (`if something(foo)`). Должны ли мы инициализировать `bars` как пустой срез либо как срез с заданной длиной или емкостью?

Здесь нет строгого правила. Это извечная проблема для программистов: что лучше — загрузить в большей степени процессор или память? Возможно, если `something(foo)` верно в 99 % случаев, стоит инициализировать `bars` с заданной длиной или емкостью. Решение зависит от конкретной ситуации.

Преобразование одного типа среза в другой — частая операция в Go-разработке. Как мы видели, если длина будущего среза уже известна, для создания в памяти сначала пустого среза нет веской причины. Наши варианты — создать срез либо с заданной емкостью, либо с заданной длиной. Мы видели, что из этих двух решений второе работает немного быстрее. Но использование среза заданной емкости и функции `append` в некоторых контекстах может быть проще для реализации и чтения.

Дальше поговорим о разнице между нулевыми и пустыми срезами и обсудим, почему это важно для Go-разработчиков.

3.6. ОШИБКА #22: ПУТАТЬ ПУСТЫЕ И НУЛЕВЫЕ СРЕЗЫ

Разработчики Go довольно часто путают нулевые и пустые срезы. В зависимости от конкретной ситуации может понадобиться использовать то или другое. Между тем некоторые библиотеки делают между ними различие. Для продуктивной работы со срезами важно разобраться в этих понятиях. Прежде чем рассматривать какие-либо примеры, приведу определения:

- Срез считается пустым, если его длина равна 0.
- Срез считается нулевым, если его значение равно `nil`.

Рассмотрим способы инициализации среза. Сможете догадаться, к чему приводит выполнение следующего кода? Каждый раз мы будем выводить, является срез пустым или нулевым:

```
func main() {
    var s []string  <-- Вариант 1 (значение 0)
    log(1, s)

    s = []string(nil)  <-- Вариант 2
    log(2, s)

    s = []string{}  <-- Вариант 3
    log(3, s)

    s = make([]string, 0)  <-- Вариант 4
    log(4, s)
}

func log(i int, s []string) {
    fmt.Printf("%d: empty=%t\tnil=%t\n", i, len(s) == 0, s == nil)
}
```

Вот что выведет этот код:

```
1: empty=true nil=true
2: empty=true nil=true
3: empty=true nil=false
4: empty=true nil=false
```

Все срезы пусты, то есть их длина равна 0. Поэтому и нулевой срез тоже пустой. Но только первые два среза нулевые. Если есть несколько способов инициализировать срез, то какой вариант стоит предпочесть? Отмечу две вещи:

- Одно из основных различий между нулевым и пустым срезом касается выделения памяти. Инициализация нулевого среза не требует выделения памяти, чего нельзя сказать о пустом срезе.
- Вызов встроенной функции `append` будет работать независимо от того, является ли срез нулевым. Например:

```
var s1 []string
fmt.Println(append(s1, "foo")) // [foo]
```

Следовательно, если функция возвращает срез, мы не должны делать так, как в других языках, и возвращать ненулевую коллекцию из соображений безопасности. Поскольку нулевой срез не требует какого-либо резервирования памяти, следует возвращать нулевой, а не пустой срез. Посмотрим на функцию, которая возвращает срез строк:

```
func f() []string {
    var s []string
    if foo() {
        s = append(s, "foo")
    }
    if bar() {
        s = append(s, "bar")
    }
    return s
}
```

Если и `foo`, и `bar` равны `false`, мы получаем пустой срез. Чтобы предотвратить создание пустого среза без особого на то основания, следует выбрать вариант 1 (`var s []string`). Можно выбрать и вариант 4 (`make([]string, 0)`) со строкой нулевой длины, но по сравнению с вариантом 1 это не принесет никакой пользы, а кроме того, еще и потребует резервирования места в памяти.

Но в случае, когда нужно создать срез известной длины, следует использовать вариант 4 (`s := make([]string, length)`), как показано в следующем примере:

```
func intsToStrings(ints []int) []string {
    s := make([]string, len(ints))
    for i, v := range ints {
        s[i] = strconv.Itoa(v)
    }
    return s
}
```

Как уже обсуждалось при разборе ошибки #21 (неэффективная инициализация среза), при таком сценарии нужно устанавливать его длину (или емкость), чтобы избежать дополнительных выделений памяти и копирования. Поэтому в случае нашего примера, где указаны разные способы инициализации среза, остается два варианта:

- Вариант 2: `s := []string(nil)`
- Вариант 3: `s := []string{}`

Вариант 2 нечасто используемый, но может быть полезен как синтаксический сахар, поскольку мы можем задать нулевой срез одной строкой, например, используя `append`:

```
s := append([]int(nil), 42)
```

Если бы мы использовали вариант 1 (`var s []string`), то потребовалось бы две строки кода. Хотя это и не самая важная оптимизации читабельности, но о таком варианте тоже стоит знать.

> **ПРИМЕЧАНИЕ** В разделе с ошибкой #24 (неправильно создавать копии срезов) мы рассмотрим один случай, когда добавление элемента к нулевому срезу имеет смысл.

Теперь рассмотрим вариант 3 (`s := []string{}`). Он рекомендуется для создания среза с начальными элементами:

```
s := []string{"foo", "bar", "baz"}
```

Если создавать срез с начальными элементами не нужно, не используйте этот вариант. Он дает те же преимущества, что и вариант 1 (`var s []string`), за исключением того, что срез оказывается ненулевым, следовательно, требует выделения памяти. Поэтому избегайте варианта 3, если указание начальных элементов необязательно.

> **ПРИМЕЧАНИЕ** Некоторые линтеры могут распознавать использование варианта 3 без начальных значений и рекомендуют изменить его на вариант 1. Но помните, что это также меняет семантику с ненулевого на нулевой срез.

Упомянем, что в некоторых библиотеках учитываются различия между нулевыми и пустыми срезами. Так обстоит дело, например, с пакетом `encoding/`

json. В следующих примерах маршалируются две структуры, одна из которых содержит нулевой, а вторая — ненулевой, но пустой срез:

```
var s1 []float32  ◄——  Нулевой срез
    customer1 := customer{
    ID: "foo",
    Operations: s1,
}
b, _ := json.Marshal(customer1)
fmt.Println(string(b))
s2 := make([]float32, 0)  ◄——  Ненулевой пустой срез
    customer2 := customer{
    ID: "bar",
    Operations: s2,
}
b, _ = json.Marshal(customer2)
fmt.Println(string(b))
```

Запустив код из этого примера, мы увидим, что результаты маршалинга для этих двух структур различны:

```
{"ID":"foo","Operations":null}
{"ID":"bar","Operations":[]}
```

Здесь нулевой срез маршалируется как элемент null, тогда как ненулевой пустой срез маршалируется как пустой массив. Если мы работаем в контексте строгих JSON-клиентов, которые различают null и [], об этом различии очень важно помнить.

Пакет encoding/json — не единственный пакет из стандартной библиотеки, в котором проводится такое различие. Например, Reflect.DeepEqual возвращает false, если мы сравниваем нулевой и ненулевой пустой срез, что следует помнить, например, в контексте юнит-тестов. В любом случае при работе со стандартной библиотекой или какими-то внешними библиотеками нужно следить за тем, чтобы при использовании той или иной версии наш код не приводил к неожиданным результатам.

Подводя итог, можно сказать, что в Go есть различие между нулевыми и пустыми срезами. Нулевой срез равен nil, тогда как пустой срез имеет нулевую длину. Нулевой срез пуст, но пустой срез не обязательно равен nil. Кроме того, нулевой срез не требует никакого резервирования памяти. В этом разделе мы видели, как инициализировать срез в зависимости от контекста, используя:

- var s []string, если нет определенности в отношении конечной длины и срез может быть пустым.

- `[]string(nil)` как синтаксический сахар для создания нулевого и пустого среза.
- `make([]string, length)`, если будущая длина известна.

Последнего возможного варианта — `[]string{}` — следует избегать, если инициализируется срез без элементов. Наконец, следует проверять, предусмотрены ли в используемых библиотеках различия между нулевыми и пустыми срезами, чтобы предотвратить неожиданное поведение.

В следующем разделе посмотрим на лучший способ проверки пустого среза после вызова функции.

3.7. ОШИБКА #23: НЕПРАВИЛЬНО ПРОВЕРЯТЬ ПУСТОТУ СРЕЗА

В предыдущем разделе мы узнали, что между нулевыми и пустыми срезами есть различие. Есть ли идиоматический способ проверить, содержит ли какой-то срез элементы? Отсутствие четкого ответа на этот вопрос может привести к малозаметным ошибкам.

В этом примере вызываем функцию `getOperations`, которая возвращает срез типа `float32`. Далее мы хотим вызывать функцию `handle` только в том случае, если этот срез содержит элементы. Вот первая (ошибочная) версия кода этих действий:

```
func handleOperations(id string) {
    operations := getOperations(id)
    if operations != nil {  <-- Проверяется, является ли срез operations нулевым
        handle(operations)
    }
}
func getOperations(id string) []float32 {
    operations := make([]float32, 0)  <-- Инициализируется срез operations
    if id == "" {
        return operations  <-- Возвращается operations, если заданный id пуст
    }
    // Добавление элементов к operations
    return operations
}
```

Мы определяем, есть ли в срезе элементы, проверяя, не является ли срез `operations` нулевым. Но у этого кода есть проблема: функция `getOperations` никогда не возвращает нулевой срез, она возвращает пустой срез. Поэтому значение проверки `operations != nil` всегда будет `true`.

Что делать? Один из подходов — изменить `getOperations` так, чтобы он возвращал нулевой срез, если `id` пуст:

```
func getOperations(id string) []float32 {
    operations := make([]float32, 0)
    if id == "" {
        return nil  ◄──  Возврат nil вместо operations
    }
    // Add elements to operations
    return operations
}
```

Вместо того чтобы возвращать `operations`, если `id` пуст, мы возвращаем `nil`. Таким образом, реализуемая нами проверка на нулевой срез соответствует действительности. Но такой подход работает не во всех ситуациях — контекст, в котором мы находимся, не всегда позволяет изменить вызываемый объект. Например, если используется внешняя библиотека, мы не будем создавать пулреквест только для того, чтобы заменить пустые срезы на нулевые.

Как тогда проверить, является срез пустым или нулевым? Нужно проверить его длину:

```
func handleOperations(id string) {
    operations := getOperations(id)
    if len(operations) != 0 {  ◄──  Проверка длины среза
        handle(operations)
    }
}
```

В предыдущем разделе мы упоминали, что пустой срез по определению имеет нулевую длину. При этом нулевые срезы всегда пусты. Поэтому, проверяя длину среза, мы учитываем все возможные сценарии:

- Если срез равен `nil`, то `len(operations) != 0` принимает значение `false`.
- Если срез не равен `nil`, а является пустым, то значение `len(operations) != 0` также будет `false`.

Следовательно, проверка длины — лучший способ, поскольку мы не всегда можем контролировать то, что будет происходить в результате выполнения вызываемых функций. Как говорится в Вики по Go, при проектировании интерфейсов следует избегать различий между нулевыми и пустыми срезами, которые могут приводить к малозаметным ошибкам программирования. При возврате срезов не должно быть ни семантической, ни технической разницы, возвращаем мы `nil` или пустой срез. Оба варианта должны означать одно и то же для вызывающей

стороны. Тот же принцип применим и к картам. Чтобы проверить, пуста ли карта, проверяйте ее длину, а не то, равна ли она `nil`.

В следующем разделе узнаем, как правильно создавать копии срезов.

3.8. ОШИБКА #24: НЕПРАВИЛЬНО СОЗДАВАТЬ КОПИИ СРЕЗОВ

Встроенная функция `copy` позволяет копировать элементы из исходного среза в другой. Хотя эта встроенная функция удобна, Go-разработчики не всегда правильно ее понимают. Рассмотрим одну распространенную ошибку, которая приводит к копированию неправильного количества элементов.

В следующем примере мы создаем один срез и копируем его элементы в другой. Что выведет этот код?

```
src := []int{0, 1, 2}
var dst []int
copy(dst, src)
fmt.Println(dst)
```

В результате мы получим `[]`, а не `[0 1 2]`. Что же мы упустили?

Чтобы эффективно использовать функцию `copy`, важно понимать, что число элементов, скопированных в другой срез, определяется минимумом между:

- длиной исходного среза;
- длиной второго среза.

В предыдущем примере `src` — это срез длиной 3, а `dst` — срез с нулевой длиной, поскольку он инициализируется со своим нулевым значением. Поэтому функция `copy` копирует количество элементов, равное минимуму в наборе 3 и 0: здесь этот минимум будет равен 0. Поэтому полученный срез будет пустым.

Если мы хотим выполнить полное копирование, второй срез должен иметь длину больше или равную длине исходного. Здесь мы устанавливаем длину, отталкиваясь от параметров исходного среза:

```
src := []int{0, 1, 2}
dst := make([]int, len(src))    ◄—— Создается срез dst, но уже с заданной длиной
copy(dst, src)
fmt.Println(dst)
```

Поскольку `dst` теперь срез, инициализированный с длиной, равной 3, то копируются три элемента. На этот раз, если мы запустим код, его результатом будет `[0 1 2]`.

> **ПРИМЕЧАНИЕ** Другая распространенная ошибка — инвертировать порядок аргументов при вызове функции `copy`. Помните, что срез, в который происходит копирование, — первый аргумент, а срез-источник — второй.

Использование встроенной функции `copy` — не единственный способ копирования элементов среза. Есть альтернативы, самая известная из которых следующая (в ней используется `append`):

```
src := []int{0, 1, 2}
dst := append([]int(nil), src...)
```

Мы присоединяем элементы из исходного среза в другой, нулевой. Следовательно, этот код создает копию среза длиной 3 и емкостью 3. Эта альтернатива имеет то преимущество, что все действия выполняются в одной строке. Но использование функции `copy` более идиоматично и, следовательно, легче для понимания, даже несмотря на то, что требует дополнительной строки.

Копирование элементов из одного среза в другой — довольно частая операция. При использовании функции `copy` мы должны помнить, что количество скопированных элементов определяется минимумом между длинами двух срезов.

Имейте в виду, что есть и альтернативные способы копирования среза, поэтому не стоит удивляться, если мы будем находить их в текстах кодов.

Продолжим изучать срезы и связанную с ними распространенную ошибку при использовании `append`.

3.9. ОШИБКА #25: НЕОЖИДАННЫЕ ПОБОЧНЫЕ ЭФФЕКТЫ ПРИ ИСПОЛЬЗОВАНИИ APPEND В ОПЕРАЦИЯХ СО СРЕЗАМИ

В этом разделе обсуждается распространенная ошибка при использовании `append`, которая в некоторых ситуациях может приводить к неожиданным побочным эффектам. В следующем примере мы инициализируем срез `s1`, создаем `s2` путем нарезки `s1` и создаем `s3`, добавляя элемент к `s2`:

```
s1 := []int{1, 2, 3}
s2 := s1[1:2]
s3 := append(s2, 10)
```

Мы инициализируем срез `s1`, содержащий три элемента, а `s2` создается из среза `s1`. Затем к `s3` мы вызываем функцию `append`. Каким будет состояние этих трех срезов после выполнения кода? Догадаетесь?

На рис. 3.10 показано, как будут выглядеть в памяти оба среза после выполнения второй строки, то есть после создания `s2`. `s1` — это срез длиной 3 и емкостью 3, а `s2` — срез длиной 1 и емкостью 2, за ними обоими стоит один и тот же резервный массив, о котором мы уже упоминали. При добавлении элемента с помощью `append` проверяется, заполнен ли срез (длина == емкость, то есть length == capacity). Если он не заполнен, функция `append` добавляет в него элемент, обновляя резервный массив и создавая срез, длина которого увеличивается на 1.

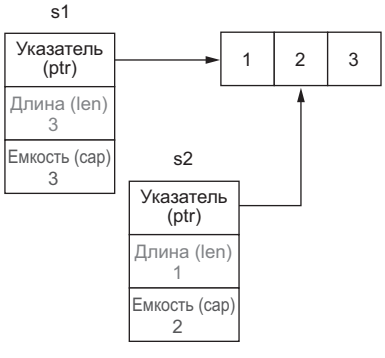


Рис. 3.10. Оба среза имеют один и тот же резервный массив, но разные длины и емкости

В этом примере `s2` не заполнен, в него можно записать еще один элемент. На рис. 3.11 показано конечное состояние этих трех срезов.

В резервном массиве мы обновили последний элемент, придав ему значение 10. Поэтому если мы выведем то, что находится во всех срезах, получим:

```
s1=[1 2 10], s2=[2], s3=[2 10]
```

Содержимое среза `s1` было изменено, хотя мы не обновляли `s1 [2]` или `s2 [1]` напрямую. Об этом нужно помнить, чтобы избежать проявления каких-то последствий, которые мы не собирались получить.

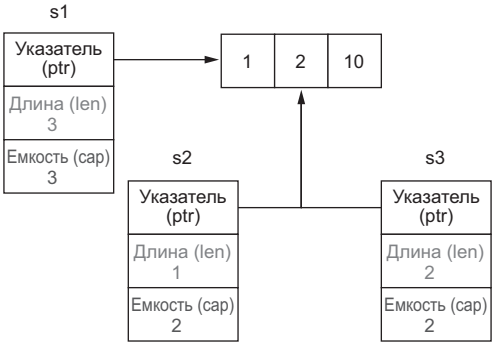


Рис. 3.11. За всеми срезами стоит один и тот же резервный массив

Посмотрим, как этот принцип влияет на передачу результата операции нарезки в функцию. В следующем фрагменте кода инициализируем срез с тремя элементами и вызываем функцию, которая имеет дело только с его первыми двумя элементами:

```
func main() {
    s := []int{1, 2, 3}
    f(s[:2])
    // Использование s
}
func f(s []int) {
    // Обновление s
}
```

В такой реализации этих действий, если `f` обновляет первые два элемента, изменения видны срезу в `main`. Но если `f` вызывает `append`, он обновляет третий элемент среза, хотя мы передаем функции только два элемента. Например,

```
func main() {
    s := []int{1, 2, 3}
    f(s[:2])
    fmt.Println(s) // [1 2 10]
}
func f(s []int) {
    _ = append(s, 10)
}
```

Если мы хотим защитить третий элемент из соображений безопасности, то есть гарантировать, что `f` не обновит его, существует два варианта.

Первый вариант — передать копию среза, а затем создать результирующий срез:

```
func main() {
    s := []int{1, 2, 3}
    sCopy := make([]int, 2)
    copy(sCopy, s)
    f(sCopy)
    result := append(sCopy, s[2])
    // Использование результата
}

func f(s []int) {
    // Обновление s
}
```

Копирование первых двух элементов s в sCopy

Присоединяет s[2] к sCopy для формирования результирующего среза

Поскольку мы передаем копию в `f`, то даже если эта функция вызывает `append`, это не приведет к появлению какого-либо побочного эффекта за пределами диапазона первых двух элементов. Недостаток этого варианта в том, что в этом случае код становится более сложным для чтения и появляется дополнительная копия, что может стать проблемой, если срез большой по размеру.

Второй вариант можно использовать для ограничения диапазона влияния возможных сайд-эффектов только первыми двумя элементами. Эта опция включает так называемое *полное выражение среза* (full slice expression): `s[low:high:max]`. Этот оператор создает срез, аналогичный созданному с помощью `s[low:high]`, за исключением того, что емкость получающегося среза равна `max-low`. Вот пример вызова `f`:

```
func main() {
    s := []int{1, 2, 3}
    f(s[:2:2])
    // Использование s
}
func f(s []int) {
    // Обновление s
}
```

Передача «подсреза» с использованием полного выражения среза

Здесь срез, переданный в `f`, — это не `s[:2]`, а `s[:2:2]`. Следовательно, емкость среза равна 2 – 0 = 2, как показано на рис. 3.12.

При передаче `s[:2:2]` мы можем ограничить диапазон влияния эффектов первыми двумя элементами. Это также избавляет нас от необходимости выполнять копирование среза.

При использовании нарезки мы должны помнить, что можем столкнуться с ситуацией, приводящей к непреднамеренным побочным эффектам. Если

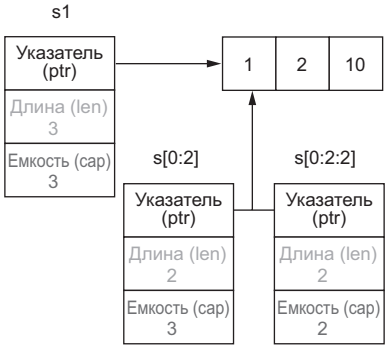


Рис. 3.12. `s[0:2]` создает срез длиной 2 и емкостью 3, тогда как `s[0:2:2]` создает срез длиной 2 и емкостью 2

результирующий срез имеет длину меньше, чем его емкость, append может изменять исходный срез. Если мы хотим ограничить диапазон влияния возможных сайд-эффектов, можно использовать либо копию среза, либо полное выражение среза, что не позволит делать копию.

В следующем разделе продолжим обсуждение срезов, но в контексте потенциальных утечек памяти.

3.10. ОШИБКА #26: СРЕЗЫ И УТЕЧКИ ПАМЯТИ

В этом разделе мы покажем, что нарезка существующего среза или массива может в некоторых случаях приводить к утечкам памяти. Обсудим два случая: один, при котором происходит утечка емкости, и другой — связанный с указателями.

3.10.1. Утечки емкости

В случае утечки емкости представим реализацию какого-то собственного двоичного протокола. Сообщение может содержать 1 миллион байтов, а в первых пяти байтах хранится тип этого сообщения. При выполнении кода мы воспринимаем эти сообщения и для целей аудита хотим сохранять в памяти последние 1000 типов сообщений. Вот скелет функции:

```
func consumeMessages() {
    for {
        msg := receiveMessage()
        // Производится какая-то операция с msg
        storeMessageType(getMessageType(msg))
    }
}
```

Происходит получение нового []-байтового среза, записываемого в msg

В память записываются 1000 последних типов сообщений

```
func getMessageType(msg []byte) []byte {
    return msg[:5]
}
```

Вычисление типа сообщения нарезкой msg

Функция `getMessageType` вычисляет тип сообщения, нарезая входной срез. При тестировании этого кода мы видим, что все в порядке. Но при развертывании приложения замечаем, что оно потребляет около 1 Гбайт памяти. Почему?

Операция нарезки сообщения `msg` с использованием `msg[:5]` создает срез длиной 5. Но его емкость остается такой же, как у исходного среза. Остальные элементы по-прежнему сохраняются в памяти, даже если в итоге отсутствуют ссылки на `msg`. Рассмотрим пример с большим сообщением — длиной 1 миллион байтов, как показано на рис. 3.13.

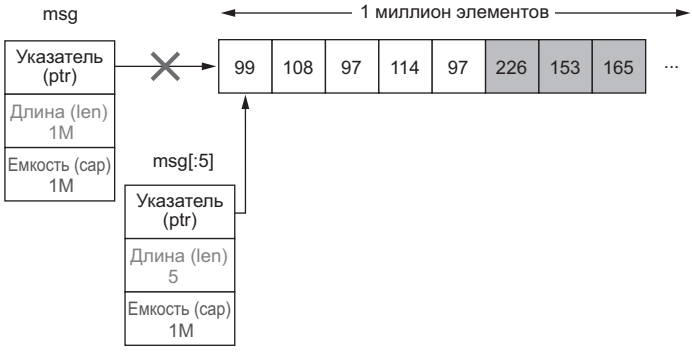


Рис. 3.13. После новой итерации в цикле msg больше не используется. Однако msg[:5] будет по-прежнему использовать его резервный массив

Резервный массив среза по-прежнему содержит 1 миллион байтов после операции нарезки. Следовательно, если мы храним в памяти 1000 сообщений, то вместо примерно 5 Кбайт мы занимаем около 1 Гбайт.

Как решить эту проблему? Можно сделать копию среза вместо нарезки `msg`:

```
func getMessageType(msg []byte) []byte {
    msgType := make([]byte, 5)
    copy(msgType, msg)
    return msgType
}
```

Поскольку мы выполняем копирование, `msgType` представляет собой срез длиной 5 и емкостью 5 независимо от размера полученного сообщения.

Следовательно, мы сохраняем в памяти только 5 байт для каждого типа сообщения.

Полные выражения среза и утечка емкости

А что насчет использования полного выражения среза для решения этой проблемы? Посмотрим пример:

```
func getMessageType(msg []byte) []byte {
   return msg[:5:5]
}
```

Здесь `getMessageType` возвращает уменьшенную версию исходного среза — срез длиной 5 и емкостью 5. Но сможет ли сборщик мусора восстановить недоступное пространство из байта 5? Спецификация Go официально не определяет его поведение. Но с помощью `runtime.Memstats` мы можем записывать статистику распределителя памяти, например количество байтов, выделенных в куче:

```
func printAlloc() {
   var m runtime.MemStats
   runtime.ReadMemStats(&m)
   fmt.Printf("%d KB\n", m.Alloc/1024)
}
```

Если мы вызываем эту функцию после вызова `getMessageType` и `runtime.GC()`, чтобы принудительно запустить сборку мусора, недоступное пространство не восстанавливается. Резервный массив целиком все еще находится в памяти. Следовательно, использование полного выражения среза недопустимо (если это не будет как-то исправлено в будущем обновлении Go).

В качестве эмпирического правила запомните, что нарезка большого среза или массива может потенциально привести к высокому потреблению памяти. Остающееся в памяти пространство не будет восстановлено сборщиком мусора, и мы можем сохранять в памяти очень большой резервный массив, несмотря на использование только нескольких элементов. Использование копии среза — это способ предотвращения такой ситуации.

3.10.2. Срез и указатели

Мы видели, что нарезка может вызвать утечку из-за тонкостей, связанных с емкостью среза. А что насчет элементов, которые все еще являются частью

резервного массива, но находятся за пределами диапазона, определяемого длиной? Собирает ли их сборщик мусора?

Возьмем структуру `Foo`, содержащую байтовый срез:

```
type Foo struct {
    v []byte
}
```

После каждого шага мы проверяем, как производится распределение пространства в памяти:

- Выделяем срез из 1000 элементов `Foo`.
- Выполняем итерацию по каждому элементу `Foo` и каждый раз выделяем 1 Мбайт для среза `v`.
- Вызываем `keepFirstTwoElementsOnly`, который возвращает только первые два элемента с использованием нарезки, а затем вызываем сборщик мусора.

Мы хотим увидеть, как поведет себя память после вызова `keepFirstTwoElementsOnly` и сборки мусора. Вот сценарий в Go (повторно используем упомянутую ранее функцию `printAlloc`):

```
func main() {
    foos := make([]Foo, 1_000)
    printAlloc()
    for i := 0; i < len(foos); i++ {
        foos[i] = Foo{
            v: make([]byte, 1024*1024),
        }
    }
    printAlloc()
    two := keepFirstTwoElementsOnly(foos)
    runtime.GC()
    printAlloc()
    runtime.KeepAlive(two)
}
func keepFirstTwoElementsOnly(foos []Foo) []Foo {
    return foos[:2]
}
```

Производится резервирование памяти под срез из 1000 элементов

Для каждого элемента создается срез в 1 Мбайт

Сохраняются только первые два элемента

Вызывается GC, чтобы принудительно вызвать очистку кучи

Сохраняется ссылка на переменную two

В этом примере мы создаем срез `foos`, затем для каждого элемента — срезы размером в 1 Мбайт, а потом вызываем `keepFirstTwoElementsOnly` и сборщик

мусора. В конце концов мы используем `runtime.KeepAlive`, чтобы после сборки мусора сохранить ссылку на переменную `two` и не удалять ее.

Мы ожидаем, что сборщик мусора удалит 998 оставшихся элементов `Foo` и данные, выделенные для среза, поскольку доступа к этим элементам больше нет. Но это не так. Например, приведенный код может вывести следующее:

```
83 KB
1024072 KB
1024072 KB  ◄——  После операции нарезки
```

Сначала выделяется около 83 Кбайт данных. Действительно, мы зарезервировали память под 1000 нулевых значений в `Foo`. Второй результат выделяет 1 Мбайт на срез, что увеличивает память. Но обратите внимание на то, что сборщик мусора не очистил оставшиеся 998 элементов после последнего шага. В чем причина?

Важно помнить об этом правиле при работе со срезами: если элемент является указателем или структурой с полями-указателями, сборщик мусора не восстановит память, зарезервированную под эти элементы. В нашем примере, поскольку `Foo` содержит срез (а срез — это указатель поверх резервного массива), память, остающаяся занятой под 998 элементов `Foo` и их срезы, не восстанавливается. Таким образом, хотя эти 998 элементов становятся недоступными, они остаются в памяти до тех пор, пока сохраняется ссылка на переменную, возвращаемую функцией `keepFirstTwoElementsOnly`.

Какие есть варианты, чтобы не допустить утечки оставшихся элементов `Foo`? Первый вариант, опять же, — создать копию среза:

```
func keepFirstTwoElementsOnly(foos []Foo) []Foo {
    res := make([]Foo, 2)
    copy(res, foos)
    return res
}
```

Поскольку мы копируем только первые два элемента среза, сборщик мусора будет знать, что на оставшиеся 998 элементов больше не будет никаких ссылок и память под них теперь можно очистить.

Если мы хотим сохранить базовую емкость в 1000 элементов, то есть пометить срезы для оставшихся элементов как `nil` в явном виде, есть такой вариант:

```
func keepFirstTwoElementsOnly(foos []Foo) []Foo {
    for i := 2; i < len(foos); i++ {
```

```
        foos[i].v = nil
    }
    return foos[:2]
}
```

Здесь мы возвращаем срез длиной 2 и емкостью 1000, но устанавливаем срезы остальных элементов равными `nil`. Следовательно, сборщик мусора теперь сможет очистить 998 резервных массивов.

Какой вариант лучше? Если мы не хотим сохранять емкость на уровне 1000 элементов, то, вероятно, первый. Но ответ на этот вопрос может зависеть и от соотношения элементов. На рис. 3.14 представлены варианты, из которых мы можем выбрать, предполагая, что срез содержит `n` элементов, а нам нужно сохранить только `i` элементов.

Рис. 3.14. Вариант 1 используется вплоть до `i`-го элемента, а далее используется вариант 2

При использовании первого варианта создается копия `i` элементов. Следовательно, он должен применяться от элемента 0 до элемента `i`. При втором варианте оставшиеся срезы делаются равными `nil`, поэтому он должен применяться от элемента `i` до элемента `n`. Если особо важна скорость выполнения операций, а `i` ближе к `n`, чем к нулю, то возможно рассмотреть использование только второго варианта. Он потребует перебора меньшего количества элементов (по крайней мере два варианта действий стоит сравнить).

В этом разделе мы рассмотрели две потенциальные проблемы с утечкой памяти. Первая заключалась в нарезке существующего среза или массива для сохранения их емкости. Если мы обрабатываем большие срезы и оставляем только их части, делая их повторную нарезку, останется зарезервированным большой объем памяти, который никак не используется. Вторая проблема заключается в том, что когда мы используем операцию нарезки с указателями или структурами с полями указателей, нужно помнить, что сборщик мусора не очистит память, используемую этими элементами. В таком случае есть два варианта: либо выполнить копирование, либо явно пометить оставшиеся элементы или их поля равными `nil`.

Теперь обсудим карты в контексте их инициализации.

3.11. ОШИБКА #27: НЕЭФФЕКТИВНО ИНИЦИАЛИЗИРОВАТЬ КАРТЫ

В этом разделе поговорим о проблеме инициализации карт. Чтобы понять, почему важна настройка их инициализации, сначала вспомним основы реализации карт в Go.

3.11.1. Концепции

Карта представляет собой неупорядоченный набор пар «ключ — значение», в котором все ключи различны. В Go карта основана на структуре данных хеш-таблицы. Хеш-таблица представляет собой массив сегментов, каждый из которых является указателем на массив пар «ключ — значение», как показано на рис. 3.15.

На рис. 3.15 за хеш-таблицей стоит массив из четырех элементов. Если присмотреться к индексу массива, мы заметим, что один сегмент состоит из одной-единственной пары «ключ — значение» (элемент): `"two"/2`. Каждый сегмент имеет фиксированный размер из восьми элементов.

Рис. 3.15. Пример хеш-таблицы с особым вниманием к сегменту 0

Каждая операция (чтение, обновление, вставка, удаление) выполняется путем ассоциирования ключа с индексом массива. Этот шаг зависит от хеш-функции.

Эта функция стабильна, потому что мы хотим, чтобы она всякий раз возвращала один и тот же сегмент, если задан один и тот же ключ. В предыдущем примере `hash("two")` возвращает 0. Следовательно, элемент хранится в сегменте, на который ссылается индекс массива 0.

Если мы вставляем другой элемент и хеширование ключа возвращает тот же индекс, Go добавляет еще один элемент в тот же сегмент. На рис. 3.16 показан этот результат.

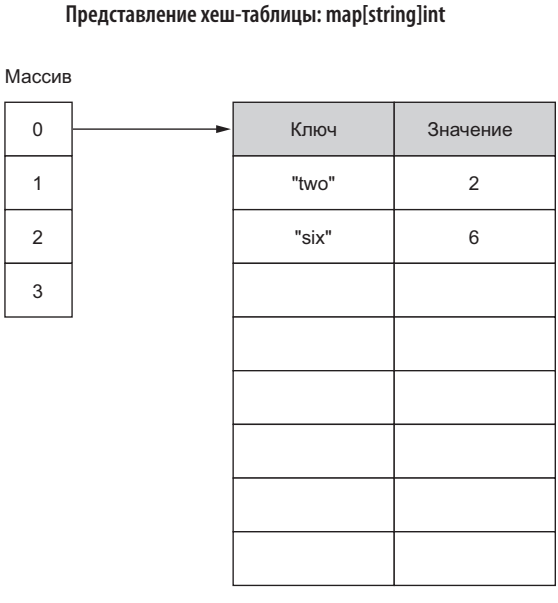


Рис. 3.16. `hash("six")` возвращает значение 0, поэтому элемент записывается в тот же сегмент

В случае вставки в уже заполненный сегмент (переполнение сегмента) Go создает еще один сегмент из восьми элементов и связывает с ним предыдущий сегмент. На рис. 3.17 показан результат этого.

Что касается операций чтения, обновления и удаления, то Go должен вычислить соответствующий индекс массива. Затем Go последовательно перебирает все ключи, пока не найдет заданный. Таким образом, в наихудшем случае временна́я сложность в результате этих трех операций равна $O(p)$, где p — общее количество элементов в сегментах (по умолчанию один сегмент, несколько сегментов в случае переполнения).

А теперь обсудим, почему эффективная инициализация карты столь важна.

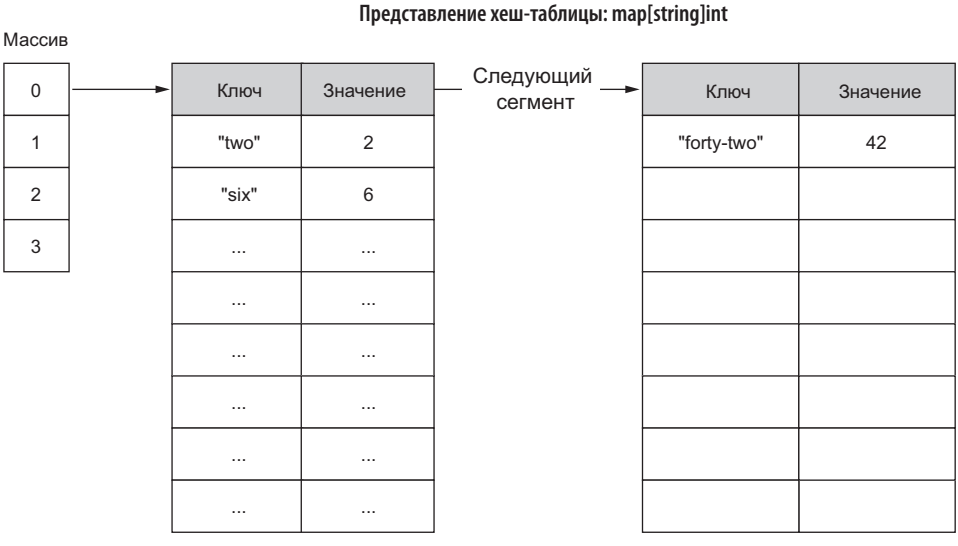


Рис. 3.17. В случае переполнения сегмента Go создает новый сегмент и устанавливает его связь с предыдущим

3.11.2. Инициализация

Чтобы понять проблемы, связанные с возможной неэффективной инициализацией карты, создадим тип `map[string]int`, содержащий три элемента:

```
m := map[string]int{
    "1": 1,
    "2": 2,
    "3": 3,
}
```

За этой картой стоит резервный массив, состоящий из одной записи, а следовательно, из одного сегмента. Что произойдет, если добавить 1 миллион элементов? В этом случае этой одной записи будет недостаточно, потому что поиск ключа в худшем случае будет означать перебор тысяч сегментов. Поэтому карта должна иметь возможность автоматически увеличиваться, чтобы соответствовать количеству элементов.

Когда карта увеличивается в размере, она удваивает количество своих сегментов. Каковы условия для такого увеличения карты?

- Среднее количество элементов в сегментах (называемое коэффициентом загрузки) превышает определенную константу, которая равна 6.5 (но ее значение может измениться в будущих версиях, поскольку это внутренний параметр Go).
- Слишком много сегментов оказываются переполненными (содержат более восьми элементов).

Когда размер карты растет, все ключи снова пересылаются во все сегменты. Вот почему при худшем сценарии вставка ключа может быть операцией $O(n)$, где n — это общее количество элементов в карте.

Мы видели при рассмотрении срезов, что если бы заранее знать количество элементов, которые нужно добавить в какой-то срез, то можно было бы инициализировать его с заданным размером или емкостью. Это позволяет избежать повторения дорогостоящей операции роста среза. Аналогичная ситуация и с картами. Можно использовать встроенную функцию `make`, чтобы при создании карты указать ее начальный размер. Например, если мы хотим инициализировать карту, которая должна содержать 1 миллион элементов, то сделать это можно так:

```
m := make(map[string]int, 1_000_000)
```

В случае с картами можно задавать встроенной функции `make` в качестве аргумента только начальный размер, а не емкость, как это было в случае со срезами. Здесь мы оперируем только с одним аргументом.

Указав размер, мы делаем некий намек на количество элементов, которые, как ожидается, будут содержаться в карте. Внутренними средствами карта создается с количеством сегментов, соответствующим хранению одного миллиона элементов. Это экономит процессорное время, поскольку карте не нужно на лету создавать какие-либо сегменты и перебалансировать их.

Кроме того, указание размера n не означает создания карты с максимальным количеством элементов, равным n. При необходимости мы можем добавить больше чем n элементов. Это указание означает запрос к среде выполнения Go на выделение места в памяти для карты с как минимум n элементами, что будет полезно, если мы заранее знаем размер.

Чтобы понять, почему важно указывать размер, запустим два бенчмарка. Первый вставляет 1 миллион элементов в карту без установки ее начального размера; во втором случае мы инициализируем карту с заданным размером:

```
BenchmarkMapWithoutSize-4 6 227413490 ns/op
BenchmarkMapWithSize-4 13 91174193 ns/op
```

Второй вариант, в котором задается начальный размер, выполняется примерно на 60 % быстрее. Задавая размер, мы предотвращаем операции по увеличению размера карты, необходимые для корректной вставки элементов.

Поэтому, как и в случае со срезами, если мы заранее знаем количество элементов, которые будет содержать карта, следует ее создавать, указывая ее начальный размер. Это позволяет избежать потенциального увеличения размера карты, что потребует значительных вычислительных ресурсов, поскольку влечет повторное выделение достаточного пространства в памяти и перебалансировку всех ее элементов.

Продолжим наш разговор о картах и рассмотрим распространенную ошибку, приводящую к утечкам памяти.

3.12. ОШИБКА #28: КАРТЫ И УТЕЧКИ ПАМЯТИ

При работе с картами в Go нужно понимать некоторые важные характеристики того, как карты увеличиваются и уменьшаются. Углубление в эту тему поможет предотвратить проблемы, вызывающие утечки памяти.

Для начала, чтобы рассмотреть конкретный пример этой проблемы, создадим сценарий, где будем работать со следующей картой:

```
m := make(map[int][128]byte)
```

Каждый элемент *m* представляет собой массив из 128 байтов. Сделаем следующее:

- Создадим в памяти пустую карту.
- Добавим в нее 1 миллион элементов.
- Сотрем все эти элементы и вызовем выполнение GC.

После каждого шага выведем размер кучи (в этот раз используя мегабайты). Это покажет, как в этом примере ведет себя память:

```
n := 1_000_000
m := make(map[int][128]byte)
printAlloc()

for i := 0; i < n; i++ {   <-- Добавление 1 миллиона элементов
    m[i] = randBytes()
}
printAlloc()
```

```
for i := 0; i < n; i++ {  ◄—— Удаление 1 миллиона элементов
    delete(m, i)
}

runtime.GC()  ◄—— Ручной запуск сборки мусора (GC)
printAlloc()
runtime.KeepAlive(m)  ◄—— Сохранение ссылки на m, чтобы GC не «чистил» карту
```

Создаем пустую карту, добавляем в нее 1 миллион элементов, затем удаляем 1 миллион элементов, а затем запускаем сборщик мусора. Обязательно сохраняем ссылку на карту с помощью `runtime.KeepAlive`, чтобы сборщик мусора не очищал карту. Запустим код:

```
0 MB    ◄—— После резервирования в памяти места под m
461 MB  ◄—— После добавления 1 миллиона элементов
293 MB  ◄—— После удаления 1 миллиона элементов
```

Что мы видим? Вначале размер кучи минимален. Затем, после добавления в карту 1 миллиона элементов, он значительно увеличивается. Мы ожидали, что после удаления всех элементов размер кучи сильно уменьшится, однако карты в Go работают иначе. Несмотря на то что сборщик мусора удалил все элементы, размер кучи по-прежнему значителен и составляет 293 Мбайт. Память освободилась, но не в той мере, как мы могли бы ожидать. Почему?

В предыдущем разделе мы говорили, что карта состоит из восьмиэлементных сегментов. На самом деле карта в Go — это указатель на структуру `runtime.hmap`. Эта структура содержит несколько полей, в том числе поле `B`, задающее количество сегментов в карте:

```
type hmap struct {
    B uint8 // log_2 of # of buckets
        // (can hold up to loadFactor * 2^B items)
    // ...
}
```

После добавления 1 миллиона элементов значение `B` равно 18, что означает 2^{18} = 262 144 сегмента. Каково будет значение `B`, когда мы удалим 1 миллион элементов? Все так же 18. Следовательно, карта по-прежнему содержит такое же количество сегментов.

Причина в том, что количество сегментов в карте не может сокращаться. Поэтому удаление элементов из карты не влияет на количество существующих сегментов, оно просто обнуляет слоты в сегментах. Карта может только расти и иметь больше сегментов, но она никогда не уменьшается.

В предыдущем примере мы перешли от 461 Мбайт к 293 Мбайт, потому что элементы были удалены, но запуск сборщика мусора не повлиял на саму карту. Даже количество дополнительных сегментов (сегментов, созданных вследствие переполнений) остается прежним.

Сделаем шаг назад и обсудим ситуацию, когда то, что карта не может уменьшаться, рискует стать проблемой. Представьте себе создание кэша с помощью `map[int][128]byte`. Эта карта содержит для каждого ID клиента (`int`) последовательность из 128 байтов. Теперь предположим, что мы хотим сохранить последние 1000 клиентов. Размер карты останется постоянным, поэтому не стоит беспокоиться, что карта не может сжаться.

Допустим, что мы хотим сохранять данные за один час. Но компания решила к Черной пятнице провести большую промоакцию: тогда за один час к нашей системе могут подключиться миллионы клиентов. Через несколько дней после этой Черной пятницы в карте будет столько же сегментов, сколько их было в пиковое время. Это объясняет, почему мы можем столкнуться с высокой загрузкой памяти, которая при таком сценарии существенно не уменьшается.

Какие могут быть решения этой проблемы, если мы не хотим вручную перезапускать сервис для очистки того объема памяти, который потребляется картой? Одним из решений может быть регулярное повторное создание копии текущей карты. Например, каждый час мы можем создавать новую карту, копировать все элементы и освобождать предыдущую. Главный недостаток такого варианта в том, что после копирования и до следующей сборки мусора за небольшой промежуток времени может требоваться в два раза больший объем текущей памяти.

Другим решением было бы изменить тип карты для хранения указателя массива: `map[int]*[128]byte`. Это не отменяет того факта, что у нас будет значительное количество сегментов. Однако каждый сегмент будет резервировать память только в соответствии с размером указателя, а не 128 байтов (то есть 8 байтов в 64-разрядных и 4 байта в 32-разрядных системах).

Возвращаясь к исходному сценарию, сравним потребление памяти для каждого типа карты после каждого шага. Такое сравнение показано в таблице:

Шаг	`map[int][128]byte`	`map[int]*[128]byte`
Создание пустой карты	0 Мбайт	0 Мбайт
Добавление 1 миллиона элементов	461 Мбайт	182 Мбайт
Удаление всех элементов и исполнение GC	293 Мбайт	38 Мбайт

Как мы видим, после удаления всех элементов, с типом `map[int]*[128]byte` требуется значительно меньший объем памяти. Кроме того, объем требуемой памяти менее значителен в пиковое время из-за оптимизаций, производимых для уменьшения потребления памяти.

> **ПРИМЕЧАНИЕ** Если ключ или значение превышает 128 байт, Go не будет хранить их непосредственно в сегменте карты. Вместо этого хранится указатель — для ссылки на ключ или на значение.

Как мы видели, добавление *n* элементов в карту и последующее удаление из нее всех элементов означает то, что в памяти сохраняется такое же количество сегментов. Нужно помнить, что поскольку карта Go может только увеличиваться в размере, то и потребление памяти только увеличивается. Каких-либо автоматизированных процедур для его уменьшения нет. Если в результате мы сталкиваемся со слишком высоким потреблением ресурсов памяти, то можно пробовать разные варианты его уменьшения: например, сделать так, чтобы Go пересоздал карту, или использовать указатели, чтобы проверить возможность оптимизации.

В последнем разделе этой главы обсудим сравнение значений в Go.

3.13. ОШИБКА #29: НЕКОРРЕКТНОЕ СРАВНЕНИЕ ЗНАЧЕНИЙ

Сравнение значений — обычная операция в программировании. Мы часто применяем сравнения: пишем функцию для сравнения двух объектов, сравниваем значения с какой-то ожидаемой величиной и т. д. Интуитивное желание — использовать оператор `==` везде. Но как будет показано дальше, это не всегда верно. Когда уместно использовать `==` и каковы альтернативы?

Начнем с примера. Мы создаем базовую структуру клиента и используем `==` для сравнения двух ее экземпляров. Что выведет этот код?

```
type customer struct {
    id string
}
func main() {
    cust1 := customer{id: "x"}
    cust2 := customer{id: "x"}
    fmt.Println(cust1 == cust2)
}
```

Сравнение этих двух структур `customer` в Go допустимая операция, и оно выведет значение `true`. А что, если мы немного изменим структуру `customer` и добавим поле среза?

```
type customer struct {
    id string
    operations []float64
}
func main() {
    cust1 := customer{id: "x", operations: []float64{1.}}
    cust2 := customer{id: "x", operations: []float64{1.}}
    fmt.Println(cust1 == cust2)
}
```

Можно ожидать, что этот код выдаст `true`. Но он даже не компилируется:

```
invalid operation:

    cust1 == cust2 (struct containing []float64 cannot be compared)
```

Проблема связана с тем, как работают операторы `==` и `!=`. Со срезами или картами они не работают вообще. И поскольку структура `customer` содержит срез, она не компилируется.

Важно понимать, как использовать `==` и `!=` для того, чтобы проводить сравнение корректно. Мы можем использовать эти операторы для сравнения сопоставимых операндов:

- *булевых значений*: равны ли два логических значения;
- *чисел* (`int`, `float` *и типы комплексных чисел*): равны ли два числовых значения;
- *строк*: равны ли две строки;
- *каналов*: были ли два канала созданы одним и тем же вызовом функции `make` либо равны ли оба канала `nil`;
- *интерфейсов*: имеют ли два интерфейса одинаковые динамические типы и одинаковые динамические значения либо равны ли оба интерфейса `nil`;
- *указателей*: указывают ли два указателя на одно и то же значение в памяти либо равны ли оба указателя `nil`;
- *структур и массивов*: состоят ли они из похожих типов.

ПРИМЕЧАНИЕ Можно использовать операторы `?`, `>=`, `<` и `>` как с числовыми типами для сравнения значений, так и со строками для сравнения их лексического порядка.

В последнем примере код не скомпилировался, так как структура была составлена на основе типа, не подлежащего сравнению (среза).

Нужно знать и о возможных проблемах использования == и != с типами any. Например, разрешено сравнение двух целых чисел, присвоенных типам any:

```
var a any = 3
var b any = 3
fmt.Println(a == b)
```

В результате этот код выведет true.

Но что, если мы инициализируем два типа customer (в последней версии, содержащей поле среза) и присваиваем значения типам any? Вот пример:

```
var cust1 any = customer{id: "x", operations: []float64{1.}}
var cust2 any = customer{id: "x", operations: []float64{1.}}
fmt.Println(cust1 == cust2)
```

Этот код компилируется. Но поскольку оба типа нельзя сравнивать, так как структура customer содержит поле среза, выполнение кода приводит к ошибке:

```
panic: runtime error: comparing uncomparable type main.customer
```

Имея в виду такое поведение, как можно сравнить два среза, две карты или две структуры, содержащие не подлежащие сравнению типы? Если мы придерживаемся стандартной библиотеки, один из вариантов — использовать отражение во время выполнения с пакетом reflect.

Отражение — это форма метапрограммирования, которая относится к способности приложения анализировать и изменять свою структуру и поведение. Например, в Go можно использовать Reflect.DeepEqual. Эта функция сообщает, являются ли два элемента *глубоко равными* (*deeply equal*), рекурсивно обходя два значения. Элементы, которые могут быть ее аргументами, являются базовыми типами, а также массивами, структурами, срезами, картами, указателями, интерфейсами и функциями.

> **ПРИМЕЧАНИЕ** Reflect.DeepEqual ведет себя определенным образом в зависимости от типа, который мы задаем. Прежде чем использовать эту функцию, внимательно прочитайте документацию.

Запустим код из первого примера еще раз, добавив `Reflect.DeepEqual`:

```
cust1 := customer{id: "x", operations: []float64{1.}}
cust2 := customer{id: "x", operations: []float64{1.}}
fmt.Println(reflect.DeepEqual(cust1, cust2))
```

Несмотря на то что структура `customer` содержит не подлежащие сравнению типы (срез), она работает, как и ожидалось, выдавая значение `true`.

При использовании `Reflect.DeepEqual` важно помнить о двух вещах. Во-первых, эта функция делает различие между пустой и нулевой коллекцией, как обсуждалось в рассмотрении ошибки #22 (путать пустые и нулевые срезы). Является ли это проблемой? Не обязательно — всё зависит от конкретного случая. Например, если нужно сравнить результаты двух операций демаршалинга (например, из JSON в структуру Go), то может потребоваться подчеркнуть это различие. Стоит помнить о таком поведении, чтобы эффективно использовать `Reflect.DeepEqual`.

Другая загвоздка довольно стандартна для большинства языков. Поскольку эта функция использует отражение, которое интроспективно исследует значения во время выполнения, чтобы узнать, как они формируются, у нее есть проблемы с производительностью. Выполнение нескольких локальных бенчмарков со структурами разного размера показывает, что в среднем `Reflect.DeepEqual` примерно в 100 раз медленнее, чем `==`. Это может быть причиной, по которой лучше использовать ее в контексте тестирования, а не во время выполнения.

Если производительность — решающий фактор, другим вариантом может быть реализация собственного метода сравнения. Вот пример, который сравнивает две структуры `customer` и возвращает результат логического типа:

```
func (a customer) equal(b customer) bool {
    if a.id != b.id {
        return false
    }
    if len(a.operations) != len(b.operations) {
        return false
    }
    for i := 0; i < len(a.operations); i++ {
        if a.operations[i] != b.operations[i] {
            return false
        }
    }
    return true
}
```

Проводится сравнение полей id

Проверяется длина обоих срезов

Проводится сравнение каждого элемента обоих срезов

Здесь мы создаем собственный метод сравнения со своими способами проверки различных полей структуры `customer`. Запуск локального бенчмарка на срезе, состоящем из 100 элементов, показывает, что этот метод `equal` примерно в 96 раз быстрее, чем `Reflect.DeepEqual`.

Нужно помнить, что применение оператора `==` довольно ограниченно. Например, он не работает со срезами и картами. В большинстве случаев задача сравнения решается использованием `Reflect.DeepEqual`, но основным недостатком становится снижение производительности. В контексте юнит-тестов возможны другие варианты: использование внешних библиотек с `go-cmp` (https://github.com/google/go-cmp) или `testify` (https://github.com/stretchr/testify).

Но если при выполнении кода важна производительность, то использование собственного метода может оказаться лучшим решением.

Важно помнить, что в стандартной библиотеке уже есть некоторые методы сравнения. Например, можно использовать оптимизированную функцию `bytes.Compare` для сравнения двух срезов байтов. Перед использованием собственного метода убедитесь, что не занимаетесь велосипедостроением.

ИТОГИ

- При чтении существующего кода имейте в виду, что целочисленные литералы, начинающиеся с 0, являются восьмеричными числами. Для удобочитаемости делайте восьмеричные целые числа явными, добавляя к ним префикс `0o`.
- Поскольку целочисленные переполнения и антипереполнения в Go обрабатываются автоматически, можно реализовать собственные функции для их обнаружения.
- Выполнение сравнений чисел с плавающей точкой по принципу попадания их разницы в пределы заданной дельты может обеспечивать переносимость кода.
- При выполнении операций сложения или вычитания для повышения точности группируйте операции с числами, переменными, значениями одинаковых порядков величины. Сначала делайте умножение и деление, а потом сложение и вычитание.
- Понимание разницы между длиной и емкостью среза очень важно в разработке на Go. Длина среза — это количество доступных в нем элементов, а емкость — количество элементов в резервном массиве.
- При создании среза инициализируйте его с заданной длиной или емкостью, если его длина заранее известна. Это уменьшает количество операций по ре-

зервированию места в памяти и повышает производительность. Та же логика применима и к картам: при инициализации задайте их размер.

- Использование копирования или полного выражения среза — это способ предотвратить возникновение конфликтов при использовании функции `append`, если две разные функции используют срезы с одним и тем же резервным массивом. Но только создание копии среза предотвращает утечку памяти, если вы хотите уменьшить срез большого размера.
- Если вы хотите скопировать один срез в другой с помощью встроенной функции `copy`, помните, что количество копируемых элементов соответствует минимуму из длин двух этих срезов.
- Работая со срезом указателей или со структурами с полями указателей, можно избежать утечек памяти, сделав исключенные операцией нарезки элементы равными `nil`.
- Чтобы избежать часто возникающей путаницы при использовании пакетов `encoding/json` или `reflect`, нужно понимать разницу между нулевыми и пустыми срезами. Они оба являются срезами нулевых длины и емкости, но только нулевой срез не требует для себя выделения места в памяти.
- Чтобы убедиться, что срез вообще не содержит элементов, проверьте его длину. Эта проверка работает независимо от того, нулевой срез или пустой. То же самое касается и карт.
- Для разработки однозначных API не следует проводить различие между нулевыми и пустыми срезами.
- Карта в памяти всегда может увеличиваться в размере, но никогда не уменьшается. И если это приводит к проблемам с памятью, попробуйте разные варианты действий: например, принудительно пересоздавать карты с помощью внутренних средств Go или использовать указатели.
- Для сравнения типов в Go используйте операторы `==` и `!=`, если два типа можно сравнивать в принципе: логические значения, числа, строки, указатели, каналы и структуры, полностью состоящие из сопоставимых друг с другом типов. В противном случае можно использовать `Reflect.DeepEqual` и заплатить цену за отражение либо использовать пользовательские реализации и библиотеки.

4 Управляющие структуры

В этой главе:

- ✓ Как цикл с ключевым словом `range` присваивает значения элементам и оценивает заданное выражение
- ✓ Работа с циклами и указателями с `range`
- ✓ Предотвращение типичных ошибок итераций карт и нарушения выполнения цикла
- ✓ Использование функции `defer` внутри цикла

Управляющие структуры в Go одновременно и похожи на аналогичные в C или Java, и отличаются от них. Например, в Go нет циклов `do` или `while`, а есть только обобщенный цикл `for`. Рассмотрим типичные ошибки, связанные с управляющими структурами, и особое внимание уделим ключевому слову `range`, которое часто понимают неверно.

4.1. ОШИБКА #30: ИГНОРИРОВАТЬ ТО, ЧТО ЭЛЕМЕНТЫ В ЦИКЛЕ RANGE КОПИРУЮТСЯ

`range` — это удобный способ итераций по различным структурам данных. Не нужно иметь дело с индексами и проверять состояние завершенности цикла.

Но Go-разработчики могут забыть или не знать, как `range` присваивает значения, что приводит к распространенным ошибкам. Об этом и поговорим далее.

4.1.1. Концепция

Цикл `range` позволяет проводить итерации по различным структурам данных:

- строкам;
- массивам;
- указателям на массивы;
- срезам;
- картам;
- принимающим каналам.

По сравнению с классическим циклом `for` цикл с `range` — это удобный способ перебора всех элементов одной из этих структур данных благодаря лаконичному синтаксису. Он также в меньшей степени подвержен ошибкам, поскольку не требует обрабатывать условия и переменную цикла вручную, что позволяет избежать ошибки на единицу (off-by-one error). Вот пример с итерацией по срезу строк:

```
s := []string{"a", "b", "c"}
for i, v := range s {
    fmt.Printf("index=%d, value=%s\n", i, v)
}
```

Этот код перебирает каждый элемент среза. На каждой итерации, когда мы перебираем срез, `range` создает пару значений: индекс и значение элемента, присваиваемые в `i` и `v` соответственно. Как правило, `range` создает два значения для каждой структуры данных, кроме принимающего канала, для которого создает только один элемент (значение).

В некоторых случаях нужно только значение элемента, а не его индекс. Так как неиспользование локальной переменной приводит к ошибке компиляции, можно использовать пустой идентификатор для замены индексной переменной:

```
s := []string{"a", "b", "c"}
for _, v := range s {
    fmt.Printf("value=%s\n", v)
}
```

Благодаря пустому идентификатору мы перебираем каждый элемент, игнорируя его индекс и присваивая в `v` только значение элемента.

Если же значение нас не интересует, второй элемент можно опустить:

```
for i := range s {}
```

Теперь, когда мы освежили в памяти особенности использования `range`, посмотрим, какие значения возвращаются во время итераций.

4.1.2. Копия значения

Чтобы эффективно использовать `range`, важно понимать, как во время каждой итерации обрабатывается значение. Рассмотрим пример.

Создадим структуру `account`, содержащую единственное поле `balance`:

```
type account struct {
    balance float32
}
```

Затем создадим срез структуры `account` и переберем каждый элемент, используя цикл `range`. Во время каждой итерации мы увеличиваем `balance` для каждой структуры `account`:

```
accounts := []account{
    {balance: 100.},
    {balance: 200.},
    {balance: 300.},
}
for _, a := range accounts {
    a.balance += 1000
}
```

Как вы думаете, какой из этих двух вариантов показывает содержимое среза в соответствии с приведенным кодом?

- `[{100} {200} {300}]`
- `[{1100} {1200} {1300}]`

Правильный ответ — `[{100} {200} {300}]`. В этом примере `range` не влияет на содержимое среза.

В Go все, что мы присваиваем, является копией:

- Если мы присваиваем результат выполнения функции, возвращающей *структуру*, Go создает копию этой структуры.
- Если мы присваиваем результат выполнения функции, возвращающей *указатель*, Go создает копию адреса памяти (в 64-битной архитектуре адрес имеет длину 64 бита).

Об этом важно помнить, чтобы избежать типичных ошибок, в том числе связанных с циклами `range`. Когда `range` совершает итерацию по структуре данных, выполняется копирование каждого элемента в переменную-значение (второй элемент).

Вернемся к нашему примеру: перебор каждого элемента `account` приводит к тому, что копия структуры присваивается переменной значения `a`. Следовательно, увеличение `balance` с помощью `a.balance += 1000` изменяет только переменную значения (`a`), а не элемент в срезе.

Что будет, если нужно обновить элементы среза? Есть два варианта получить доступ к элементу с помощью индекса среза. Этого можно добиться либо с помощью классического цикла `for`, либо с помощью цикла `range`, используя индекс вместо переменной значения:

```
for i := range accounts {
    accounts[i].balance += 1000
}
for i := 0; i < len(accounts); i++ {
    accounts[i].balance += 1000
}
```

Используется переменная индекса для доступа к элементу среза

Используется традиционный цикл for

Оба этих варианта приводят к одинаковому эффекту: к обновлению элементов в срезе `accounts`.

Какой из них предпочтительнее? Зависит от контекста. Если нужно просмотреть каждый элемент, первый цикл будет короче для записи и чтения. Если же нужно проконтролировать, какой конкретно элемент мы хотим обновить (например, один из двух), то следует использовать второй цикл.

Помните, что элемент значения в цикле `range` является копией. Поэтому если значение представляет собой структуру, которую нужно изменить, мы будем обновлять только копию, а не сам элемент (при условии, что модифицируемое

значение или поле не являются указателем). Предпочтительным вариантом является доступ к элементу через индекс с использованием цикла `range` или классического цикла `for`.

Обновление элементов среза: третий вариант

Другой вариант: продолжить использовать цикл `range` и получить доступ к значению, но изменить тип среза на срез указателей `account`:

```
accounts := []*account{   <-- Обновление типа среза до []*account
    {balance: 100.},
    {balance: 200.},
    {balance: 300.},
}
for _, a := range accounts {
    a.balance += 1000   <-- Прямое обновление элементов среза
}
```

Как мы уже говорили, переменная `a` является копией указателя `account`, хранящегося в срезе. Но поскольку оба указателя ссылаются на одну и ту же структуру, оператор `a.balance += 1000` обновляет элемент среза.

У этого варианта есть два недостатка. Во-первых, требуется обновить тип среза, что не всегда возможно. Во-вторых, если важна производительность, то итерация по срезу указателей может быть менее эффективной для центрального процессора из-за отсутствия предсказуемости (обсудим этот момент в ошибке #91 (не понимать устройство кэша CPU)).

В следующем разделе мы продолжим работу с циклами `range` и посмотрим, как оценивается задаваемое выражение.

4.2. ОШИБКА #31: ИГНОРИРОВАТЬ ТО, КАК В ЦИКЛАХ RANGE ВЫЧИСЛЯЮТСЯ АРГУМЕНТЫ

Синтаксис цикла `range` требует наличия выражения. Например, в цикле `for i, v := range exp, exp` — это выражение. Как мы видели, это может быть строка, массив, указатель на массив, срез, карта или канал. Теперь поговорим о том, как вычисляется это выражение. Это важный момент, позволяющий избежать многих типичных ошибок.

Рассмотрим пример, где к срезу добавляется элемент, по которому мы выполняем итерацию. Как вы считаете, завершится ли этот цикл?

```
s := []int{0, 1, 2}
for range s {
    s = append(s, 10)
}
```

Чтобы понять суть, следует помнить, что при использовании цикла `range` указываемое выражение вычисляется только один раз — перед началом цикла. В этом контексте слово «вычисляется» означает, что предоставленное выражение копируется во временную переменную, а затем цикл `range` выполняет итерации над этой переменной. В этом примере при вычислении выражения `s` результатом будет копия среза, как показано на рис. 4.1.

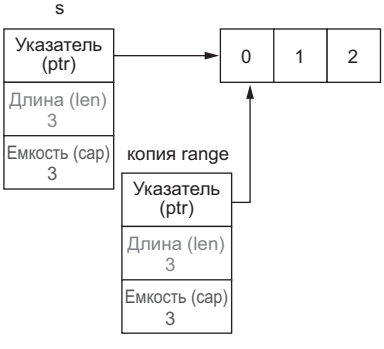


Рис. 4.1. `s` копируется во временную переменную, используемую в цикле `range`

Цикл `range` использует эту временную переменную. Исходный срез `s` также обновляется во время каждой итерации. Следовательно, после трех итераций состояние будет таким, как на рис. 4.2.

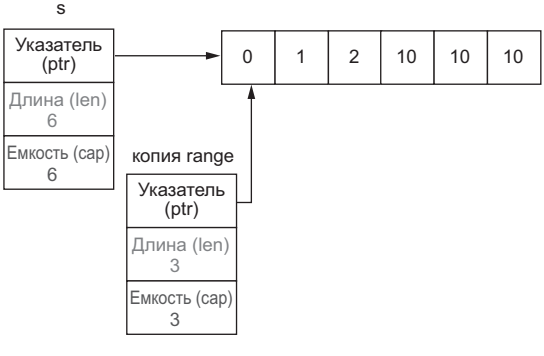


Рис. 4.2. Временная переменная остается срезом длиной 3, поэтому итерации прекращаются

Каждый шаг приводит к добавлению нового элемента. Но за три шага мы прошлись по всем его элементам. Длина временного среза, используемого в `range`, остается равна 3, поэтому цикл завершается после трех итераций.

Такое поведение отличается от классического цикла `for`:

```
s := []int{0, 1, 2}
for i := 0; i < len(s); i++ {
    s = append(s, 10)
}
```

В этом примере цикл никогда не закончится. Значение выражения `len(s)` вычисляется во время каждой итерации, и раз мы продолжаем добавлять элементы, то никогда не достигнем состояния завершения цикла. Чтобы правильно использовать циклы в Go, важно помнить об этой разнице.

При использовании `range` помните, что вышеописанное поведение (выражение вычисляется только один раз) также применимо ко всем типам данных. В качестве примера посмотрим на последствия такого поведения для двух других типов: каналов и массивов.

4.2.1. Каналы

Рассмотрим пример, где цикл `range` осуществляет итерации по каналу. Мы создаем две горутины, каждая из которых отправляет элементы в два канала. Затем в родительской горутине реализуем потребителя на одном канале, используя цикл `range`, который пытается переключиться на другой канал во время выполнения цикла:

```
ch1 := make(chan int, 3)    <-- Создается первый канал, содержащий элементы 0, 1, 2
go func() {
    ch1 <- 0
    ch1 <- 1
    ch1 <- 2
    close(ch1)
}()

ch2 := make(chan int, 3)    <-- Создается второй канал, содержащий элементы 10, 11, 12
go func() {
    ch2 <- 10
    ch2 <- 11
    ch2 <- 12
    close(ch2)
}()

ch := ch1    <-- ch присваивается значение первого канала
```

```
for v := range ch {  ◄── Создается клиент канала путем итераций по ch
    fmt.Println(v)
    ch = ch2  ◄── ch присваивается значение второго канала
}
```

Та же логика применяется в отношении того, как вычисляется выражение `range`. Выражение, задаваемое для `range`, представляет собой канал `ch`, указывающий на `ch1`. Следовательно, `range` вычисляет `ch`, выполняет его копирование во временную переменную и итерирует по элементам из этого канала. Несмотря на то что `ch = ch2`, цикл `range` продолжается по `ch1`, а не `ch2`:

```
0
1
2
```

Но оператор `ch = ch2` все-таки оказывает некоторое влияние. Поскольку мы присвоили второй переменной значение `ch`, то если после этого кода вызовем `close(ch)`, закроется второй канал, а не первый.

Теперь посмотрим, как оператор `range` вычисляет каждое выражение только один раз и влияет на результат при использовании его с массивом.

4.2.2. Массив

Как влияет использование `range` с массивом? Поскольку выражение, по которому проводятся итерации в цикле `range`, вычисляется до начала цикла, то, что присваивается временной переменной цикла, является копией массива. Посмотрим на этот принцип на примере, где некоторый индекс массива обновляется во время выполнения цикла:

```
a := [3]int{0, 1, 2}  ◄── Создается массив из трех элементов
for i, v := range a {  ◄── Проводятся итерации по массиву
    a[2] = 10  ◄── Обновляется последний индекс
    if i == 2 {  ◄── На печать выводится содержимое последнего индекса
       fmt.Println(v)
    }
}
```

Этот код меняет значение последнего индекса на 10. Но в результате выполнения кода будет выведено не `10`, а `2`, как показано на рис. 4.3.

Как мы уже говорили, оператор `range` создает копию массива. По мере выполнения цикла обновляется не копия, а исходный массив `a`. Следовательно, значение `v` на последней итерации равно `2`, а не `10`.

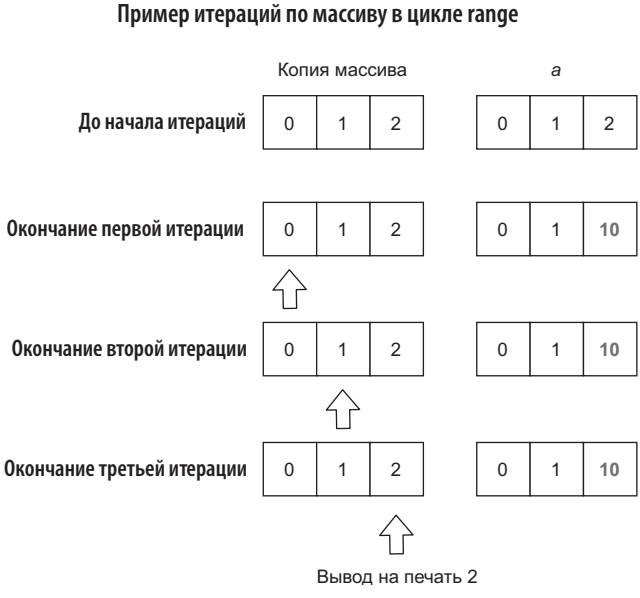


Рис. 4.3. range проводит итерации по копии массива (*слева*), в то время как цикл модифицирует a (*справа*)

Если нужно вывести фактическое значение последнего элемента, то сделать это можно двумя способами:

- Получая доступ к элементу по его индексу:

```
a := [3]int{0, 1, 2}
for i := range a {
    a[2] = 10
    if i == 2 {
       fmt.Println(a[2])
    }
}
```

Обращение к a[2], а не к переменной, по которой осуществляется цикл range

Поскольку мы обращаемся к исходному массиву, этот код выводит 2 вместо 10.

- Используя указатель массива:

```
a := [3]int{0, 1, 2}
for i, v := range &a {
    a[2] = 10
    if i == 2 {
       fmt.Println(v)
    }
}
```

Цикл range осуществляется по &a, а не по a

Мы присваиваем копию указателя массива временной переменной, используемой оператором `range`. Но поскольку оба указателя ссылаются на один и тот же массив, обращение к `v` также возвращает `10`.

Оба варианта допустимы и приводят к правильному результату. Но во втором варианте не производится копирование всего массива, о чем следует помнить, если массив имеет значительный размер.

Таким образом, в цикле `range` выражение, по которому проводятся итерации, вычисляется только один раз — перед началом цикла, при этом выполняется копирование этого выражения (независимо от типа). Помните о таком поведении, чтобы избежать ошибок, которые могут привести к тому, что вы обратитесь к неправильному элементу.

В следующем разделе посмотрим, как избежать ошибок при использовании `range` с указателями.

4.3. ОШИБКА #32: ИГНОРИРОВАТЬ ВЛИЯНИЕ, КОТОРОЕ ОКАЗЫВАЕТ ИСПОЛЬЗОВАНИЕ ЭЛЕМЕНТОВ УКАЗАТЕЛЯ В ЦИКЛАХ RANGE

Рассмотрим специфическую ошибку при использовании цикла `range` с элементами указателя: ссылки не на те элементы, которые на самом деле нужны.

Прежде всего проясним причину использования среза или карты элементов-указателей. Есть три основных сценария:

- Хранение данных с применением семантики указателей подразумевает совместное использование элемента. Например, следующий метод содержит логику для вставки элемента в кэш:

```
type Store struct {
    m map[string]*Foo
}
func (s Store) Put(id string, foo *Foo) {
    s.m[id] = foo
    // ...
}
```

Здесь применение семантики указателя подразумевает, что элемент `Foo` общий как для вызывающего объекта `Put`, так и для структуры `Store`.

- Бывают случаи, когда мы уже производим какие-то действия с указателями. Поэтому вместо значений может быть удобно сохранять непосредственно указатели.
- Если мы держим в памяти структуры больших размеров и они часто изменяются, то можно использовать указатели, чтобы избежать операции копирования и вставки при каждом таком изменении:

```
func updateMapValue(mapValue map[string]LargeStruct, id string) {
    value := mapValue[id]  <-- Копирование
    value.foo = "bar"
    mapValue[id] = value  <-- Вставка
}
func updateMapPointer(mapPointer map[string]*LargeStruct, id string) {
    mapPointer[id].foo = "bar"  <-- Прямое изменение элемента карты
}
```

Поскольку аргумент функции `updateMapPointer` — это карта указателей, изменение поля `foo` можно выполнить за один шаг.

Обсудим распространенную ошибку с элементами-указателями, возникающую при работе с `range`. Рассмотрим две структуры:

- `Customer`, представляющую собой потребитель.
- `Store`, которая содержит карту указателей `Customer`.

```
type Customer struct {
    ID string
    Balance float64
}
    type Store struct {
    m map[string]*Customer
}
```

Следующий код итерирует по срезу элементов `Customer` и сохраняет их в карте `m`:

```
func (s *Store) storeCustomers(customers []Customer) {
    for _, customer := range customers {
        s.m[customer.ID] = &customer  <-- Сохранение указателя customer в карте
    }
}
```

Мы итерируем по входному срезу с помощью оператора `range` и сохраняем указатели `Customer` в карте. Но приводит ли такой метод к тому, что ожидается?

Вызовем его с помощью фрагмента из трех разных структур `Customer`:

```
s.storeCustomers([]Customer{
    {ID: "1", Balance: 10},
    {ID: "2", Balance: -10},
    {ID: "3", Balance: 0},
})
```

Вот результат этого кода, если вывести содержимое карты:

```
key=1, value=&main.Customer{ID:"3", Balance:0}
key=2, value=&main.Customer{ID:"3", Balance:0}
key=3, value=&main.Customer{ID:"3", Balance:0}
```

Вместо сохранения трех разных структур `Customer` все элементы, хранящиеся в карте, ссылаются на одну и ту же структуру `Customer` — третью. Что пошло не так?

Итерация по срезу клиентов с использованием цикла `range` создает независимо от количества элементов одну-единственную переменную `customer` с фиксированным адресом. Проверим это, выводя адрес указателя во время каждой итерации:

```
func (s *Store) storeCustomers(customers []Customer) {
    for _, customer := range customers {
        fmt.Printf("%p\n", &customer)  <-- Вывод адреса customer
        s.m[customer.ID] = &customer
    }
}
```

```
0xc000096020
0xc000096020
0xc000096020
```

Почему это важно? Рассмотрим каждую итерацию:

- При первой итерации `customer` ссылается на первый элемент — `Customer 1`. Мы сохраняем указатель в структуре `customer`.
- При второй итерации `customer` теперь ссылается на другой элемент — `Customer 2`. Также сохраняем указатель в структуре `customer`.
- Наконец, при последней итерации `customer` ссылается на последний элемент — `Customer 3`. И опять тот же указатель сохраняется в карте.

В конце каждой итерации мы трижды сохраняли один и тот же указатель в карте (рис. 4.4). Последняя запись этого указателя в память является ссылкой на

последний элемент среза — `Customer 3`. Поэтому все элементы карты ссылаются на один и тот же потребитель.

Рис. 4.4. Переменная customer имеет постоянный адрес, поэтому мы храним в карте один и тот же указатель

Как решить эту проблему? Есть два способа. Первый похож на тот, что мы видели при разборе ошибки #1 (непреднамеренно затенять переменные). При этом потребуется создание локальной переменной:

```
func (s *Store) storeCustomers(customers []Customer) {
    for _, customer := range customers {
        current := customer
        s.m[current.ID] = &current    ◄── Запись указателя в карту
    }
}
```

В этом примере мы не храним указатель, ссылающийся на `customer`, а сохраняем указатель, ссылающийся на `current` — переменную, ссылающуюся на уникальный `Customer` во время каждой итерации. По мере выполнения цикла мы сохраняли в карте разные указатели, ссылающиеся на разные структуры `Customer`. Другое решение состоит в том, чтобы сохранять указатель, ссылающийся на каждый элемент, используя индекс среза:

```
func (s *Store) storeCustomers(customers []Customer) {
    for i := range customers {
        customer := &customers[i]  ◄── Присвоение customer значения указателя на элемент i
        s.m[customer.ID] = customer  ◄── Сохранение указателя customer
    }
}
```

В этом решении `customer` теперь указатель. Поскольку он инициализируется во время каждой итерации, то имеет уникальный адрес. Поэтому мы храним в картах разные указатели.

При итерациях по структурам данных с использованием `range` помните, что все значения присваиваются уникальной переменной с одним уникальным

адресом. И если во время каждой итерации хранить указатель, ссылающийся на эту переменную, то мы окажемся в ситуации, когда будет сохраняться один и тот же указатель, ссылающийся на один и тот же элемент — самый последний. Эту проблему можно решить, принудительно создав локальную переменную в области действия цикла или указатель, ссылающийся на элемент среза через его индекс. Оба решения хороши. Хотя мы взяли в качестве входных данных структуру среза, с картами может возникнуть аналогичная проблема.

В следующем разделе рассмотрим ошибки, связанные с итерациями по картам.

4.4. ОШИБКА #33: ДЕЛАТЬ НЕВЕРНЫЕ ДОПУЩЕНИЯ ВО ВРЕМЯ ИТЕРАЦИЙ КАРТЫ

Итерации карт вызывают много непонимания и ошибок, в основном из-за того, что разработчики делают неверные допущения. В этом разделе обсудим два разных случая:

- упорядочивание;
- обновление карты во время итераций.

При итерациях карты возможны две ошибки, основанные на неверных допущениях.

4.4.1. Упорядочивание

Важно понять несколько фундаментальных свойств структуры данных `map`:

- Она не хранит данные, отсортированные по ключу (карта не основана на двоичном дереве).
- Она не сохраняет порядок, в котором были добавлены данные. Например, если мы вставляем пару A перед парой B, то не должны делать никаких предположений, основанных на таком порядке вставки.

Более того, при итерациях карты мы вообще не должны делать никаких допущений относительно упорядочивания. Посмотрим, что вытекает из этого утверждения.

Рассмотрим карту на рис. 4.5, состоящую из четырех сегментов (элементы представляют ключ). Каждый индекс резервного массива ссылается на какой-то сегмент.

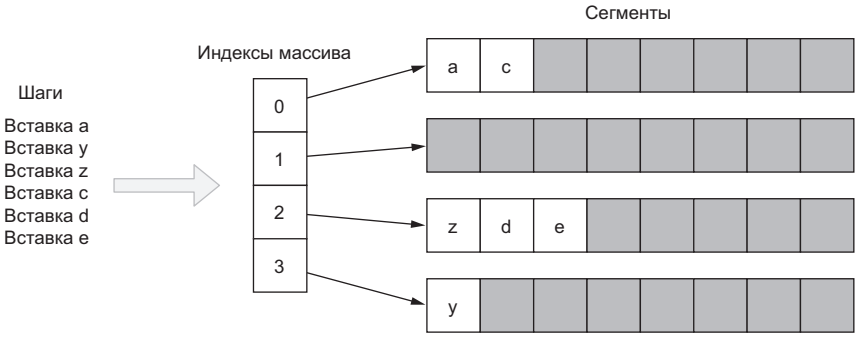


Рис. 4.5. Карта с четырьмя сегментами

С помощью цикла `range` переберем элементы этой карты и выведем все ключи:

```
for k := range m {
    fmt.Print(k)
}
```

Мы уже говорили, что сортировка данных по ключу не делается. Следовательно, мы не можем ожидать, что в результате выполнения этого кода будет выведено `acdeyz`. Кроме того, мы также уже сказали, что в картах не сохраняется порядок вставки. Следовательно, мы также не можем ожидать, что будет выведено `ayzcde`.

Можно ли ожидать, что код выведет ключи в том порядке, в каком они сейчас хранятся в карте, то есть `aczdey`? Нет. В Go порядок итераций по карте *не определяется и не указывается*. Нет никакой гарантии, что порядок будет одинаковым от одной итерации к другой. Следует помнить о таком поведении карты, чтобы не основывать код на неверных допущениях.

Справедливость всех этих утверждений подтверждается разными запусками кода:

```
zdyaec
czyade
```

Вывод отличается от одной итерации к другой.

> **ПРИМЕЧАНИЕ** Хотя нет никакой гарантии относительно порядка итераций, распределение итераций неравномерно. Именно поэтому в официальной спецификации Go говорится, что итерация неопределенная, а не случайная.

Почему в Go такой причудливый способ итераций карты? Это сознательный выбор разработчиков языка. Они хотели добавить элемент случайности, чтобы программисты никогда не основывались на каких-либо предположениях об упорядочивании при работе с картами (см. http://mng.bz/M2JW).

Go-разработчикам никогда не следует делать какие-либо предположения относительно порядка при итерациях карт. Отметим, что использование пакетов из стандартной или внешней библиотеки может приводить к разному поведению. Например, когда пакет `encoding/json` маршалирует карту в JSON, он выстраивает данные в алфавитном порядке по ключам, независимо от порядка вставки. Но это не свойство самой карты в Go. Если нужно какое-либо упорядочивание, рекомендую использовать другие структуры данных, например двоичную кучу (библиотека GoDS на https://github.com/emirpasic/gods содержит полезные реализации структурирования данных).

Рассмотрим вторую ошибку, связанную с обновлением карты в процессе итераций по ней.

4.4.2. Вставка карты во время итераций

В Go разрешено обновление карты (вставка или удаление элемента) во время итераций — это не приводит к ошибкам компиляции или выполнения. Но есть нюанс, который следует учитывать при добавлении элемента в карту во время итерации, чтобы избежать недетерминированных результатов.

В примере ниже проводятся итерации по `map[int]bool`. Если значение пары равно `true`, мы добавляем еще один элемент. Как вы думаете, что выведет этот код?

```
m := map[int]bool{
    0: true,
    1: false,
    2: true,
}
for k, v := range m {
    if v {
        m[10+k] = true
    }
}
fmt.Println(m)
```

Результат непредсказуем. Посмотрите на несколько примеров вывода:

```
map[0:true 1:false 2:true 10:true 12:true 20:true 22:true 30:true]
map[0:true 1:false 2:true 10:true 12:true 20:true 22:true 30:true 32:true]
map[0:true 1:false 2:true 10:true 12:true 20:true]
```

Вот что говорится в спецификации Go по поводу создания нового элемента карты во время итераций:

> *Если запись карты создается во время итерации, она может быть произведена во время итерации или пропущена. Выбор может варьироваться для каждой созданной записи и от одной итерации к другой.*

Когда элемент добавляется к карте во время итерации, он может быть либо создан, либо нет при последующей итерации. В Go нет возможности как-то «навязать» поведение кода. Оно может варьироваться от одной итерации к другой, и поэтому мы трижды получали разные результаты.

Важно помнить о таком поведении, чтобы код не выдавал непредсказуемых результатов. Если нужно обновить карту во время итерации по ней и убедиться, что добавленные записи не часть этой итерации, то одним из решений будет работа с копией карты:

```
m := map[int]bool{
    0: true,
    1: false,
    2: true,
}
m2 := copyMap(m)  <-- Создается копия первоначальной карты

for k, v := range m {
    m2[k] = v
    if v {
        m2[10+k] = true  <-- Обновляется m2 вместо m
    }
}
fmt.Println(m2)
```

В этом примере мы отделяем читаемую карту от обновляемой. Мы продолжаем итерировать по `m`, но все обновления делаются на `m2`. Эта новая версия кода ведет к предсказуемому и повторяемому результату:

```
map[0:true 1:false 2:true 10:true 12:true]
```

При работе с картой не полагайтесь на следующее:

- на то, что данные упорядочиваются по ключам;
- на то, что порядок вставки сохранится;
- на детерминированность порядка итераций;

- на то, что элемент будет создан во время той же итерации, во время которой он был добавлен.

Помня об этих особенностях поведения, мы сможем избежать распространенных ошибок, основанных на неверных предположениях. В следующем разделе рассмотрим ошибку, которая довольно часто допускается при прерывании циклов.

4.5. ОШИБКА #34: ИГНОРИРОВАТЬ ОСОБЕННОСТИ РАБОТЫ ОПЕРАТОРА BREAK

Оператор `break` обычно используется для прекращения цикла. Когда циклы используются в сочетании со `switch` или `select`, разработчики часто совершают ошибку, прерывая не тот оператор.

Рассмотрим пример. Мы используем `switch` внутри цикла `for`. Когда индекс цикла получает значение 2, требуется прервать цикл:

```
for i := 0; i < 5; i++ {
    fmt.Printf("%d ", i)

    switch i {
    default:
    case 2:  <-- Если i равен 2, то вызывается оператор break
        break
    }
}
```

На первый взгляд код может казаться правильным, но он не приводит к выполнению ожидаемых действий. Оператор `break` не завершает цикл `for`, он завершает действие оператора `switch`. Следовательно, вместо итерации от 0 до 2 этот код выполняет итерацию от 0 до 4: `0 1 2 3 4`.

Важное правило, о котором следует помнить, заключается в том, что оператор `break` завершает выполнение самого последнего оператора `for`, `switch` или `select`. И здесь он прерывает действие `switch`.

Как написать код, который будет прерывать цикл, а не действие оператора `switch`? Самый идиоматический способ — использовать метку:

```
loop:  <-- Определяется метка loop
    for i := 0; i < 5; i++ {
        fmt.Printf("%d ", i)
        switch i {
```

```
        default:
        case 2:
           break loop  ◄──  Прерывается цикл, привязанный к метке loop, а не к switch
        }
    }
```

Здесь мы связываем `loop` с циклом `for`. Поскольку мы указываем эту метку в операторе `break`, то он прерывает цикл, а не действие оператора `switch`. Новый код выведет `0 1 2`, как и требовалось.

break с меткой — это то же самое, что и goto?

Некоторые разработчики могут поставить под сомнение то, что `break` с меткой идиоматичен, и рассматривать его как причудливый оператор `GOTO`. Но это не так, и такой код используется в стандартной библиотеке. Например, в пакете `net/http` при чтении строк из буфера:

```
readlines:
    for {
        line, err := rw.Body.ReadString('\n')
        switch {
        case err == io.EOF:
           break readlines
        case err != nil:
           t.Fatalf("unexpected error reading from CGI: %v", err)
        }
        // ...
    }
```

В этом примере используется говорящая сама за себя метка `readlines`, чтобы подчеркнуть цель цикла. Поэтому прерывание действия оператора с помощью меток — это идиоматический подход в Go.

Прерывание не того оператора также может произойти и в случае с `select`, находящимся внутри цикла. В примере ниже мы хотим использовать `select` с двумя случаями и выйти из цикла, если контекст отменяется:

```
for {
    select {
    case <-ch:
        // Какие-то действия
     case <-ctx.Done():
        break  ◄──  Цикл прерывается, если контекст отменяется
    }
}
```

Здесь самым внутренним оператором из списка `for`, `switch`, `select` является `select`, а не `for`. Поэтому цикл продолжается. Чтобы прервать сам цикл, используем метку:

```
loop:  ◄──  Определяется метка loop
    for {
        select {
        case <-ch:
            // Какие-то действия
        case <-ctx.Done():
            break loop  ◄──  Прерывается выполнение привязанного к loop цикла, а не select
        }
    }
```

Как и ожидалось, оператор `break` приводит к выходу из цикла, а не к прерыванию выполнения `select`.

> **ПРИМЕЧАНИЕ** Можно использовать `continue` с меткой, чтобы перейти к следующей итерации отмеченного цикла.

При использовании `switch` или `select` внутри цикла нужно быть очень внимательными. При использовании `break` тщательно проверяйте, на какой оператор он повлияет. Использование меток — это идиоматическое решение для принудительного прекращения определенного оператора.

В последнем разделе обсудим особенности циклов в сочетании с ключевым словом `defer`.

4.6. ОШИБКА #35: ИСПОЛЬЗОВАТЬ DEFER ВНУТРИ ЦИКЛОВ

Оператор `defer` откладывает выполнение вызова до возвращения соседней функции. В основном он используется для сокращения шаблонного кода. Например, если ресурс в итоге должен быть закрыт, можно использовать `defer`, чтобы избежать повторения вызовов для закрытия перед каждым возвратом. Распространенная ошибка — незнание последствий использования `defer` внутри цикла.

Например, нужно реализовать функцию, которая будет открывать какую-то группу файлов, причем пути к ним получены из канала. Мы делаем итерации по этому каналу, открываем файлы и обрабатываем закрытие. Вот первая версия:

```
func readFiles(ch <-chan string) error {
    for path := range ch {  <-- Итерации по каналу
        file, err := os.Open(path)  <-- Открывается файл
        if err != nil {
            return err
        }
        defer file.Close()  <-- Откладывается вызов file.Close()
        // Какие-то действия с файлом
    }
    return nil
}
```

ПРИМЕЧАНИЕ Обработка ошибок, связанных с `defer`, рассмотрена в разделе, посвященном ошибке #54 (не выполнять обработку ошибки оператора `defer`).

В такой реализации есть проблема. `Defer` планирует вызов функции, когда возвращается окружающая функция. В данном случае вызов `defer` выполняется не во время каждой итерации цикла, а при возвращении `readFiles`. Если `readFiles` не вернет значение, дескрипторы файлов останутся открытыми навсегда, что приведет к утечкам.

Как решить эту проблему? Можно было бы избавиться от `defer` и обработать закрытие файла вручную. Но в таком случае пришлось бы отказаться от удобной функции набора инструментов Go только потому, что мы находимся внутри какого-то цикла. Какие варианты, если продолжить использовать `defer`? Создать вокруг `defer` другую окружающую функцию, которая будет вызываться во время каждой итерации.

Например, можно реализовать функцию `readFile`, содержащую логику для каждого нового полученного пути к файлу:

```
func readFiles(ch <-chan string) error {
    for path := range ch {
        if err := readFile(path); err != nil {  <-- Вызов readFile, которая содержит основную логику
            return err
        }
    }
    return nil
}
func readFile(path string) error {
    file, err := os.Open(path)
    if err != nil {
        return err
    }
    defer file.Close()  <-- Сохраняет вызов defer
```

```
    // Какие-то действия с файлом
    return nil
}
```

Здесь `defer` вызывается после возврата `readFile`, то есть в конце каждой итерации. Поэтому мы не держим дескрипторы файлов открытыми до возвращения родительской функции `readFiles`.

Другой подход в том, чтобы сделать функцию `readFile` закрывающей:

```
func readFiles(ch <-chan string) error {
    for path := range ch {
        err := func() error {
             // ...
           defer file.Close()
           // ...
        }()  ◄—— Выполнение уже определенной операции закрытия
        if err != nil {
          return err
        }
    }
    return nil
}
```

По сути, это остается тем же самым решением: добавление еще одной окружающей функции для выполнения `defer` во время каждой итерации. Преимущество простой старой функции в том, что она немного понятнее, и мы также можем написать для нее специальный юнит-тест.

При использовании `defer` помните, что она планирует вызов функции, когда возвращается окружающая функция. Вызов `defer` внутри цикла будет складывать все вызовы в стек: они не будут выполняться во время каждой итерации, что может привести к утечке памяти, например, если цикл так и не завершится. Наиболее удобный подход к решению этой проблемы — введение еще одной функции, которая будет вызываться на каждой итерации. Но если важна производительность, то недостатком этого метода будет оверхед на требуемое время на вызовы функции. Если это ваш случай, то следует избавиться от `defer` и обрабатывать вызов `defer` вручную перед выполнением цикла.

ИТОГИ

- Значение элемента, по которому проводится цикл `range`, является копией. Поэтому чтобы изменить структуру, обращайтесь к ней, например, через ее

индекс или используйте классический цикл `for` (если только элемент или поле, которые вы хотите изменить, не являются указателем).

- Понимание того, что выражение, переданное оператору `range`, вычисляется только один раз перед началом цикла, поможет избежать неэффективного присваивания в итерации по каналу или срезу.
- Используя локальную переменную или обращаясь к элементу по индексу, можно предотвратить ошибки при копировании указателей внутри цикла.
- Чтобы обеспечить предсказуемость результатов при использовании карт, помните, что эта структура данных:
  - не упорядочивает данные по ключам;
  - не сохраняет порядок их вставки;
  - не имеет детерминированного порядка итерации;
  - не гарантирует, что элемент, добавленный во время итерации, будет создан во время этой итерации.
- Использование `break` или `continue` с меткой приводит к прерыванию какого-то конкретного оператора. Это может быть полезно при работе с операторами `switch` или `select` внутри циклов.
- Извлечение логики цикла внутри функции приводит к выполнению оператора `defer` в конце каждой итерации.

5 Строки

В этой главе:

- ✓ Понимание фундаментальной концепции рун в Go
- ✓ Предотвращение распространенных ошибок при итерации и обрезке строк
- ✓ Избавление от неэффективного кода, возникающего из-за конкатенации строк или бесполезных преобразований
- ✓ Предотвращение утечек памяти при работе с подстроками

В Go строка — это неизменяемая структура данных, содержащая:

- указатель на неизменяемую последовательность байтов;
- общее количество байтов в этой последовательности.

В языке Go есть уникальный способ работы со строками. Go вводит концепцию рун: она очень важна для понимания и может сбивать с толку новичков. Когда мы разберемся, как в этом языке обрабатываются строки, то сможем избежать распространенных ошибок при итерациях строк. Рассмотрим и типичные ошибки при использовании или создании строк. Кроме того, увидим, что иногда можно

работать напрямую с `[]byte`, избегая дополнительных выделений памяти. Наконец, обсудим, как избежать распространенной ошибки, которая может привести к утечкам из подстрок. Основная цель этой главы — помочь вам понять, как работают строки в Go, разобрав несколько ошибок.

5.1. ОШИБКА #36: НЕ ПОНИМАТЬ КОНЦЕПЦИИ РУН

Для начала обсудим концепцию рун в Go. Эта концепция — ключ к пониманию того, как обрабатываются строки, что позволяет избежать распространенных ошибок. Для начала освежим основные понятия.

Важно понимать разницу между кодировкой символов (charset) и кодированием (encoding):

- Кодировка символов, charset, — это просто набор символов. Например, кодировка Unicode содержит 2^{21} символ.
- Кодирование — это перевод списка символов в двоичный код. Например, UTF-8 — это стандарт кодирования, определяющий способ того, как возможно закодировать все символы Unicode в переменном количестве байтов (от 1 до 4 байт).

Мы упомянули слово «символы», чтобы упростить определение кодировки. Но в Unicode мы используем концепцию *кодовой точки* для ссылки на элемент, представленный одним значением. Например, символ 汉 определяется кодовой точкой U+6C49. Используя UTF-8, 汉 кодируется тремя байтами: 0xE6, 0xB1 и 0x89. Почему это важно? Потому что в Go руна — это кодовая точка Unicode.

Мы сказали, что UTF-8 кодирует символы в количестве байтов от 1 до 4 байт, следовательно, до 32 бит. Вот почему в Go руна — это псевдоним типа `int32`:

```
type rune = int32
```

Еще одна вещь, важная для UTF-8: некоторые считают, что строки Go всегда имеют кодировку UTF-8, но это не так. Рассмотрим пример:

```
s := "hello"
```

Мы присваиваем строковый литерал (строковую константу) переменной `s`. В Go исходный код представлен в UTF-8, то есть все строковые литералы кодируются в последовательность байтов с использованием UTF-8. Но строка представляет

собой последовательность произвольных байтов, и она не обязательно основана на UTF-8. Когда мы работаем с переменной, которая не была инициализирована из строкового литерала (например, при чтении из файловой системы), мы не можем считать по умолчанию, что она использует кодировку UTF-8.

> **ПРИМЕЧАНИЕ** golang.org/x — репозиторий, предоставляющий расширения стандартной библиотеки, — содержит пакеты для работы с UTF-16 и UTF-32.

Вернемся к примеру с приветствием. Есть строка, состоящая из пяти символов: *h, e, l, l* и *o*.

Эти *простые* символы кодируются с использованием одного байта каждый. Вот почему вызов функции запроса длины `s` возвращает 5:

```
s := "hello"
fmt.Println(len(s)) // 5
```

Но символ не всегда кодируется одним байтом. Возвращаясь к символу 汉, мы упомянули, что в UTF-8 он кодируется тремя байтами. Это подтверждается примером:

```
s := "汉"
fmt.Println(len(s)) // 3
```

Вместо `1` в этом примере выводится `3`. Применяемая к строке встроенная функция `len` возвращает не количество символов, а число байтов.

И наоборот, мы можем создать строку, отталкиваясь от списка байтов. Мы уже упоминали, что символ 汉 кодируется тремя байтами: 0xE6, 0xB1 и 0x89:

```
s := string([]byte{0xE6, 0xB1, 0x89})
   fmt.Printf("%s\n", s)
```

Здесь мы создаем строку из этих трех байтов. Когда мы выводим ее, то получаем не три символа, а один: 汉.

Выводы:

- Кодировка символов — это набор символов. Кодирование же описывает, как кодировка преобразовывается в двоичный код.
- В Go строка ссылается на неизменяемый срез произвольных байтов.

- Исходный код Go использует UTF-8. Все строковые литералы — строки UTF-8. Но поскольку строка может содержать какие угодно произвольные байты, если получена откуда-то еще (а не из исходного кода), то нет гарантии, что она будет основана на кодировке UTF-8.
- Руна соответствует понятию кодовой точки Unicode, означающей элемент, представленный одним значением.
- При использовании UTF-8 кодовая точка Unicode может быть закодирована с помощью одного, двух, трех или четырех байтов.
- Применение функции `len` к строке возвращает количество байтов, а не количество рун.

Знать эти понятия необходимо, потому что руны в Go встречаются повсюду. Посмотрим на конкретное применение этих знаний в связи с распространенной ошибкой, совершаемой при итерации строк.

5.2. ОШИБКА #37: НЕТОЧНАЯ ИТЕРАЦИЯ СТРОК

Итерация строк — распространенное действие. Возможно, мы хотим выполнить какую-то операцию для каждой руны в строке или реализовать пользовательскую функцию для поиска определенной подстроки. В обоих случаях мы должны осуществлять перебор разных рун строки. Но в том, как работает итерация, легко запутаться.

Рассмотрим пример, где хотим вывести разные руны в строке и их соответствующие позиции:

```
s := "hêllo"  ◄——  Литерал строки содержит специальную руну — ê.
for i := range s {
   fmt.Printf("position %d: %c\n", i, s[i])
}
fmt.Printf("len=%d\n", len(s))
```

Мы используем оператор `range` для итерации по `s`, а затем выводим каждую руну, используя ее индекс в строке. Вот результат:

```
position 0: h
position 1: Ã
position 3: l
position 4: l
position 5: o
len=6
```

Этот код делает не то, что мы хотим. Выделим три момента:

- Вторая руна в выводе на печать — Ã, а не ê.
- Мы перепрыгнули с позиции 1 сразу на позицию 3... Но что находится на позиции 2?
- `len` возвращает число 6, тогда как `s` содержит только 5 рун.

Начнем с последнего момента. Мы уже упоминали, что `len` возвращает количество байтов в строке, а не количество рун. Поскольку мы присвоили `s` значение строкового литерала, то `s` будет строкой UTF-8. При этом специальный символ ê не кодируется одним байтом — для этого требуется два байта. Следовательно, вызов `len(s)` возвращает 6.

Подсчет количества рун в строке

А что, если мы хотим получить количество рун в строке, а не количество байтов? То, как мы сможем это сделать, будет зависеть от кодировки.

В предыдущем примере мы присвоили `s` значение строкового литерала, поэтому можно использовать пакет `unicode/utf8`:

```
fmt.Println(utf8.RuneCountInString(s)) // 5
```

Вернемся к рассматриваемому циклу, чтобы понять оставшиеся сюрпризы:

```
for i := range s {
   fmt.Printf("position %d: %c\n", i, s[i])
}
```

Мы должны признать, что в этом примере итерируем не каждую руну, а каждый начальный индекс руны, как показано на рис. 5.1.

При выводе на печать `s[i]` выводится не `i`-я руна, а байт с индексом `i` в представлении UTF-8. Следовательно, мы вывели `hÃllo` вместо `hêllo`. Как исправить код, чтобы он выводил все разнообразные руны? Есть два варианта.

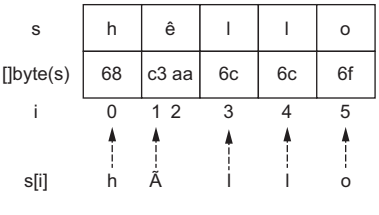


Рис. 5.1. Печать `s[i]` выводит представление UTF-8 каждого байта с индексом `i`

Мы должны использовать значение элемента оператора `range`:

```
s := "hêllo"
for i, r := range s {
   fmt.Printf("position %d: %c\n", i, r)
}
```

Чтобы не выводить руну с помощью `s[i]`, мы используем переменную `r`. Использование цикла `range` для строки возвращает две переменные: начальный индекс руны и саму руну:

```
position 0: h
position 1: ê
position 3: l
position 4: l
position 5: o
```

Другой подход заключается в преобразовании строки в срез рун и итерации по нему:

```
s := "hêllo"
runes := []rune(s)
for i, r := range runes {
   fmt.Printf("position %d: %c\n", i, r)
}
position 0: h
position 1: ê
position 2: l
position 3: l
position 4: o
```

Здесь мы преобразуем `s` в срез рун, используя `[]rune(s)`. Затем мы проводим итерацию по этому срезу и используем значение элемента оператора `range` для вывода всех рун. Единственная разница связана с позицией: вместо вывода начального индекса последовательности байтов руны код выводит непосредственно индекс руны.

Это решение приводит к оверхеду на время выполнения по сравнению с предыдущим. Действительно, преобразование строки в срез рун требует выделения места в памяти для дополнительного среза и преобразования байтов в руны: временна́я сложность $O(n)$, где n — количество байтов в строке. Поэтому если нужно выполнить итерацию по всем рунам, то используйте первое решение.

Если мы хотим получить доступ к `i`-й руне строки с помощью первого варианта, важно понимать, что доступа к индексу рун не будет, скорее мы будем знать

только начальный индекс какой-то руны в последовательности байтов. В большинстве таких случаев предпочтительнее второй вариант:

```
s := "hêllo"
r := []rune(s)[4]
fmt.Printf("%c\n", r) // o
```

Этот код выводит четвертую руну, сначала преобразуя строку в срез руны.

Возможная оптимизация доступа к определенной руне

Если строка состоит из однобайтовых рун, то возможен один метод оптимизации: например, когда строка содержит буквы от A до Z и от a до z. Мы можем получить доступ к `i`-й руне без преобразования всей строки в срез рун, обратившись к байту напрямую с помощью `s[i]`:

```
s := "hello"
fmt.Printf("%c\n", rune(s[4])) // o
```

Если требуется выполнить итерацию по рунам строки, можно использовать цикл `range` напрямую по этой строке. Но следует помнить, что индекс соответствует не индексу руны, а начальному индексу последовательности байтов руны. Если мы хотим получить доступ к самой руне, нужно использовать значение элемента оператора `range`, а не индекс в строке, потому что руна может состоять из нескольких байтов. А если нужно получить `i`-ю руну строки, то в большинстве случаев следует преобразовывать строку в срез рун.

Далее рассмотрим часто встречающийся источник путаницы при использовании функций обрезки в пакете `strings`.

5.3. ОШИБКА #38: НЕПРАВИЛЬНО ИСПОЛЬЗОВАТЬ ФУНКЦИИ ОБРЕЗКИ

Одна из распространенных ошибок при использовании пакета `strings` — некоторая неразбериха, связанная с использованием `TrimRight` и `TrimSuffix`. Обе функции служат одной цели, и их довольно легко спутать.

В следующем примере мы используем `TrimRight`. Что выведет этот код?

```
fmt.Println(strings.TrimRight("123oxo", "xo"))
```

Ответ: `123`. Но этого ли вы ожидали? Если нет, то, вероятно, вы ожидали результата функции `TrimSuffix`. Рассмотрим их обе.

`TrimRight` удаляет все завершающие руны, содержащиеся в заданном множестве. В нашем примере мы передали множество `xo`, которое содержит две руны: `x` и `o`. На рис. 5.2 показана логика этого действия.

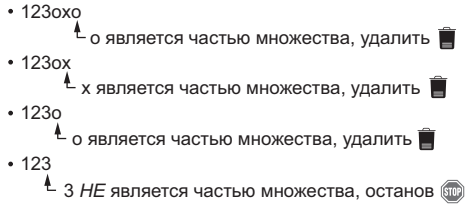


Рис. 5.2. `TrimRight` выполняет итерацию в обратном направлении, пока не найдется руна, не входящая в это множество

`TrimRight` перебирает каждую руну в обратном порядке. Если руна является частью предоставленного множества, то функция удаляет ее, если нет, то останавливает итерации и возвращает оставшуюся строку. Вот почему наш пример возвращает `123`.

С другой стороны, `TrimSuffix` возвращает строку без указанного завершающего суффикса:

```
fmt.Println(strings.TrimSuffix("123oxo", "xo"))
```

Поскольку `123oxo` заканчивается на `xo`, этот код выводит `123o`. Кроме того, удаление завершающего суффикса не является повторяющейся операцией, поэтому `TrimSuffix("123xoxo", "xo")` возвращает `123xo`.

Принцип будет тем же для левой части строки с `TrimLeft` и `TrimPrefix`:

```
fmt.Println(strings.TrimLeft("oxo123", "ox")) // 123
fmt.Println(strings.TrimPrefix("oxo123", "ox")) /// o123
```

`strings.TrimLeft` удаляет все начальные руны, содержащиеся в множестве, и, следовательно, выводит `123`. `TrimPrefix` удаляет заданный начальный префикс, выводя `o123`.

Последнее замечание по теме: `Trim` применяет к строке как `TrimLeft`, так и `TrimRight`. Поэтому он удаляет все ведущие и последующие руны, содержащиеся в множестве:

```
fmt.Println(strings.Trim("oxo123oxo", "ox")) // 123
```

Таким образом, мы должны убедиться, что понимаем разницу между `TrimRight`/`TrimLeft` и `TrimSuffix`/`TrimPrefix`:

- `TrimRight`/`TrimLeft` удаляет замыкающие/ведущие руны в наборе.
- `TrimSuffix`/`TrimPrefix` удаляет указанный суффикс/префикс.

В следующем разделе углубимся в рассмотрение конкатенации строк.

5.4. ОШИБКА #39: НЕДОСТАТОЧНАЯ СТЕПЕНЬ ОПТИМИЗАЦИИ ПРИ КОНКАТЕНАЦИИ СТРОК

Для конкатенации строк в Go предусмотрены два основных подхода, но один из них в некоторых условиях может быть очень неэффективным. Разберемся, какой вариант следует предпочесть и когда.

Напишем код с функцией `concat`, которая объединяет все строковые элементы среза с помощью оператора `+=`:

```
func concat(values []string) string {
   s := ""
   for _, value := range values {
      s += value
   }
   return s
}
```

Во время каждой итерации оператор `+=` объединяет `s` со строкой `value`. На первый взгляд эта функция может показаться правильной. Но в этой реализации мы забываем одну из основных характеристик строки: ее неизменность. Следовательно, с каждой итерацией `s` не обновляется, вместо этого в памяти создается новая строка, что сильно влияет на время выполнения этой функции.

К счастью, у этой проблемы есть решение — пакет `strings` и структура `Builder`:

```
func concat(values []string) string {
   sb := strings.Builder{}  <-- Создается strings.Builder
   for _, value := range values {
      _, _ = sb.WriteString(value)  <-- Добавляется строка
   }
   return sb.String()  <-- Возвращается результирующая строка
}
```

Здесь мы сначала создали структуру `strings.Builder`, задав ей нулевое значение. Во время каждой итерации мы создавали результирующую строку, вызывая

метод `WriteString`, который добавляет содержимое `value` во внутренний буфер, сводя к минимуму копирование памяти.

Обратите внимание, что `WriteString` в качестве второго вывода возвращает ошибку, но мы намеренно ее игнорируем. Действительно, этот метод никогда не вернет ненулевую ошибку. Так для чего же он возвращает ошибку как часть своей сигнатуры? `strings.Builder` реализует интерфейс `io.StringWriter`, который содержит единственный метод: `Write-String(s string) (n int, err error)`. Следовательно, чтобы соответствовать этому интерфейсу, `WriteString` должен возвращать ошибку.

> **ПРИМЕЧАНИЕ** Идиоматическое игнорирование ошибок мы обсудим в ошибке # 53 (не выполнять обработку ошибки).

Используя `strings.Builder`, мы также можем добавить:

- срез байта с помощью `Write`;
- одиночный байт с помощью `WriteByte`;
- одиночную руну с помощью `WriteRune`.

`strings.Builder` содержит внутри себя байтовый срез. Каждый вызов `WriteString` приводит к вызову `append`, применяемому к этому срезу. Это приводит к двум последствиям. Во-первых, эту структуру не следует использовать в режиме конкурентного выполнения, так как вызовы `append` приведут к состоянию гонки. Во-вторых, будет иметь место то, что мы уже видели при разборе ошибки #21 (неэффективная инициализация среза): если будущая длина среза уже известна, нужно заранее выделить под него место в памяти. Для этой цели в `strings.Builder` есть метод `Grow(n int)`, он помогает гарантировать наличие места для еще `n` байт.

Взглянем на другую версию метода `concat`, вызвав `Grow` с общим количеством байтов:

```
func concat(values []string) string {
   total := 0
   for i := 0; i < len(values); i++ {
      total += len(values[i])
   }
   sb := strings.Builder{}
   sb.Grow(total)
   for _, value := range values {
      _, _ = sb.WriteString(value)
   }
   return sb.String()
}
```

Проводятся итерации по каждой строке для вычисления общего числа байтов

Вызывается Grow с аргументом, равным этому общему числу

Перед началом итераций мы вычисляем общее количество байтов, которое будет содержать окончательная строка, и присваиваем это значение переменной `total`. Обратите внимание, что нас интересует не количество рун, а количество байтов, поэтому мы используем функцию `len`. Затем мы вызываем `Grow`, чтобы гарантировать наличие места для байтов `total`, прежде чем проводить итерации по строкам.

Запустим бенчмарк для сравнения трех версий (v1 с использованием +=, v2 с использованием `strings.Builder{}` без предварительного резервирования места в памяти и v3 с использованием `strings.Builder{}` с предварительным резервированием). Входной срез содержит 1000 строк, и каждая строка содержит 1000 байт:

```
BenchmarkConcatV1-4 16 72291485 ns/op
BenchmarkConcatV2-4 1188 878962 ns/op
BenchmarkConcatV3-4 5922 190340 ns/op
```

Как мы видим, последний способ самый эффективный: на 99 % быстрее, чем v1, и на 78 % быстрее, чем v2. Мы можем спросить себя, как двукратное итерирование по входному срезу может ускорить код? Ответ кроется в ошибке # 21 (неэффективная инициализация среза): если для среза с заданной длиной или емкостью не выделено место заранее, то этот срез будет продолжать расти каждый раз, когда окажется заполненным, что приведет к дополнительным выделениям памяти и копиям. Следовательно, двукратное итерирование в этом случае — наиболее эффективный вариант.

`strings.Builder` — рекомендуемое решение для конкатенации списка строк. Обычно это решение следует использовать в циклах. Если просто нужно объединить несколько строк (например, имя и фамилию), использование `strings.Builder` не рекомендуется, так как это сделает код менее читаемым, чем использование оператора `+=` или `fmt.Sprintf`.

С точки зрения производительности решение с использованием `strings.Builder` будет быстрее с того момента, когда нужно будет объединять более пяти строк. Несмотря на то что точное число зависит от многих факторов (например, от размера объединенных строк и от конкретного процессора), это может быть эмпирическим правилом, которое поможет понять, когда предпочесть одно решение другому. Также не стоит забывать, что если количество байтов будущей строки заранее известно, то следует использовать метод `Grow` для предварительного выделения места под внутренний байтовый срез.

Ниже обсудим пакет `bytes` и то, как с его помощью предотвращать бесполезные преобразования строк.

5.5. ОШИБКА #40: БЕСПОЛЕЗНЫЕ ПРЕОБРАЗОВАНИЯ СТРОК

Выбирая между работой со строками или с `[]byte`, большинство программистов выберут строки из соображений удобства. Но большая часть операций ввода/вывода на самом деле выполняется с помощью `[]byte`. Например, `io.Reader`, `io.Writer` и `io.ReadAll` работают с `[]byte`, а не со строками. Следовательно, работа со строками требует дополнительных преобразований, хотя пакет `bytes` содержит многие из тех же операций, что и пакет `strings`.

Рассмотрим пример того, что *не следует* делать. Мы реализуем функцию `getBytes`, которая принимает данные `io.Reader` в качестве входных, читает из них и вызывает функцию `sanitize`. Очищение будет выполнено путем обрезки всех начальных и конечных пробелов. Вот скелет функции `getBytes`:

```
func getBytes(reader io.Reader) ([]byte, error) {
   b, err := io.ReadAll(reader)  ◄——  b это []byte
   if err != nil {
      return nil, err
   }
   // Вызов очистки
}
```

Мы вызываем `ReadAll` и присваиваем `b` значение байтового среза. Как реализовать функцию `sanitize`? Одним из вариантов может быть создание функции `sanitize(string) string` с использованием пакета `strings`:

```
func sanitize(s string) string {
   return strings.TrimSpace(s)
}
```

Теперь вернемся к `getBytes`: когда мы проводим какие-то действия с `[]byte`, сначала нужно преобразовать его в строку, прежде чем вызывать `sanitize`. Затем нужно преобразовать результаты обратно в `[]byte`, потому что `getBytes` возвращает байтовый срез:

```
return []byte(sanitize(string(b))), nil
```

В чем проблема этой реализации? Двойная плата: за преобразование `[]byte` в строку, а затем за преобразование строки в `[]byte`. С точки зрения использования памяти каждое из этих преобразований требует дополнительного выделения места в ней. Даже если за строкой стоит какой-то `[]byte`, для преобразования `[]byte` в строку требуется создание копии байтового среза. Это означает новое выделение места в памяти и копирование всех байтов.

Неизменяемость строк

Используйте следующий код, чтобы проверить тот факт, что создание строки из `[]byte` приводит к копированию:

```
b := []byte{'a', 'b', 'c'}
s := string(b)
b[1] = 'x'
fmt.Println(s)
```

Выполнение этого кода выводит `abc`, а не `axc`. Ведь в Go строка неизменяема.

Как реализовать функцию `sanitize`? Вместо того чтобы принимать и возвращать строку, нужно произвести действия над байтовым срезом:

```
func sanitize(b []byte) []byte {
   return bytes.TrimSpace(b)
}
```

В пакете `bytes` также есть функция `TrimSpace` для обрезки всех начальных и конечных пробелов. Тогда вызов функции `sanitize` не потребует дополнительных преобразований:

```
return sanitize(b), nil
```

Как мы уже упоминали, большая часть операций ввода/вывода выполняется с помощью `[]byte`, а не строк. Когда мы задаемся вопросом, с чем работать — со строками или с `[]byte`, вспомним, что работа с `[]byte` не обязательно менее удобна. Все экспортируемые функции пакета `strings` также имеют альтернативы в пакете `bytes`: `Split`, `Count`, `Contains`, `Index` и т. д. Независимо от того, выполняем ли мы ввод/вывод или нет, сначала нужно проверить, можно ли реализовать весь процесс, используя байты вместо строк, и избежать затрат на дополнительные преобразования.

В последнем разделе обсудим, как операция с подстрокой может приводить к утечке памяти.

5.6. ОШИБКА #41: ПОДСТРОКИ И УТЕЧКИ ПАМЯТИ

При обсуждении ошибки #26 (срезы и утечки памяти) мы увидели, как нарезка среза или массива может привести к утечке памяти. Это применимо

и к операциям со строками и подстроками. Посмотрим, как в Go обрабатываются подстроки для предотвращения утечек памяти.

Чтобы извлечь подмножество строки, можно использовать такой синтаксис:

```
s1 := "Hello, World!"
s2 := s1[:5] // Hello
```

`s2` создается как подстрока `s1`. В этом примере создается строка из первых пяти байтов, а не первых пяти рун. Мы не должны использовать этот синтаксис в случае рун, закодированных несколькими байтами. Вместо этого сначала нужно преобразовать входную строку в тип `[]rune`:

```
s1 := "Hêllo, World!"
s2 := string([]rune(s1)[:5]) // Hêllo
```

Теперь рассмотрим конкретную проблему, показывающую возможные утечки памяти.

Мы будем получать сообщения журнала в виде строк. Каждый журнал сначала будет отформатирован с использованием универсального уникального идентификатора (UUID, 36 символов), за которым последует само сообщение. Требуется хранить эти UUID в памяти: например, чтобы хранить в кэше последние *n* UUID. Отметим, что такие записи из журнала могут быть довольно тяжелыми (до тысяч байтов). Вот так будет выглядеть реализация:

```
func (s store) handleLog(log string) error {
   if len(log) < 36 {
      return errors.New("log is not correctly formatted")
   }
   uuid := log[:36]
   s.store(uuid)
   // Какие-то действия
}
```

Чтобы извлечь UUID, мы используем операцию подстроки с `log[:36]`, поскольку знаем, что UUID закодирован в 36 байтах. Затем мы передаем эту переменную `uuid` в метод `store`, который сохранит ее в памяти. Несет ли это решение в себе какие-либо проблемы? Да, несет.

Для выполнения операции с подстрокой в спецификации Go не указывается, должны ли результирующая строка и строка, участвующая в этой операции, использовать одни и те же данные. Однако стандартный компилятор Go позволяет им совместно использовать один и тот же резервный массив, что с точки зрения распределения памяти и достижения лучшей производительности — наиболее

удачное решение, поскольку предотвращает новое резервирование места в памяти и копирование.

Сообщения в журнале могут быть довольно тяжелыми. `log[:36]` создаст новую строку, ссылающуюся на тот же резервный массив. Поэтому каждая строка `uuid`, которую мы храним в памяти, будет содержать не просто 36 байт, а количество байтов в исходной строке `log`: потенциально — тысячи байтов.

Как исправить ситуацию? Сделать глубокую копию подстроки, чтобы внутренний байтовый срез `uuid` ссылался на новый резервный массив, состоящий всего из 36 байт:

```
func (s store) handleLog(log string) error {
   if len(log) < 36 {
      return errors.New("log is not correctly formatted")
   }
   uuid := string([]byte(log[:36]))
   s.store(uuid)
   // Какие-то действия
}
```

Выполняется []byte, а затем — преобразование строки

Копирование выполняется путем преобразования подстроки сначала в `[]byte`, а затем снова в строку. Так мы предотвращаем утечку памяти. За строкой `uuid` стоит массив, состоящий всего из 36 байт.

Некоторые IDE или линтеры могут предупреждать, что преобразование `string([]byte(s))` не требуется. Например, GoLand — среда разработки Go от JetBrains — предупреждает об избыточном преобразовании типов. Это верно в том смысле, что мы преобразуем строку в строку, но фактически эта операция имеет реальное воздействие на поведение программы. Как уже говорилось, это предотвращает ситуацию, когда за новой строкой стоит тот же резервный массив, что и за `uuid`. Помните, что предупреждения от IDE или линтеров иногда могут быть неточными.

> **ПРИМЕЧАНИЕ** Поскольку строка, как правило, является указателем, вызов функции для передачи строки не приводит к глубокому копированию байтов. Скопированная строка по-прежнему будет ссылаться на тот же резервный массив.

Начиная с Go 1.18, стандартная библиотека также включает решение с `strings.Clone`, которое возвращает новую копию строки:

```
uuid := strings.Clone(log[:36])
```

Вызов `strings.Clone` создает копию `log[:36]` в новом месте памяти, предотвращая утечку памяти.

При совершении операции с подстрокой в Go помните о двух вещах. Во-первых, задаваемый интервал основан на числе байтов, а не рун. Во-вторых, операция с подстрокой может привести к утечке памяти, поскольку результирующая подстрока будет использовать тот же резервный массив, что и исходная строка. Чтобы этого не произошло, можно выполнить копирование строки вручную или использовать `strings.Clone` из Go 1.18.

ИТОГИ

- Для правильной работы со строками в Go важно понимать, что руна соответствует концепции кодовой точки Unicode и может состоять из нескольких байтов.
- Итерация строки с помощью `range` выполняет итерацию по рунам с индексом, соответствующим начальному индексу последовательности байтов руны. Чтобы получить доступ к определенному индексу рун (например, к третьей руне), преобразуйте строку в `[]rune`.
- `strings.TrimRight/strings.TrimLeft` удаляет все последующие/ведущие руны, содержащиеся в заданном множестве, тогда как `strings.TrimSuffix/strings.TrimPrefix` возвращает строку без указанного суффикса/префикса.
- Конкатенация списка строк должна выполняться с помощью `strings.Builder`, чтобы предотвратить резервирование места в памяти для новой строки во время каждой итерации.
- Помните, что пакет `bytes` позволяет совершать те же операции, что и пакет `strings`, — это поможет избежать лишних преобразований байт/строка.
- Использование копий вместо подстрок может предотвратить утечку памяти, поскольку строка, возвращаемая операцией над подстрокой, будет поддерживаться тем же самым массивом байтов.

6 Функции и методы

В этой главе:

- ✓ Когда использовать получатели значений или указателей
- ✓ Когда использовать именованные параметры результата и какие у них побочные эффекты
- ✓ Как избежать распространенной ошибки при возврате нулевого получателя
- ✓ Почему использование функций, которые принимают имя файла, не является лучшей практикой
- ✓ Обращение с аргументами `defer`

Функция оборачивает последовательность инструкций в модуль, который может быть вызван в другом месте. Она может принимать некоторые входные данные и производить некоторые выходные данные. А вот *метод* — это функция, привязанная к какому-то конкретному типу. Этот тип называется *получателем* и может быть указателем или значением.

Начнем с обсуждения того, как выбрать тип получателя. Затем обсудим именованные параметры, их использование и возникающие при этом ошибки. Также

обсудим типичные ошибки при создании функций или возврате ими определенных значений, таких как нулевой получатель.

6.1. ОШИБКА #42: НЕ ЗНАТЬ, КАКОЙ ТИП ПОЛУЧАТЕЛЯ ИСПОЛЬЗОВАТЬ

Выбор типа получателя для метода не всегда прост. Когда использовать получатели, являющиеся значениями? Когда использовать получатели, являющиеся указателями? В этом разделе рассмотрим условия для принятия правильного решения.

В главе 12 мы подробно поговорим о значениях и указателях, а также об основных различиях между ними. Данный раздел лишь поверхностно коснется этой темы с точки зрения производительности. Кроме того, во многих случаях использование значения или указателя в качестве получателя должно диктоваться не соображениями производительности, а чем-то другим, что мы и обсудим. Но сначала вспомним, как работают получатели.

В Go можно привязать к методу получатель либо значения, либо указателя. При использовании получателя значения Go создает копию значения и передает ее методу. Любые изменения объекта остаются локальными для метода. Исходный объект остается неизмененным.

В этом примере изменяется получатель значения:

```
type customer struct {
   balance float64
}
func (c customer) add(v float64) {   <-- Получатель значения
   c.balance += v
}
func main() {
   c := customer{balance: 100.}
   c.add(50.)
   fmt.Printf("balance: %.2f\n", c.balance)   <-- customer balance остается неизмененным
}
```

Поскольку мы используем получатель значения, увеличение `balance` в методе `add` не изменяет поле `balance` исходной структуры `customer`:

```
100.00
```

В случае с получателем указателя Go передает методу адрес объекта. По своей сути то, что передается методу, также является копией, но копируется только указатель, а не сам объект (передачи по ссылке в Go нет). Любые модификации получателя выполняются на исходном объекте. Вот тот же пример, но теперь получатель является указателем:

```
type customer struct {
   balance float64
}
func (c *customer) add(operation float64) {  <-- Получатель указателя
   c.balance += operation
}
func main() {
   c := customer{balance: 100.0}
   c.add(50.0)
   fmt.Printf("balance: %.2f\n", c.balance)  <-- customer balance обновляется
}
```

Поскольку мы используем получатель указателя, увеличение `balance` изменяет поле `balance` исходной структуры `customer`:

```
150.00
```

Выбор между получателями значений и указателей не всегда прост. Обсудим некоторые условия, которые помогут сделать выбор.

Получатель *должен* быть указателем:

- Если метод должен изменить получатель. Это правило также действует, если получатель является срезом, а метод должен добавлять элементы:

  ```
  type slice []int
  func (s *slice) add(element int) {
     *s = append(*s, element)
  }
  ```

- Если в получателе метода есть поле, которое нельзя скопировать: например, тип, входящий в пакет синхронизации (подробнее об этом в разделе об ошибке #74 (копировать тип sync).

Получатель *следует* сделать указателем:

- Если получатель — крупный объект. Использование указателя может сделать вызов более эффективным, так как предотвращает создание большой по размеру копии этого объекта. Если вы сомневаетесь в том, крупный ли это

объект, хорошей подсказкой станет бенчмаркинг. Практически невозможно указать конкретный размер, поскольку он зависит от многих факторов.

Получатель *должен* быть значением:

- Если нужно обеспечить неизменность получателя.
- Если получателем является карта, функция или канал. Иначе возникнет ошибка при компиляции.

Получатель *следует* сделать значением:

- Если получатель представляет собой срез, который не нужно изменять.
- Если получатель представляет собой небольшой массив или структуру, которая является типом значения без изменяемых полей, например `time.Time`.
- Если получатель является базовым типом — `int`, `float64` или `string`.

Один случай нужно пояснить. Допустим, мы разрабатываем другую структуру `customer`. Ее изменяемые поля не являются частью структуры напрямую, а находятся внутри другой структуры:

```
type customer struct {
   data *data
}
type data struct {
   balance float64
}
func (c customer) add(operation float64) {
   c.data.balance += operation
}
func main() {
   c := customer{data: &data{
   balance: 100,
   }}
   c.add(50.)
   fmt.Printf("balance: %.2f\n", c.data.balance)
}
```

balance не является частью структуры customer напрямую, но содержится в структуре, на которую ссылается поле указателя

Используется получатель значения

Несмотря на то что получатель является значением, вызов `add` в конце концов изменяет фактический `balance`:

```
150.00
```

В этом случае не обязательно, чтобы получатель был указателем для изменения `balance`. Но для ясности можно предпочесть получатель указателя, чтобы подчеркнуть, что `customer` как объект — изменяемый.

Смешивание типов получателей

Можно ли смешивать типы получателей, например, в структуре, содержащей несколько методов, где некоторые содержат получатели указателя, а другие — получатели значения? Общее мнение таково, что это следует запретить. Но в стандартной библиотеке есть контрпримеры, например `time.Time`.

Разработчики хотели обеспечить неизменяемость структуры `time.Time`. Следовательно, большинство методов, таких как `After`, `IsZero` и `UTC`, имеют получатели значения. Но для совместимости с существующими интерфейсами — `encoding.TextUnmarshaler` — структура `time.Time` должна реализовать метод `UnmarshalBinary([]byte) error`, который изменяет получатель, задаваемый байтовым срезом. Этот метод имеет получатель указателя.

В целом следует избегать смешивания типов получателей, но это не запрещено в 100 % случаев.

Невозможно дать исчерпывающие рекомендации по поводу использования получателей, так как всегда будут крайние случаи. Цель этого раздела в том, чтобы предоставить руководство, охватывающее большинство сценариев. По умолчанию мы можем выбирать получатель значения, если нет веских причин не делать этого. Если возникают сомнения, используйте получатель указателя.

Далее обсудим именованные параметры результата: что это и когда их использовать.

6.2. ОШИБКА #43: НЕ ИСПОЛЬЗОВАТЬ ИМЕНОВАННЫЕ ПАРАМЕТРЫ РЕЗУЛЬТАТА

Именованные параметры результата — это редко используемая в Go опция. В этом разделе рассмотрим случаи, когда целесообразно использовать именованные параметры результата, чтобы сделать API более удобным. Но сначала вспомним, как с ними работать.

Когда мы возвращаем параметры в функции или методе, мы можем присвоить этим параметрам имена и использовать их как обычные переменные. Когда параметру результата присваивается какое-то имя, то при запуске функции/метода он инициализируется с нулевым значением. С именованными параметрами результата мы также можем вызвать пустой оператор `return` (то есть

без задания его аргументов). В этом случае в качестве возвращаемых значений используются текущие значения параметров результата.

Вот пример, в котором используется именованный параметр результата b:

```
func f(a int) (b int) {  <-- Именует параметр int результата b
   b = a
   return  <-- Возвращается текущее значение b
}
```

В этом примере мы присваиваем имя b параметру результата. Когда мы вызываем return без аргументов, он возвращает текущее значение b.

Когда рекомендуется использовать именованные параметры результата? Сначала рассмотрим следующий интерфейс, который содержит метод для получения координат с заданного адреса:

```
type locator interface {
   getCoordinates(address string) (float32, float32, error)
}
```

Поскольку этот интерфейс неэкспортируемый, то подробная документация необязательна. Но сможете ли вы догадаться, прочитав этот код, что представляют собой два результата float32? Возможно, это широта и долгота, но что из них что? Зависит от конкретных соглашений: широта не всегда первый элемент. И чтобы понять суть, придется выполнить код.

Используем именованные параметры результата для удобочитаемости кода:

```
type locator interface {
   getCoordinates(address string) (lat, lng float32, err error)
}
```

Здесь уже понятны значения сигнатуры метода: сначала широта (latitude), потом долгота (longitude).

Теперь поговорим о том, когда использовать именованные параметры результата в реализации метода. Нужно ли использовать их как часть кода самой реализации?

```
func (l loc) getCoordinates(address string) (
   lat, lng float32, err error) {
   // ...
}
```

В этом конкретном случае наличие выразительной сигнатуры метода поможет читателям кода. Поэтому лучше использовать именованные параметры результата.

> **ПРИМЕЧАНИЕ** Если нужно вернуть несколько результатов одного и того же типа, можно подумать о создании специальной структуры с осмысленными именами полей. Но это не всегда возможно: например, в случае существующего интерфейса, который мы не можем обновить.

Рассмотрим еще одну сигнатуру функции, позволяющую хранить тип `Customer` в базе данных:

```
func StoreCustomer(customer Customer) (err error) {
   // ...
}
```

Здесь присваивание параметру ошибки имени `err` никак дополнительно не проясняет ситуацию и не помогает читателям. В этом случае нужно отказаться от использования именованных параметров результата.

Решение о том, использовать или нет именованные параметры результата, зависит от контекста. Если нет уверенности в том, что их использование делает код более читабельным, именованные параметры результата использовать не нужно.

Также обратите внимание, что наличие уже инициализированных параметров результата может быть весьма удобным в некоторых контекстах, даже если они не обязательно улучшают читабельность. Следующий пример, предложенный в *Effective Go* (https://go.dev/doc/effective_go), вдохновлен функцией `io.ReadFull`:

```
func ReadFull(r io.Reader, buf []byte) (n int, err error) {
   for len(buf) > 0 && err == nil {
      var nr int
      nr, err = r.Read(buf)
      n += nr
      buf = buf[nr:]
   }
   return
}
```

В этом примере именованные параметры результата не улучшают читабельность. Но поскольку и `n`, и `err` инициализируются нулевыми значениями, то такая реализация функции короче. С другой стороны, с первого взгляда эта функция может немного сбивать с толку. И это вопрос поиска баланса.

Замечание о пустых `return` (`return` без аргументов): они считаются допустимыми в коротких функциях. В противном случае они могут навредить удобочитаемости, потому что читатель должен помнить выходные данные на протяжении всей функции. Будьте последовательными в рамках функции: используйте либо только пустые операторы `return`, либо только их, но с аргументами.

В большинстве случаев использование именованных параметров результата в контексте определения интерфейса может повысить удобочитаемость без каких-либо побочных эффектов. Но в контексте реализации какого-либо метода строгого правила нет. В некоторых случаях именованные параметры результата могут повысить читаемость, например, если два параметра имеют одинаковый тип. В других случаях их также можно использовать для удобства. Именованные параметры результата следует использовать только тогда, когда от этого есть очевидная выгода.

> **ПРИМЕЧАНИЕ** При обсуждении ошибки #54 (не выполнять обработку ошибки оператора `defer`) обсудим еще один вариант использования именованных параметров результата в контексте вызовов `defer`.

При недостаточном внимании применение именованных параметров результата может привести к некоторым побочным эффектам и непредвиденным последствиям, что мы увидим в следующем разделе.

6.3. ОШИБКА #44: ПОБОЧНЫЕ ЭФФЕКТЫ ОТ ИМЕНОВАННЫХ ПАРАМЕТРОВ РЕЗУЛЬТАТА

Именованные параметры результата могут оказаться полезны в некоторых ситуациях, но поскольку инициализация этих результирующих параметров происходит с присваиванием им нулевого значения, то их применение иногда может привести к малозаметным багам. В данном разделе поговорим об этом подробнее.

Усовершенствуем наш предыдущий пример метода, который возвращает широту и долготу по заданному адресу. Поскольку мы возвращаем два числа `float32`, мы решили использовать именованные параметры результата, чтобы сделать широту и долготу явными. Эта функция сначала проверит заданный адрес, а затем определит координаты. В промежутке она выполнит проверку входного контекста, чтобы убедиться, что он не был отменен и что срок его выполнения не истек.

ПРИМЕЧАНИЕ Мы подробнее поговорим о том, что в Go подразумевается под контекстом, при разборе ошибки #60 (неверно понимать контексты Go). Если вы не знакомы с контекстами, вкратце: контекст может нести в себе сигналы отмены или крайнего срока. Мы можем проверить состояние этих сигналов, вызвав метод `Err` и убедившись, что возвращаемая ошибка не равна `nil`.

Вот новая реализация метода `getCoordinates`. Как вы думаете, что не так с этим кодом?

```
func (l loc) getCoordinates(ctx context.Context, address string) (
   lat, lng float32, err error) {
   isValid := l.validateAddress(address)   <-- Валидация адреса
   if !isValid {
      return 0, 0, errors.New("invalid address")
   }
   if ctx.Err() != nil {   <-- Проверка, был ли отменен контекст
      return 0, 0, err         и не истек ли крайний срок
   }
   // Получение и возврат координат
}
```

На первый взгляд ошибка может быть неочевидной. Ошибка, возвращаемая в области видимости `if ctx.Err() != nil`, — это `err`. Но мы не присвоили переменной `err` никакого значения. Ей по-прежнему присвоено нулевое значение типа ошибки: `nil`. Следовательно, этот код всегда будет возвращать ошибку `nil`.

Этот код компилируется, потому что `err` была инициализирована нулевым значением благодаря именованным параметрам результата. Без присвоения имени мы получили бы ошибку компиляции:

```
Unresolved reference 'err'
```

Один из возможных выходов — сделать переменную `err` равной `ctx.Err()`:

```
if err := ctx.Err(); err != nil {
   return 0, 0, err
}
```

Мы продолжаем возвращать `err`, но сначала присваиваем ей результат `ctx.Err()`. Обратите внимание, что `err` в этом примере затеняет переменную результата.

Завершим это обсуждение, еще раз подчеркнув, что именованные параметры результата могут в некоторых случаях улучшить читаемость кода (например, возврат одного и того же типа несколько раз) и быть весьма удобными в других.

Но помните, что каждый такой параметр инициализируется своим нулевым значением. Как мы видели в этом разделе, это может привести к неочевидным ошибкам, которые не всегда легко обнаружить. Будьте осторожны при использовании именованных параметров результата, чтобы избежать возможных побочных эффектов.

Использование пустого оператора return

Другой вариант — использовать пустой оператор `return`:

```
if err = ctx.Err(); err != nil {
   return
}
```

Но при этом нарушается правило, утверждающее, что не нужно смешивать в одном фрагменте кода пустые операторы `return` с такими же операторами, но с аргументами. В этом примере, вероятно, следует придерживаться первого варианта. Помните, что применение именованных параметров результата не всегда равно требованию применять пустые операторы `return`. Иногда можно просто использовать именованные параметры результата, чтобы сделать сигнатуру более чистой.

В следующем разделе обсудим распространенную ошибку, возникающую, когда функция возвращает интерфейс.

6.4. ОШИБКА #45: ВОЗВРАТ ПОЛУЧАТЕЛЯ NIL

Обсудим случаи, когда возвращается интерфейс, и поговорим, почему в некоторых обстоятельствах это может приводить к ошибкам. Эта ошибка, вероятно, одна из самых распространенных в Go, потому что ее можно считать контринтуитивной, по крайней мере до того, как ее совершили.

Рассмотрим пример. Мы будем работать над структурой `Customer` и реализуем метод `Validate` для проверки ее работоспособности (sanity check). Вместо того чтобы возвращать первую ошибку, мы хотим возвращать список ошибок. Для этого создадим собственный тип для передачи за раз нескольких ошибок:

```
type MultiError struct {
   errs []string
}

func (m *MultiError) Add(err error) {  ◄—— Добавление ошибки
```

```
   m.errs = append(m.errs, err.Error())
}

func (m *MultiError) Error() string {   <-- Реализация интерфейса возврата ошибок
   return strings.Join(m.errs, ";")
}
```

`MultiError` удовлетворяет интерфейсу ошибок, поскольку реализует строку `Error()`. Он раскрывает метод `Add` для добавления ошибки. С помощью этой структуры мы можем реализовать метод `Customer.Validate`, чтобы проверить возраст и имя клиента. Если проверка на работоспособность пройдена успешно, мы хотим вернуть ошибку `nil`:

```
func (c Customer) Validate() error {
   var m *MultiError   <-- Создается пустой *MultiError
   if c.Age < 0 {
      m = &MultiError{}
      m.Add(errors.New("age is negative"))   <-- Добавляется ошибка, если значение возраста отрицательно
   }
   if c.Name == "" {
      if m == nil {
         m = &MultiError{}
      }
      m.Add(errors.New("name is nil"))   <-- Добавляется ошибка, если значение имени равно nil
   }
   return m
}
```

В этой реализации `m` инициализируется нулевым значением `*MultiError`, то есть `nil`. Когда же проверка на работоспособность не проходит, мы создаем новую `MultiError` (если необходимо), а затем добавляем ошибку. В конце концов мы возвращаем `m`, которая может быть либо нулевым указателем, либо указателем на структуру `MultiError`, в зависимости от результатов проверки.

Проверим эту реализацию, запустив код с валидной структурой `Customer`:

```
customer := Customer{Age: 33, Name: "John"}
if err := customer.Validate(); err != nil {
      log.Fatalf("customer is invalid: %v", err)
}
```

В результате получим:

```
2021/05/08 13:47:28 customer is invalid: <nil>
```

Результат может удивить. Структура `Customer` валидна, но и условие `err != nil` истинно, и запись на ошибки привела к выводу `<nil>`. В чем проблема?

Мы должны знать, что в Go получатель указателя может равняться `nil`. Поэкспериментируем, создав фиктивный тип и вызвав метод с нулевым получателем указателя:

```
type Foo struct{}
func (foo *Foo) Bar() string {
   return "bar"
}

func main() {
   var foo *Foo
   fmt.Println(foo.Bar())    ◄—— foo равна nil
}
```

`foo` инициализируется нулевым значением указателя `nil`. Но этот код компилируется и выводит `bar`, если мы его запустим. Нулевой указатель является допустимым получателем.

Почему так? В Go метод — это синтаксический сахар для функции, первым параметром которой является получатель. Следовательно, метод `Bar`, который мы рассматривали, похож на эту функцию:

```
func Bar(foo *Foo) string {
   return "bar"
}
```

Мы знаем, что передача нулевого указателя на функцию допустима. Поэтому и использование нулевого указателя в качестве получателя допустимо.

Вернемся к исходному примеру:

```
func (c Customer) Validate() error {
   var m *MultiError
   if c.Age < 0 {
      // ...
   }
   if c.Name == "" {
      // ...
   }
   return m
}
```

`m` инициализируется нулевым значением указателя `nil`. Затем, если все проверки проходят успешно, аргумент оператора `return` является не самим `nil`, а нулевым указателем. Поскольку нулевой указатель является допустимым получателем, преобразование результата в интерфейс не даст нулевого значения. То есть вызывающий объект `Validate` всегда будет получать ненулевую ошибку.

Чтобы прояснить этот момент, вспомним, что в Go интерфейс — это обертка диспетчеризации (dispatch wrapper). Здесь то, что содержится в этой обертке, равно нулю (указатель `MultiError`), а сама обертка — нет (интерфейс `error`) (рис. 6.1).

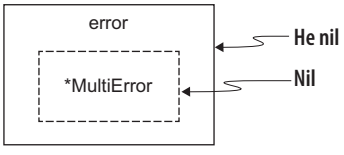


Рис. 6.1. Обертка `error` ненулевая

Независимо от того, чем является `Customer`, вызывающая эту функцию сторона всегда будет получать ненулевую ошибку. Понимание такого поведения крайне важно, потому что эта ошибка широко распространена.

Итак, что нужно, чтобы сделать код этого примера корректным? Самое простое решение — возвращать `m`, только если она не равна нулю:

```
func (c Customer) Validate() error {
   var m *MultiError
   if c.Age < 0 {
      // ...
   }
   if c.Name == "" {
      // ...
   }
   if m != nil {
      return m   <-- m возвращается, только если была хотя бы одна ошибка
   }
   return nil   <-- В противном случае возвращается nil
}
```

В самом конце метода мы проверяем, не равна ли `m` нулю. Если да, то возвращаем `m`. В противном случае мы возвращаем `nil` в явном виде. Следовательно, если структура `Customer` валидная, мы возвращаем нулевой интерфейс, а не нулевой получатель, преобразованный в ненулевой интерфейс.

Мы увидели, что в Go допускается использование нулевого получателя, а интерфейс, преобразованный из нулевого указателя, не является нулевым интерфейсом. По этой причине, когда нужно вернуть интерфейс, нужно возвращать не нулевой указатель, а непосредственно нулевое значение. Как правило, наличие нулевого указателя нежелательно и означает вероятное наличие ошибки.

Рассмотренный здесь случай — наиболее распространенный и приводящий к ошибке. Но такая проблема связана не только с ошибками: это может произойти с любым интерфейсом, реализованным с использованием получателями указателей.

В следующем разделе обсудим типичную ошибку при использовании имени файла в качестве входных данных функции.

6.5. ОШИБКА #46: ИСПОЛЬЗОВАТЬ ИМЯ ФАЙЛА В КАЧЕСТВЕ ВХОДНЫХ ДАННЫХ ФУНКЦИИ

При создании новой функции, которая должна прочитать файл, передача ей имени файла не считается хорошей практикой и может приводить к негативным последствиям, например к сложностям в написании юнит-тестов. Углубимся в эту проблему и поймем, как ее устранить.

Предположим, нужно реализовать функцию для подсчета количества пустых строк в файле. Одним из способов реализации этой функции было бы принятие имени файла и использование `bufio.NewScanner` для сканирования и проверки каждой его строки:

```
func countEmptyLinesInFile(filename string) (int, error) {
   file, err := os.Open(filename)  <-- Открытие filename
   if err != nil {
      return 0, err
   }
   // Обработка закрытия файла
   scanner := bufio.NewScanner(file)  <-- Создание функции scanner из переменной *os.File, которая разбивает входной файл на строки
   for scanner.Scan() {  <-- Итерации по каждой строке
      // ...
   }
}
```

Мы открываем файл `filename`. Затем используем `bufio.NewScanner` для сканирования каждой строки (по умолчанию вводимый файл разбивается на строки).

У этой функции ожидаемое поведение. Пока задаваемое имя файла валидно, мы будем читать данные из него и возвращать количество пустых строк. Но в чем проблема?

Допустим, нужно реализовать юнит-тесты, чтобы покрыть следующие случаи:

- номинальный случай;
- пустой файл;
- файл, содержащий только пустые строки.

Каждый юнит-тест потребует создания в нашем Go-проекте какого-то файла. Чем сложнее функция, тем больше случаев нужно протестировать и тем больше файлов мы создадим. В некоторых случаях может потребоваться создать десятки файлов, и процесс станет неуправляемым.

Кроме того, эту функцию нельзя переиспользовать. Например, если бы пришлось реализовать ту же логику, но подсчитать количество пустых строк при HTTP-запросе, пришлось бы продублировать основную логику:

```
func countEmptyLinesInHTTPRequest(request http.Request) (int, error) {
   scanner := bufio.NewScanner(request.Body)
   // Копируется та же самая логика
}
```

Один из способов преодолеть эти ограничения — сделать так, чтобы функция принимала `*bufio.Scanner` (вывод, возвращаемый `bufio.NewScanner`). Обе функции имеют одинаковую логику с момента создания переменной `scanner`, поэтому такой подход будет работать. Но в Go идиоматический способ — начать с абстракции считывания данных.

Напишем новую версию функции `countEmptyLines`, которая вместо этого получает абстракцию `io.Reader`:

```
func countEmptyLines(reader io.Reader) (int, error) {
   scanner := bufio.NewScanner(reader)
   for scanner.Scan() {
      // ...
   }
}
```

Получение io.Reader в качестве входных данных

Создание *bufio.Scanner из io.Reader, а не из *os.File

Поскольку `bufio.NewScanner` принимает `io.Reader`, мы можем напрямую передать переменную `reader`.

В чем преимущества такого подхода? Прежде всего, эта функция абстрагируется от источника данных. Он является файлом? HTTP-запросом? Входом сокета? Для функции это не важно. Поскольку `*os.File` и поле `Body` в `http.Request` реализуют `io.Reader`, мы можем переиспользовать одну и ту же функцию независимо от типа ввода.

Еще одно преимущество связано с тестированием. Мы упоминали, что создание файла для каждого теста может стать громоздкой процедурой. Теперь, когда `countEmptyLines` принимает `io.Reader`, мы можем реализовать юнит-тесты, создав `io.Reader` из строки:

```
func TestCountEmptyLines(t *testing.T) {
   emptyLines, err := countEmptyLines(strings.NewReader(
        `foo
           bar
           baz
           `))
    // Логика теста
}
```

Передача io.Reader из строки

В этом тесте мы создаем `io.Reader`, используя `strings.NewReader` напрямую из строкового литерала. Поэтому не нужно создавать для каждого теста свой файл. Каждый тест-кейс может быть автономным, что улучшает читабельность и удобство сопровождения, поскольку не нужно открывать другой файл, чтобы увидеть содержимое.

Использование имени файла в качестве входных данных функции для чтения из файла в большинстве случаев следует рассматривать как код с душком (за исключением определенных функций, например `os.Open`). Мы увидели, что это усложняет юнит-тесты, поскольку может потребоваться создать несколько файлов. Это также снижает возможность многократного использования функции (хотя не все функции предназначены для такого использования). Использование интерфейса `io.Reader` абстрагирует источник данных. Независимо от того, являются ли входные данные файлом, строкой, HTTP-запросом или запросом gRPC, функцию с такой реализацией можно переиспользовать и легко протестировать.

В последнем разделе этой главы обсудим типичную ошибку, связанную с оператором `defer`: как вычисляются аргументы функции/метода и получатели метода.

6.6. ОШИБКА #47: ИГНОРИРОВАТЬ ТО, КАК ВЫЧИСЛЯЮТСЯ АРГУМЕНТЫ И ПОЛУЧАТЕЛИ ОПЕРАТОРА DEFER

В предыдущем разделе мы говорили, что оператор `defer` задерживает выполнение вызова до тех пор, пока окружающая функция не вернет результат. Go-разработчики не всегда понимают, как вычисляются аргументы. Углубимся в эту проблему в двух подразделах: первый относится к аргументам функций и методов, а второй связан с получателями методов.

6.6.1. Вычисление аргументов

Чтобы понять, как вычисляются аргументы `defer`, рассмотрим пример. В нем некая функция должна вызывать две функции — `foo` и `bar`. Кроме того, она также должна отслеживать статус выполнения:

- `StatusSuccess`, если и `foo` и `bar` не возвращают никаких ошибок.
- `StatusErrorFoo`, если возвращает ошибку `foo`.
- `StatusErrorBar`, если возвращает ошибку `bar`.

Будем использовать этот статус для нескольких действий, например для уведомления другой горутины и для инкремента счетчиков. Чтобы избежать повторения этих вызовов перед каждым оператором `return`, будем использовать `defer`. Это первая реализация:

```
const (
   StatusSuccess = "success"
   StatusErrorFoo = "error_foo"
   StatusErrorBar = "error_bar"
)
func f() error {
   var status string
   defer notify(status)  <-- Откладывает вызов notify
   defer incrementCounter(status)  <-- Откладывает вызов incrementCounter
   if err := foo(); err != nil {
      status = StatusErrorFoo  <-- Устанавливает значение status в error foo
      return err
   }
   if err := bar(); err != nil {
      status = StatusErrorBar  <-- Устанавливает значение status в error bar
      return err
   }
   status = StatusSuccess  <-- Устанавливает значение status в success
   return nil
}
```

Сначала мы объявляем переменную `status`. Затем откладываем вызовы `notify` и `incrementCounter` с помощью `defer`. При выполнении всей этой функции и в зависимости от пути выполнения мы соответствующим образом обновляем `status`.

Но если попробовать запустить выполнение этой функции, мы увидим, что независимо от пути выполнения `notify` и `incrementCounter` всегда вызываются с одним и тем же статусом: пустая строка. Почему?

Важно понять одну вещь о вычислении аргументов функции `defer`: они вычисляются *сразу*, а не после возврата окружающей функции. В нашем примере мы вызываем `notify(status)` и `incrementCounter(status)` как отложенные функции. Следовательно, Go отложит выполнение этих вызовов до того момента, как `f` вернется с текущим значением `status` на этапе, на котором мы использовали `defer`, передав таким образом пустую строку. Как решить эту проблему, если мы хотим продолжать использовать `defer`? Есть два способа.

Первое решение — передать строковый указатель функциям `defer`:

```
func f() error {
   var status string
   defer notify(&status)
   defer incrementCounter(&status)
   // Далее код функции остается неизменным
   if err := foo(); err != nil {
      status = StatusErrorFoo
      return err
   }
   if err := bar(); err != nil {
      status = StatusErrorBar
      return err
   }
   status = StatusSuccess
   return nil
}
```

Указатель на строку передается функции notify как аргумент

Указатель на строку передается функции incrementCounter как аргумент

Продолжаем обновлять `status` в зависимости от случая, но теперь `notify` и `incrementCounter` получают в качестве аргумента указатель на строку. Почему этот подход работает?

При использовании `defer` аргументы вычисляются сразу — в данном примере они являются адресом `status`. Да, сама переменная `status` изменяется на протяжении выполнения функции, но ее адрес остается неизменным вне зависимости от того, что происходит. Следовательно, если `notify` или `incrementCounter` использует значение, на которое ссылается указатель строки, они будут работать так, как и ожидалось. Но это решение требует изменения сигнатуры двух функций, что не всегда возможно.

Есть и другое решение: вызов замыкания как оператора `defer`. Напоминаем, что замыкание (closure) — это анонимная функция, которая ссылается на переменные вне своего тела. Аргументы, переданные в функцию `defer`, вычисляются сразу. Но мы должны знать, что переменные, на которые ссылается замыкание `defer`, вычисляются *во время выполнения* замыкания (следовательно, когда возвращается окружающая функция).

Вот пример, показывающий работу замыкания `defer`. Оно ссылается на две переменные, одна из которых аргумент функции, а вторая — переменная вне ее тела:

```
func main() {
    i := 0
    j := 0
    defer func(i int) {
        fmt.Println(i, j)
    }(i)
    i++
    j++
}
```

Вызов в качестве функции defer замыкания, которому в качестве входных данных передается целое число

i — это входная переменная функции, а j — внешняя переменная

Передача i замыканию (вычисляется сразу)

Здесь замыкание использует переменные `i` и `j`. `i` передается как аргумент функции, поэтому вычисляется немедленно. И наоборот, `j` ссылается на переменную вне тела замыкания, поэтому вычисляется при выполнении замыкания. Если запустить этот пример, будет выведено `0 1`.

Следовательно, можно использовать замыкание для реализации новой версии нашей функции:

```
func f() error {
   var status string
   defer func() {
      notify(status)
      incrementCounter(status)
   }()
   // Далее код функции остается неизменным
}
```

Вызов замыкания как отложенной функции

Вызов notify внутри замыкания и ссылка на status

Вызов incrementCounter внутри замыкания и ссылка на status

Здесь мы включаем вызовы `notify` и `incrementCounter` в замыкание. Это замыкание ссылается на переменную `status` вне своего тела. Таким образом, `status` вычисляется после выполнения замыкания, а не при вызове `defer`. Это решение также работает и не требует изменения сигнатуры `notify` и `incrementCounter`.

А что происходит при применении `defer` к методу с указателем или получателем значения? Давайте посмотрим.

6.6.2. Получатели значений или указателей

При разборе ошибки #42 (не знать, какой тип получателя использовать) мы говорили, что получатель может быть либо значением, либо указателем. Та же логика, связанная с оценкой аргумента, применяется, когда мы используем `defer` в методе: получатель вычисляется сразу. Разберем последствия применения обоих типов получателей.

Во-первых, вот пример, в котором вызывается метод в применении к получателю значения с использованием `defer`, но впоследствии получатель меняется:

```
func main() {
   s := Struct{id: "foo"}
   defer s.print()  <-- s вычисляется немедленно
   s.id = "bar"  <-- обновление s.id (невидимое)
}
type Struct struct {
   id string
}
func (s Struct) print() {
   fmt.Println(s.id)  <-- foo
}
```

Мы откладываем выполнение вызова метода `print`. Как и в случае с аргументами, вызов `defer` немедленно вычисляет получателя. Следовательно, `defer` задерживает выполнение метода со структурой, которая содержит поле `id`, равное `foo`. Поэтому этот код выводит `foo`.

И наоборот, если указатель является получателем, потенциальные изменения получателя после вызова `defer` будут видны:

```
func main() {
   s := &Struct{id: "foo"}
   defer s.print()
   s.id = "bar"  <-- Обновление s.id (видимое)
}
type Struct struct {
   id string
}
func (s *Struct) print() {
   fmt.Println(s.id)  <-- bar
}
```

s — указатель, поэтому он вычисляется немедленно, но может ссылаться на другую переменную при выполнении метода defer

Получатель `s` также вычисляется немедленно. Однако вызов метода приводит к копированию получателя указателя. Следовательно, изменения, внесенные в структуру, на которую ссылается указатель, видимые. Этот код выводит `bar`.

Когда мы вызываем `defer` в применении к функции или методу, их аргументы вычисляются немедленно. Если мы потом захотим изменить задаваемые перед `defer` аргументы, мы можем использовать указатели или замыкания. Для метода сразу вычисляется и получатель. Следовательно, поведение зависит от того, является получатель значением или указателем.

ИТОГИ

- Решение об использовании получателя значения или указателя должно приниматься на основе его типа, необходимости изменения, наличия поля, которое не может быть скопировано, и размера объекта. Если сомневаетесь, используйте получатель указателя.
- Применение именованных параметров результата может быть эффективным способом улучшить читаемость функции/метода, особенно если несколько параметров результата имеют один и тот же тип. В некоторых случаях этот подход также может быть удобен, так как именованные параметры результата инициализируются нулевым значением. Но будьте очень внимательны в связи с возможными побочными эффектами.
- В случае возврата интерфейса будьте очень внимательны, если потребуется возвращать его нулевое значение: надо вернуть не нулевой указатель, а явное значение `nil`. В противном случае могут возникнуть непредвиденные последствия, поскольку вызывающая сторона получит значение не `nil`.
- Определение функций, которые получают типы `io.Reader` вместо имен файлов, повышает шансы на возможность переиспользования этих функций и упрощает тестирование.
- Передача указателя функции `defer` и перенос вызова внутрь замыкания — два возможных решения, позволяющих обойти ситуацию, при которой аргументы и получатели вычисляются сразу.

7 Обработка ошибок

В этой главе:

- ✓ Когда нужен режим паники
- ✓ Когда следует оборачивать ошибку
- ✓ Эффективное сравнение типов и значений ошибок, начиная с версии Go 1.13
- ✓ Идиоматическая обработка ошибок
- ✓ Как следует игнорировать ошибку
- ✓ Обработка ошибок в вызовах `defer`

Обработка ошибок — это фундаментальный аспект создания надежных и наблюдаемых приложений, и этот аспект должен быть столь же важным, как и любая другая часть кода. В Go, в отличие от большинства языков программирования, обработка ошибок не основывается на традиционном механизме try/catch. В Go ошибки обрабатываются с помощью возвращения значения ошибки вместе с другими значениями из функции.

В этой главе рассмотрим распространенные проблемы, связанные с обработкой ошибок.

7.1. ОШИБКА #48: ПАНИКА

Начинающие Go-разработчики часто путаются в обработке ошибок. В Go ошибки обычно обрабатываются функциями или методами, которые в качестве своего последнего параметра возвращают тип `error`. Но некоторым разработчикам такой подход может показаться неожиданным, и у них возникнет соблазн обработать ошибки с помощью `panic` и `recover` — так же, как это делается в Java или Python. Освежим представления о концепции паники и обсудим, когда паниковать считается уместным.

В Go `panic` — это встроенная функция, которая останавливает обычный поток:

```
func main() {
   fmt.Println("a")
   panic("foo")
   fmt.Println("b")
}
```

Этот код выводит `a`, а затем останавливается перед выводом `b`:

```
a
panic: foo

goroutine 1 [running]:
main.main()
        main.go:7 +0xb3
```

После запуска паники она продолжается вверх по стеку вызовов до тех пор, пока либо не произойдет возврат из текущей горутины, либо `panic` не будет перехвачен с помощью `recovery`:

```
func main() {
   defer func() {
      if r := recover(); r != nil {
         fmt.Println("recover", r)
      }
   }()
   f()
}
func f() {
   fmt.Println("a")
   panic("foo")
   fmt.Println("b")
}
```

Вызовы восстанавливаются внутри отложенного замыкания

Вызов f, которая запускает панику. Паника отлавливается предыдущим восстановлением

Когда вызывается `panic`, текущее выполнение функции `f` останавливается и функция поднимается вверх по стеку вызовов: в `main`. Поскольку паника

перехватывается с помощью recover, в `main` она не останавливает выполнение горутины:

```
a
recover foo
```

Вызов функции `recovery()` для перехвата паники горутины полезен только внутри функции `defer`; в противном случае функция просто вернет `nil` и более ни на что не будет влиять. Это связано с тем, что функции `defer` также выполняются, когда окружающая функция вызывает панику.

Теперь подумаем, а когда уместно вызывать панику? В Go `panic` используется для обозначения по-настоящему исключительных ситуаций, таких как ошибка программиста. Например, если мы обратимся к пакету `net/http`, то заметим, что в методе `WriteHeader` есть вызов функции `checkWriteHeaderCode` для проверки правильности кода состояния:

```
func checkWriteHeaderCode(code int) {
   if code < 100 || code > 999 {
      panic(fmt.Sprintf("invalid WriteHeader code %v", code))
   }
}
```

Эта функция вызывает панику, если код состояния недействителен, что в чистом виде является ошибкой программиста.

Другой пример, в основе которого лежит ошибка программиста, можно найти в пакете `database/sql` при регистрации драйвера базы данных:

```
func Register(name string, driver driver.Driver) {
   driversMu.Lock()
   defer driversMu.Unlock()
   if driver == nil {
      panic("sql: Register driver is nil")  <-- Паника, если драйвер нулевой
   }
   if _, dup := drivers[name]; dup {
      panic("sql: Register called twice for driver " + name)  <-- Паника, если драйвер уже зарегистрирован
   }
   drivers[name] = driver
}
```

Эта функция вызывает состояние паники, если драйвер равен `nil` (`driver.Driver` — это интерфейс) или уже зарегистрирован. Оба случая снова будут считаться ошибками программиста. Кроме того, в большинстве случаев (например, с `go-sql-driver/mysql` (https://github.com/go-sql-driver/mysql), самым популярным драйвером MySQL для Go) `Register` вызывается через функцию `init`,

что ограничивает возможности по обработке ошибок. По всем этим причинам разработчики сделали так, что в случае ошибки функция вызывает панику.

Другой случай паники — когда наше приложение требует зависимость, но ее не удается инициализировать. Допустим, мы предоставляем сервис для создания новых учетных записей клиентов. На каком-то этапе сервису нужно проверить действительность введенного адреса электронной почты. Чтобы реализовать это, мы решаем использовать регулярное выражение.

В Go пакет `regexp` предоставляет две функции для создания регулярного выражения из строки: `Compile` и `MustCompile`. Первая возвращает `*regexp.Regexp` и ошибку, а вторая возвращает только `*regexp.Regexp`, но в случае ошибки вызывает панику. В этом случае регулярное выражение — обязательная зависимость. Действительно, если не удастся его скомпилировать, мы никогда не сможем проверить правильность введенного адреса электронной почты. Поэтому следует использовать `MustCompile` и в случае обнаружения ошибки вызвать панику.

Панику в Go следует использовать с осторожностью. Мы рассмотрели два важных случая: в одном из них должен возникнуть сигнал об ошибке программиста, а в другом приложение не может создать обязательную зависимость, то есть когда налицо появление неких исключительных ситуаций, которые должны заставить нас остановить приложение. В большинстве других случаев управление ошибками должно выполняться с помощью функции, которая возвращает правильный тип `error` в качестве последнего возвращаемого аргумента.

Приступим к обсуждению ошибок. В следующем разделе посмотрим, когда следует оборачивать ошибку.

7.2. ОШИБКА #49: ИГНОРИРОВАТЬ ОБОРАЧИВАНИЕ ОШИБКИ

Начиная с Go 1.13, директива `%w` позволяет удобно оборачивать ошибки. Но не всегда понятно, когда это нужно делать, а когда нет. Вспомним, что такое оборачивание ошибок и когда его использовать.

Оборачивание — это упаковка ошибки внутри контейнера-обертки, который делает доступной и исходную ошибку (рис. 7.1). Есть два основных сценария использования оборачивания ошибок:

- добавление дополнительного контекста к ошибке;
- маркировка ошибки как специфической.

Рис. 7.1. Оборачивание ошибок

Рассмотрим пример. Мы получаем запрос от конкретного пользователя на доступ к ресурсу базы данных, но во время запроса получаем ошибку «Отказ в доступе» («permission denied»). Если информация об этой ошибке заносится в какой-то журнал, то для целей отладки нужно добавить дополнительный контекст. В этом случае мы можем обернуть ошибку, чтобы указать, кто был пользователем и к какому ресурсу был запрос (рис. 7.2).

Рис. 7.2. Добавление дополнительного контекста к ошибке «Отказ в доступе»

Теперь предположим, что вместо добавления контекста мы хотим пометить ошибку. Например, реализовать обработчик HTTP, который проверяет, все ли ошибки, полученные при вызове функций, относятся к типу `Forbidden`, чтобы можно было вернуть код состояния 403. В этом случае мы можем обернуть эту ошибку внутри `Forbidden` (рис. 7.3).

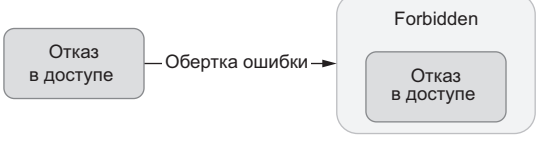


Рис. 7.3. Маркировка ошибки `Forbidden`

В обоих случаях исходная ошибка остается доступной. Следовательно, вызывающая сторона также может обработать ошибку, развернув ее и перепроверив источник ошибки. Обратите внимание, что иногда есть смысл использовать оба подхода: и добавлять контекст, и маркировать ошибку.

Теперь, когда мы выяснили основные ситуации, в которых можно использовать оборачивание ошибки, посмотрим на способы возврата полученной нами ошибки. Рассмотрим фрагмент кода и изучим опции внутри блока `if err != nil`:

```
func Foo() error {
   err := bar()
   if err != nil {
   // ?  <-- Как возвратить ошибку?
   }
   // ...
}
```

Первый вариант — вернуть эту ошибку напрямую. Если мы не хотим ее пометить и понимаем, что нет какого-либо полезного контекста, который имеет смысл добавить, то такой подход вполне подойдет:

```
if err != nil {
   return err
}
```

На рис. 7.4 показано, что мы возвращаем ту же ошибку, что и `bar`.

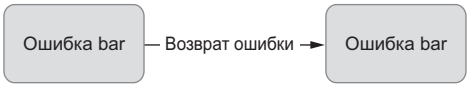


Рис. 7.4. Мы можем вернуть ошибку напрямую

До версии Go 1.13 для оборачивания ошибки единственным вариантом без применения внешней библиотеки было создание пользовательского типа ошибки:

```
type BarError struct {
   Err error
}
func (b BarError) Error() string {
   return "bar failed:" + b.Err.Error()
}
```

Затем вместо прямого возврата `err` мы обернули эту ошибку в `BarError` (см. рис. 7.5):

```
if err != nil {
   return BarError{Err: err}
}
```

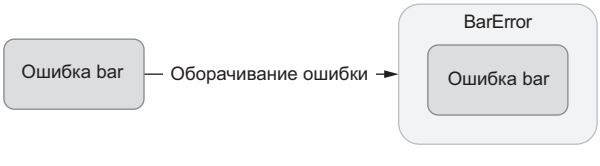


Рис. 7.5. Оборачивание ошибки внутри `BarError`

Преимущество такого варианта в гибкости. Поскольку `BarError` — это пользовательская структура, то при необходимости мы можем добавить в нее любой

дополнительный контекст. Но если потребуется повторить эту операцию, то необходимость создавать специфический тип ошибки может загрязнить код.

Для решения этой проблемы в Go 1.13 появилась директива `%w`:

```
if err != nil {
   return fmt.Errorf("bar failed: %w", err)
}
```

Этот код оборачивает исходную ошибку, чтобы можно было добавлять дополнительный контекст без необходимости создания другого типа ошибки (рис. 7.6).

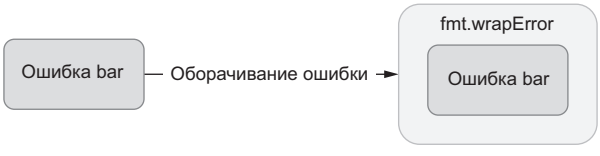


Рис. 7.6. Оборачивание ошибки в стандартную ошибку

Поскольку исходная ошибка остается доступной, клиент может развернуть родительскую ошибку, а затем проверить, относится ли исходная ошибка к какому-либо специфическому типу или значению (эти вопросы обсуждаются в следующих разделах).

Последний вариант, который мы обсудим, — использование директивы `%v`:

```
if err != nil {
   return fmt.Errorf("bar failed: %v", err)
}
```

Отличие заключается в том, что сама ошибка не обернутая. Мы преобразуем ее в другую ошибку, чтобы добавить контекст, и исходная ошибка становится недоступной (рис. 7.7).

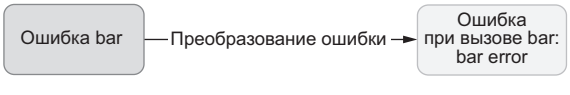


Рис. 7.7. Преобразование ошибки

Информация об источнике проблемы остается доступной. Но вызывающая сторона не может развернуть эту ошибку и проверить, была ли источником всех неприятностей `bar error`. Так что в некотором смысле эта опция носит более ограничительный характер, чем `%w`. Нужно ли предотвращать такие ситуации, поскольку стала доступной директива `%w`? Не обязательно.

Оборачивание ошибки делает исходную ошибку доступной для вызывающей стороны. Следовательно, это означает введение их потенциальной связки.

Представьте, что мы используем обертку, и вызывающая `Foo` сторона проверяет, является ли `bar error` исходной ошибкой. А что, если мы изменим код и воспользуемся другой функцией, которая будет возвращать другой тип ошибки? Это будет нарушать процедуру проверки ошибок, сделанную вызывающей стороной.

Чтобы убедиться, что наши клиенты не полагаются на то, что мы считаем деталями реализации, возвращаемая ошибка должна быть преобразована, а не обернута. В таком случае вместо `%w` можно использовать `%v`.

Подведем промежуточный итог этих вариантов.

Вариант/случай	Дополнительный контекст	Пометка ошибки	Исходная ошибка доступна?
Возврат ошибки напрямую	Нет	Нет	Да
Пользовательский тип ошибки	Возможен (например, если тип ошибки содержит строковое поле)	Да	Возможно (если исходная ошибка экспортируется или доступна через метод)
`fmt.Errorf c %w`	Да	Нет	Да
`fmt.Errorf c %v`	Да	Нет	Нет

При обработке ошибки мы можем обернуть ее. Обертка — это добавление к ошибке дополнительного контекста и/или ее маркировка как специфической: если нужно пометить ошибку, мы должны создать для нее собственный тип. Но если нужно лишь добавить дополнительный контекст, то следует использовать `fmt.Errorf` с директивой `%w`, так как при этом не нужно создавать новый тип ошибки. Тем не менее при оборачивании ошибок создается потенциальная связь, поскольку исходная ошибка становится доступной для вызывающей стороны. Если надо предотвратить это, то следует использовать не обертку, а преобразование ошибки, например `fmt.Errorf` с директивой `%v`.

В этом разделе я показал, как обернуть ошибку с помощью директивы `%w`. Но как ее использование повлияет на проверку типа ошибки?

7.3. ОШИБКА #50: НЕТОЧНАЯ ПРОВЕРКА ТИПА ОШИБКИ

В предыдущем разделе мы рассмотрели возможный способ оборачивания ошибок с помощью директивы `%w`. Но при использовании этого подхода важно изменить

и способ проверки типа ошибки на его специфичность, иначе обработка ошибок может оказаться неточной.

Рассмотрим пример. Напишем обработчик HTTP для возврата суммы транзакции по идентификатору (ID). Обработчик будет анализировать запрос, чтобы получить данные об этом ID и информацию о сумме из базы данных (БД). Реализация такой функции может приводить к ошибкам в двух случаях:

- Если ID недействителен (длина строки не равна пяти символам).
- Если запрос к БД не удался.

В первом случае мы хотим вернуть `StatusBadRequest (400)`, а во втором — `ServiceUnavailable (503)`. Для этого создадим тип `transientError`, чтобы подчеркнуть, что ошибка временная. Родительский обработчик проверит тип ошибки. Если ошибка будет иметь тип `transientError`, обработчик вернет код состояния 503, в противном случае — код 400.

Сосредоточимся на определении типа ошибки и функции, которую будет вызывать обработчик:

```
type transientError struct {
   err error
}
func (t transientError) Error() string {
   return fmt.Sprintf("transient error: %v", t.err)
}
func getTransactionAmount(transactionID string) (float32, error) {
   if len(transactionID) != 5 {
      return 0, fmt.Errorf("id is invalid: %s",
         transactionID)
   }
   amount, err := getTransactionAmountFromDB(transactionID)
   if err != nil {
      return 0, transientError{err: err}
   }
   return amount, nil
}
```

Создается пользовательский тип transientError

Возврат простой ошибки, если ID транзакции недействителен

Возврат ошибки типа transientError в случае сбоя при запросе в БД

`getTransactionAmount` возвращает ошибку, используя `fmt.Errorf`, если ID недействителен. Но если получить данные о сумме транзакции из БД не удается, `getTransactionAmount` оборачивает ошибку в тип `transientError`.

Теперь напишем код HTTP-обработчика, который проверяет тип ошибки и возвращает соответствующий HTTP-код состояния:


```
func handler(w http.ResponseWriter, r *http.Request) {
   transactionID := r.URL.Query().Get("transaction")
   amount, err := getTransactionAmount(transactionID)
   if err != nil {
      switch err := err.(type) {
      case transientError:
         http.Error(w, err.Error(), http.StatusServiceUnavailable)
      default:
         http.Error(w, err.Error(), http.StatusBadRequest)
      }
      return
   }
   // Текст ответа
}
```

Применяя оператор `switch` к типу ошибки, мы возвращаем соответствующий код состояния HTTP: 400 в случае неудачи обращения к БД или 503 в случае переходящей ошибки.

Этот код вполне работоспособен. Но предположим, что нужно выполнить небольшой рефакторинг `getTransactionAmount`. Теперь `TransientError` будет возвращаться функцией `getTransactionAmountFromDB` вместо `getTransactionAmount`. При этом `getTransactionAmount` оборачивает эту ошибку с помощью директивы `%w`:

```
func getTransactionAmount(transactionID string) (float32, error) {
   // Проверка действительности транзакционного ID
   amount, err := getTransactionAmountFromDB(transactionID)
   if err != nil {
      return 0, fmt.Errorf("failed to get transaction %s: %w",
         transactionID, err)
   }
   return amount, nil
}
func getTransactionAmountFromDB(transactionID string) (float32, error) {
   // ...
   if err != nil {
      return 0, transientError{err: err}
   }
   // ...
}
```

Оборачивание ошибки вместо ее прямого возврата в виде transientError

Эта функция возвращает transientError

Если мы запустим этот код, он всегда вернет 400, независимо от типа ошибки, поэтому случай, когда должна быть выдана `TransientError`, никогда не появится. Как объяснить такое поведение?

Перед рефакторингом функция `getTransactionAmount` возвращала `transientError` (рис. 7.8). После же него `transientError` возвращается функцией `getTransactionAmountFromDB` (рис. 7.9).

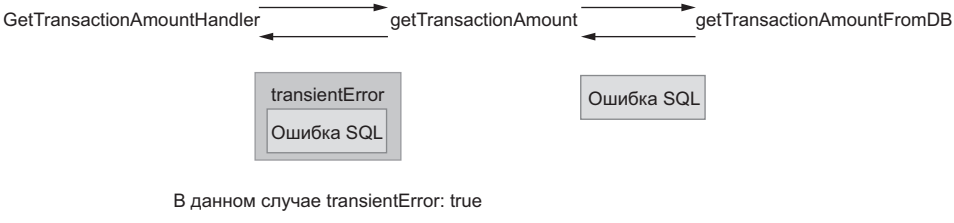


Рис. 7.8. Поскольку `transientError` возвращался функцией `getTransactionAmount`, в случае сбоя в обращении к базе данных его значение было `true`

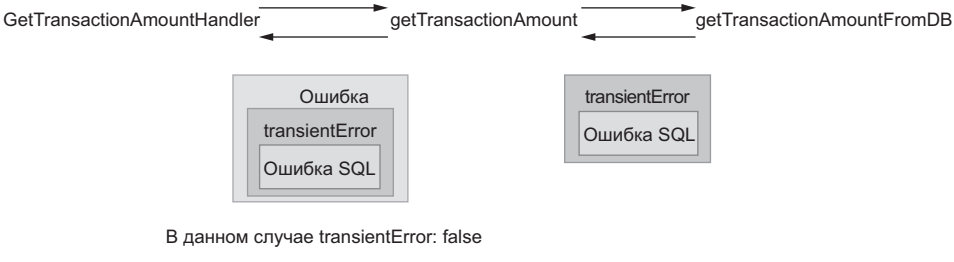


Рис. 7.9. Теперь `getTransactionAmount` возвращает обернутую ошибку, поэтому `transientError` принимает значение `false`

То, что возвращается в результате выполнения `getTransactionAmount`, не является непосредственно `transientError`: это обертка ошибки `transientError`. Поэтому в данном случае `transientError` принимает значение `false`.

Именно для этой цели в Go 1.13 появилась директива для оборачивания ошибок и способ проверки того, относится ли обернутая ошибка к некоторому определенному типу, с помощью `errors.As`. Эта функция рекурсивно разворачивает ошибку и возвращает `true`, если какая-то ошибка в цепочке соответствует ожидаемому типу.

Перепишем нашу реализацию вызова с помощью `errors.As`:

```
func handler(w http.ResponseWriter, r *http.Request) {
   // Получение ID транзакции
   amount, err := getTransactionAmount(transactionID)
   if err != nil {
      if errors.As(err, &transientError{}) {
```

Вызов errors.As с указателем на transientError в качестве аргумента

```
            http.Error(w, err.Error(),
               http.StatusServiceUnavailable)
         } else {
            http.Error(w, err.Error(),
               http.StatusBadRequest)
         }
         return
      }
      // Текст ответа
}
```

Возврат 503, если ошибка является transient

Возврат 400 в любом другом случае

В новой версии кода мы избавились от случаев с типом `switch` и теперь используем `errors.As`. Эта функция требует, чтобы второй аргумент (целевая ошибка) был указателем. В противном случае функция будет компилироваться, но вызывать панику во время выполнения. Независимо от того, является ли ошибка, проявляющаяся во время выполнения, непосредственно типа `transientError` или ошибкой, обертывающей `transientError`, функция `errors.As` возвращает значение `true`. Следовательно, наш обработчик вернет код состояния 503.

Если мы используем оборачивание ошибок в среде Go 1.13, следует использовать `errors.As`, чтобы проверить, относится ли ошибка к какому-то определенному типу. Независимо от того, возвращается ли ошибка непосредственно функцией, которую мы вызываем, или обернутой внутрь какой-то еще ошибки, `errors.As` сможет рекурсивно развернуть основную ошибку и посмотреть, относится ли одна из ошибок к какому-либо определенному типу.

Мы только что увидели, как сравнивать типы ошибок. Теперь поговорим о сравнении значений ошибок.

7.4. ОШИБКА #51: НЕТОЧНАЯ ПРОВЕРКА ЗНАЧЕНИЯ ОШИБКИ

Этот раздел аналогичен предыдущему, но здесь речь пойдет о сигнальных ошибках (значениях ошибок). Сначала мы определим понятие сигнальных ошибок. Затем увидим, как сравнить ошибку с каким-то значением.

Сигнальная ошибка (sentinel error) — это ошибка, определенная как глобальная переменная:

```
import "errors"
var ErrFoo = errors.New("foo")
```

В общем случае принято начинать с `Err`, за которым следует тип ошибки: здесь `ErrFoo`. Сигнальная ошибка сообщает об *ожидаемой* ошибке. Но что мы подразумеваем под этим? Обсудим это в контексте библиотеки SQL.

Мы собираемся создать метод `Query`, который позволит выполнять запрос к базе данных. Этот метод возвращает срез строк. Как поступить в случае, когда строки не найдены? Есть два варианта:

- Вернуть контрольное значение, например нулевой срез (вспомните о `strings.Index`, который возвращает контрольное значение –1, если подстрока отсутствует).
- Вернуть конкретную ошибку, которую клиент может проверить.

Рассмотрим второй подход: метод может возвращать конкретную ошибку, если не найдено никаких строк. Мы можем классифицировать это как *ожидаемую* ошибку, потому что допускается передача запроса, который не вернет какие-либо строки. И наоборот, такие ситуации, как проблемы с сетью и ошибки при подключении и опросе соединения, являются *непредвиденными* ошибками. Это означает не то, что мы не хотим обрабатывать непредвиденные ошибки, а то, что семантически эти ошибки несут в себе разные смыслы.

Если посмотреть на стандартную библиотеку, можно найти много примеров сигнальных ошибок:

- `sql.ErrNoRows` — возвращается, когда запрос не возвращает ни одной строки (что как раз соответствует нашему случаю).
- `io.EOF` — `io.Reader` возвращает эту ошибку, когда больше нет доступных входных данных.

Это общий принцип, стоящий за сигнальными ошибками. Они сообщают об ошибках, которые можно ожидать заранее и наличие которых, как предполагается, клиенты будут проверять. Поэтому в качестве общих указаний:

- Ожидаемые ошибки должны быть сделаны в виде значений (сигнальных ошибок): `var ErrFoo = errors.New("foo")`.
- Непредвиденные ошибки должны быть оформлены как типы ошибок: `type BarError struct { … }`, где `BarError` реализует интерфейс `error`.

Вернемся к обсуждению типичной ошибки. Как мы можем сравнивать ошибку с каким-то конкретным значением? Используя оператор `==`:

```
err := query()
if err != nil {
   if err == sql.ErrNoRows {  ◄—— Сравнение ошибки с переменной sql.ErrNoRows
      // ...
   } else {
      // ...
   }
}
```

Здесь мы вызываем функцию `query` и получаем ошибку. Проверка того, является ли она ошибкой `sql.ErrNoRows`, выполняется с помощью оператора `==`. Но как говорилось в предыдущем разделе, сигнальную ошибку также можно обернуть. Если `sql.ErrNoRows` обернут с использованием `fmt.Errorf` и директивы `%w`, `err == sql.ErrNoRows` всегда будет принимать значение `false`.

Go 1.13 дает на это ответ. Мы видели, как `errors.As` используется для проверки типа ошибки. В случаях со значениями ошибок мы можем использовать его аналог: `errors.Is`. Перепишем предыдущий пример:

```
err := query()
if err != nil {
   if errors.Is(err, sql.ErrNoRows) {
      // ...
   } else {
      // ...
   }
}
```

Использование `errors.Is` вместо оператора `==` позволяет выполнять сравнение, даже если ошибка обернута с помощью `%w`.

Таким образом, если в приложении мы используем обертку ошибок с помощью директивы `%w` и `fmt.Errorf`, проверка ошибки на ее равенство определенному значению должна выполняться с использованием `errors.Is`, а не `==`. И даже если сигнальная ошибка обернута, `errors.Is` может рекурсивно ее развернуть и сравнивать каждую ошибку в цепочке с заданным значением.

Теперь обсудим один из наиболее важных аспектов обработки ошибок: не обрабатывать ошибку дважды.

7.5. ОШИБКА #52: ДВОЙНАЯ ОБРАБОТКА ОШИБКИ

Многократная обработка программных сбоев — это оплошность, которую часто допускают разработчики, и это не какая-то особенность Go. Разберемся, почему это может стать проблемой и как эффективно обрабатывать ошибки.

Напишем код функции `GetRoute` для расчета маршрута, отталкиваясь от пар координат начальной и конечной точек. Предположим, что эта функция будет вызывать неэкспортированную функцию `getRoute`, содержащую бизнес-логику для расчета оптимального маршрута. Перед вызовом `getRoute` мы должны проверить координаты исходной и конечной точек, используя `validateCoordinates`. Мы также хотим, чтобы возможные ошибки регистрировались в журнале. Реализация может выглядеть так:

```
func GetRoute(srcLat, srcLng, dstLat, dstLng float32) (Route, error) {
   err := validateCoordinates(srcLat, srcLng)
   if err != nil {
      log.Println("failed to validate source coordinates")
      return Route{}, err
   }
   err = validateCoordinates(dstLat, dstLng)
   if err != nil {
      log.Println("failed to validate target coordinates")
      return Route{}, err
   }
   return getRoute(srcLat, srcLng, dstLat, dstLng)
}
func validateCoordinates(lat, lng float32) error {
   if lat > 90.0 || lat < -90.0 {
      log.Printf("invalid latitude: %f", lat)
      return fmt.Errorf("invalid latitude: %f", lat)
   }
   if lng > 180.0 || lng < -180.0 {
      log.Printf("invalid longitude: %f", lng)
      return fmt.Errorf("invalid longitude: %f", lng)
    }
   return nil
}
```

Регистрация и возврат ошибки

Регистрация и возврат ошибки

Регистрация и возврат ошибки

Регистрация и возврат ошибки

Что не так с этим кодом? Прежде всего, `validateCoordinates` будет весьма громоздким и неуклюжим из-за повторений сообщений об ошибках `invalid latitude` или `invalid longitude` одновременно в журнале и в виде возвращаемой ошибки. Кроме того, если мы запустим код, например, с недопустимым значением широты, то в журнал регистрации ошибок будут внесены следующие строки:

```
2021/06/01 20:35:12 invalid latitude: 200.000000
2021/06/01 20:35:12 failed to validate source coordinates
```

Почему наличие двух строк, относящихся к одной ошибке, — проблема? Это усложняет отладку. Например, если рассматриваемая функция вызывается несколько раз в режиме конкурентного выполнения, эти два сообщения могут не следовать в журналах одно за другим, что усложнит процесс отладки.

Ошибка должна быть обработана только один раз. Регистрация — это такая же обработка ошибки, как и возврат ошибки. Следовательно, мы должны либо регистрировать, либо возвращать ошибку, но никогда не делать и то и другое вместе.

Перепишем код функции так, чтобы она обрабатывала ошибки только один раз:

```
func GetRoute(srcLat, srcLng, dstLat, dstLng float32) (Route, error) {
   err := validateCoordinates(srcLat, srcLng)
   if err != nil {
      return Route{}, err  <-- Только возврат ошибки

   }
   err = validateCoordinates(dstLat, dstLng)
   if err != nil {
      return Route{}, err  <-- Только возврат ошибки
   }
   return getRoute(srcLat, srcLng, dstLat, dstLng)
}
func validateCoordinates(lat, lng float32) error {
   if lat > 90.0 || lat < -90.0 {
      return fmt.Errorf("invalid latitude: %f", lat)  <-- Только возврат ошибки
   }
   if lng > 180.0 || lng < -180.0 {
      return fmt.Errorf("invalid longitude: %f", lng)  <-- Только возврат ошибки
   }
   return nil
}
```

В этой версии каждая ошибка обрабатывается только один раз путем ее прямого возврата. Далее, предполагая, что вызывающая сторона `GetRoute` обрабатывает возможные ошибки, регистрируя их в журнале, код выведет сообщение в случае недопустимого значения широты:

```
2021/06/01 20:35:12 invalid latitude: 200.000000
```

Идеальна ли эта новая версия кода? Вовсе нет. Действительно, в случае задания недопустимой широты первая реализация приводила к двум записям в журнале. Но при этом мы знали, какой вызов `validateCoordinates` был ошибочным: по координатам начальной или конечной точки. Во втором случае эта информация теряется, поэтому нужно добавить к ошибке дополнительный контекст.

Модифицируем последнюю версию кода, используя оборачивание ошибок Go 1.13 (опускаем строки, определяющие функцию `validateCoordinates`, так как она остается неизменной):

```
func GetRoute(srcLat, srcLng, dstLat, dstLng float32) (Route, error) {
   err := validateCoordinates(srcLat, srcLng)
```

```
   if err != nil {
      return Route{},
         fmt.Errorf("failed to validate source coordinates: %w",
            err)  <-- Возврат обернутой ошибки
   }
   err = validateCoordinates(dstLat, dstLng)
   if err != nil {
      return Route{},
         fmt.Errorf("failed to validate target coordinates: %w",
            err)  <-- Возврат обернутой ошибки
   }
   return getRoute(srcLat, srcLng, dstLat, dstLng)
}
```

Каждая ошибка, возвращаемая `validateCoordinates`, теперь обернута для получения дополнительного контекста, а именно связана с координатами начальной или конечной точки. И если мы запустим выполнение новой версии, вот что будет зарегистрировано на вызывающей стороне в случае неверной широты начальной точки:

```
2021/06/01 20:35:12 failed to validate source coordinates:
    invalid latitude: 200.000000
```

В этой версии мы учли все случаи и свели их к одной записи в журнале ошибок без потери какой-либо ценной информации. Кроме того, каждая ошибка обрабатывается только один раз, что упрощает код, позволяя избегать повторяющихся сообщений об одних и тех же ошибках.

Обработка ошибки должна быть выполнена только один раз. Регистрация ошибки — это обработка ошибки. Следовательно, нужно либо зарегистрировать, либо вернуть ошибку. Делая это, мы упрощаем код и достигаем лучшего понимания ситуаций с ошибками. Использование обертки ошибок — наиболее удобный подход, поскольку позволяет указывать на исходную ошибку и добавлять к ошибке контекст.

В следующем разделе рассмотрим подходящий способ игнорирования ошибок в Go.

7.6. ОШИБКА #53: НЕ ВЫПОЛНЯТЬ ОБРАБОТКУ ОШИБКИ

В некоторых случаях требуется проигнорировать ошибку, возвращаемую функцией. В Go делать это нужно только одним способом.

Рассмотрим пример, где вызываем функцию `notify`, возвращающую единственный аргумент `error`. Допустим, что эта ошибка нас не интересует, поэтому мы намеренно опускаем какую-либо обработку ошибок:

```
func f() {
   // ...
   notify()  <-- Обработка ошибки пропущена
}
func notify() error {
   // ...
}
```

Поскольку мы хотим проигнорировать ошибку, то в этом примере просто вызываем функцию `notify`, не присваивая результат ее выполнения классической переменной `err`. С функциональной точки зрения в этом коде нет ничего плохого: он компилируется и работает, как и ожидалось.

Но с точки зрения сопровождаемости такой код может вызвать проблемы. Поставим себя на место человека, который видит его впервые. Он заметит, что `notify` возвращает ошибку, но эта ошибка не обрабатывается родительской функцией. Как догадаться, что пропуск обработки ошибки был преднамеренным? Как понять: предыдущий разработчик просто забыл сделать такую обработку или сделал это сознательно?

Поэтому когда мы хотим игнорировать ошибку в Go, есть только один способ отобразить это намерение в коде:

```
_ = notify()
```

Вместо того чтобы вообще не присваивать значение ошибки какой-либо переменной, мы присваиваем ее пустому идентификатору. С точки зрения компиляции и времени выполнения такой подход ничего не меняет по сравнению с первым вариантом. Но эта новая версия ясно показывает всем, что в данном случае ошибка нас никак не интересует.

Такой код также может сопровождаться комментарием, но не таким, как в этом примере:

```
// Игнорировать ошибку
_ = notify()
```

Этот комментарий дублирует то, что делает код, — этого следует избегать. Хорошей практикой будет написание комментария, указывающего на причину, по которой ошибка игнорируется, например:

```
// Доставка не более одного раза.
// Поэтому в случае ошибок некоторые из них допустимо просто пропускать.
_ = notify()
```

Ситуации игнорирования ошибок в Go должны быть исключительными. Во многих случаях лучше предпочесть их регистрацию, пусть даже на низком уровне журнала. Но если вы уверены, что ошибку можно и нужно игнорировать, то делайте это явно, присвоив ее пустому идентификатору. Так читатель кода поймет, что ошибку проигнорировали намеренно.

В последнем разделе этой главы обсудим, как обрабатывать ошибки, возвращаемые функцией `defer`.

7.7. ОШИБКА #54: НЕ ВЫПОЛНЯТЬ ОБРАБОТКУ ОШИБКИ ОПЕРАТОРА DEFER

Отказ от обработки ошибок в операторах `defer` — это оплошность, которую часто допускают разработчики. Поговорим, в чем тут проблема, и обсудим пути решения.

В примере ниже реализуем функцию для запроса к базе данных, чтобы получить баланс по идентификатору клиента (ID). Будем использовать `database/sql` и метод `Query`.

> **ПРИМЕЧАНИЕ** Не будем вдаваться в работу этого пакета. Поговорим об этом в разделе, посвященном разбору ошибки #78 (типичные ошибки, связанные с SQL).

Вот возможная реализация (фокусируемся на самом запросе, а не на разборе результатов):

```
const query = "..."
func getBalance(db *sql.DB, clientID string) (
   float32, error) {
   rows, err := db.Query(query, clientID)
   if err != nil {
      return 0, err
   }
   defer rows.Close()  ◄—— Откладывание вызова rows.Close()
   // Используется rows
}
```

`rows` — это тип `*sql.Rows`. Он реализует интерфейс `Closer`:

```
type Closer interface {
   Close() error
}
```

Этот интерфейс содержит единственный метод `Close`, который возвращает ошибку (рассмотрим эту тему при разборе ошибки #79 (не закрывать временные ресурсы)). В предыдущем разделе я говорил, что ошибки всегда должны обрабатываться. Но в этом случае ошибка, возвращаемая вызовом `defer`, игнорируется:

```
defer rows.Close()
```

Если мы не хотим обрабатывать ошибку, нужно проигнорировать ее в явном виде, используя пустой идентификатор:

```
defer func() { _ = rows.Close() }()
```

Эта версия более многословна, но с точки зрения удобства сопровождения она лучше, поскольку мы явно указываем, что игнорируем ошибку.

Однако в данном случае, вместо того чтобы слепо игнорировать все ошибки от отложенных вызовов, следует задуматься, нет ли другого, более правильного подхода. Здесь вызов `Close()` возвращает ошибку, если не удается освободить соединение с БД из пула. Следовательно, игнорирование этой ошибки — не лучший вариант. Более грамотным подходом будет регистрация сообщения:

```
defer func() {
   err := rows.Close()
   if err != nil {
      log.Printf("failed to close rows: %v", err)
   }
}()
```

Теперь, если закрытие `rows` не проходит, код зарегистрирует сообщение об ошибке, чтобы мы знали о ней.

А если вместо обработки ошибки мы захотим передать ее клиенту, который вызывает функцию `getBalance`, чтобы он решил, как ее обработать?

```
defer func() {
   err := rows.Close()
   if err != nil {
      return err
   }
}()
```

Этот код не скомпилируется. Действительно, оператор `return` связан с анонимной функцией `func()`, а не с `getBalance`.

Если мы хотим связать ошибку, возвращаемую `getBalance`, с ошибкой, обнаруженной в вызове `defer`, надо применить именованные параметры результата. Вот первая версия реализации требуемых действий:

```
func getBalance(db *sql.DB, clientID string) (
   balance float32, err error) {
   rows, err := db.Query(query, clientID)
   if err != nil {
      return 0, err
   }
   defer func() {
      err = rows.Close()
   }()
   if rows.Next() {
      err := rows.Scan(&balance)
      if err != nil {
         return 0, err
      }
      return balance, nil
   }
   // ...
}
```

Присвоение значения ошибки выходному именованному параметру

Как только переменная `rows` будет создана, мы откладываем вызов `rows.Close()` в анонимной функции. Эта функция присваивает переменной `err`, которая инициализируется с использованием именованных параметров результата, значение ошибки.

Код может выглядеть нормально, но в нем есть проблема. Если `rows.Scan` возвращает ошибку, `rows.Close` выполняется в любом случае; но поскольку этот вызов переопределяет ошибку, возвращаемую `getBalance`, вместо ошибки мы можем вернуть ошибку `nil` при успешном возврате `rows.Close`. Другими словами, если вызов `db.Query` завершится успешно (первая строка функции), ошибка, возвращаемая `getBalance`, всегда будет соответствовать ошибке, возвращаемой `rows.Close`, а это не то, что нужно.

Логика, которую нужно реализовать, не так уж и проста.

- Если `rows.Scan` выполнена успешно, то:
  - если `rows.Close` также успешно выполняется, то ошибка не возвращается;
  - если `rows.Close` завершается с ошибкой, то возвращается эта ошибка.

Но если `rows.Scan` не проходит, то логика будет несколько сложнее, потому что, скорее всего, придется обрабатывать две ошибки.

- Если `rows.Scan` не проходит, то:
  - если `rows.Close` выполняется успешно, то возвращается ошибка из `rows.Scan`;
  - если `rows.Close` завершается с ошибкой... тогда что?

Что делать, если и `rows.Scan`, и `rows.Close` не проходят? Есть несколько вариантов. Например, мы можем вернуть какую-то пользовательскую ошибку, которая будет содержать в себе две ошибки. Другой вариант, который мы и реализуем, — это возвращать ошибку `rows.Scan`, но регистрировать в журнале ошибку `rows.Close`. Вот окончательная реализация этой анонимной функции:

```
defer func() {
   closeErr := rows.Close()  <-- Присвоение ошибки из rows.Close другой переменной
   if err != nil {  <-- Если ошибка уже не nil, определяем приоритет
      if closeErr != nil {
         log.Printf("failed to close rows: %v", err)
      }
      return
   }
   err = closeErr  <-- В противном случае возвращаем closeErr
}()
```

Ошибка `rows.Close` присваивается другой переменной: `closeErr`. Прежде чем присвоить ее `err`, мы проверяем, отличается ли `err` от `nil`. Если да, значит, `getBalance` уже вернула ошибку, поэтому мы заносим `err` в журнал и возвращаем существующую ошибку.

Ошибки всегда должны обрабатываться. В случае ошибок, возвращаемых `defer`, как минимум нужно явно их проигнорировать. Если этого недостаточно, следует обработать ошибку напрямую, зарегистрировав ее в журнале или передав вызывающей стороне, как показано в этом разделе.

ИТОГИ

- Использование паники в Go — это вариант борьбы с ошибками. Но такой вариант следует применять с осторожностью только в самых крайних случаях: например, чтобы сигнализировать об ошибке программиста или когда невозможно загрузить обязательную зависимость.

- Оборачивание ошибки позволяет пометить ошибку и/или дополнить ее каким-то контекстом. Однако это создает потенциальную связанность, поскольку делает исходную ошибку доступной для вызывающей функции. Если вы хотите предотвратить это, не используйте оборачивание ошибок.
- Если вы используете оборачивание ошибок с помощью директивы `%w` и `fmt.Errorf`, сравнение ошибки с типом или значением должно выполняться с помощью `errors.As` или `errors.Is` соответственно. В противном случае, если возвращаемая ошибка, которую вы хотите проверить, обернута, она не пройдет проверку.
- Чтобы передать информацию об ожидаемой ошибке, используйте сигнальные ошибки (значения ошибок). Непредвиденная ошибка должна быть определенного типа.
- В большинстве ситуаций ошибку следует обрабатывать только один раз. Регистрация ошибки — это тоже обработка. Поэтому выбирайте между ведением журнала или возвратом ошибки. Во многих случаях оборачивание ошибок — это хорошее решение, поскольку позволяет снабдить ошибку дополнительным контекстом, а также вернуть исходную ошибку.
- Игнорирование ошибки, будь то во время вызова какой-либо функции или в функции `defer`, должно выполняться в явном виде с использованием пустого идентификатора. Иначе читатели вашего кода запутаются и не поймут, игнорируете вы ошибку намеренно или случайно.
- Не игнорируйте ошибки, возвращаемые функцией `defer`. Лучше обработать их напрямую либо передать их вызывающей функции — в зависимости от контекста. Если вы все же хотите проигнорировать ошибку, используйте пустой идентификатор.

8 Конкурентность: основы

В этой главе:

- ✓ Понимание конкурентности и параллелизма
- ✓ Почему конкурентность не всегда быстрее
- ✓ Влияние рабочих нагрузок, увязанных с процессором (CPU-bound) и системой ввода/вывода (I/O-bound)
- ✓ Каналы и мьютексы — особенности использования
- ✓ Понимание различий между гонкой данных и состоянием гонки
- ✓ Работа с контекстами в Go

В последние десятилетия производители процессоров перестали концентрироваться только на вопросах повышения их тактовой частоты. Современные процессоры имеют несколько ядер и отличаются гиперпоточностью (hyperthreading — наличие нескольких логических ядер в одном физическом ядре). Для эффективного использования возможностей этих архитектур разработчики ПО должны хорошо понимать особенности конкурентного выполнения программ.

Несмотря на то что в Go есть простые примитивы, их наличие не обязательно означает, что написание конкурентного кода — это легко. В данной главе обсудим основные понятия, связанные с конкурентностью, а затем сосредоточимся на ее практических аспектах.

8.1. ОШИБКА #55: ПУТАТЬ КОНКУРЕНТНОСТЬ И ПАРАЛЛЕЛИЗМ

Даже после многих лет опыта конкурентного программирования разработчики могут недостаточно четко понимать разницу между конкурентностью и параллелизмом. Прежде чем углубляться в эти темы в контексте Go, попробуем прояснить эти концепции в целом. Проиллюстрируем их на примере из реальной жизни: работе кофейни.

В этой кофейне один бариста, он отвечает за прием заказов и их приготовление с помощью одной кофемашины. Клиенты делают заказы, а затем ждут кофе (рис. 8.1).

Рис. 8.1. Простая кофейня

Если бариста трудно обслуживать всех клиентов быстро и качественно, а заведение хочет ускорить общий процесс, то можно нанять второго бариста и купить вторую кофемашину. Клиент в очереди будет ожидать, когда кто-то из двух сотрудников сможет его обслужить (рис. 8.2).

В новом процессе каждая часть системы независима. Кофейня должна обслуживать потребителей в два раза быстрее. Это — *параллельная* реализация системы обслуживания.

Если владелец хочет масштабировать свой кофейный бизнес, он может привлекать новых сотрудников и устанавливать дополнительные кофемашины. Но это не единственный возможный вариант. Другой подход состоит в том, чтобы разделить обязанности сотрудников: например, один отвечает за прием

заказов, а другой — за помол кофейных зерен, из которых он сварит кофе в единственной кофемашине. Кроме того, чтобы не заставлять клиентов ждать в общей очереди до тех пор, пока не обслужат текущего клиента, можно ввести еще одну очередь — для клиентов, ожидающих выполнения принятых заказов (как в Starbucks) (рис. 8.3).

Рис. 8.2. Кофейня: дублирование

Рис. 8.3. Разделение обязанностей сотрудников кофейни

В такой новой системе обслуживания клиентов мы не совершаем действия параллельно. Вместо этого пересматривается общая структура: мы разбиваем одну роль на две и вводим еще одну очередь. В отличие от параллелизма, смысл которого в том, что одни и те же действия совершаются одновременно, *конкурентность* связана со структурой.

Предполагая, что один поток соответствует бариста, принимающему заказы, а другой — кофемашине, мы ввели третий поток — помола кофейных зерен. Каждый поток независим, но все действия внутри него должны координироваться

с другими. Поток принимающего заказы сотрудника должен сообщать, какой объем кофейных зерен нужно смолоть. А поток кофемолки должен обмениваться сообщениями с потоком кофеварки.

А если нужно увеличить пропускную способность, обслуживая все больше клиентов в единицу времени? Поскольку помол зерен занимает больше времени, чем прием заказов, возможное изменение может состоять в том, чтобы нанять еще одного сотрудника на помол кофе (рис. 8.4).

Рис. 8.4. Нанимаем еще одного сотрудника для помола кофейных зерен

Структура кофейни остается прежней. Это по-прежнему трехэтапная схема: принять заказ, смолоть кофе и сварить кофе. Следовательно, с точки зрения конкурентности нет никаких изменений. Но мы вернулись к добавлению элемента параллелизма, пусть для одного конкретного шага — этапа подготовки к окончательному выполнению заказа.

Предположим, что узкое место, замедляющее весь процесс, — это кофеварка. Использование только одной кофемашины приводит к конкуренции между разными потоками помола кофе, поскольку они оба ожидают, когда поток кофемашины станет для них доступным. Как можно решить эту задачу? Добавить дополнительные потоки кофемашины (рис. 8.5).

Вместо одной кофемашины мы увеличили соответствующий уровень параллелизма, установив больше машин. И снова структура не изменилась: она остается трехступенчатой. Но пропускная способность всей системы должна увеличиться, потому что уровень конкуренции между потоками помола кофе уменьшится.

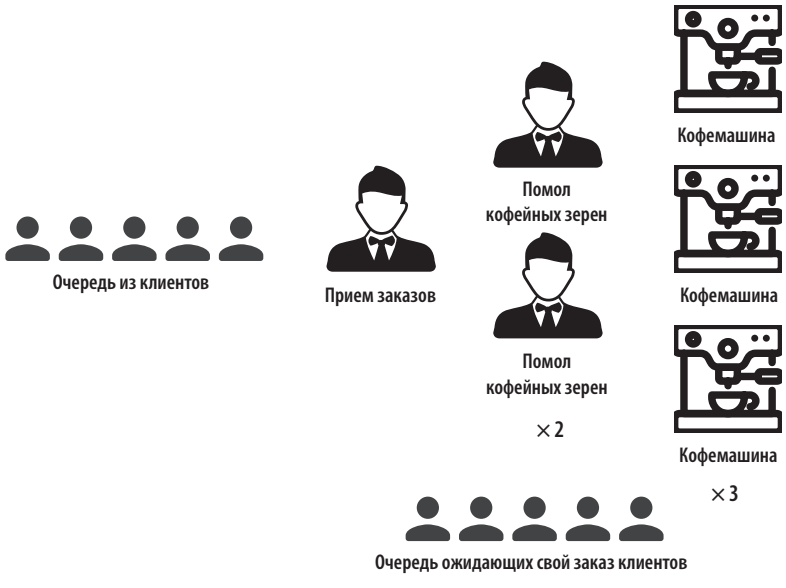


Рис. 8.5. Установка большего числа кофемашин

В таком дизайне мы можем заметить нечто очень важное: *конкурентность обеспечивает возможность параллелизма.* Конкурентность подталкивает к созданию структуры для решения всей проблемы через ее разбиение на части, действия внутри которых можно распараллелить.

> *Конкурентность — это о работе с большим количеством вещей одновременно. Параллелизм — это о выполнении множества дел одновременно.*
>
> *Роб Пайк (Rob Pike)*

Итак, конкурентность и параллелизм — это разные вещи. Конкурентность связана со структурой: мы можем превратить последовательную реализацию в параллельную, вводя различные шаги, которые могут совершаться в рамках отдельных параллельно выполняемых потоков. А параллелизм касается выполнения: можно использовать его на уровне шагов, добавляя больше параллельных потоков. Понимание этих двух концепций очень важно для качественной разработки на Go.

Теперь поговорим о распространенной ошибке: убеждении, что конкурентность — это всегда правильный путь.

8.2. ОШИБКА #56: ПОЛАГАТЬ, ЧТО КОНКУРЕНТНОСТЬ БЫСТРЕЕ

Многие разработчики ошибочно полагают, что конкурентное решение всегда быстрее, чем последовательное. И это в корне неверно. Общая производительность выбранного решения зависит от многих факторов, таких как эффективность структуры (конкурентность), от того, какие части могут выполняться параллельно, и от уровня конкуренции между вычислительными блоками. Этот раздел освежит некоторые фундаментальные знания о конкурентности в Go. Мы увидим пример, где конкурентное решение не обязательно будет более быстрым.

8.2.1. Планирование в Go

Поток — это наименьшая единица обработки, которую может выполнять операционная система (ОС). Если процесс хочет выполнить несколько действий одновременно, он запускает несколько потоков. Потоки могут быть:

- *конкурентными*: два или более потока могут запускаться, выполняться и завершаться в перекрывающиеся периоды времени — как поток бариста и поток кофемашины в предыдущем разделе;
- *параллельными*: одна и та же задача может выполняться несколько раз одновременно, как потоки нескольких бариста.

ОС отвечает за оптимальное планирование процессов потоков, чтобы:

- все потоки могли использовать процессорное время, не слишком долго ожидая своей очереди;
- рабочая нагрузка распределялась между различными ядрами процессора максимально равномерно.

> **ПРИМЕЧАНИЕ** Термин *поток* (thread) на уровне CPU может иметь другое значение. Каждое физическое ядро может состоять из нескольких логических ядер (концепция гиперпоточности — hyperthreading), которые также называются потоками (thread). В этом разделе слово «поток» используется тогда, когда имеется в виду единица обработки, а не логическое ядро.

Ядро процессора выполняет различные потоки. Когда оно переключается с одного потока на другой, выполняется операция под названием *переключение контекста*. Активный поток, потребляющий циклы процессора и находящийся

в состоянии *выполнения* (executing state), переходит в состояние *готовности выполнения* (runnable state), это означает, что он готов к запуску и ожидает доступное ядро. Переключение контекста считается дорогостоящей операцией, поскольку ОС нужно сохранить текущее состояние выполнения потока перед переключением (например, текущие значения регистров).

В Go нельзя создавать потоки напрямую, но есть возможность создавать горутины, которые можно рассматривать как потоки на уровне приложения. При этом если поток ОС управляется самой ОС, то горутина управляется средой выполнения Go. Кроме того, по сравнению с потоком ОС горутина занимает меньше места в памяти: 2 Кбайт из Go 1.4. Размер потока ОС зависит от ОС, но, например, в Linux/x86-32 размер по умолчанию составляет 2 Мбайт (см. http://mng.bz/DgMw). Меньший размер делает переключение контекста более быстрым.

> **ПРИМЕЧАНИЕ** Переключение контекста горутины по сравнению с таковым для потока происходит примерно на 80–90 % быстрее в зависимости от архитектуры.

Теперь обсудим, как работает планировщик Go, и поймем, как обрабатываются горутины. Здесь используется следующая терминология (см. http://mng.bz/N611):

- G — горутина
- M — поток ОС (M означает machine, *машина*)
- P — ядро CPU (P означает processor, *процессор*)

Каждый поток ОС (M) назначается ядру CPU (P) планировщиком ОС. Затем каждая горутина (G) запускается на M. Переменная `GOMAXPROCS` определяет предел количества потоков M, отвечающих за одновременное исполнение кода пользовательского уровня. Но если поток заблокирован в системном вызове (например, ввода/вывода), планировщик может запустить больше потоков M. Начиная с Go 1.5, `GOMAXPROCS` по умолчанию равен количеству доступных ядер CPU.

Горутина имеет более простой жизненный цикл, чем поток ОС. Она может совершать одно из следующих действий:

- *исполнение (executing)* — на M назначено исполнение горутины, и входящие в нее инструкции выполняются;
- *готовность к выполнению (runnable)* — горутина ожидает перехода в состояние выполнения;

- *ожидание (waiting)* — горутина остановлена и ожидает завершения чего-либо, например системного вызова или операции синхронизации (например, получения мьютекса).

Остался последний шаг к пониманию того, как в Go реализуется планирование: что происходит, когда горутина создана, но еще не может быть выполнена, например, все остальные M уже выполняют G. Что тогда будет делать среда выполнения Go? Ответ: поставит в очередь. Среда выполнения Go обрабатывает два типа очередей: по одной локальной очереди для каждого P и глобальная очередь, которая ориентирована на выполнение на всех P.

На рис. 8.6 показана заданная ситуация планирования на четырехъядерной машине с `GOMAXPROCS`, равным 4. Частями являются логические ядра (P), горутины (G), потоки ОС (M), локальные очереди и глобальная очередь.

Заметьте, мы видим пять M, тогда как для `GOMAXPROCS` установлено значение 4. Но как я уже сказал, при необходимости среда выполнения Go может создать больше потоков ОС, чем значение `GOMAXPROCS`.

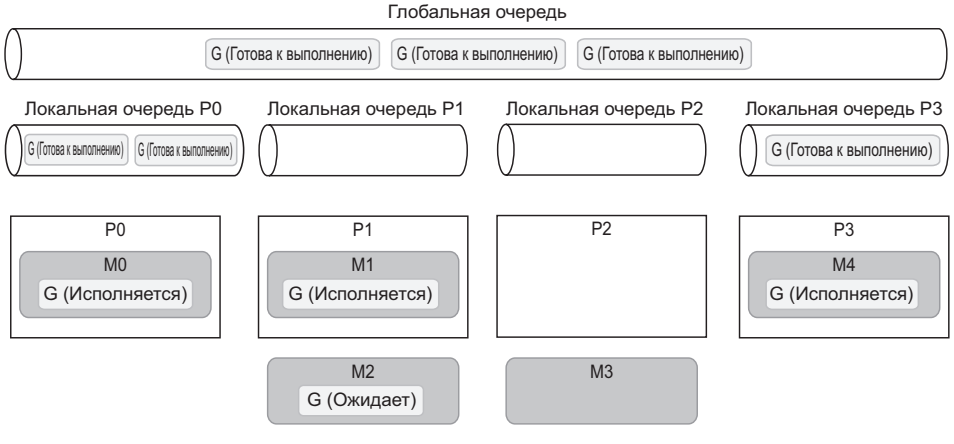


Рис. 8.6. Пример текущего состояния приложения Go, выполняемого на четырехъядерной машине. Горутины, которые не находятся в состоянии выполнения, либо готовы к выполнению (ожидают в очереди на выполнение), либо ожидают (блокирующую операцию)

В настоящее время P0, P1 и P3 заняты выполнением потоков среды Go. Но P2 при этом простаивает, так как M3 отключен от P2, и никакой горутины для выполнения нет. Это не очень хорошая ситуация, потому что шесть готовых

к запуску горутин ожидают выполнения, некоторые в глобальной очереди, а некоторые в других локальных очередях. Как среда выполнения Go справится с этой ситуацией? Вот реализация планирования в псевдокоде (см. http://mng.bz/lxY8):

```
runtime.schedule() {
    // Только 1/61 от всего времени, проверка глобальной очереди выполнения
    // на наличие G.
    // Если ничего не найдено, проверка локальной очереди.
    // Если ничего не найдено,
    //      попытка украсть у других P.
    //      Если опять ничего не найдено, проверка глобальной очереди
            готовых к выполнению.
    //      Если ничего не найдено, опрос сети.
}
```

При каждом шестьдесят первом выполнении планировщик Go будет проверять, доступны ли горутины из глобальной очереди. Если нет, он проверит свою локальную очередь. Если же и глобальная и локальная очереди пусты, планировщик может перехватить горутины из других локальных очередей. Этот принцип в планировании называется *кражей задач* (work stealing), и он позволяет недостаточно загруженному процессору активно искать горутины, ожидающие своего выполнения на другом процессоре, и *украсть* некоторые из них.

Обратите внимание: до версии Go 1.14 планировщик был кооперативным, что означало, что горутина могла быть контекстно отключена от потока только в определенных случаях блокировки (например, отправка или получение канала, операции ввода/вывода, ожидание получения мьютекса). Начиная с Go 1.14, планировщик стал вытесняющим (preemptible): когда горутина выполняется в течение некоторого заданного отрезка времени (10 мс), она будет помечена как вытесняемая и может быть контекстно отключена и заменена другой горутиной. Это позволяет использовать процессор в тот период, когда выполняется какое-то длительное задание, а также для выполнения и других задач.

Теперь, когда мы понимаем основы планирования в Go, рассмотрим пример: параллельная реализация сортировки слиянием.

8.2.2. Параллельная сортировка слиянием

Рассмотрим, как работает алгоритм сортировки слиянием. Реализуем параллельную версию. Цель состоит не в том, чтобы написать код, который будет наиболее производительным, а в том, чтобы показать, почему конкурентность не всегда быстрее.

Суть алгоритма сортировки слиянием состоит в многократном разбиении какого-то списка на два подсписка до тех пор, пока каждый из них не будет состоять из одного элемента. Затем эти подсписки объединяются таким образом, что в результате получается отсортированный список (рис. 8.7). Каждая операция разделения разбивает список на два подсписка, тогда как операция слияния объединяет два подсписка в один отсортированный список.

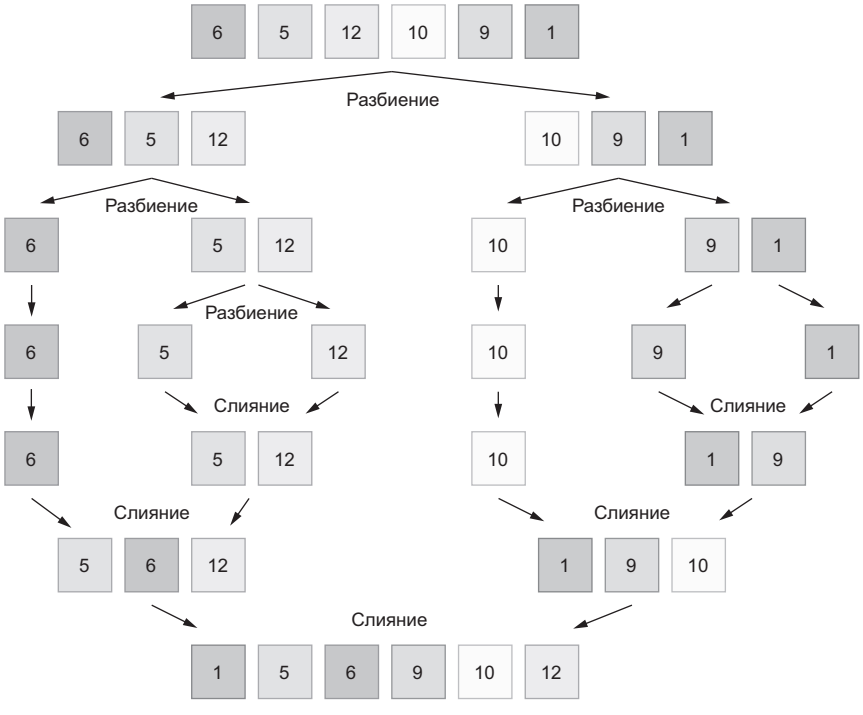


Рис. 8.7. Применение алгоритма сортировки слиянием многократно разбивает каждый список на два подсписка. Затем применяется операция слияния, чтобы полученный список оказался отсортированным

Вот последовательная реализация этого алгоритма. Код сокращен, так как здесь это не главное:

```
func sequentialMergesort(s []int) {
   if len(s) <= 1 {
      return
   }
   middle := len(s) / 2
   sequentialMergesort(s[:middle])   ◄—— Первая половина
```

```
   sequentialMergesort(s[middle:])  ◄──  Вторая половина
   merge(s, middle)  ◄──  Объединение двух половин
}
func merge(s []int, middle int) {
   // ...
}
```

Этот алгоритм имеет структуру, которая делает его открытым для конкурентности. Поскольку каждая операция `sequenceMergesort` работает с независимым набором данных, которые не нужно полностью копировать (здесь независимое представление основного массива с использованием нарезки), мы можем распределить эту рабочую нагрузку между ядрами CPU, запустив каждую операцию `sequenceMergesort` в другой горутине. Вот код первой параллельной реализации:

```
func parallelMergesortV1(s []int) {
   if len(s) <= 1 {
      return
   }
   middle := len(s) / 2
   var wg sync.WaitGroup
   wg.Add(2)
   go func() {  ◄──  Запускается первая половина работы в горутине
      defer wg.Done()
      parallelMergesortV1(s[:middle])
   }()
   go func() {  ◄──  Запускается вторая половина работы в горутине
      defer wg.Done()
      parallelMergesortV1(s[middle:])
   }()
   wg.Wait()
   merge(s, middle)  ◄──  Объединение этих половин
}
```

В этой версии кода каждая половина рабочей нагрузки обрабатывается отдельной горутиной. Родительская горутина ожидает завершения выполнения обеих частей с помощью оператора `sync.WaitGroup`. Таким образом, мы вызываем метод `Wait` перед операцией слияния.

> **ПРИМЕЧАНИЕ** Если вы еще не знакомы с оператором `sync.WaitGroup`, то имейте в виду, что мы рассмотрим его подробнее в разделе, посвященном разбору ошибки #71 (неправильно использовать `sync.WaitGroup`). Вкратце: он позволяет дождаться завершения *n* операций (обычно горутины, как в предыдущем примере).

Теперь у нас есть параллельная версия алгоритма сортировки слиянием. И если мы запустим бенчмарк для сравнения этой версии с последовательной, первая

версия должна быть быстрее. Так ведь? Запустим его на четырехъядерной машине с 10 000 элементов:

```
Benchmark_sequentialMergesort-4 2278993555 ns/op
Benchmark_parallelMergesortV1-4 17525998709 ns/op
```

Удивительно, но параллельная версия оказалась на порядок медленнее. Почему? Как получилось, что параллельная версия, в которой рабочая нагрузка распределяется между четырьмя ядрами, работает медленнее, чем последовательная версия, работающая на одной машине? Давайте проанализируем.

Если у нас есть срез из 1024 элементов, родительская горутина запускает две другие горутины, каждая из которых отвечает за обработку соответствующей половины, состоящей из 512 элементов. Каждая из этих горутин запустит две новые горутины, отвечающие за обработку 256 элементов, затем 128, и так далее, пока мы не запустим горутину, вычисляющую один элемент.

Если рабочая нагрузка, которую мы хотим распараллелить, слишком мала, то есть мы собираемся провести соответствующие ей вычисления слишком быстро, преимущество распределения задания по ядрам теряется: время, необходимое для создания горутины и ее выполнения планировщиком, слишком велико по сравнению с прямым слиянием небольшого количества элементов в текущей горутине. Хотя горутины леговесны и запускаются быстрее, чем потоки, мы все же можем столкнуться со случаями, когда рабочая нагрузка слишком мала.

> **ПРИМЕЧАНИЕ** О том, как выявить неудачное распараллеливание, поговорим при разборе ошибки #98 (не использовать диагностический инструментарий Go).

Какой можно сделать вывод? Алгоритм сортировки слиянием нельзя распараллелить? Погодите, не будем спешить.

Используем другой подход. Поскольку применение новой горутины для слияния небольшого количества элементов неэффективно, определим порог. Он будет указывать на то, какое минимальное число элементов должна содержать половина списка, чтобы обрабатывать ее параллельно имело смысл. Если количество элементов в половине меньше этого значения, будем обрабатывать ее последовательно. Вот новая версия кода:

```
const max = 2048  ◄——  Задание величины порога
func parallelMergesortV2(s []int) {
   if len(s) <= 1 {
```

```
      return
   }
   if len(s) <= max {
      sequentialMergesort(s)  ◄—— Вызов последовательной версии
   } else {  ◄—— Если порог превышен, то выполняется параллельная версия
      middle := len(s) / 2
      var wg sync.WaitGroup
      wg.Add(2)
      go func() {
         defer wg.Done()
         parallelMergesortV2(s[:middle])
      }()
      go func() {
         defer wg.Done()
         parallelMergesortV2(s[middle:])
      }()
      wg.Wait()
      merge(s, middle)
   }
}
```

Если количество элементов в срезе `s` меньше `max`, мы вызываем последовательную версию. В противном случае продолжаем вызывать параллельную реализацию. Влияет ли такой подход на результат? Да, влияет:

```
Benchmark_sequentialMergesort-4 2278993555 ns/op
Benchmark_parallelMergesortV1-4 17525998709 ns/op
Benchmark_parallelMergesortV2-4 1313010260 ns/op
```

Вторая версия параллельной реализации более чем на 40 % быстрее последовательной, и все это благодаря идее порога, определяющего эффективность параллельной обработки.

> **ПРИМЕЧАНИЕ** Почему в качестве порогового значения я взял 2048? Потому что это было оптимальным значением для этой конкретной рабочей нагрузки на моей машине. В общем, такие магические числа следует тщательно определять с помощью бенчмарков (работающих в среде выполнения, аналогичной той, которая будет использована при реальном применении программы на практике). Интересно отметить, что запуск одного и того же алгоритма на языке без концепции горутин влияет на это значение. Например, выполнение того же примера на Java с использованием потоков означает, что оптимальное значение будет ближе к 8192. Это иллюстрирует тот факт, насколько горутины более эффективны, чем потоки.

В этой главе мы рассмотрели фундаментальные концепции планирования в Go: различия между потоком и горутиной и то, как среда выполнения Go планирует

выполнение горутин. Используя пример с параллельной сортировкой слиянием, мы показали, что конкурентность не всегда приводит к более быстрому выполнению кода. Как мы увидели, развертывание механизма горутин для обработки незначительных рабочих нагрузок (на примере слияния относительно небольшого набора элементов) сводит на нет преимущества, которые мы могли бы получить от параллелизма.

Каков вывод? Помните, что конкурентность не всегда быстрее и ее не следует по умолчанию рассматривать как способ решения всех проблем. Во-первых, она все усложняет. Во-вторых, современные CPU стали невероятно эффективны при выполнении последовательного и предсказуемого кода. Например, суперскалярный процессор может с высокой эффективностью распараллелить выполнение инструкций на одном ядре.

Означает ли это, что не следует использовать конкурентность? Конечно нет. Но важно помнить о вышеприведенных рассуждениях. Если мы не уверены, что параллельная версия будет быстрее, то правильным подходом будет начать с простой последовательной версии и отталкиваться от нее, используя, например, профилирование (см. разбор ошибки #98 (не использовать диагностический инструментарий Go) и бенчмарки (см. разбор ошибки #89 (писать неточные бенчмарки). Это может быть единственным способом убедиться, что конкурентность того стоит.

В следующем разделе узнаем, когда следует использовать каналы, а когда мьютексы.

8.3. ОШИБКА #57: ПУТАТЬСЯ В ТОМ, КОГДА ИСПОЛЬЗОВАТЬ КАНАЛЫ, А КОГДА МЬЮТЕКСЫ

При возникновении проблемы конкурентности не всегда ясно, как лучше реализовать решение: с помощью каналов либо мьютексов. Поскольку Go поощряет совместное использование памяти при обмене данными, одной из ошибок может быть постоянное принудительное использование каналов, независимо от ситуации. Но следует рассматривать оба варианта как взаимодополняющие. В этом разделе поговорим, когда мы должны отдавать предпочтение одному варианту перед другим. Цель главы в том, чтобы дать общие рекомендации, которые помогут принимать верные решения.

Для начала напомню о каналах в Go: каналы — это механизм коммуникации. Внутри себя канал — это некий трубопровод, который мы можем использовать

для отправки и получения значений и который позволяет соединять конкурентные горутины. Канал может быть:

- *небуферизованным*: отправляющая горутина блокируется до тех пор, пока получающая горутина не будет готова;
- *буферизованным*: отправляющая горутина блокируется только тогда, когда буфер оказывается полностью заполненным.

Вернемся к изначальному вопросу. Когда нужно использовать каналы, а когда мьютексы? Возьмем пример на рис. 8.8. В нем есть три разные горутины, которые соотносятся друг с другом следующим образом:

- G1 и G2 — параллельные горутины. Это могут быть две горутины, выполняющие одну и ту же функцию, которая продолжает получать сообщения из канала, или две горутины, одновременно выполняющие один и тот же обработчик HTTP.
- С другой стороны, G1 и G3 являются конкурентными горутинами — так же, как и G2 и G3. Все горутины — это часть общей конкурентной структуры, но G1 и G2 выполняют первый шаг, а G3 — следующий шаг.

Рис. 8.8. Горутины G1 и G2 параллельны, тогда как G2 и G3 конкурентны

Как правило, параллельные горутины должны *синхронизироваться*, например, когда им нужно получить доступ или изменить общий ресурс, такой как срез. Синхронизация производится с помощью мьютексов, но не с любыми типами каналов (не с буферизованными каналами). Следовательно, в общем случае синхронизация между параллельными горутинами должна достигаться с помощью мьютексов.

И наоборот, в общем случае конкурентные горутины должны *координировать* и *оркестровать* свои действия. Например, если G3 должна объединить результаты как из G1, так и из G2, то G1 и G2 должны сообщить G3, что новый промежуточный результат доступен. Эта координация относится к коммуникации, следовательно, должна осуществляться с помощью каналов.

Что касается конкурентных горутин, то можно представить себе случай, когда мы захотим передать право собственности на ресурс с одного шага (G1 и G2) на другой (G3), например, если G1 и G2 расширяют возможности какого-то общего ресурса и в какой-то момент мы посчитаем их работу завершенной. В этом случае мы должны использовать каналы, чтобы сигнализировать о том, что конкретный ресурс готов, и обработать передачу права собственности.

Мьютексы и каналы имеют разную семантику. Всякий раз, когда нужно разделить состояние или получить доступ к общему ресурсу, мьютексы обеспечивают эксклюзивный доступ к этому ресурсу. И наоборот, канал — это механизм для передачи сигналов с данными или без них (`chan struct{}` или нет). Координация или передача права собственности должна осуществляться по каналам. Важно знать, являются ли горутины параллельными или конкурентными, потому что обычно для параллельных горутин нужны мьютексы, а для конкурентных каналы.

Давайте обсудим широко распространенную проблему, связанную с конкурентностью: проблему гонки.

8.4. ОШИБКА #58: НЕ ПОНИМАТЬ ПРОБЛЕМ ГОНКИ

Проблемы гонки — это одни из самых сложных и коварных ошибок, с которыми сталкиваются программисты. Мы должны понимать некоторые важные вопросы: гонка данных и состояние гонки, их возможные последствия, а также способы их избежать. Сначала обсудим гонку данных и состояние гонки в сравнении друг с другом, затем изучим модель памяти Go и поговорим, почему все это столь важно.

8.4.1. Гонка данных и состояние гонки

Сосредоточимся на гонке данных (data race), которая происходит, когда две или более горутины одновременно обращаются к одной и той же ячейке памяти и по крайней мере одна из них выполняет запись в эту ячейку. Вот пример, когда две горутины увеличивают значение общей переменной:

```
i := 0
go func() {
   i++  <-- Увеличение значения i
}()
go func() {
   i++
}()
```

Если запустить этот код с использованием детектора гонок Go (опция `-race`), он предупредит о том, что произошла гонка данных:

```
==================
WARNING: DATA RACE
Write at 0x00c00008e000 by goroutine 7:
   main.main.func2()
Previous write at 0x00c00008e000 by goroutine 6:
   main.main.func1()
==================
```

Окончательное значение `i` также непредсказуемо. Иногда это может быть 1, а иногда и 2.

Какая проблема в этом коде? Оператор `i++` производит три действия:

- чтение `I`;
- увеличение значения `i` на 1;
- перезапись нового значения в `i`.

Если первая горутина выполняется и ее выполнение завершается раньше, чем у второй, то происходит вот что.

Горутина 1	Горутина 2	Операция	i
			0
Чтение		<-	0
Увеличение на 1			0
Перезапись		->	1
	Чтение	<-	1
	Увеличение на 1		1
	Перезапись	->	2

Первая горутина читает, увеличивает и записывает значение 1 обратно в `i`. Затем вторая горутина выполняет тот же набор действий, но начинает уже со значения `i`, равного 1. Следовательно, окончательный результат, записанный в `i`, равен 2.

Однако нет никакой гарантии, что первая горутина запустится или завершится раньше второй. Мы также можем столкнуться со случаем чередующегося выполнения, когда обе горутины выполняются одновременно и конкурируют за доступ к `i`. Вот еще один возможный сценарий.

Горутина 1	Горутина 2	Операция	i
			0
Чтение		<-	0
	Чтение	<-	0
Увеличение на 1			0
	Увеличение на 1		0
Перезапись		->	1
	Перезапись	->	1

В таком случае обе горутины сначала читают `i` и получают значение этой переменной, равное 0. Затем они обе увеличивают это значение и записывают свой локальный результат, то есть 1, обратно, а это не является ожидаемым результатом.

Это возможное последствие гонки данных. Если две горутины одновременно обращаются к одной и той же ячейке памяти и хотя бы одна записывает в нее данные, результат может быть опасным. Хуже того, в некоторых ситуациях в ячейке памяти может оказаться значение, содержащее бессмысленную комбинацию битов.

> **ПРИМЕЧАНИЕ** При обсуждении ошибки #83 (не включать флаг `-race`) мы увидим, как среда Go помогает обнаружить гонку данных.

Как предотвращать гонки данных? Рассмотрим несколько разных методик. Я не буду перечислять все возможные варианты (например, опустим `atomic.Value`), а покажу основные из них.

Первый вариант — сделать операцию инкремента атомарной, то есть выполняемой целиком за один шаг. Это предотвращает запутанное выполнение операций.

Горутина 1	Горутина 2	Операция	i
			0
Чтение и увеличение на 1		<->	1
	Чтение и увеличение на 1	<->	2

Даже если вторая горутина запустится перед первой, то результатом останется 2.

Атомарные операции можно выполнять в Go с помощью пакета `sync/atomic`. Вот пример, как атомарно увеличивать `int64`:

```
var i int64
go func() {
   atomic.AddInt64(&i, 1)  <-- Атомарное увеличение значения i
}()
go func() {
   atomic.AddInt64(&i, 1)  <-- То же самое
}()
```

Обе горутины обновляют `i` атомарно. Атомарная операция не может быть прервана, что предотвращает ситуацию с одновременным предоставлением доступа из двух мест. Независимо от порядка выполнения горутин `i` в итоге будет равно 2.

> **ПРИМЕЧАНИЕ** Пакет `sync/atomic` предоставляет примитивы для `int32`, `int64`, `uint32` и `uint64`, но не для `int`. Вот почему в этом примере `i` имеет тип `int64`.

Другой вариант — синхронизировать две горутины с помощью специальной структуры данных, мьютекса. Слово *мьютекс* (mutex) образовано от «mutual exclusion», что означает «взаимное исключение». Мьютекс обеспечивает обращение к так называемой критической секции не более одной горутины. В пакете `sync` определяется тип `Mutex`:

```
i := 0
mutex := sync.Mutex{}
go func() {
   mutex.Lock()  <-- Начало критического раздела
   i++  <-- Увеличение значения i на единицу
   mutex.Unlock()  <-- Конец критического раздела

}()
go func() {
   mutex.Lock()
   i++
   mutex.Unlock()
}()
```

В этом примере инкрементирование `i` является критической секцией. Независимо от порядка горутин, выполнение этого кода также выдает детерминированное значение `i`, равное 2.

Какой подход лучше? Все довольно просто. Как я говорил, пакет `sync/atomic` работает только с определенными типами данных. Если нужно обрабатывать данные каких-то других видов (например, срезы, карты или структуры), то мы не можем полагаться на `sync/atomic`.

Другой возможный вариант — запретить совместное использование одного и того же места в памяти и отдать предпочтение взаимодействию между горутинами. Например, создать канал, который каждая горутина использует для получения значения инкремента:

```
i := 0
ch := make(chan int)
go func() {
   ch <- 1  <-- Уведомление горутины об увеличении на 1
}()
go func() {
   ch <- 1
}()
i += <-ch  <-- Увеличение i от того ее значения, которое было получено из канала
i += <-ch
```

Каждая горутина отправляет по каналу уведомление о том, что нужно увеличить `i` на 1. Родительская горутина собирает эти уведомления и увеличивает `i`. Поскольку она единственная горутина, непосредственно пишущая в ячейку `i`, в таком решении не возникнет гонки данных.

Подведем итоги. Гонка данных происходит, когда несколько горутин одновременно обращаются к одной и той же ячейке памяти (например, к одной и той же переменной) и по крайней мере одна из горутин выполняет запись. Мы рассмотрели варианты предотвращения этой проблемы с помощью трех синхронизирующих подходов:

- использование атомарных операций;
- защита критических секций с помощью мьютексов;
- использование связи и каналов для обеспечения того, чтобы переменная обновлялась только одной горутиной.

Во всех этих трех случаях значение `i` в итоге будет равно 2, независимо от порядка выполнения двух горутин. Но в зависимости от того, какую операцию мы хотим

выполнить, обязательно ли приложение, свободное от гонки данных, означает детерминированный результат? Рассмотрим на другом примере.

Теперь горутины не увеличивают общую переменную, а каждая выполняет операцию присваивания значения. Используем мьютекс для предотвращения гонки данных:

```
i := 0
mutex := sync.Mutex{}
go func() {
   mutex.Lock()
   defer mutex.Unlock()
   i = 1  ◄——  Первая горутина присваивает переменной i значение, равное 1
}()
go func() {
   mutex.Lock()
   defer mutex.Unlock()
   i = 2  ◄——  Вторая горутина присваивает переменной i значение, равное 2
}()
```

Первая горутина присваивает переменной `i` значение 1, а вторая — 2.

Есть ли здесь гонка данных? Нет. Обе горутины обращаются к одной и той же переменной, но не одновременно, так как ее защищает мьютекс. Но является ли этот пример детерминированным? Тоже нет.

В зависимости от порядка выполнения `i` в итоге будет равно либо 1, либо 2. Этот пример не приводит к гонке данных. Но в нем есть *состояние гонки* (race condition). Оно возникает, когда поведение зависит от последовательности или времени выполнения событий, которые невозможно контролировать. В данном случае время событий — это порядок выполнения горутин.

Обеспечение определенной последовательности выполнения горутин — вопрос координации и оркестровки. Если мы хотим сначала перейти из состояния 0 в состояние 1, а затем из состояния 1 в состояние 2, нужно найти способ гарантировать, что горутины выполнятся по порядку. Эту проблему можно решить с помощью каналов. Координация и оркестровка также могут гарантировать, что к определенному разделу будет обращаться только одна горутина, что также может означать исключение мьютекса в предыдущем примере.

При работе с конкурентными приложениями важно понимать, что ситуация гонки данных отличается от ситуации состояния гонки. Гонка данных возникает, когда несколько горутин одновременно обращаются к одной и той же ячейке памяти и по крайней мере одна из них выполняет запись в эту ячейку. Гонка данных означает возможность неожиданного поведения. Тем не менее

приложение, в котором обеспечено отсутствие ситуаций с гонками данных, не обязательно будет выдавать детерминированные результаты. Приложение может быть свободным от гонок данных, но по-прежнему зависеть от неконтролируемых событий (выполнение горутины, скорость распространения сообщения по каналу или длительность обращения к базе данных). В таких случаях будет наблюдаться состояние гонки. Понимание этих концепций очень важно при профессиональной разработке конкурентных приложений.

Теперь рассмотрим модель памяти Go и разберемся, почему это важно.

8.4.2. Модель памяти Go

В предыдущем разделе мы обсудили три основных метода синхронизации горутин: атомарные операции, мьютексы и каналы. Но есть еще несколько основных принципов, о которых важно знать. Например, буферизованные и небуферизованные каналы предоставляют разные гарантии. Чтобы избежать неожиданных гонок, вызванных непониманием основных спецификаций языка, посмотрим на модель памяти Go.

Модель памяти Go (https://golang.org/ref/mem) — это спецификация, определяющая условия, при которых чтение из переменной в одной горутине может гарантированно произойти только после записи в ту же переменную другой горутиной. Другими словами, она предоставляет определенные гарантии, о которых разработчики должны помнить, чтобы избежать гонки данных и обеспечить детерминированный результат.

В рамках одной горутины нет возможности несинхронизированного доступа. То, что одно действие происходит раньше другого, гарантируется порядком, заданным программой.

При работе с несколькими горутинами помните о некоторых из этих гарантий. Мы будем использовать выражение типа «A < B» для обозначения того, что событие A происходит до события B. Рассмотрим эти гарантии (некоторые из них скопированы из модели памяти Go):

- Создание горутины происходит до начала выполнения этой горутины. Следовательно, чтение переменной, а затем запуск новой горутины, которая производит запись в эту переменную, не приводит к гонке данных:

```
i := 0
go func() {
   i++
}()
```

- И наоборот, выход из горутины не обязательно произойдет до наступления какого-либо события. Следующий пример содержит гонку данных:

```
i := 0
go func() {
   i++
}()
fmt.Println(i)
```

Если мы хотим предотвратить гонку данных, следует синхронизировать эти горутины.

- Отправка по каналу происходит до завершения соответствующего приема из этого канала. В следующем примере родительская горутина увеличивает значение переменной перед отправкой, а другая горутина считывает ее после чтения канала:

```
i := 0
ch := make(chan struct{})
go func() {
   <-ch
   fmt.Println(i)
}()
i++
ch <- struct{}{}
```

Порядок такой:

```
variable increment < channel send < channel receive < variable read[1]
```

С помощью транзитивности мы можем обеспечить то, что доступ к `i` будет синхронизирован и, следовательно, не будет гонки данных.

- Закрытие канала происходит до получения замыкания. Следующий пример аналогичен предыдущему, за исключением того, что вместо отправки сообщения мы закрываем канал:

```
i := 0
ch := make(chan struct{})
go func() {
   <-ch
   fmt.Println(i)
}()
i++
close(ch)
```

[1] Увеличение значения переменной < чтение канала < получение канала < чтение значения переменной. — *Примеч. ред.*

Поэтому этот пример также свободен от гонки данных.

- Последняя гарантия в отношении каналов может показаться на первый взгляд нелогичной: прием из небуферизованного канала происходит *до* завершения отправки по этому каналу.

  Для начала рассмотрим пример с буферизованным каналом вместо небуферизованного. Есть две горутины, и та, что является родительской, отправляет сообщение и читает переменную, в то время как дочерняя обновляет эту переменную и получает из канала:

```
i := 0
ch := make(chan struct{}, 1)
go func() {
   i = 1
   <-ch
}()
ch <- struct{}{}
fmt.Println(i)
```

  Этот код приводит к возникновению гонки данных. На рис. 8.9 видно, что и чтение и запись в `i` могут происходить одновременно; следовательно, `i` не синхронизирован.

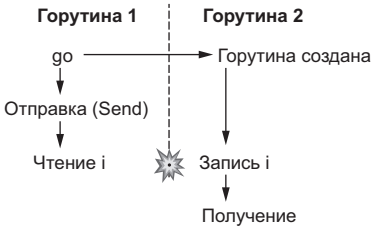


Рис. 8.9. Если канал буферизован, то это приводит к гонке данных

Изменим канал так, чтобы он стал небуферизованным, — для иллюстрации гарантии со стороны модели памяти:

```
i := 0
ch := make(chan struct{})   <-- Канал делается небуферизованным
go func() {
   i = 1
   <-ch
}()
ch <- struct{}{}
fmt.Println(i)
```

В этом примере изменение типа канала устраняет гонку данных (рис. 8.10). Здесь мы видим главное: запись гарантированно происходит до чтения. Обратите

внимание, что стрелки не представляют причинно-следственную связь (хотя, конечно, получение невозможно без предварительной отправки); они представляют собой гарантии порядка, предоставляемые моделью памяти Go. Поскольку прием из небуферизованного канала происходит до отправки, запись в `i` всегда будет происходить до чтения.

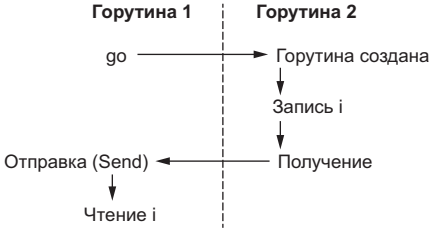


Рис. 8.10. Если канал не буферизован, то он не приводит к гонке данных

В этом разделе мы рассмотрели основные гарантии модели памяти Go. Их понимание должно лежать в основе написания конкурентного кода и может уберечь от неверных предположений и допущений, приводящих к гонке данных и состоянию гонки.

В следующем разделе поговорим, почему так важно понимать тип рабочей нагрузки.

8.5. ОШИБКА #59: НЕ ПОНИМАТЬ ВЛИЯНИЕ ТИПА РАБОЧЕЙ НАГРУЗКИ НА КОНКУРЕНТНОСТЬ

В этом разделе рассматривается влияние типа рабочей нагрузки на конкурентные реализации. В зависимости от того, влияет ли какая-то рабочая нагрузка в большей степени на процессор (CPU) или на систему ввода/вывода (I/O), соответствующие проблемы будем решать по-разному. Для начала определимся с понятиями.

Время выполнения рабочей нагрузки ограничено одним из следующих факторов:

- Тактовой частотой/скоростью работы центрального процессора: например, это главный фактор при выполнении алгоритма сортировки слиянием. Такая рабочая нагрузка называется *CPU-bound*.
- Скоростью работы системы ввода/вывода: например, это главный фактор при выполнении вызова REST или запроса к базе данных. Рабочая нагрузка в этом случае называется *I/O-bound*.

- Объемом доступной памяти: такая рабочая нагрузка называется *memory-bound*.

ПРИМЕЧАНИЕ Последний фактор самый редкий, учитывая, что память в последние десятилетия сильно подешевела. Поэтому в этом разделе основное внимание уделяется двум первым типам рабочей нагрузки: CPU-bound и I/O-bound.

Почему так важно классифицировать рабочую нагрузку в контексте конкурентных приложений? Разберемся с этим, рассмотрев один из паттернов конкурентности: пул рабочих процессов (worker pooling).

В следующем примере реализована функция `read`, которая принимает `io.Reader` и многократно считывает из него 1024 байта. Мы передаем эти 1024 байта функции `task`, которая выполняет какие-то задачи (позднее увидим какие). Функция `task` возвращает целое число, а мы должны вернуть сумму всех результатов. Вот последовательная реализация:

```
func read(r io.Reader) (int, error) {
   count := 0
   for {
      b := make([]byte, 1024)
      _, err := r.Read(b)   <-- Чтение 1024 байт
      if err != nil {
         if err == io.EOF {   <-- Остановка цикла, когда достигнут его конец
            break
         }
         return 0, err
      }
      count += task(b)   <-- Увеличение count на величину, получающуюся на основе результата функции task
   }
   return count, nil
}
```

Эта функция создает переменную `count`, считывает входные данные `io.Reader`, вызывает задачу и увеличивает `count`. А что будет, если мы захотим запускать все функции `task` параллельно?

Одним из вариантов является использование так называемого *паттерна Worker Pool*. Для этого необходимо создать воркеры (workers — рабочие процессы-горутины) фиксированного размера, которые опрашивают задачи из общего канала (рис. 8.11).

Сначала мы запускаем фиксированный пул горутин (позднее обсудим, сколько их будет). Затем создаем общий канал, в котором публикуем задачи после

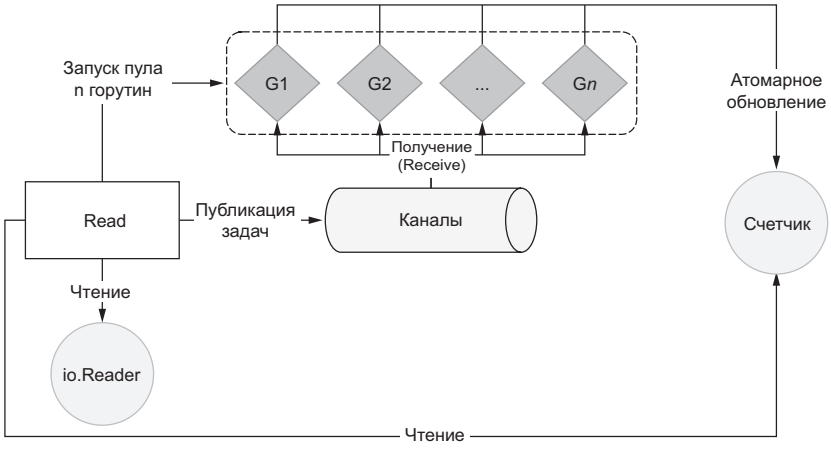


Рис. 8.11. Каждая горутина из фиксированного пула получает данные из общего канала

каждого чтения в `io.Reader`. Каждая горутина из пула получает данные из этого канала, выполняет свою работу, а затем атомарно обновляет общий счетчик.

Вот способ написать соответствующий код на Go с пулом в 10 горутин. Каждая горутина атомарно обновляет общий счетчик:

```
func read(r io.Reader) (int, error) {
   var count int64
   wg := sync.WaitGroup{}
   var n = 10
   ch := make(chan []byte, n)        // Создание канала емкостью, равной пулу
   wg.Add(n)                         // Добавление n к WaitGroup
   for i := 0; i < n; i++ {          // Создание пула из n горутин
      go func() {
         defer wg.Done()             // Вызов метода Done, когда горутина получила данные из канала
         for b := range ch {         // Каждая горутина получает данные из общего канала
             v := task(b)
             atomic.AddInt64(&count, int64(v))
         }
      }()
   }
   for {
      b := make([]byte, 1024)
      // Чтение из r в  b
      ch <- b                        // Публикация новой задачи в канале после каждого чтения
   }
   close(ch)
   wg.Wait()                         // Ожидание завершения группы Wait перед возвратом
   return int(count), nil
}
```

В этом примере мы используем `n` для определения размера пула. Мы создаем канал с той же емкостью, что и этот пул, и группу ожидания (`Wait`) с дельтой, равной `n`. Таким образом, мы уменьшаем потенциальную конкуренцию в родительской горутине в процессе публикации сообщений. Мы повторяем эти действия `n` раз, чтобы создать новую горутину, которая получает данные из общего канала. Каждое полученное сообщение обрабатывается путем выполнения `task` и атомарного увеличения значения общего счетчика. После чтения из канала каждая горутина уменьшает группу ожидания.

В родительской горутине мы продолжаем читать из `io.Reader` и публикуем каждую задачу в канале. И последнее, но не менее важное: мы закрываем канал и ждем завершения работы группы ожидания (это будет означать, что все дочерние горутины завершили свою работу) перед возвратом.

Наличие фиксированного количества горутин ограничивает влияние недостатков, которые мы уже обсуждали. Это сужает влияние ресурсов и предотвращает переполнение внешней системы. А теперь встает основной вопрос: каким должно быть значение размера пула? Ответ зависит от типа рабочей нагрузки.

Если рабочая нагрузка — типа I/O-bound, то ответ зависит от внешней системы. С каким количеством конкурентных обращений сможет справиться система, если мы хотим максимизировать ее пропускную способность?

Если рабочая нагрузка — типа CPU-bound, то рекомендуется полагаться на `GOMAXPROCS` — переменную, которая устанавливает количество потоков ОС, выделенных для выполнения горутин. По умолчанию это значение равно количеству логических процессоров.

Использование runtime.GOMAXPROCS

Мы можем использовать функцию `runtime.GOMAXPROCS (int)` для обновления значения `GOMAXPROCS`. Вызов с 0 в качестве аргумента не меняет, а просто возвращает текущее значение:

```
n := runtime.GOMAXPROCS(0)
```

Итак, зачем нужно сопоставление размера пула с `GOMAXPROCS`? Рассмотрим пример и предположим, что будем запускать наше приложение на четырехъядерной машине. Таким образом, Go создаст четыре потока ОС, в которых будут выполняться горутины. Поначалу все может быть неидеально: мы можем столкнуться со сценарием, когда есть четыре ядра CPU и четыре горутины, но при этом выполняться будет только одна горутина, как показано на рис. 8.12.

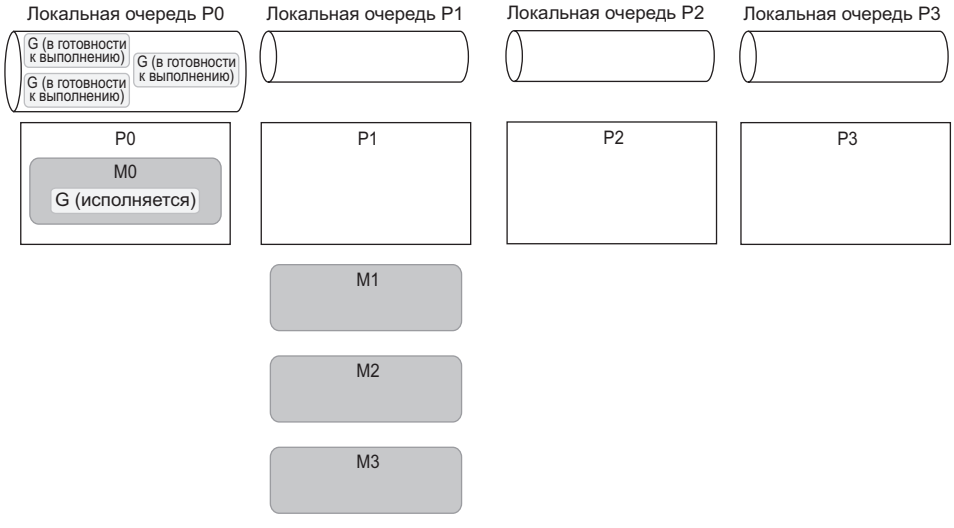


Рис. 8.12. Выполняется не более одной горутины

На M0 выполняется горутина из пула рабочих процессов. Следовательно, все эти горутины начинают получать сообщения из канала и выполнять свои задания. Но для трех других горутин из пула еще не назначены соответствующие M. Следовательно, они находятся в состоянии готовности к выполнению. M1, M2 и M3 не имеют горутин, которые могли бы на них выполняться, поэтому остаются не у дел. Таким образом, работает только одна горутина.

В конце концов, учитывая концепцию кражи задач, которую мы уже описывали, P1 может украсть горутины из локальной очереди P0. На рис. 8.13 P1 украл три горутины у P0. В этой ситуации планировщик Go также может назначить в итоге все горутины другому потоку ОС, но нет никакой определенности относительно того, когда это должно произойти. Но поскольку одной из основных целей планировщика Go является оптимизация ресурсов (в данном случае назначение выполнения горутин различными M), мы должны прийти к сценарию рис. 8.13 с учетом характера рабочих нагрузок:

Этот сценарий по-прежнему не оптимален, потому что работает не более двух горутин. Допустим, на машине запущено только наше приложение (кроме процессов ОС), поэтому P2 и P3 остаются свободными. В конце концов ОС должна задействовать M2 и M3 так, как показано на рис. 8.14.

Здесь планировщик ОС решил назначить M2 на P2 и M3 на P3. Опять же, нет никакой уверенности, когда это произойдет. Но учитывая, что машина выполняет

только наше четырехпоточное приложение, это должно быть окончательной картиной.

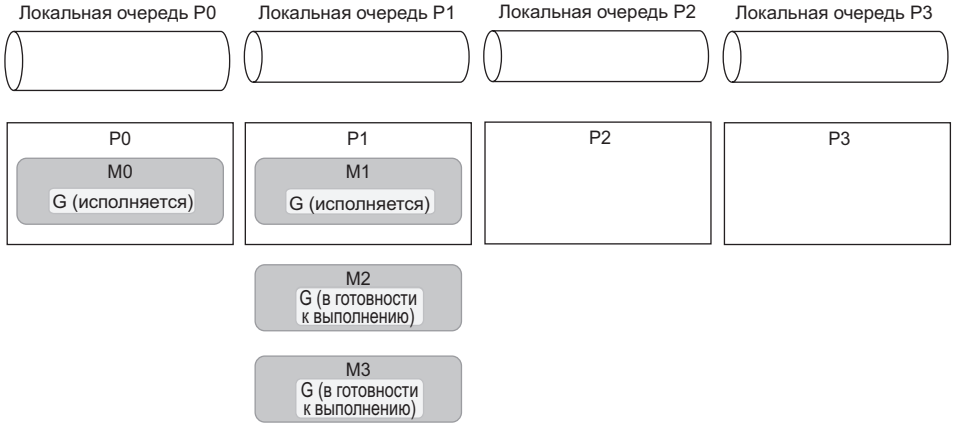


Рис. 8.13. Выполняется не более двух горутин

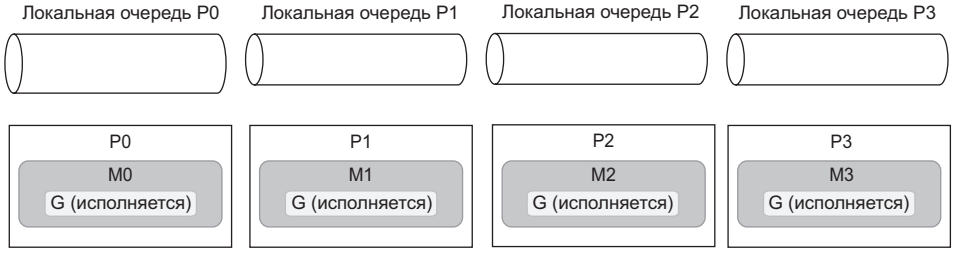


Рис. 8.14. Выполняется не более четырех горутин

Ситуация изменилась — она стала оптимальной. Четыре горутины выполняются в отдельных потоках, а потоки на отдельных ядрах. Такой подход уменьшает количество переключений контекста как на уровне горутин, так и на уровне потоков.

Эта глобальная картина не может быть разработана и запрошена нами (программистами Go). Но как мы видели, мы можем добиться такой благоприятной ситуации в случае рабочих нагрузок типа CPU-bound, то есть при наличии пула рабочих процессов и на основе `GOMAXPROCS`.

ПРИМЕЧАНИЕ Если при каких-то особых условиях мы захотим, чтобы количество горутин было привязано к количеству ядер CPU, почему бы не положиться на функцию `runtime.NumCPU()`, которая возвращает количество логических ядер CPU? Как я говорил, параметр `GOMAXPROCS` может быть изменен и может быть меньше, чем количество ядер CPU. В случае рабочей нагрузки типа CPU-bound, если количество ядер равно четырем, но есть только три потока, нужно запустить три горутины, а не четыре. В противном случае поток будет делить время выполнения между двумя горутинами, увеличивая количество переключений контекста.

При реализации паттерна Worker Pool мы увидели, что оптимальное количество горутин в пуле зависит от типа рабочей нагрузки. Если рабочая нагрузка, выполняемая рабочими процессами, является I/O-bound, то это значение зависит от внешней системы. И наоборот, если рабочая нагрузка будет типа CPU-bound, то оптимальное количество горутин близко к количеству доступных потоков. Знание типа рабочей нагрузки (I/O- или CPU-bound) очень важно при разработке конкурентных приложений.

И последнее, но не менее важное: помните, что в большинстве случаев нужно проверять предположения с помощью бенчмарков. Конкурентность — не простой принцип, и отталкиваясь только от нее, можно сделать поспешные предположения, которые окажутся неверными.

В последнем разделе этой главы обсудим важнейшую тему в Go — контексты.

8.6. ОШИБКА #60: НЕВЕРНО ПОНИМАТЬ КОНТЕКСТЫ GO

Разработчики часто неправильно понимают тип `context.Context`, несмотря на то что он является одним из ключевых понятий языка и основой конкурентности в Go. Изучим эту концепцию и убедимся, что мы понимаем, как эффективно ее использовать. Согласно официальной документации (https://pkg.go.dev/context):

> *Контекст переносит крайний срок, сигнал отмены и другие значения через границы API.*

Разберемся в этом определении и во всех концепциях, связанных в Go с контекстом.

8.6.1. Крайний срок

Крайний срок (deadline) указывает на некий момент времени, определяемый одним из следующих способов:

- `time.Duration` с настоящего момента (например, через 250 мс);
- `time.Time` (например, 2023-02-07 00:00:00 UTC).

Семантика крайнего срока означает, что текущая деятельность должна быть остановлена, если этот крайний срок наступил. «Деятельность» — это, например, запрос типа ввод/вывод или горутина в состоянии ожидания получения сообщения из канала.

Рассмотрим приложение, которое каждые 4 секунды получает от радара данные о позиции самолета. Получив позицию, мы хотим поделиться ею с другими приложениями, для которых интерес представляет только последняя позиция.

В нашем распоряжении есть интерфейс `publisher`, содержащий в себе один-единственный метод:

```
type publisher interface {
   Publish(ctx context.Context, position flight.Position) error
}
```

Этот метод принимает контекст и позицию. Предполагается, что конкретная реализация вызывает функцию для публикации сообщения брокеру (например, использование Sarama для публикации сообщения Kafka). Эта функция *контекстно зависимая* (context aware), это означает, что она может отменить запрос после отмены контекста.

Предполагая, что мы не получаем существующий контекст, что нужно предоставить методу `Publish` в качестве аргумента контекста? Я говорил, что для других приложений нужна информация только о самой последней позиции. Поэтому создаваемый контекст должен сообщать о ней через 4 секунды, а если мы не смогли опубликовать позицию, то следует остановить вызов `Publish`:

```
type publishHandler struct {
   pub publisher
}
func (h publishHandler) publishPosition(position flight.Position) error {
   ctx, cancel := context.WithTimeout(context.Background(), 4*time.Second)
   defer cancel()
   return h.pub.Publish(ctx, position)
}
```

Создание контекста, который будет ожидать только 4 секунды

Откладывание отмены

Передача созданного контекста

Этот код создает контекст с помощью функции `context.WithTimeout`, которая принимает тайм-аут и контекст. Поскольку `publishPosition` не получает существующий контекст, мы создаем его из пустого контекста с помощью `context.Background`. Между тем `context.WithTimeout` возвращает две переменные: созданный контекст и функцию отмены `func()`, которая отменит контекст после вызова. Передача созданного контекста в метод `Publish` должна произойти не позднее чем через 4 секунды.

В чем смысл вызова функции `cancel` как функции `defer`? Внутри себя `context.WithTimeout` создает горутину, которая будет храниться в памяти в течение 4 секунд или до тех пор, пока не будет вызвана `cancel`. Следовательно, вызов `cancel` в качестве функции `defer` означает, что при выходе из родительской функции контекст будет отменен, а созданная горутина остановлена. Это мера предосторожности, чтобы при возвращении мы не оставили в памяти сохраненные объекты.

Теперь перейдем ко второму аспекту контекстов Go — сигналам отмены (cancellation signals).

8.6.2. Сигналы отмены

Другой вариант использования контекстов в Go — передача сигнала отмены. Допустим, нужно создать приложение, которое вызывает `CreateFileWatcher(ctx context.Context, filename string)` внутри другой горутины. Эта функция создает специальный файловый монитор (file watcher), который постоянно читает файл и улавливает его обновления. Когда предоставленный контекст становится неактуальным или отменяется, эта функция обрабатывает его, чтобы закрыть дескриптор файла.

Наконец, когда происходит возврат из `main`, мы хотим, чтобы все обрабатывалось корректно путем закрытия этого файлового дескриптора. Поэтому нам нужно передать сигнал. Возможный подход — использовать `context.WithCancel`, возвращающий контекст (возвращается первая переменная), который отменяется после вызова функции `cancel` (возвращается вторая переменная):

```
func main() {
   ctx, cancel := context.WithCancel(context.Background())
   defer cancel()
   go func() {
      CreateFileWatcher(ctx, "foo.txt")
   }()
   // ...
}
```

Создание контекста, который может быть отменен

Откладывание вызова cancel

Вызов функции с использованием созданного контекста

Когда происходит возврат из `main` , происходит и вызов функции `cancel` для отмены контекста, переданного в `CreateFileWatcher`, — чтобы дескриптор файла корректно закрылся.

Обсудим еще один аспект контекстов Go: значения.

8.6.3. Значения контекстов

Последний вариант использования контекстов Go — перенос списка «ключ — значение». Для начала посмотрим, как это можно применить.

Контекст, передающий значения, может быть создан так:

```
ctx := context.WithValue(parentCtx, "key", "value")
```

Как и `context.WithTimeout`, `context.WithDeadline` и `context.WithCancel`, `context.WithValue` создается из родительского контекста (в данном примере `parentCtx`). В этом случае мы создаем новый контекст `ctx`, содержащий те же характеристики, что и `parentCtx`, но также передающий ключ и значение.

Можно получить доступ к значению, используя метод `Value`:

```
ctx := context.WithValue(context.Background(), "key", "value")
fmt.Println(ctx.Value("key"))
value
```

Задаваемые ключ и значения имеют тип `any`. Действительно, для значений мы хотим передавать тип `any`. Но почему ключ должен быть еще и пустым интерфейсом, а не, например, строкой? Потому, что это может привести к коллизиям: две функции из разных пакетов могут в качестве ключа использовать одно и то же строковое значение. Следовательно, последнее из них переопределит предыдущее. Лучшей практикой при обработке контекстных ключей будет создание неэкспортируемого пользовательского типа:

```
package provider
type key string
const myCustomKey key = "key"
func f(ctx context.Context) {
   ctx = context.WithValue(ctx, myCustomKey, "foo")
   // ...
}
```

Константа `myCustomKey` не экспортируется. Поэтому нет риска, что другой пакет, использующий тот же контекст, может переопределить уже заданное значение.

Даже если другой пакет создает тот же `myCustomKey` на основе типа `key`, это будет другой ключ.

Так какой смысл иметь контекст, содержащий список «ключ — значение»? Поскольку контексты Go повсеместны, есть множество сценариев использования.

Например, при трассировке может потребоваться, чтобы разные подфункции использовали один и тот же идентификатор корреляции. Некоторые разработчики могут посчитать его слишком агрессивным (invasive), чтобы быть частью сигнатуры функции. В связи с этим мы могли бы также решить включить его как часть задаваемого контекста.

Другой пример: если мы хотим реализовать промежуточное ПО HTTP. Если вы не знакомы с таким понятием, то поясню: промежуточное ПО (middleware) — это промежуточная функция, выполняемая перед выполнением запроса. Например, на рис. 8.15 мы настроили два промежуточных модуля, которые должны быть выполнены перед выполнением самого обработчика. Если требуется, чтобы промежуточные программы взаимодействовали, они должны пройти через контекст, обработанный в `*http.Request`.

Рис. 8.15. Прежде чем достичь обработчика, запрос проходит через настроенное сконфигурированное промежуточное ПО

Создадим пример промежуточного ПО, которое отмечает, является ли исходный хост валидным:

```
type key string
const isValidHostKey key = "isValidHost"   <-- Создание ключа контекста
func checkValid(next http.Handler) http.Handler {
   return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
      validHost := r.Host == "acme"   <-- Проверка валидности хоста
      ctx := context.WithValue(r.Context(), isValidHostKey, validHost)
      next.ServeHTTP(w, r.WithContext(ctx))
   })
}
```

Создание нового контекста со значением, чтобы сообщить, действителен ли исходный хост

Вызов следующего шага с новым контекстом

Сначала мы определяем специфический контекстный ключ `isValidHostKey`. Затем промежуточное ПО `checkValid` проверяет, валиден ли исходный хост. Эта информация передается в новом контексте, который далее передается на следующий шаг HTTP с помощью `next.ServeHTTP` (следующим шагом может быть другое промежуточное ПО HTTP или конечный обработчик HTTP).

Этот пример показал, как контекст со значениями можно использовать в конкретных приложениях. В предыдущих разделах мы видели, как создавать контекст для передачи крайнего срока, сигнала отмены и/или значений. Мы можем использовать этот контекст и передавать его *контекстно-зависимым* библиотекам, то есть библиотекам, содержащим в себе функции, которые принимают контекст.

Теперь представим, что нужно создать библиотеку и требуется, чтобы какие-то внешние клиенты предоставляли контекст, который можно было бы отменить.

8.6.4. Перехват отмены контекста

Тип `context.Context` экспортирует метод `Done`, который возвращает односторонний канал уведомления (то есть он может только получать): `<-chan struct{}`. Этот канал закрывается, когда работа, связанная с контекстом, должна быть отменена. Например:

- Канал `Done`, связанный с контекстом, созданным с помощью `context.WithCancel`, закрывается при вызове функции `cancel`.
- Канал `Done`, связанный с контекстом, созданным с помощью `context.WithDeadline`, закрывается по истечении крайнего срока.

Внутренний канал следует закрывать, когда контекст отменяется или наступает крайний срок, а не тогда, когда он получит некоторое определенное значение, потому что закрытие канала — это единственное действие с ним, сигнал о котором получат все горутины-потребители. Таким образом, все потребители будут уведомлены об отмене контекста или достижении крайнего срока.

Более того, `context.Context` экспортирует метод `Err`, возвращающий `nil`, если канал `Done` еще не закрыт. В противном случае возвращается ненулевая ошибка, объясняющая, почему канал `Done` был закрыт. Например:

- ошибка `context.Canceled`, если канал отменен;
- ошибка `context.DeadlineExceeded`, если крайний срок действия контекста прошел.

Реализация функции, получающей контекст

В функции, которая получает контекст, сообщающий о возможной отмене или тайм-ауте, действия по получению или отправке сообщения в канал не должны выполняться блокирующим образом. Например, в следующей функции мы отправляем сообщение в канал и получаем его из другого канала:

```
func f(ctx context.Context) error {
   // ...
   ch1 <- struct{}{}  <-- Получение
   v := <-ch2  <-- Отправка
   // ...
}
```

Проблема здесь в том, что если контекст отменяется или истекает время, в течение которого он актуален, то, возможно, придется ожидать, пока сообщение не будет отправлено или получено. Вместо этого мы должны использовать `select` либо для ожидания завершения действий канала, либо для ожидания отмены контекста:

```
func f(ctx context.Context) error {
   // ...
   select {  <-- Отправка сообщения в ch1 или ожидание отмены контекста
   case <-ctx.Done():
      return ctx.Err()
   case ch1 <- struct{}{}:
   }
   select {  <-- Получение сообщения из ch2 или ожидание отмены контекста
   case <-ctx.Done():
      return ctx.Err()
   case v := <-ch2:
      // ...
   }
}
```

В этой новой версии, если `ctx` отменяется или завершается, мы немедленно возвращаемся, не блокируя канал отправки или получения.

Рассмотрим пример, в котором требуется и дальше получать сообщения из канала. В то же время наша реализация должна учитывать контекст и возвращать значение, если предоставленный контекст выполнен:

```
func handler(ctx context.Context, ch chan Message) error {
   for {
      select {
      case msg := <-ch:  <-- Продолжение получения сообщения от ch
         // Какие-то действия с msg
```

```
        case <-ctx.Done():  ◄──  Если контекст выполнен, возврат связанной с ним ошибки
            return ctx.Err()
        }
    }
}
```

Мы создаем цикл `for` и используем `select` в двух случаях: получение сообщений от `ch` или получение сигнала о том, что контекст выполнен и работу надо остановить. Это пример того, как сделать функцию контекстно зависимой.

Опытные Go-разработчики должны понимать, что такое контекст и как его использовать. `context.Context` доступен в стандартной библиотеке и во всех внешних библиотеках. Как я говорил, контекст позволяет передавать крайний срок, сигнал отмены и/или список «ключ — значение». В общем случае функция, на которую пользователи ожидают ответа, должна принимать контекст, поскольку это позволяет вызывающим сторонам решать, когда прервать вызов этой функции.

Если вы сомневаетесь, какой контекст использовать, выбирайте `context.TODO()` вместо передачи пустого контекста с помощью `context.Background`. `context.TODO()` возвращает пустой контекст, но семантически сообщает, что используемый контекст либо неясен, либо еще недоступен (например, еще не передан родителем).

Все доступные контексты в стандартной библиотеке безопасны для конкурентного использования несколькими горутинами.

ИТОГИ

- Понимание фундаментальных различий между конкурентностью и параллелизмом — краеугольный камень в знаниях Go-разработчика. Конкурентность — это о структуре, тогда как параллелизм — о выполнении.
- Конкурентность не всегда ведет к более быстрым решениям. Варианты, предусматривающие распараллеливание минимальных рабочих нагрузок, не обязательно будут быстрее, чем последовательная реализация. Сравнение бенчмарков решений с последовательной и конкурентной реализацией должно быть способом проверки допущений.
- Знание того, как горутины взаимодействуют друг с другом, полезно, когда вы выбираете между каналами и мьютексами. Обычно параллельные горутины требуют синхронизации и, следовательно, использования мьютексов. Конкурентные горутины обычно требуют координации и оркестровки и, следовательно, использования каналов.

- Гонка данных и состояние гонки — это разные понятия. Гонка данных происходит, когда несколько горутин одновременно обращаются к одной и той же ячейке памяти и по крайней мере одна из горутин выполняет запись в эту ячейку. Отсутствие гонки данных не обязательно означает детерминированность в результате выполнения неких действий. Когда поведение зависит от последовательности или времени наступления событий, которые невозможно проконтролировать, то получается состояние гонки.
- Понимание модели памяти в Go и лежащих в ее основе гарантий с точки зрения упорядочения и синхронизации важно для предотвращения возможной гонки данных или состояния гонки.
- При создании нескольких горутин учитывайте тип рабочей нагрузки. Если создаются CPU-bound горутины, то это число должно ограничиваться значением переменной `GOMAXPROCS` (которая по умолчанию отражает количество ядер CPU на хосте). При создании I/O-bound горутин следует учитывать характеристики внешней системы.
- Контексты в Go также очень важны для понимания конкурентности. Контекст позволяет передавать информацию о крайних сроках, сигналы отмены и/или списки «ключ — значение».

9 Конкурентность: практика

В этой главе:

- ✓ Предотвращение типичных ошибок при работе с горутинами и каналами
- ✓ Последствия применения стандартных структур данных вместе с конкурентным кодом
- ✓ Использование стандартной библиотеки и некоторых расширений
- ✓ Предотвращение гонки данных и взаимоблокировок

В предыдущей главе мы рассмотрели базовые понятия конкурентности. Теперь поговорим об ошибках, возникающих при работе с примитивами конкурентности.

9.1. ОШИБКА #61: ПЕРЕДАВАТЬ НЕПОДХОДЯЩИЙ КОНТЕКСТ

Контексты — это неотъемлемая часть работы с конкурентностью в Go, их рекомендуется передавать во многих ситуациях. Но иногда передача контекста может

приводить к малозаметным ошибкам, мешающим правильному выполнению подфункций.

Рассмотрим пример, где реализуем HTTP-обработчик, выполняющий некоторые задачи и возвращающий ответ. Но прежде чем вернуть ответ, его нужно отправить в тему Kafka. Мы не хотим наказывать HTTP-потребителя какой-либо задержкой, поэтому действие публикации должно обрабатываться асинхронно — в новой горутине. Мы предполагаем, что у нас есть функция `publish`, принимающая контекст, чтобы действие публикации сообщения можно было прервать, если, например, контекст отменен. Вот возможная реализация:

```
func handler(w http.ResponseWriter, r *http.Request) {
   response, err := doSomeTask(r.Context(), r)
   if err != nil {
      http.Error(w, err.Error(), http.StatusInternalServerError)
      return
   }
   go func() {
      err := publish(r.Context(), response)
      // Какие-то действия с err
   }()
   writeResponse(response)
}
```

Выполняется некоторая задача по вычислению ответа HTTP

Создание горутины для публикации ответа в Kafka

Запись HTTP-ответа

Сначала вызываем функцию `doSomeTask`, чтобы получить переменную `response`. Она используется в горутине, вызывающей `publish`, и для форматирования HTTP-ответа. При вызове `publish` мы передаем контекст, прикрепленный к HTTP-запросу. Как вы думаете, что не так с этим кодом?

Мы должны знать, что контекст, прикрепленный к HTTP-запросу, может быть отменен при некоторых условиях:

- Когда соединение клиента закрывается.
- В случае запроса HTTP/2, когда он оказывается отмененным.
- Когда ответ был направлен клиенту обратно.

В первых двух случаях мы, вероятно, поступаем правильно. Например, если получаем ответ от `doSomeTask`, но при этом клиент закрыл соединение, то нормально будет вызвать `publish` с уже отмененным контекстом, так как сообщение не будет опубликовано. Но как быть с последним случаем?

В то время как ответ будет отправляться клиенту, контекст, связанный с запросом, будет отменяться. Таким образом, мы сталкиваемся с состоянием гонки:

- Если ответ отправлен после публикации в Kafka, то мы и возвращаем ответ, и успешно публикуем сообщение.
- Но если ответ написан до или во время публикации в Kafka, то сообщение не должно быть опубликовано.

В последнем случае вызов `publish` приведет к возврату ошибки, потому что мы быстро вернули ответ HTTP.

Как решить эту проблему? Одна идея состоит в том, чтобы не передавать родительский контекст. Вместо этого мы вызовем `publish` с пустым контекстом:

```
err := publish(context.Background(), response)
```

Используется пустой контекст вместо контекста HTTP-запроса

Вот это сработает. Независимо от того, сколько времени потребуется для обратной записи HTTP-ответа, мы можем вызывать `publish`.

Но что, если контекст содержит полезные значения? Например, если бы контекст содержал идентификатор корреляции для распределенной трассировки, мы могли бы соотнести HTTP-запрос и публикацию в Kafka. В идеале нам бы хотелось иметь новый контекст, который не зависит от возможной отмены родительского контекста, но все же передает значения.

Стандартный пакет не предоставляет немедленного решения этой проблемы. Поэтому возможным решением будет реализация собственного контекста Go, аналогичного предоставленному контексту, за исключением того, что он не несет сигнала отмены.

`Context.Context` — это интерфейс, содержащий четыре метода:

```
type Context interface {
   Deadline() (deadline time.Time, ok bool)
   Done() <-chan struct{}
   Err() error
   Value(key any) any
}
```

Крайние сроки, содержащиеся в контексте, обрабатывает метод `Deadline`, а сигналами отмены управляют методы `Done` и `Err`. Когда какой-то крайний срок прошел или контекст был отменен, `Done` должен вернуть закрытый канал, а `Err` — ошибку. Наконец, все значения передаются с помощью метода `Value`.

Создадим собственный контекст, который отвязывает сигнал отмены от родительского контекста:

```
type detach struct {
   ctx context.Context
}
func (d detach) Deadline() (time.Time, bool) {
   return time.Time{}, false
}
func (d detach) Done() <-chan struct{} {
   return nil
}
func (d detach) Err() error {
   return nil
}
func (d detach) Value(key any) any {
   return d.ctx.Value(key)
}
```

Пользовательская структура, действующая как обертка исходного контекста

Делегирует вызов get value родительскому контексту

За исключением метода `Value`, который вызывает родительский контекст для извлечения значения, другие методы возвращают значение по умолчанию, поэтому контекст никогда не считается просроченным или отмененным.

Используя свой пользовательский контекст, мы можем вызвать публикацию и отвязать сигнал отмены:

```
err := publish(detach{ctx: r.Context()}, response)
```

Использует detach поверх контекста HTTP

Теперь переданный для публикации контекст никогда не окажется устаревшим или отмененным — он будет нести в себе все значения из родительского контекста.

Таким образом, передавать контекст следует осторожно. Я показал это на примере обработки асинхронного действия на основе контекста, связанного с HTTP-запросом. Поскольку контекст отменяется в то время, когда мы возвращаем ответ, асинхронное действие может быть неожиданно остановлено. Помните о последствиях передачи данного контекста и о том, что для конкретного действия всегда можно создать собственный контекст.

В следующем разделе обсудим запуск горутины без плана по ее остановке.

9.2. ОШИБКА #62: ЗАПУСКАТЬ ГОРУТИНУ И НЕ ЗНАТЬ, КОГДА ЕЕ ОСТАНОВИТЬ

Горутины запускать легко и дешево, причем настолько, что вы можете не понимать, когда следует останавливать новую горутину, а это может приводить

к утечкам. Незнание того, когда остановить горутину, является проблемой проектирования и распространенной ошибкой конкурентности в Go. Разберемся, как ее предотвратить.

Для начала давайте определим, что такое утечка горутины. С точки зрения потребления памяти горутина требует минимального размера стека в 2 Кбайт, который может меняться по мере необходимости (максимальный размер стека составляет 1 Гбайт для 64-разрядных и 250 Мбайт для 32-разрядных систем). Горутина может содержать ссылки на переменные, место под которые выделено и зарезервировано в куче. Между тем горутина может содержать такие ресурсы, как HTTP-соединения или соединения с базой данных, открытые файлы и сетевые сокеты, которые в итоге должны быть корректно закрыты. Если происходит утечка горутины, то будут и утечки таких ресурсов.

Рассмотрим пример, в котором момент остановки горутины неясен. Здесь родительская горутина вызывает функцию, которая возвращает канал, а затем создает новую горутину, которая продолжает получать сообщения от этого канала:

```
ch := foo()
go func() {
   for v := range ch {
      // ...
   }
}()
```

Созданная горутина завершится, когда `ch` будет закрыт. Но знаем ли мы точно, когда такое закрытие произойдет? Это может быть неочевидно, потому что `ch` создается функцией `foo`. Если канал никогда не будет закрыт, то возникнет утечка. Нужно быть осторожными с точками выхода из горутины и убедиться, что они достигнуты.

Обсудим пример. Разработаем приложение, которое должно отслеживать некоторую внешнюю конфигурацию (например, с помощью подключения к базе данных). Вот его первая реализация:

```
func main() {
   newWatcher()
   // Запуск выполнения приложения
}
type watcher struct { /* Некоторые ресурсы */ }
func newWatcher() {
   w := watcher{}
   go w.watch()  <-- Создание горутины, отслеживающей некоторую внешнюю конфигурацию
}
```

Мы вызываем `newWatcher`, который создает структуру `watcher` и запускает горутину, отвечающую за наблюдение за конфигурацией. Проблема с этим кодом в том, что при выходе из основной горутины (возможно, из-за сигнала ОС или из-за того, что она имеет конечную рабочую нагрузку) приложение останавливается. Ресурсы, созданные в `watcher`, не закрываются корректно. Как предотвратить это?

Один из вариантов — передача в `newWatcher` контекста, который будет отменен при возвращении `main`:

```
func main() {
   ctx, cancel := context.WithCancel(context.Background())
   defer cancel()
   newWatcher(ctx)  <-- Передача в newWatcher контекста, который в дальнейшем будет отменен
   // Запуск выполнения приложения
}
func newWatcher(ctx context.Context) {
   w := watcher{}
   go w.watch(ctx)  <-- Распространение этого контекста
}
```

Мы передаем созданный контекст на метод `watch`. Когда контекст отменяется, структура `watcher` должна закрыть свои ресурсы. Но есть ли гарантия, что у `watch` будет время, чтобы это сделать? Конечно нет. И это недостаток дизайна нашего кода.

Проблема в том, что мы использовали подачу сигнала, чтобы сообщить, что горутина должна быть остановлена. Мы не заблокировали родительскую горутину до тех пор, пока ресурсы не окажутся закрыты. Сделаем это:

```
func main() {
   w := newWatcher()
   defer w.close()  <-- Откладывание вызова метода close
   // Запуск выполнения приложения
}
func newWatcher() watcher {
   w := watcher{}
   go w.watch()
   return w
}
func (w watcher) close() {
   // Закрытие ресурсов
}
```

Теперь в `watcher` есть новый метод — `close`. Чтобы не посылать в `watcher` сигнал о том, что пришло время закрыть его ресурсы, теперь вызывается метод `close` с помощью `defer`, что обеспечивает закрытие этих ресурсов до выхода из приложения.

Горутина — это такой же ресурс, как и любой другой, который должен быть закрыт для освобождения памяти или других ресурсов. Запуск горутины без понимания того, когда ее остановить, — это проблема дизайна программы. Всякий раз, когда горутина запускается, вы должны четко понимать, когда она остановится. И последнее, но не менее важное: если горутина создает ресурсы и время ее жизни привязано к времени жизни приложения, то перед выходом из приложения, вероятно, будет безопаснее дождаться завершения этой горутины. Это гарантирует, что ресурсы будут освобождены.

Теперь обсудим одну из самых типичных ошибок при работе в Go — неправильное обращение с горутинами и переменными цикла.

9.3. ОШИБКА #63: НЕОСТОРОЖНО ОБРАЩАТЬСЯ С ГОРУТИНАМИ И ПЕРЕМЕННЫМИ ЦИКЛА

Неправильное обращение с горутинами и переменными цикла — это, пожалуй, одна из самых распространенных ошибок в Go при написании конкурентных приложений. Рассмотрим пример и определим условия, при которых эта ошибка возникает, а также узнаем, как ее предотвращать.

В следующем примере инициализируется срез. Затем в замыкании, которое выполняется как новая горутина, мы получаем доступ к этому элементу:

```
s := []int{1, 2, 3}
for _, i := range s {  <-- Итерации по каждому элементу
   go func() {
      fmt.Print(i)  <-- Обращение к переменной цикла
   }()
}
```

Мы могли бы ожидать, что этот код выведет `123` в произвольном порядке (поскольку нет гарантии, что горутина, созданная первой, первой и завершится). Но вывод этого кода — не детерминированный. Иногда он выводит `233`, а иногда `333`. Почему?

В этом примере мы создаем новые горутины внутри замыкания. Напоминаем, что замыкание — это функция, которая ссылается на переменные вне своего тела: в данном случае это переменная `i`. Помните, что когда горутина выполняется из замыкания, она не фиксирует значения, которые были при ее создании. Вместо этого все горутины ссылаются на одну и ту же переменную. Когда горутина запускается, она выводит значение переменной `i` на тот момент,

когда выполняется `fmt.Print`. Следовательно, с момента запуска горутины `i` могла измениться.

На рис. 9.1 показан возможный вариант выполнения кода, когда он выводит `233`. Со временем значение `i` меняется: `1`, `2`, а затем `3`. На каждой итерации мы запускаем новую горутину. Поскольку нет гарантии, когда каждая горутина запустится и завершится, то и результат может быть разным. В этом примере первая горутина выводит значение `i`, когда оно равно `2`. Затем другие горутины выводят `i`, когда оно уже равно `3`. Этот пример выводит `233`. Поведение этого кода не детерминировано.

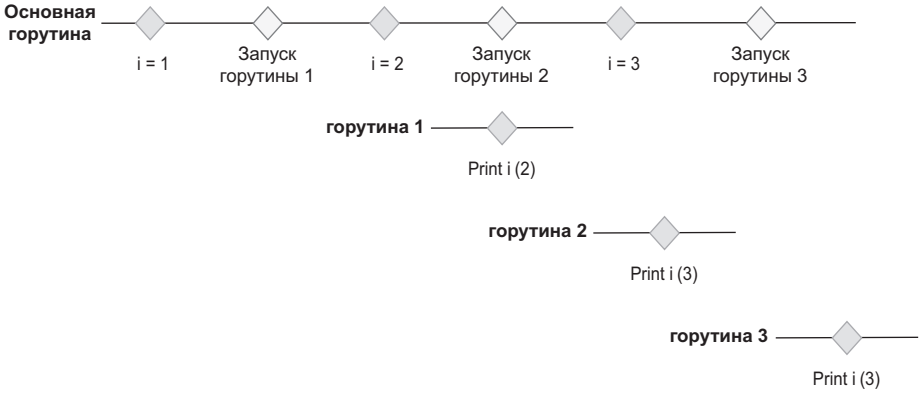


Рис. 9.1. Горутины обращаются к переменной `i`, которая не является фиксированной, а меняется со временем

А что делать, если нужно, чтобы каждое замыкание обращалось к тому значению, которое имела переменная `i` при создании горутины? Если мы хотим продолжать использовать замыкание, то первый вариант предполагает создание новой переменной:

```
for _, i := range s {
   val := i  <-- Создание переменной, локальной для каждой итерации
   go func() {
      fmt.Print(val)
   }()
}
```

Почему этот код будет работать правильно? На каждой итерации мы создаем новую локальную переменную `val`. Эта переменная фиксирует текущее значение `i` перед созданием горутины. Следовательно, каждая горутина из замыкания

выполняет оператор `Print` с тем значением `i`, которое для нас ожидаемо. Этот код выводит `123` (опять же, в произвольном порядке).

Второй вариант больше не зависит от замыкания и использует фактическую функцию:

```
for _, i := range s {
   go func(val int) {
      fmt.Print(val)
   }(i)
}
```

Выполнение функции, которая принимает целое число в качестве аргумента

Вызов этой функции с текущим значением переменной i

По-прежнему в новой горутине выполняется анонимная функция (например, не запускается `go f(i)`), но на этот раз это не замыкание. Функция не ссылается на `val` как на переменную вне своего тела; `val` теперь является частью входных данных функции. Так мы фиксируем `i` на каждой итерации, и приложение работает так, как ожидалось.

Следует осторожно обращаться с горутинами и переменными цикла. Если горутина представляет собой замыкание, которое обращается к переменной итерации, объявленной вне ее тела, то это может вызывать проблемы. Их можно решить, либо создав локальную переменную (например, используя `val := i` перед выполнением горутины), либо сделав функцию, не являющуюся замыканием. Оба варианта работают одинаково хорошо, и нет никаких оснований для того, чтобы предпочесть один другому. Некоторым разработчикам подход с замыканием может показаться более удобным, в то время как другие найдут подход с фактической функцией более выразительным.

Что происходит с оператором `select` на нескольких каналах? Давайте выясним.

9.4. ОШИБКА #64: ОЖИДАТЬ ДЕТЕРМИНИРОВАННОЕ ПОВЕДЕНИЕ ПРИ ИСПОЛЬЗОВАНИИ SELECT И КАНАЛОВ

При работе с каналами Go-разработчики могут делать неверные предположения о том, как `select` ведет себя с несколькими каналами. Это может привести к скрытым ошибкам, которые трудно выявить и отловить.

Допустим, мы хотим реализовать горутину, которая должна получать данные из двух каналов:

- `messageCh` — для новых сообщений, которые нужно будет обработать;
- `disconnectCh` — для получения уведомлений об отключениях. В этом случае мы хотим вернуться из родительской функции.

При получении данных из этих двух каналов требуется отдать приоритет `messageCh`. Например, если происходит какое-то отключение, нужно убедиться, что перед возвратом получены все сообщения. Приоритизацию можно установить следующим образом:

```
for {
   select {   <-- Использование оператора select для получения данных из нескольких каналов
   case v := <-messageCh:   <-- Получение новых сообщений
      fmt.Println(v)
   case <-disconnectCh:   <-- Получение сообщений об отключениях
      fmt.Println("disconnection, return")
      return
   }
}
```

Здесь мы используем оператор `select` для получения сообщений из нескольких каналов. Поскольку мы хотим отдать приоритет тому, что приходит из `messageCh`, то можно предположить, что в коде сначала следует обработать `messageCh`, а уже потом `disconnectCh`. Но будет ли этот код работать? Попробуем это проверить, написав фиктивную горутину-производитель, которая будет отправлять 10 сообщений, а затем уведомление об отключении:

```
for i := 0; i < 10; i++ {
   messageCh <- i
}
disconnectCh <- struct{}{}
```

Если мы запустим этот пример, то вывод может выглядеть так (если `messageCh` буферизируется):

```
0
1
2
3
4
disconnection, return
```

Мы получили не 10 сообщений, а только пять. В чем причина? Она кроется в спецификации оператора `select` с несколькими каналами (https://go.dev/ref/spec):

> *Если одна или несколько коммуникаций могут быть выполнены, то будет выбрана только одна из них с помощью равномерного псевдослучайного выбора.*

В отличие от оператора `switch`, где выполняется первый совпадающий случай, оператор `select` случайным образом выбирает один из возможных вариантов, если их несколько.

Поначалу такое поведение может показаться странным, но для этого есть веская причина: стремление предотвратить возможное голодание (starvation). Предположим, что первая возможная коммуникация выбрана на основе исходного порядка. Тогда мы можем попасть в ситуацию, когда получим данные только по одному каналу, которому соответствует быстрый отправитель. Чтобы предотвратить такие ситуации, разработчики языка решили использовать случайный выбор.

Возвращаясь к нашему примеру, можно сказать, что, даже несмотря на то, что `case v := <-messageCh` стоит первым в исходном порядке, если и в канале `messageCh`, и в канале `disabledCh` есть какие-то сообщения, то нет никакой уверенности, какое из них будет выбрано. По этой причине поведение кода из примера — не детерминированное. Можно получить и 0 сообщений, и 5, и 10.

Каким может быть решение? Существуют разные способы получения всех сообщений перед возвратом по отключению.

Если есть только одна горутина, которая генерирует сообщения и уведомления об отключении, то возможны два варианта:

- Сделать канал `messageCh` небуферизованным вместо буферизованного. Поскольку горутина-отправитель блокируется до тех пор, пока горутина-получатель не будет готова, такой подход гарантирует, что все сообщения от `messageCh` будут получены до отключения по уведомлению из `disconnectCh`.
- Использовать один канал вместо двух. Например, можно определить структуру, которая будет передавать либо новое сообщение, либо сигнал об отключении. Каналы гарантируют, что порядок отправления сообщений будет таким же, как и порядок их получения. Так мы сможем гарантировать, что уведомление об отключении будет получено последним.

Если в нашем случае несколько горутин-производителей, не всегда можно гарантировать, сообщение какой конкретно из них будет записано первым. И независимо от того, имеем мы небуферизованный канал `messageCh` или единственный канал, такая конфигурация приводит к состоянию гонки среди горутин-производителей. Тогда возможно следующее решение:

1. Получение данных либо из `messageCh`, либо из `disconnectCh`.

2. Если получено уведомление об отключении, то:
   - прочитать все имеющиеся в `messageCh` сообщения, если они там есть;
   - вернуться.

Вот реализация этого решения:

```
for {
   select {
   case v := <-messageCh:
      fmt.Println(v)
   case <-disconnectCh:
      for {  <-- Вложенные for/select
         select {
         case v := <-messageCh:  <-- Чтение остающихся сообщений
            fmt.Println(v)
         default:  <-- Затем возврат
            fmt.Println("disconnection, return")
            return
         }
      }
   }
}
```

В этом решении используется внутренний блок `for/select`, в котором обрабатываются два случая (`cases`): один — `messageCh`, другой — `default`. Переход к `default` в операторе `select` происходит только в том случае, если ни один из других случаев не происходит. Другими словами, это означает, что мы перейдем к возврату только после того, как получим все оставшиеся в `messageCh` сообщения.

Посмотрим, как работает этот код на конкретном примере. Мы рассмотрим ситуацию, когда в `messageCh` есть два сообщения и одно отключение в `disconnectCh` (рис. 9.2).

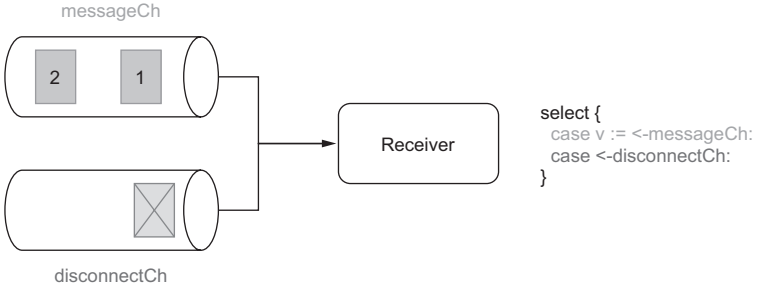


Рис. 9.2. Начальное состояние

В этой ситуации `select` случайным образом выбирает вариант. Предположим, что `select` выбирает второй (рис. 9.3).

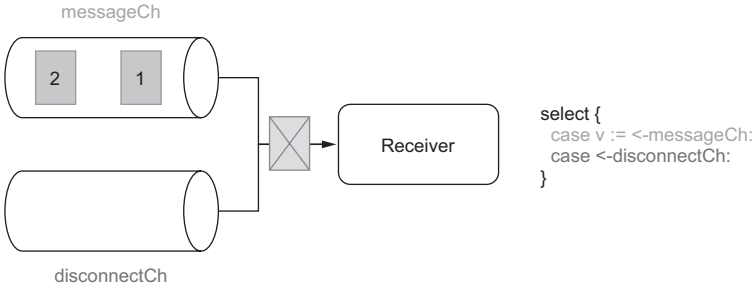


Рис. 9.3. Получение сигнала об отключении

Мы получаем сигнал об отключении и входим в блок, который находится внутри `select` (рис. 9.4). В этом сценарии, пока сообщения остаются в `messageCh`, `select` всегда будет отдавать приоритет первому случаю перед `default` (рис. 9.5).

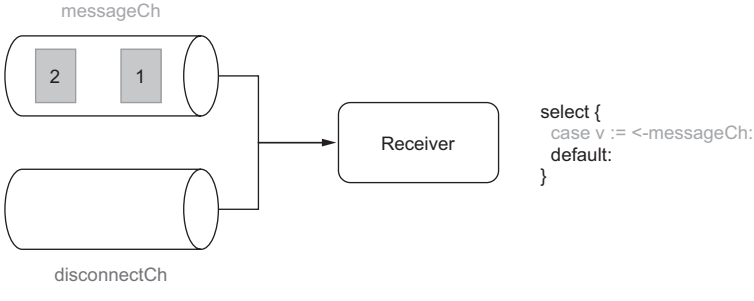


Рис. 9.4. Вложенный `select`

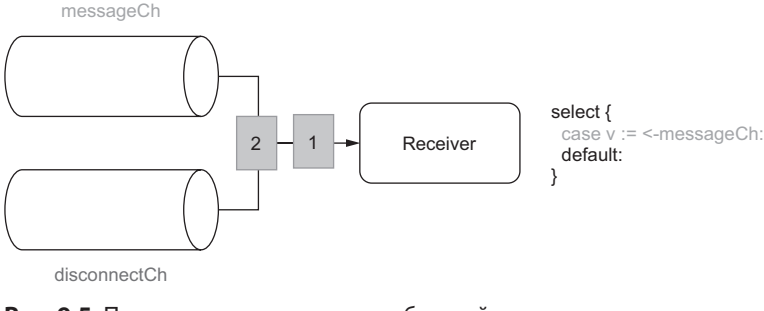


Рис. 9.5. Получение оставшихся сообщений

Как только мы получили все сообщения из `messageCh`, `select` не блокирует и выбирает вариант `default` (рис. 9.6). Происходит возврат и остановка горутины.

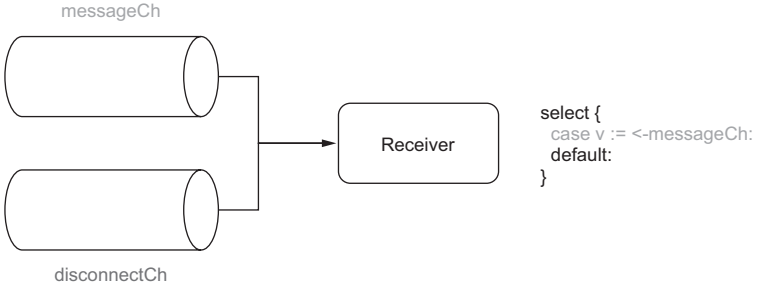


Рис. 9.6. Переход к случаю `default`

Описанный способ позволяет убедиться, что мы получим все оставшиеся сообщения из канала с получателем из нескольких каналов. Конечно, если сообщение отправлено в `messageCh` после возврата из горутины (например, если у нас есть несколько горутин-производителей), то мы это сообщение пропустим.

При использовании `select` с несколькими каналами нужно помнить, что если при этом возможны несколько вариантов, то первый из них в исходном порядке не выбирается автоматически. Go действует случайным образом, поэтому нет никакой уверенности, какой из вариантов будет выбран. Чтобы преодолеть эту трудность, в случае одной горутины-производителя мы можем либо использовать небуферизованные каналы, либо свести все производители в один канал. В случае с несколькими горутинами-производителями для установки приоритетов можно использовать операторы `select` и оператор `default`.

Ниже обсудим распространенный тип канала — каналы уведомлений.

9.5. ОШИБКА #65: НЕ ИСПОЛЬЗОВАТЬ КАНАЛЫ УВЕДОМЛЕНИЙ

Каналы — это механизм взаимодействия между горутинами с помощью сигналов. Сигнал может быть как с данными, так и без них. И для программистов Go последний случай не всегда простой.

Рассмотрим пример. Создадим канал, который будет уведомлять нас всякий раз, когда происходит какое-то отключение. Одна из идей — обращаться с ним как с `chan bool`:

```
disconnectCh := make(chan bool)
```

Теперь предположим, что мы взаимодействуем с API, предоставляющим такой канал. Поскольку это канал логических значений, мы можем получать сообщения вида как `true`, так и `false`. Наверное, понятно, какой смысл несет в себе `true`. Но что означает `false`? Означает ли это, что мы не отключились? И в таком случае как часто мы будем получать такой сигнал? Означает ли это, что мы восстановили соединение?

Должны ли мы вообще ожидать получения `false`? Возможно, следует ожидать только получения сообщений со значением `true`. Если это так, то это означает, что для передачи некоторой информации не нужно задавать или определять конкретное значение — нужен канал, по которому *не передаются* данные. Идиоматический способ справиться с этим — создать канал пустых структур: `chan struct{}`.

В Go пустая структура — это структура без каких-либо полей. Независимо от архитектуры она занимает в памяти 0 байт, в чем мы можем убедиться, используя `unsafe.Sizeof`:

```
var s struct{}
fmt.Println(unsafe.Sizeof(s))
```

```
0
```

> **ПРИМЕЧАНИЕ** Почему не следует использовать пустой интерфейс (`var i interface{}`)? Потому что пустой интерфейс имеет ненулевой объем. Он занимает 8 байт в 32-битной архитектуре и 16 байт в 64-битной архитектуре.

Пустая структура — стандарт де-факто для обозначения отсутствия смысла. Например, если нужна структура в виде хеш-множества (коллекция уникальных элементов), то следует использовать в качестве значения пустую структуру `map[K]struct{}`.

Применительно к каналам: если мы хотим создать канал для отправки уведомлений без данных, подходящий способ сделать это в Go — `chan struct{}`. Одно из самых известных применений канала пустых структур связано с контекстами Go, которые мы обсудим в этой главе.

Канал может быть с данными или без них. Если нужно разработать идиоматический API в соответствии со стандартами Go, помните, что канал без данных

должен быть выражен типом `chan struct{}`. Таким образом, получатели сигнала понимают, что не должны ожидать никакого смысла от содержания сообщения — имеет значение только сам факт получения ими сообщения. В Go такие каналы называются *каналами уведомлений* (notification channels).

В следующем разделе обсудим, как Go ведет себя с нулевыми каналами, и узнаем, зачем их использовать.

9.6. ОШИБКА #66: НЕ ИСПОЛЬЗОВАТЬ НУЛЕВЫЕ КАНАЛЫ

При работе с каналами в Go разработчики часто забывают о том, что нулевые каналы иногда могут быть полезны. Так что же это такое и почему об этом нужно помнить?

Рассмотрим горутину, которая создает нулевой канал и ожидает получения сообщения. Что должен делать ее код?

```
var ch chan int  ◄——  Нулевой канал (nil channel)
<-ch
```

Тип переменной `ch` — `chan int`. Нулевое значение канала — это nil, то есть `ch` равно `nil`. Горутина не будет вызывать панику, но заблокируется навсегда.

Принцип тот же и в случае, если мы отправляем сообщение в нулевой канал. Эта горутина блокируется навсегда:

```
var ch chan int
ch <- 0
```

Тогда почему Go позволяет получать сообщения из нулевого канала или отправлять их в него? Рассмотрим пример.

Реализуем функцию `func merge(ch1, ch2 <-chan int) <-chan int` для объединения двух каналов в один. Под их слиянием (см. рис. 9.7) мы подразумеваем, что каждое сообщение, полученное в `ch1` или в `ch2`, будет отправлено в возвращаемый канал.

Как мы можем сделать это в Go? Напишем наивную реализацию этого действия, в которой запускаем горутину и получаем данные из обоих каналов (результирующий канал будет буферизованным каналом с одним элементом):

```
func merge(ch1, ch2 <-chan int) <-chan int {
   ch := make(chan int, 1)
   go func() {
      for v := range ch1 {
         ch <- v
      }
      for v := range ch2 {
         ch <- v
      }
      close(ch)
   }()
   return ch
}
```

Получение данных из канала ch1 и публикация их в объединенном канале

Получение данных из канала ch2 и публикация их в объединенном канале

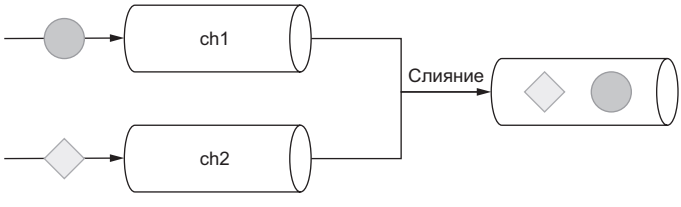


Рис. 9.7. Слияние двух каналов в один

В другой горутине мы получаем данные из обоих каналов, и каждое сообщение в итоге публикуется в `ch`.

Проблема первой версии в том, что мы получаем данные сначала из `ch1`, а затем из `ch2`. Это означает, что мы не будем ничего получать из `ch2`, пока `ch1` не будет закрыт. Это не подходит для нашего случая, так как может оказаться, что `ch1` станет открытым навсегда. Поэтому надо сделать так, чтобы получать данные из обоих каналов одновременно.

Улучшим код с конкурентным получателями, используя `select`:

```
func merge(ch1, ch2 <-chan int) <-chan int {
   ch := make(chan int, 1)
   go func() {
      for {
         select {
         case v := <-ch1:
            ch <- v
         case v := <-ch2:
            ch <- v
         }
      }
      close(ch)
   }()
   return ch
}
```

Получение из обоих каналов ch1 и ch2 конкурентно

Оператор `select` позволяет горутине одновременно ожидать выполнения нескольких операций. Поскольку мы обернули его в цикл `for`, то должны получать сообщения из того или иного канала многократно. Будет ли работать этот код?

Одна из его проблем в том, что оператор `close(ch)` недоступен. Зацикливание канала с использованием оператора `range` прерывается, когда канал оказывается закрытым. Но в способе, которым мы реализовали `for/select`, не улавливаются те моменты, когда закрывается `ch1` или `ch2`. И что еще хуже, если в какой-то момент `ch1` или `ch2` закроется, вот что получит получатель объединенного канала при регистрации значения:

```
received: 0
received: 0
received: 0
received: 0
received: 0
...
```

Получатель будет постоянно получать целое число, равное нулю. Почему? Прием из закрытого канала — неблокирующая операция:

```
ch1 := make(chan int)
close(ch1)
fmt.Print(<-ch1, <-ch1)
```

Пока мы ожидаем, что этот код либо вызовет состояние паники, либо приведет к блокировке, он запускается и выводит `0 0`. Здесь мы перехватываем событие закрытия, а не то, что фактически содержится в сообщении. Вот как проверить, получаем мы сообщение или сигнал закрытия:

```
ch1 := make(chan int)
close(ch1)
v, open := <-ch1
fmt.Print(v, open)
```

Присваивание значения переменной open независимо от того, открыт канал или нет

Используя переменную `open` булева типа, мы увидим, открыт ли еще `ch1`:

```
0 false
```

Между тем `0` присваивается переменной `v`, потому что это нулевое значение целого числа.

Вернемся ко второму решению. Оно работает не очень хорошо, если `ch1` закрыт. Поскольку при обращении к `select` будет `case v := <-ch1`, продолжим обращаться к этому случаю и публиковать нулевое целое число в объединенном канале.

Сделаем шаг назад и посмотрим, как лучше решить эту проблему (рис. 9.8). Требуется получать данные из обоих каналов. Тогда либо:

- `ch1` закрывается первым, поэтому мы должны получать данные из `ch2`, пока он не будет закрыт;
- `ch2` закрывается первым, поэтому мы должны получать данные из `ch1`, пока он не будет закрыт.

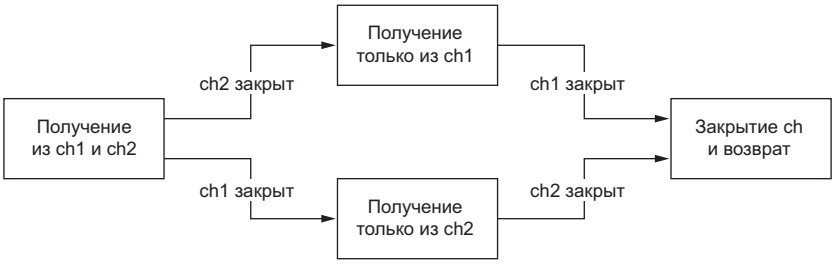


Рис. 9.8. Обработка различных случаев в зависимости от того, какой канал будет закрыт первым

Как реализовать это средствами Go? Напишем версию кода, похожую на то, что мы могли бы сделать, используя подход конечного автомата и булевы значения:

```
func merge(ch1, ch2 <-chan int) <-chan int {
   ch := make(chan int, 1)
   ch1Closed := false
   ch2Closed := false
   go func() {
      for {
         select {
         case v, open := <-ch1:
            if !open {   <-- Обработка в случае, когда ch1 закрыт
               ch1Closed = true
               break
            }
            ch <- v
         case v, open := <-ch2:
            if !open {   <-- Обработка в случае, когда ch2 закрыт
               ch2Closed = true
               break
            }
            ch <- v
         }
```

```
            if ch1Closed && ch2Closed {   <-- Закрытие и возврат, когда закрыты оба канала

               close(ch)
               return
            }
         }
      }()
      return ch
}
```

Вводятся две логические переменные: `ch1Closed` и `ch2Closed`. Как только мы получаем сообщение из какого-либо канала, то проверяем, является ли оно сигналом закрытия. Если да, то мы обрабатываем его, помечая канал как закрытый (например, `ch1Closed = true`). После закрытия обоих каналов закрываем и объединенный канал и останавливаем горутину.

В чем проблема этого кода, кроме того, что он усложняется? Есть один важный момент: когда один из двух каналов закрыт, цикл `for` будет действовать как цикл ожидания сигнала занятости, то есть он будет продолжаться, даже если в другом канале не будет получено новое сообщение. Нужно помнить о поведении оператора `select` в этом примере. Допустим, канал `ch1` закрыт (поэтому мы не будем получать из него новые сообщения). При обращении к оператору `select` он будет ждать выполнения одного из трех условий:

- `ch1` закрыт;
- в `ch2` есть новое сообщение;
- `ch2` закрыт.

Первое условие — `ch1` закрыт — всегда будет истинным. Поэтому пока мы не получим сообщение в `ch2` и этот канал не окажется закрыт, цикл продолжится в рамках первого случая. Это приведет к бессмысленной трате процессорных циклов, чего следует избегать. Поэтому такое решение неразумно.

Можно было бы улучшить часть конечного автомата и реализовать вложенные циклы `for/select` в каждом случае. Но это сделало бы код еще более сложным и трудным для понимания.

Поэтому пришло время вернуться к нулевым каналам. Как я говорил, получение с нулевого канала будет блокировать выполнение кода навсегда. А как насчет использования этой идеи в нашем решении? Чтобы после закрытия канала не задавать значение логической переменной, присвоим этому каналу значение `nil`. Напишем окончательную версию кода:

```
func merge(ch1, ch2 <-chan int) <-chan int {
   ch := make(chan int, 1)
   go func() {
      for ch1 != nil || ch2 != nil {
         select {
         case v, open := <-ch1:
            if !open {
               ch1 = nil
               break
            }
            ch <- v
         case v, open := <-ch2:
            if !open {
               ch2 = nil
               break
            }
            ch <- v
         }
      }
      close(ch)
   }()
   return ch
}
```

Продолжить выполнение, если хотя бы один канал ненулевой

Присвоение каналу ch1 значения nil после того, как он закрывается

Присвоение каналу ch2 значения nil после того, как он закрывается

Для начала здесь мы остаемся внутри цикла до тех пор, пока хотя бы один канал остается открытым. Затем, если `ch1` оказывается закрыт, мы присваиваем ему значение `nil`. Следовательно, во время последующей итерации цикла оператор `select` будет ждать только двух условий:

- в `ch2` есть новое сообщение;
- `ch2` закрыт.

`ch1` больше не часть решаемого уравнения, поскольку это нулевой канал. Между тем мы сохраняем ту же логику для `ch2` и присваиваем ему значение `nil` после его закрытия. Наконец, когда оба канала закрыты, происходит закрытие объединенного канала и возврат. На рис. 9.9 показана схема этой реализации.

Это нужная реализация. В ней охватываются все возможные случаи и не требуется использование цикла ожидания сигнала занятости, который будет впустую тратить процессорное время.

Мы увидели, что ожидание из нулевого канала или отправка в него — это блокирующее действие, и такое его поведение небесполезно. Как мы заметили на примере слияния двух каналов в один, возможно использовать нулевые каналы для реализации элегантного конечного автомата, который будет *удалять* из оператора `select` один из возможных случаев. Помните об этой идее: нулевые

каналы полезны в некоторых ситуациях и должны быть частью инструментария Go-разработчика при работе с конкурентным кодом.

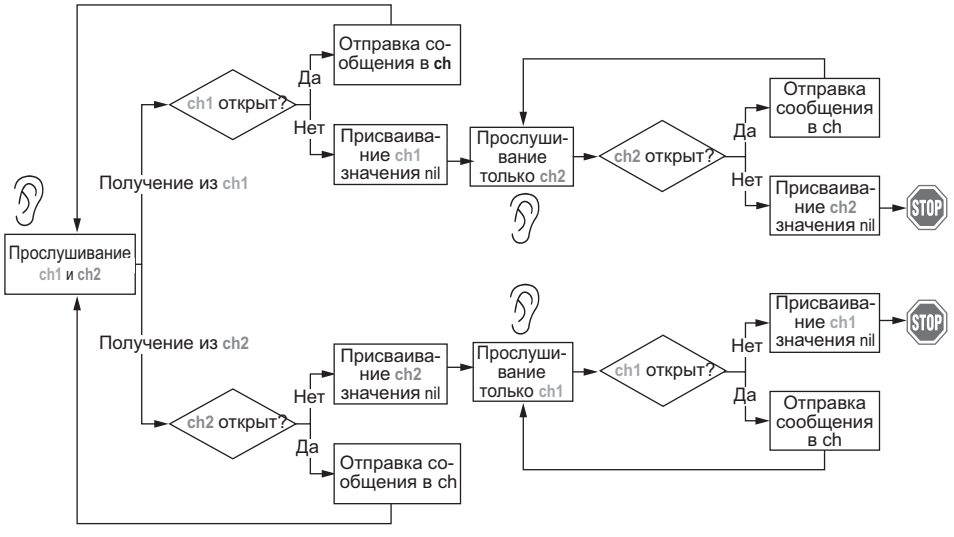


Рис. 9.9. Прием из обоих каналов. Если один из них закрыт, мы присваиваем ему значение nil, чтобы получать данные только из одного канала

В следующем разделе обсудим, какой размер устанавливать при создании канала.

9.7. ОШИБКА #67: ГАДАТЬ НАСЧЕТ РАЗМЕРА КАНАЛА

Канал, созданный с помощью встроенной функции `make`, может быть либо буферизованным, либо небуферизованным. И здесь часто возникают две ошибки: путать, какой тип канала выбрать, а в случае буферизованного канала — гадать, какой его размер задать. Разберемся в этих моментах.

Вспомним основные понятия. Небуферизованный канал — это канал без емкости. Его можно создать, опустив указание размера или указав размер `0`:

```
ch1 := make(chan int)
ch2 := make(chan int, 0)
```

Используя небуферизованный (или *синхронный*) канал, отправитель будет блокироваться до тех пор, пока получатель не получит данные из канала.

И наоборот, буферизованный канал — это канал с емкостью, и он должен быть создан с размером больше или равным `1`:

```
ch3 := make(chan int, 1)
```

В случае с буферизованным каналом отправитель может отправлять сообщения, пока канал не оказывается заполненным. Как только канал заполнится, он будет заблокирован до тех пор, пока горутина получателя не прочтет сообщение. Например:

```
ch3 := make(chan int, 1)
ch3 <-1  ◄——  Отсутствие блокировки
ch3 <-2  ◄——  Блокировка
```

Первая отправка не блокируется, а вторая блокируется, так как на данном этапе канал переполнен.

Сделаем шаг назад и обсудим принципиальные различия между двумя типами каналов. Каналы — это абстракция конкурентности, обеспечивающая связь между горутинами. Но как обстоит дело с синхронизацией? В рамках принципов конкурентности синхронизация означает, что мы можем гарантировать, что несколько горутин в какой-то момент будут находиться в известном состоянии. Например, мьютекс обеспечивает синхронизацию, потому что гарантирует, что только одна горутина может одновременно находиться в критической секции. Что касается каналов:

- Небуферизованный канал обеспечивает синхронизацию. Есть гарантия, что две горутины будут в известном состоянии: одна получает, а другая отправляет сообщение.
- Буферизованный канал не обеспечивает сильной синхронизации. Горутина-производитель может отправить сообщение, и если канал не заполнен, продолжить выполнение. Единственная гарантия заключается в том, что горутина не получит сообщение до того, как оно будет отправлено. Но эта гарантия проистекает только из-за наличия причинно-следственной связи (вы не выпьете кофе, пока не приготовите его).

Важно помнить об этом фундаментальном различии. Оба типа каналов обеспечивают связь, но только один из них обеспечивает синхронизацию. Если нужна синхронизация, используйте небуферизованные каналы. Небуферизованные каналы легче понимать: буферизованные каналы могут приводить к неочевидным взаимоблокировкам, которые в случаях с небуферизованными каналами были бы очевидны сразу.

Есть и другие случаи, когда предпочтительнее небуферизованные каналы: например, в случае канала уведомлений, где уведомление обрабатывается с помощью закрытия канала (`close(ch)`). Здесь использование буферизованного канала не принесет никакой пользы.

Но что, если нужен буферизованный канал? Какой размер ему задать? Значение по умолчанию, которое используют для буферизованных каналов, равно его минимуму, то есть единице. Можно подойти к решению данной задачи со следующей точки зрения: есть ли веская причина *не* использовать значение 1? Вот список возможных случаев, когда нужно использовать другой размер:

- При использовании паттерна вроде пула рабочих процессов (worker pooling), что означает создание фиксированного количества горутин, которые должны отправлять данные в общий канал. Тогда можно связать размер канала с количеством созданных горутин.
- При использовании каналов для решения проблем ограничения скорости выполнения. Например, если нужно ограничить использование ресурсов, устанавливая ограничение на количество запросов, следует настроить размер канала в соответствии с этим ограничением.

Если ваш случай не связан с этими ситуациями, то будьте осторожны с выбором других размеров канала. Довольно часто можно увидеть код, использующий для установки размера канала магические числа:

```
ch := make(chan int, 40)
```

Почему тут вдруг число 40? В чем причина? Почему не 50 или даже не 1000? Значение нужно задавать исходя из веских оснований. Возможно, оно вытекало из какого-то бенчмарка или из тестов производительности. Во многих случаях хорошей идеей будет прокомментировать такое решение и привести обоснование.

Имейте в виду, что решение о точном размере очереди — непростая задача. Прежде всего это вопрос баланса между загрузкой процессора и памяти. Чем меньше это значение, тем с большей конкуренцией за ресурс центрального процессора мы можем столкнуться. Но чем это значение больше, тем больше памяти нужно выделить и зарезервировать.

Еще один важный момент упоминается в документе 2011 года о LMAX Disruptor (Martin Thompson et al.; https://lmax-exchange.github.io/disruptor/files/Disruptor-1.0.pdf):

> *Очереди, как правило, всегда близки к заполнению или почти пусты из-за разницы в темпах работы потребителей и производителей. Они очень редко работают в сбалансированном среднем положении, когда темпы производства и потребления совпадают.*

Поэтому редко можно найти размер канала, который будет стабильно точным, то есть точное значение которого не приведет к слишком большому количеству конфликтов или напрасному выделению памяти.

Вот почему, за исключением вышеописанных случаев, лучше всего начинать с размера канала, равного 1. Если вы сомневаетесь, его всегда можно будет измерить, например, с помощью бенчмарков.

Как и везде в программировании, здесь тоже есть исключения. Поэтому цель данного раздела не в том, чтобы охватить все, а в том, чтобы дать азы понимания того, какой размер использовать при создании каналов. Синхронизация гарантирована при использовании небуферизованных каналов, с буферизованными такой гарантии нет. Кроме того, если нужен буферизованный канал, не забывайте использовать по умолчанию 1 в качестве значения размера канала. Осмотрительно принимайте решение о выборе какого-либо другого значения, а обоснование выбора должно быть закомментировано в коде. И последнее, но не менее важное: помните, что использование буферизованных каналов также может приводить к непредсказуемым взаимоблокировкам, которые легче обнаруживать и выявлять в случаях с небуферизованными каналами.

В следующем разделе обсудим возможные побочные эффекты при форматировании строк.

9.8. ОШИБКА #68: ЗАБЫВАТЬ О ВОЗМОЖНЫХ ПОБОЧНЫХ ЭФФЕКТАХ ПРИ ФОРМАТИРОВАНИИ СТРОК

Форматирование строк — обычная операция для разработчиков, независимо от того, возвращают они ошибку или регистрируют сообщение в логе. Но довольно легко забыть о возможных побочных эффектах форматирования строк при работе в конкурентном приложении. В этом разделе рассмотрим два примера: один (взятый из репозитория etcd) приводит к возникновению гонки данных, а другой — к взаимоблокировке.

9.8.1. Гонка данных в etcd

etcd — это распределенное хранилище ключей и значений, реализованное на Go. Оно используется во многих проектах, включая Kubernetes, для хранения всех данных кластера. Оно предоставляет API для взаимодействия с кластером. Например, интерфейс `Watcher` используется для уведомления об изменении данных:

```
type Watcher interface {
   // Watch отслеживает ключ или префикс. Просмотренные события будут возвращены
   // через возвращаемый канал.
   // ...
   Watch(ctx context.Context, key string, opts ...OpOption) WatchChan
   Close() error
}
```

API использует потоковую передачу gRPC. Вратце: это технология непрерывного обмена данными между клиентом и сервером. Сервер должен вести список всех клиентов, использующих эту функцию. Следовательно, интерфейс `Watcher` реализуется структурой `watcher`, содержащей все активные потоки:

```
type watcher struct {
   // ...

   // streams содержат все активные потоки gRPC, отмеченные значением ctx.
   streams map[string]*watchGrpcStream
}
```

Ключ карты основан на контексте, предоставленном при вызове метода `Watch`:

```
func (w *watcher) Watch(ctx context.Context, key string,
   opts ...OpOption) WatchChan {
   // ...
   ctxKey := fmt.Sprintf("%v", ctx)
   // ...
   wgs := w.streams[ctxKey]
   // ...
```

Форматирование ключа карты в зависимости от заданного контекста

`ctxKey` — это ключ карты, отформатированный исходя из контекста, предоставленного клиентом. При форматировании строки из контекста, созданного со значениями (`context.WithValue`), Go будет считывать все значения, содержащиеся в этом контексте. В этом случае разработчики etcd обнаружили, что контекст, предоставляемый `Watch`, был контекстом, содержащим изменяемые в некоторых условиях значения (например, указатель на структуру). Они обнаружили случай, когда одна горутина обновляла одно из значений контекста, а другая выполняла `Watch` и считывала все значения в этом контексте. Это приводило к гонке данных.

Суть исправления (https://github.com/etcd-io/etcd/pull/7816) — не полагаться на `fmt.Sprintf` для форматирования ключа карты, что предотвращает обход и чтение цепочки обернутых значений в контексте. Вместо этого была реализована специальная функция `streamKeyFromCtx` для извлечения ключа из определенного значения контекста, которое не было изменяемым.

> **ПРИМЕЧАНИЕ** Потенциально изменяемое значение в контексте может создать дополнительную сложность в деле предотвращения гонки данных. Вероятно, это решение, касающееся дизайна кода, и к нему нужно подходить очень тщательно.

Этот пример показывает, что нужно быть осторожными с побочными эффектами от форматирования строк в конкурентных приложениях: в этом примере таким эффектом стала гонка данных. В следующем примере увидим побочный эффект, приводящий к взаимоблокировке.

9.8.2. Взаимоблокировка

Предположим, что мы работаем со структурой `Customer`, доступ к которой возможен конкурентно. Используем `sync.RWMutex` для защиты доступа, будь то чтение или запись. Реализуем метод `UpdateAge` для обновления возраста клиента и проверки, что он имеет положительное значение. Также реализуем интерфейс `Stringer`.

Заметите ли вы проблему со структурой `Customer` в этом коде, предоставляющем метод `UpdateAge` и реализующем интерфейс `fmt.Stringer`?

```
type Customer struct {
   mutex sync.RWMutex  <-- Использование sync.RWMutex для защиты конкурентных доступов
   id string
   age int
}
func (c *Customer) UpdateAge(age int) error {
   c.mutex.Lock()  <-- Блокировка и откладывание разблокировки по мере обновления Customer
   defer c.mutex.Unlock()
   if age < 0 {  <-- Возврат ошибки, если значение age отрицательно
      return fmt.Errorf("переменная age для customer %v должна быть
                        положительным числом", c)
   }
   c.age = age
   return nil
}
```

```
func (c *Customer) String() string {
   c.mutex.RLock()  ◄——  Блокировка и откладывание разблокировки по мере чтения Customer
   defer c.mutex.RUnlock()
   return fmt.Sprintf("id %s, age %d", c.id, c.age)
}
```

Проблема здесь неочевидна. Если указанный возраст отрицательный, то мы возвращаем ошибку. Поскольку ошибка форматируется с использованием директивы `%s` на получателе, будет вызван метод `String` для форматирования объекта `Customer`. Но поскольку метод `UpdateAge` уже захватывает блокировку мьютекса, метод `String` не сможет ее захватить (рис. 9.10).

Рис. 9.10. Выполнение `UpdateAge`, если переменная `age` отрицательна

Это приводит к взаимоблокировке. Если все горутины также находятся в спящем состоянии, то это приводит к панике:

```
fatal error: all goroutines are asleep – deadlock!
goroutine 1 [semacquire]:
sync.runtime_SemacquireMutex(0xc00009818c, 0x10b7d00, 0x0)
...
```

Как быть в этой ситуации? Прежде всего она иллюстрирует важность юнит-тестирования. В таком случае можно возразить, что создавать тест с заданием отрицательного возраста не стоит, так как логика в этом случае достаточно проста. Но без надлежащего покрытия тестами эту проблему можно не заметить и пропустить.

Что здесь можно улучшить, так это ограничить область действия блокировки мьютекса. В `UpdateAge` мы сначала получаем блокировку и проверяем, корректен ли ввод. Сделаем наоборот: сначала проверим ввод, и если ввод корректен, получим блокировку. Преимущество этого заключается в уменьшении потенциальных побочных эффектов, но также может повлиять на производительность — блокировка устанавливается только тогда, когда действительно требуется, а не до того, как это выяснится:

```
func (c *Customer) UpdateAge(age int) error {
   if age < 0 {
      return fmt.Errorf("переменная age для customer %v должна быть
                         положительным числом", c)
   }
   c.mutex.Lock()  <-- Блокировка мьютекса только после того, как ввод был проверен
   defer c.mutex.Unlock()
   c.age = age
   return nil
}
```

В нашем случае блокировка мьютекса только после проверки корректности ввода возраста позволяет избежать взаимоблокировок. Если возраст отрицательный, `String` вызывается без предварительной блокировки мьютекса.

В некоторых случаях непросто или вообще невозможно ограничить область блокировки мьютекса. В таких условиях нужно быть крайне осторожными с форматированием строк. Возможно, мы захотим вызвать другую функцию, которая не будет пытаться получить мьютекс, или захотим изменить только способ форматирования ошибки, чтобы она не вызывала метод `String`. Например, следующий код не приводит к взаимоблокировке, потому что идентификатор клиента (customer ID) регистрируется только при прямом доступе к полю `id`:

```
func (c *Customer) UpdateAge(age int) error {
   c.mutex.Lock()
   defer c.mutex.Unlock()
   if age < 0 {
      return fmt.Errorf("age should be positive for customer id %s", c.id)
   }
   c.age = age
   return nil
}
```

Мы видели два конкретных примера: в одном форматируется ключ из контекста, а в другом — возвращается ошибка, которая форматирует структуру. В обоих случаях форматирование строки приводит к проблеме: к гонке данных и взаимоблокировке соответственно. Поэтому в конкурентных приложениях нужно быть очень осторожными и помнить о побочных эффектах форматирования строк.

В следующем разделе обсудим поведение `append` при конкурентных вызовах.

9.9. ОШИБКА #69: СОЗДАВАТЬ СИТУАЦИЮ ГОНКИ ДАННЫХ ИЗ-ЗА ОПЕРАТОРА APPEND

Ранее я говорил, что такое гонка данных и какие могут быть последствия. Теперь посмотрим на срезы и на то, будет ли добавление элемента в срез с помощью оператора `append` вызывать гонку данных.

В следующем примере инициализируем срез и создадим две горутины, которые будут использовать `append` для создания нового среза с дополнительным элементом:

```
s := make([]int, 1)
go func() {   <-- Добавление к s нового элемента в новой горутине
   s1 := append(s, 1)
   fmt.Println(s1)
}()
go func() {   <-- То же самое
   s2 := append(s, 1)
   fmt.Println(s2)
}()
```

Есть ли здесь гонка данных? Нет.

Вспомним основные свойства срезов из главы 3. За любым срезом стоит резервный массив, а сам срез имеет два свойства: длину и емкость. Длина — это количество доступных элементов в срезе, а емкость — это общее количество элементов в резервном массиве. Когда мы используем `append`, поведение зависит от того, заполнен ли срез (длина == емкости). Если да, то для добавления нового элемента среда выполнения Go создает новый резервный массив. В противном случае она добавляет элемент в имеющийся резервный массив.

В этом примере мы с помощью `make([]int, 1)` создаем срез длиной 1 и емкостью 1. Поскольку срез заполнен, использование `append` в каждой горутине возвращает

срез, за которым стоит новый массив. Оператор `append` не изменяет существующий массив, и это не приводит к гонке данных.

Теперь запустим тот же пример с небольшим изменением инициализации `s`. Вместо создания среза длиной 1 создадим его длиной 0 и емкостью 1:

```
s := make([]int, 0, 1)  ◄——  Изменение способа инициализации среза
// То же самое
```

Что можно сказать об этом новом примере? Содержит ли он гонку данных? Да:

```
==================
WARNING: DATA RACE
Write at 0x00c00009e080 by goroutine 10:
...
Previous write at 0x00c00009e080 by goroutine 9:
...
==================
```

Мы создаем срез с помощью `make([]int, 0, 1)`. Следовательно, массив не заполнен. Обе горутины пытаются обновить один и тот же индекс резервного массива (индекс 1), что и вызывает гонку данных.

Как предотвратить гонку данных, если мы хотим, чтобы обе горутины работали над срезом, содержащим начальные элементы из `s` и дополнительный элемент? Одно из решений — создание копии `s`:

```
s := make([]int, 0, 1)
go func() {
   sCopy := make([]int, len(s), cap(s))
   copy(sCopy, s)  ◄——  Создание копии для использования append на копии среза
   s1 := append(sCopy, 1)
   fmt.Println(s1)
}()
go func() {
   sCopy := make([]int, len(s), cap(s))
   copy(sCopy, s)  ◄——  То же самое
   s2 := append(sCopy, 1)
   fmt.Println(s2)
}()
```

Обе горутины делают копию среза. Затем применяют `append` к этой копии, а не к исходному срезу. Это предотвращает гонку данных, поскольку обе горутины работают с изолированными данными.

При работе со срезами в конкурентных контекстах помните, что применение `append` к срезам не всегда исключает гонку. В зависимости от конкретного

среза и от того, заполнен ли он, поведение будет меняться. Если срез заполнен, append выполняется без гонок. В противном случае несколько горутин могут конкурировать за обновление одного и того же индекса массива, что приведет к гонке данных.

Гонки данных в срезах и картах

Как гонки данных влияют на срезы и карты? Когда есть несколько горутин, верно следующее:

Доступ к одному и тому же индексу среза со стороны по крайней мере одной горутины, обновляющей соответствующее значение, — это ситуации с гонкой данных. Горутины обращаются к одному и тому же месту в памяти.

Доступ к различным индексам срезов независимо от операции не вызывает гонку данных. Разные индексы означают разные ячейки памяти.

Доступ к одной и той же карте (независимо от того, тот же это или другой ключ) со стороны по крайней мере одной горутины, обновляющей эту карту, ведет к гонке данных. Чем это отличается от структуры данных среза? Как я говорил в главе 3, карта — это массив сегментов, и каждый сегмент — это указатель на массив, состоящий из пар «ключ — значение». Алгоритм хеширования используется для определения индекса массива сегмента. Поскольку этот алгоритм содержит в себе некоторый элемент случайности во время инициализации карты, одно выполнение может привести к тому же индексу массива, а другое — нет. Детектор гонок обрабатывает этот случай, выдавая предупреждение независимо от того, происходит ли фактическая гонка данных.

Не должно быть разных реализаций в зависимости от того, заполнен ли срез. Учитывайте, что применение append к общему срезу в конкурентных приложениях может привести к гонке данных. Следовательно, этого следует избегать.

Теперь обсудим типичную ошибку с неточными блокировками мьютексов поверх срезов и карт.

9.10. ОШИБКА #70: НЕВЕРНО ИСПОЛЬЗОВАТЬ МЬЮТЕКСЫ СО СРЕЗАМИ И КАРТАМИ

При работе в конкурентных контекстах, где данные одновременно и изменяемы, и используются совместно, часто приходится реализовывать защищенный доступ к структурам данных с помощью мьютексов. Распространенной ошибкой

является неверное использование мьютексов при работе со срезами и картами. Рассмотрим пример и разберемся с потенциальными проблемами.

Мы реализуем структуру `Cache` для кэширования балансов клиентов. Эта структура будет содержать карту балансов для каждого идентификатора клиента (customer ID) и мьютекс для защиты конкурентных доступов:

```
type Cache struct {
   mu       sync.RWMutex
   balances map[string]float64
}
```

> **ПРИМЕЧАНИЕ** В этом решении используется `sync.RWMutex`, чтобы разрешить несколько операций чтения при отсутствии операций записи.

Затем добавляем метод `AddBalance`, который изменяет карту `balances`. Изменение выполняется в критической секции (внутри блокировки и разблокировки мьютекса):

```
func (c *Cache) AddBalance(id string, balance float64) {
   c.mu.Lock()
   c.balances[id] = balance
   c.mu.Unlock()
}
```

Нужно реализовать метод расчета среднего баланса для всех клиентов. Одна из идей состоит в том, чтобы обрабатывать минимальную критическую секцию так:

```
func (c *Cache) AverageBalance() float64 {
   c.mu.RLock()
   balances := c.balances   <-- Создание копии карты balances
   c.mu.RUnlock()
   sum := 0.
   for _, balance := range balances {   <-- Выполнение цикла с копией,
      sum += balance                        вне критической секции
   }
   return sum / float64(len(balances))
}
```

Сначала мы создаем копию карты в локальной переменной `balances`. В критической секции выполняется только копирование для того, чтобы выполнять цикл, обращаясь к каждому балансу, с целью расчета среднего, за пределами критической секции. Работает ли это решение?

Если запустить тест с использованием флага `-race` с двумя конкурентными горутинами, одна из которых вызывает `AddBalance` (тем самым изменяя

`balances`), а другая вызывает `AverageBalance`, происходит гонка данных. В чем проблема?

Внутреннее устройство карты представляет собой структуру `runtime.hmap`, содержащую в основном метаданные (например, счетчик) и указатель, ссылающийся на сегменты данных. Итак, `balances := c.balances` не копирует фактические данные. Тот же принцип работает и со срезом:

```
s1 := []int{1, 2, 3}
s2 := s1
s2[0] = 42
fmt.Println(s1)
```

Вывод `s1` возвращает `[42 2 3]`, несмотря на то что мы изменили `s2`. Причина в том, что действие `s2 := s1` создает новый срез: `s2` имеет ту же длину и такую же емкость и поддерживается тем же массивом, что и `s1`.

Возвращаясь к нашему примеру, отметим, что назначаем `balances` новую карту, ссылающуюся на те же сегменты данных, что и `c.balances`. Тем временем две горутины выполняют операции с одним и тем же набором данных, и одна из них изменяет его. Следовательно, это гонка данных. Как это исправить? Есть два варианта.

Если операция внутри цикла не тяжелая (в данном случае это так, поскольку выполняется только приращение значения), нужно защитить всю функцию:

```
func (c *Cache) AverageBalance() float64 {
   c.mu.RLock()
   defer c.mu.RUnlock()  ◄── Разблокировка при возврате из функции
   sum := 0.
   for _, balance := range c.balances {
      sum += balance
   }
   return sum / float64(len(c.balances))
}
```

Критическая секция теперь охватывает всю функцию, включая итерации внутри цикла. Это предотвращает гонку данных.

А вот если операция итерации не легковесная (то есть тяжелая), то нужно работать с актуальной копией данных и защищать только эту копию:

```
func (c *Cache) AverageBalance() float64 {
   c.mu.RLock()
   m := make(map[string]float64, len(c.balances))  ◄── Копирование карты
   for k, v := range c.balances {
```

```
      m[k] = v
   }
   c.mu.RUnlock()
   sum := 0.
   for _, balance := range m {
      sum += balance
   }
   return sum / float64(len(m))
}
```

Как только мы сделали глубокую копию, мы освобождаем мьютекс. Итерации выполняются на копии за пределами критической секции.

Поразмышляем над этим решением. Нужно перебрать в цикле все значения карты дважды: один раз для копирования и один раз для выполнения операций (в данном случае для инкремента). Но в критической секции находится только копия карты. Следовательно, это решение может подойти тогда и только тогда, когда операция не является *быстрой*. Например, если операция требует обращения к внешней базе данных, это решение, вероятно, будет более эффективным. Невозможно определить порог при выборе того или иного решения, так как выбор зависит от количества элементов и среднего размера структуры.

Так что будьте осторожны с границами блокировки мьютекса. В этом разделе я показал, почему присвоения какой-то карте значения существующей карты (или существующего среза) недостаточно для защиты от гонки данных. Новая переменная, будь то карта или срез, поддерживается тем же набором данных. Есть два основных решения для предотвращения этого: защитить всю функцию или работать с копией актуальных данных. Во всех случаях будьте внимательны при создании кода критических секций и убедитесь, что их границы точно определены.

Давайте поговорим об ошибке при использовании `sync.WaitGroup`.

9.11. ОШИБКА #71: НЕПРАВИЛЬНО ИСПОЛЬЗОВАТЬ SYNC.WAITGROUP

`sync.WaitGroup` — это механизм ожидания завершения `n` операций. Как правило, его используют, чтобы дождаться завершения `n` горутин. Вспомним сначала публичный API, затем рассмотрим довольно частую ошибку, приводящую к недетерминированному поведению.

Группа ожидания может быть создана с нулевым значением `sync.WaitGroup`:

```
wg := sync.WaitGroup{}
```

`sync.WaitGroup` содержит внутри себя счетчик, по умолчанию инициализированный нулевым значением. Мы можем увеличивать значение этого счетчика с помощью метода `Add(int)` и уменьшать его с помощью `Done()` или `Add` с отрицательным значением аргумента. Если мы хотим дождаться, когда счетчик станет равным 0, то нужно использовать блокирующий метод `Wait()`.

> **ПРИМЕЧАНИЕ** Счетчик не может быть отрицательным, иначе горутина запаникует.

В следующем примере мы инициализируем группу ожидания, запустим три горутины, которые будут обновлять счетчик атомарно, а затем дождемся их завершения. Подождем, когда эти три горутины выведут значение счетчика (которое должно быть равно 3). Как вы думаете, в чем проблема этого кода?

```
wg := sync.WaitGroup{}
var v uint64
for i := 0; i < 3; i++ {
   go func() {                                  <-- Создается горутина
      wg.Add(1)                                 <-- Увеличивается значение счетчика группы ожидания
      atomic.AddUint64(&v, 1)                   <-- Атомарно увеличивается значение переменной v
      wg.Done()
   }()
}
wg.Wait()                                       <-- Ожидание до тех пор, пока все горутины не увеличат
fmt.Println(v)                                      переменную v перед выводом ее на печать
```

Если мы запустим этот пример, то получим какое-то недетерминированное значение: код может вывести любое значение от 0 до 3. Кроме того, если мы включим флаг `-race`, Go даже поймает гонку данных. Почему так, если мы используем для обновления `v` пакет `sync/atomic`? Что не так с этим кодом?

Проблема в том, что `wg.Add(1)` вызывается во вновь созданной, а не в родительской горутине. Следовательно, нет никакой гарантии, что мы указали группе ожидания, что хотим подождать завершения трех горутин перед вызовом `wg.Wait()`.

Рисунок 9.11 показывает возможный сценарий ситуации, когда код выводит `2`. В этом сценарии основная горутина запускает три другие. Но последняя горутина выполняется после того, как две первые уже вызвали `wg.Done()`, поэтому родительская горутина уже разблокирована. Следовательно, в этом сценарии,

когда главная горутина читает переменную v, она оказывается равной 2. Детектор гонки также может обнаруживать небезопасный доступ к v.

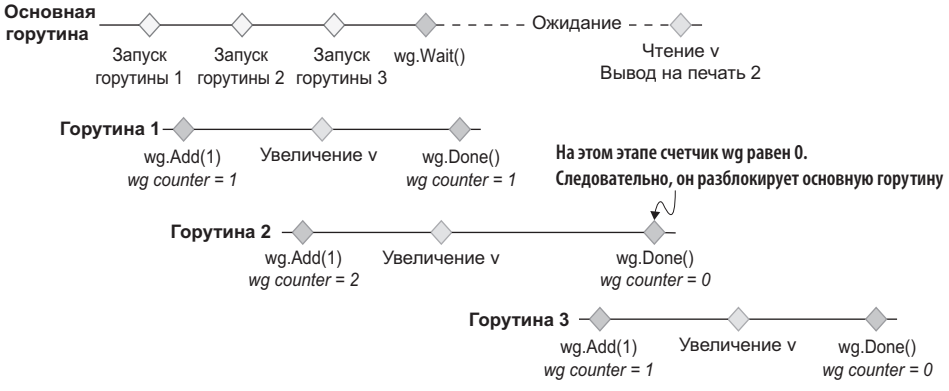


Рис. 9.11. Последняя горутина вызывает `wg.Add(1)` после того, как основная горутина уже разблокирована

При работе с горутинами важно помнить, что их выполнение не является детерминированным без синхронизации. Например, следующий код выведет либо ab, либо ba:

```
go func() {
   fmt.Print("a")
}()
go func() {
   fmt.Print("b")
}()
```

Обе горутины могут быть назначены разным потокам, и нет никакой определенности насчет того, какой поток будет выполняться первым.

CPU должен использовать *барьер памяти* (memory fence, или memory barrier), чтобы обеспечить порядок. Go предоставляет различные методы синхронизации для реализации барьеров памяти: например, `sync.WaitGroup` включает отношение между `wg.Add` и `wg.Wait` типа «происходит до».

Возвращаясь к нашему примеру, есть два варианта решения проблемы. Во-первых, перед циклом с 3 мы можем вызвать `wg.Add`:

```
wg := sync.WaitGroup{}
var v uint64
wg.Add(3)
```

```
for i := 0; i < 3; i++ {
   go func() {
      // ...
   }()
}
// ...
```

Во-вторых, мы можем вызывать `wg.Add` перед запуском дочерних горутин во время каждой итерации цикла:

```
wg := sync.WaitGroup{}
var v uint64
for i := 0; i < 3; i++ {
   wg.Add(1)
   go func() {
      // ...
   }()
}
// ...
```

Оба решения вполне хороши. Если значение, которое мы хотим в итоге установить для счетчика группы ожидания, известно заранее, первое решение избавляет нас от необходимости вызывать `wg.Add` несколько раз. Но нужно убедиться, что везде используется один и тот же счетчик, чтобы избежать неявных ошибок.

Будьте осторожны и не допускайте эту ошибку. При использовании `sync.WaitGroup` операция `Add` должна быть выполнена до запуска горутины в родительской горутине, в то время как операция `Done` должна выполняться внутри горутины.

В следующем разделе обсудим другой примитив пакета `sync`: `sync.Cond`.

9.12. ОШИБКА #72: ЗАБЫВАТЬ О SYNC.COND

Среди примитивов синхронизации, входящих в пакет `sync`, `sync.Cond`, вероятно, наименее используемый и понятный. Но он дает ту функциональность, которой нельзя достичь с помощью каналов. В этом разделе рассмотрим пример, показывающий, когда `sync.Cond` может быть полезен, а также его использование.

Пример в этом разделе реализует механизм достижения целей по сбору пожертвований: приложение, которое генерирует оповещения, когда достигаются определенные цели. Будет одна горутина, отвечающая за увеличение баланса («горутина обновления»). Другие горутины будут получать обновления

и выводить сообщение всякий раз, когда будет достигнута конкретная цель («горутины-слушатели»). Например, одна горутина ожидает достижения цели в 10 долларов, а другая — в 15 долларов.

Первое решение использует мьютексы. Горутина обновления каждую секунду отслеживает состояние баланса и увеличивает его. А горутины-слушатели находятся в состоянии постоянного выполнения соответствующих циклов до тех пор, пока заданная в них цель не будет достигнута:

```
type Donation struct {                    ← Создание и инстанцирование структуры Donation,
   mu sync.RWMutex                          содержащей текущий баланс и мьютекс
   balance int
}
donation := &Donation{}
// горутины-слушатели
f := func(goal int) {                     ← Создание замыкания
   donation.mu.RLock()
   for donation.balance < goal {          ← Проверка достижения цели
      donation.mu.RUnlock()
      donation.mu.RLock()
   }
   fmt.Printf("$%d goal reached\n", donation.balance)
   donation.mu.RUnlock()
}
go f(10)
go f(15)
// горутина обновления
go func() {
   for {                                  ← Продолжение увеличения баланса
      time.Sleep(time.Second)
      donation.mu.Lock()
      donation.balance++
      donation.mu.Unlock()
   }
}()
```

Мы защищаем доступ к общей переменной `donation.balance` с помощью мьютекса. Если запустить этот пример, он будет работать так, как и ожидалось:

```
$10 goal reached
$15 goal reached
```

Основная проблема, которая делает эту реализацию ужасной, — цикл активного ожидания (busy loop). Каждая горутина-слушатель продолжает выполнять цикл до тех пор, пока не будет достигнута цель сбора пожертвований, что приводит к трате огромного количества циклов процессора впустую и сильно нагружает его. Поищем лучшее решение.

Сделаем шаг назад. Нужно найти способ, позволяющей горутине обновления сигнализировать каждый раз, когда баланс обновляется. Всегда, когда в Go речь заходит о передаче сигналов, следует рассматривать возможность использования каналов. Попробуем другую версию реализации — с использованием примитива канала:

```
type Donation struct {
   balance int
   ch chan int   <-- Обновление Donation таким образом, что он содержит каналы
}
donation := &Donation{ch: make(chan int)}
// горутины-слушатели
f := func(goal int) {
   for balance := range donation.ch {   <-- Получение обновлений каналов
      if balance >= goal {
         fmt.Printf("$%d goal reached\n", balance)
         return
      }
   }
}
go f(10)
go f(15)
// горутина обновления
for {
   time.Sleep(time.Second)
   donation.balance++
   donation.ch <- donation.balance   <-- Отправка сообщения каждый раз, когда происходит обновление баланса
}
```

Каждая горутина-слушатель получает данные из общего канала. Между тем горутина обновления отправляет сообщения всякий раз, когда баланс обновляется. Это решение выдает такой возможный результат («goal reached» — «цель достигнута»):

```
$11 goal reached
$15 goal reached
```

Первая горутина должна была быть уведомлена, когда баланс достигнет 10, а не 11 долларов. Что же произошло?

Сообщение, отправленное в канал, принимается только одной горутиной. В нашем примере, если первая горутина получает данные из канала раньше второй, то может произойти то, что показано на рис. 9.12.

Режим распределения по умолчанию при множестве горутин, принимающих из общего канала, является циклическим. Это может измениться, если одна горутина не готова получать сообщения (не находится в состоянии ожидания на канале). Тогда Go отправляет сообщение следующей доступной горутине.

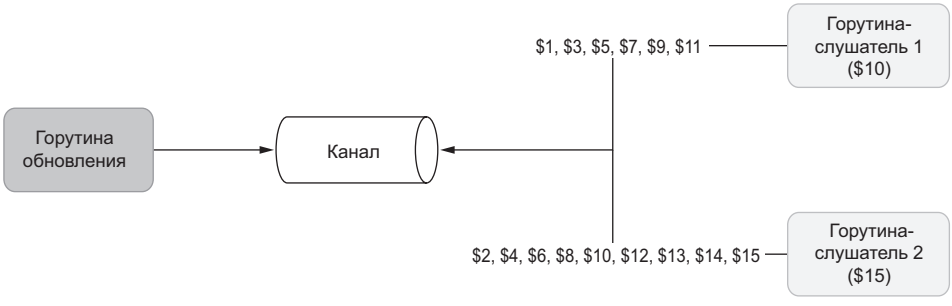


Рис. 9.12. Первая горутина получает сообщение об 1 долларе, затем вторая горутина получает сообщение о 2 долларах, затем первая горутина получает сообщение о 3 долларах и т. д.

Каждое сообщение принимается только одной горутиной. В этом примере первая горутина не получила сообщение `$10`, а вторая его получила. Нескольким горутинам одновременно может быть передано только событие закрытия канала. Но мы не хотим закрывать канал, потому что тогда горутина обновления не сможет отправлять сообщения.

Есть еще одна проблема с использованием каналов: возврат из горутин-слушателей происходит всякий раз, когда соответствующие цели сбора пожертвований достигнуты. Следовательно, горутина обновления должна знать, когда все слушатели перестанут получать сообщения из канала. В противном случае, со временем, канал переполнится и заблокирует отправителя. Возможным решением может стать применение `sync.WaitGroup`, но это все усложнит.

В идеале нужно найти способ многократно передавать уведомления нескольким горутинам всякий раз, когда баланс обновляется. К счастью, в Go есть решение: `sync.Cond`. Сначала немного теории, а затем посмотрим, как можно с помощью этого примитива решить эту задачу.

Согласно официальной документации (https://pkg.go.dev/sync):

> *Cond реализует переменную условия, точку встречи для горутин, ожидающих или объявляющих о возникновении события.*

Условная переменная — это контейнер потоков (в данном случае горутин), ожидающих выполнения определенного условия. В нашем примере условие — это обновление баланса. Горутина обновления рассылает уведомление всякий раз, когда обновляется баланс, а горутина-слушатель ожидает обновления. Кроме

того, sync.Cond использует sync.Locker (*sync.Mutex или *sync.RWMutex) для предотвращения гонки данных. Возможная реализация:

```
type Donation struct {
   cond *sync.Cond  <-- Добавление *sync.Cond
   balance int
}
donation := &Donation{
   cond: sync.NewCond(&sync.Mutex{}),  <-- sync.Cond использует мьютекс
}
// горутины-слушатели
f := func(goal int) {
   donation.cond.L.Lock()
   for donation.balance < goal {
      donation.cond.Wait()     <-- Ожидает выполнения условия (обновления баланса) в рамках блокировки/разблокировки
   }
   fmt.Printf("%d$ goal reached\n", donation.balance)
   donation.cond.L.Unlock()
}
go f(10)
go f(15)
// горутина обновления
for {
   time.Sleep(time.Second)
   donation.cond.L.Lock()
   donation.balance++  <-- Увеличение баланса в рамках блокировки/разблокировки

   donation.cond.L.Unlock()
   donation.cond.Broadcast()  <-- Трансляция факта выполнения условия (баланс обновлен)
}
```

Сначала мы создаем *sync.Cond с помощью sync.NewCond и вводим *sync.Mutex. А что насчет горутин-слушателей и горутины обновления?

Горутины-слушатели остаются в зацикленном состоянии до тех пор, пока не будет достигнут баланс пожертвований. Внутри цикла мы используем метод Wait, который блокирует горутину до выполнения условия.

> **ПРИМЕЧАНИЕ** Удостоверимся, что термин *условие* здесь понятен. В этом контексте мы говорим об обновлении баланса, а не об условии достижения целевого уровня сбора пожертвований. Таким образом, это одна условная переменная, совместно используемая двумя горутинами-слушателями.

Вызов Wait должен происходить внутри критической секции, что может показаться странным. Не помешает ли блокировка другим горутинам дождаться выполнения того же условия?

На самом деле реализация `Wait` такова:

1. Разблокировка мьютекса.
2. Приостановка выполнения горутины и ожидание уведомления.
3. Блокировка мьютекса, когда приходит уведомление.

Горутины-слушатели имеют два критических раздела:

- при доступе к `donation.balance` в `for donation.balance < goal`;
- при доступе к `donation.balance` в `fmt.Printf`.

Таким образом, все обращения к общей переменной `donation.balance` оказываются защищены.

А что насчет горутины обновления? Обновление баланса выполняется в критической секции, чтобы предотвратить гонки данных. Затем мы вызываем метод `Broadcast`, который пробуждает все горутины, ожидающие выполнения условия каждый раз при обновлении баланса.

Поэтому когда мы запустим этот пример, он выведет то, что мы ожидаем:

```
10$ goal reached
15$ goal reached
```

В нашей реализации условная переменная основана на обновляемом балансе. Поэтому каждый раз, когда делается новое пожертвование, переменные горутин-слушателей активируются, чтобы проверить, достигнута ли соответствующая цель сбора пожертвований. Это решение избавляет нас от использования циклов занятости, которые тратят процессорное время при повторных проверках.

Отмечу один возможный недостаток при использовании `sync.Cond`. Когда мы отправляем какое-то уведомление, например, в структуру `chan`, даже если активного его получателя в данный момент нет, сообщение буферизуется. Это гарантирует, что уведомление будет в конце концов получено. Использование `sync.Cond` с методом `Broadcast` пробуждает все горутины, ожидающие в данный момент выполнения условия. Если их нет, уведомление будет пропущено. Это важный принцип, о котором нужно помнить.

Передача сигналов в Go может осуществляться с помощью каналов. Единственное событие, которое смогут одновременно поймать несколько горутин, — это закрытие канала, но оно может произойти только один раз. Поэтому если мы неоднократно отправляем уведомления нескольким горутинам,

`sync.Cond` — хорошее решение такой задачи. Этот примитив основан на переменных условия, которые устанавливают контейнеры потоков, ожидающих выполнения определенного условия. Используя `sync.Cond`, мы можем транслировать сигналы, которые пробуждают все горутины, ожидающие выполнения какого-то условия.

Signal() и Broadcast()

Мы можем пробудить одну горутину, используя `Signal()` вместо `Broadcast()`. С точки зрения семантики это будет тем же самым, что и отправка сообщения в структуру `chan` неблокирующим образом:

```
ch := make(chan struct{})
select {
case ch <- struct{}{}:
default:
}
```

Расширим знания о примитивах конкурентности, используя `golang.org/x` и пакет `errgroup`.

9.13. ОШИБКА #73: НЕ ИСПОЛЬЗОВАТЬ ERRGROUP

Велосипедостроение в любом языке программирования — плохая идея. Часто в кодовых базах переопределяются способы запуска нескольких горутин и агрегирования ошибок. Но в экосистеме Go есть пакет, предназначенный для поддержки этого частого варианта использования. Рассмотрим его и узнаем, почему он должен быть частью инструментария Go-разработчика.

`golang.org/x` — это репозиторий, содержащий расширения стандартной библиотеки. Репозиторий `sync`, являющийся его частью, содержит удобный пакет: `errgroup`.

Допустим, нужно обработать некую функцию, и мы в качестве аргумента получаем какие-то данные, которые хотим использовать для вызова внешнего сервиса. Из-за ограничений мы не можем сделать только один вызов. Каждый раз мы делаем несколько вызовов с каким-то другим подмножеством. Кроме того, все эти вызовы выполняются параллельно (рис. 9.13).

В случае появления во время вызова какой-то одной ошибки мы хотим сделать ее возврат. В случае же появления нескольких ошибок мы хотим вернуть только

одну из них. Напишем основу реализации этого сценария, используя только стандартные примитивы конкурентности:

```
func handler(ctx context.Context, circles []Circle) ([]Result, error) {
   results := make([]Result, len(circles))
   wg := sync.WaitGroup{}
   wg.Add(len(results))
   for i, circle := range circles {
      i := i
      circle := circle
      go func() {
         defer wg.Done()
         result, err := foo(ctx, circle)
         if err != nil {
         // ?
         }
         results[i] = result
      }()
   }
   wg.Wait()
   // ...
}
```

Создание группы WaitGroup для ожидания запуска всех наших горутин

Создание новой переменной i, используемой в горутине (см. ошибку #63)

То же самое с circle

Запуск горутины для каждого Circle

Указание на то, когда горутина завершена

Объединение результатов

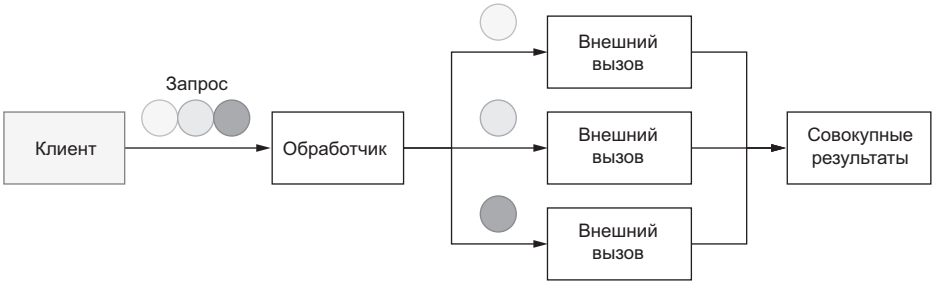


Рис. 9.13. Каждый круг приводит к параллельному вызову

Мы решили использовать `sync.WaitGroup`, чтобы дождаться завершения всех горутин и объединить результаты в срезе. Это один из способов. Другим может быть отправка каждого частичного результата в канал и объединение их в другой горутине. Основная проблема тут заключалась бы в изменении порядка входящих сообщений, если бы это потребовалось. Поэтому мы решили использовать самый простой подход и совместно используемый срез.

ПРИМЕЧАНИЕ Поскольку каждая горутина записывает данные в определенный индекс, в этой реализации нет гонки данных.

Но есть один важный случай, который мы еще не рассматривали. Что, если `foo` (вызов, сделанный в новой горутине) вернет ошибку? Как обработать эту ситуацию? Возможны варианты, в том числе такие:

- Как и в случае со срезом результатов, у нас может быть срез ошибок, который совместно используется в горутинах. Каждая горутина в случае ошибки будет что-то записывать в этот срез. Пришлось бы выполнять итерации по этому срезу в родительской горутине, чтобы определить, произошла ли ошибка (временная сложность O(n)).
- Горутины могут получить доступ к единственной переменной ошибки через общий мьютекс.
- Можно было бы подумать о совместном использовании канала ошибок, тогда родительская горутина получала бы и обрабатывала эти ошибки.

Независимо от выбранного варианта решение усложняется. По этой причине был разработан пакет `errgroup`.

Он экспортирует одну функцию `WithContext`, которая возвращает структуру `*Group` с заданным контекстом. Эта структура обеспечивает синхронизацию, передачу ошибок и отмену контекста для группы горутин и экспортирует только два метода:

- `Go`, чтобы запустить вызов в новой горутине;
- `Wait`, чтобы заблокировать выполнение, пока все горутины не будут завершены. Он возвращает первую ненулевую ошибку, если она есть.

Перепишем код, используя `errgroup`. Сначала импортируем пакет `errgroup`:

```
$ go get golang.org/x/sync/errgroup
```

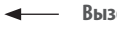
Собственно реализация:

```
func handler(ctx context.Context, circles []Circle) ([]Result, error) {
   results := make([]Result, len(circles))
   g, ctx := errgroup.WithContext(ctx)
   for i, circle := range circles {
      i := i
      circle := circle
      g.Go(func() error {
         result, err := foo(ctx, circle)
         if err != nil {
            return err
         }
```

Создание *errgroup.Group в заданном родительском контексте

Вызов Go для запуска логики обработки ошибки и объединения результатов в новой горутине

```
            results[i] = result
            return nil
         })
      }
      if err := g.Wait(); err != nil {
         return nil, err
      }
      return results, nil
}
```



Сначала мы создаем `*errgroup.Group`, предоставляя для этого родительский контекст. На каждой итерации используем `g.Go` для вызова новой горутины. Этот метод принимает `func() error` в качестве входных данных с замыканием, оборачивающим вызов `foo` и обрабатывающим и результат, и ошибку. Основное отличие от первой версии в том, что если мы получаем ошибку, то возвращаем ее из этого замыкания. Затем `g.Wait` позволяет дождаться завершения выполнения всех горутин.

Это решение проще, чем первое (которое можно посчитать частичным, поскольку мы не обрабатывали ошибку). Здесь не нужно полагаться на дополнительные примитивы конкурентности, а `errgroup.Group` оказывается достаточен для решения задачи.

Еще одно преимущество, которое мы пока не рассмотрели, — это общий контекст. Представим, что нужно инициировать три параллельных вызова:

- первый возвращает ошибку через 1 миллисекунду;
- второй и третий вызовы возвращают результат или ошибку через 5 секунд.

Мы хотим вернуть ошибку, если она вообще появится, а ждать завершения второго и третьего вызовов нет смысла. Использование `errgroup.WithContext` создает общий контекст, используемый во всех параллельных вызовах. Поскольку первый вызов возвращает ошибку через 1 миллисекунду, он отменит контекст и, следовательно, выполнение других горутин. Таким образом, не придется ждать 5 секунд, чтобы вернуть ошибку. Это еще одно преимущество при использовании группы ошибок.

> **ПРИМЕЧАНИЕ** Процесс, вызываемый `g.Go`, должен быть контекстно зависимым. В противном случае отмена контекста не будет иметь никакого эффекта.

Когда нужно запустить несколько горутин и обрабатывать ошибки, а также передавать контекст, стоит подумать, может ли `errgroup` быть решением. Как мы

видели, этот пакет обеспечивает синхронизацию для группы горутин и предоставляет инструмент для обработки ошибок и общих контекстов.

В последнем разделе этой главы обсудим ошибку, которую допускают при копировании типа `sync`.

9.14. ОШИБКА #74: КОПИРОВАТЬ ТИП SYNC

Пакет `sync` предоставляет базовые примитивы синхронизации: мьютексы, условные переменные и группы ожидания. Для всех этих типов есть жесткое правило: копировать их нельзя никогда. Разберемся в причинах и возможных проблемах.

Мы создадим потокобезопасную структуру данных для хранения счетчиков. Он будет содержать `map[string]int`, представляющий собой текущее значение для каждого счетчика. Мы также будем использовать `sync.Mutex`, потому что доступ должен быть защищен. Кроме того, добавим метод `increment` для увеличения заданного имени счетчика:

```
type Counter struct {
   mu sync.Mutex
   counters map[string]int
}
func NewCounter() Counter {   <-- Фабричная функция
   return Counter{counters: map[string]int{}}
}
func (c Counter) Increment(name string) {
   c.mu.Lock()   <-- Увеличение значения счетчика в критической секции
   defer c.mu.Unlock()
   c.counters[name]++
}
```

Логика увеличения выполняется в критической секции: между `c.mu.Lock()` и `c.mu.Unlock()`. Попробуем применить созданный метод, используя параметр `-race` для выполнения следующего примера. Этот пример запускает две горутины и увеличивает их счетчики:

```
counter := NewCounter()
go func() {
   counter.Increment("foo")
}()
go func() {
   counter.Increment("bar")
}()
```

При запуске кода возникнет гонка данных:

```
==================
WARNING: DATA RACE
...
```

Проблема этой реализации в копировании мьютекса. Поскольку получатель `Increment` является значением, всякий раз, когда мы вызываем `Increment`, выполняется копирование структуры `Counter`, которая также копирует мьютекс. Поэтому в общей критической секции приращение не выполняется.

Типы `sync` не должны копироваться. Это правило распространяется на следующие типы:

- `sync.Cond`
- `sync.Map`
- `sync.Mutex`
- `sync.RWMutex`
- `sync.Once`
- `sync.Pool`
- `sync.WaitGroup`

Копирование мьютекса не должно производиться вообще. Какие есть альтернативы? Первая — изменить тип получателя для метода `Increment`:

```
func (c *Counter) Increment(name string) {
   // Такой же код
}
```

Изменение типа получателя позволяет избежать копирования `Counter` при вызове `Increment`. Поэтому его внутренний мьютекс не копируется.

Если нужно сохранить тип получателя (значение), то второй вариант состоит в том, чтобы изменить тип поля `mu` в `Counter` на указатель:

```
type Counter struct {
   mu *sync.Mutex  <-- Изменение типа mu
   counters map[string]int
}
func NewCounter() Counter {
   return Counter{
```

```
        mu: &sync.Mutex{},  ◄——  Изменение способа инициализации mu
        counters: map[string]int{},
    }
}
```

Если у `Increment` есть получатель типа «значение», он все равно копирует структуру `Counter`. Но поскольку `mu` теперь указатель, он будет выполнять только копирование указателя, а не фактическую копию `sync.Mutex`. Поэтому такое решение тоже предотвращает гонку данных.

> **ПРИМЕЧАНИЕ** Мы также изменили способ инициализации `mu`. Поскольку `mu` — указатель, если мы опустим его при создании счетчика, он будет инициализирован нулевым значением указателя: `nil`. Это вызовет панику в горутине при вызове `c.mu.Lock()`.

С проблемой непреднамеренного копирования поля из пакета `sync` можно столкнуться в следующих случаях:

- Вызов метода с получателем типа «значение» (как мы видели).
- Вызов функции с аргументом `sync`.
- Вызов функции с аргументом, содержащим какое-то поле из пакета `sync`.

В каждом таком случае нужно быть настороже. Отмечу, что некоторые линтеры могут отлавливать эту проблему, например, с помощью `go vet`:

```
$ go vet .
./main.go:19:9: Increment passes lock by value: Counter contains sync.Mutex
```

Эмпирическое правило: всякий раз, когда несколько горутин должны получить доступ к общему элементу `sync`, нужно убедиться, что все они обращаются к одному и тому же экземпляру. Это правило применяется ко всем типам, определенным в пакете `sync`. Использование указателей — способ решить эту проблему: может быть либо указатель на какой-то элемент `sync`, либо указатель на структуру, содержащую элемент `sync`.

ИТОГИ

- Важно понимать те условия, при которых контекст может быть отменен: например, обработчик HTTP отменяет контекст после отправки ответа.

- Чтобы избежать утечек, помните, что всякий раз, когда горутина запускается, у вас должно быть четкое понимание, как и когда ее следует остановить.
- Чтобы избежать ошибок с горутинами и переменными цикла, создавайте локальные переменные или вызывайте функции вместо замыканий.
- Понимание того, что оператор `select` при использовании с несколькими каналами выбирает указанные варианты случайным образом, позволяет не делать неверных допущений, которые приводят к неочевидным ошибкам конкурентности.
- Отправляйте уведомления, используя тип `chan struct{}`.
- Использование нулевых каналов должно быть частью вашего конкурентного инструментария, поскольку позволяет, например, *исключать* какие-то варианты из операторов `select`.
- Тщательно все обдумайте, прежде чем решить, какой тип канала использовать в зависимости от конкретной проблемы. Только небуферизованные каналы дают надежные гарантии синхронизации.
- Должна быть веская причина для указания отличного от единицы размера канала в случае, если этот канал буферизованный.
- Форматирование строк может привести к вызову существующих функций, а это значит, что нужно следить за возможным появлением взаимоблокировок и других гонок данных.
- Вызов `append` не всегда исключает гонку данных, поэтому его не следует использовать одновременно в общем срезе.
- Понимание того, что срезы и карты являются указателями, позволяет предотвращать гонки данных.
- Чтобы правильно использовать `sync.WaitGroup`, перед запуском горутин вызывайте метод `Add`.
- Повторяющиеся уведомления нескольким горутинам можно отправлять с помощью `sync.Cond`.
- Синхронизировать группу горутин и обрабатывать ошибки и контексты можно с помощью пакета `errgroup`.
- Типы `sync` не подлежат копированию.

10 Стандартная библиотека

В этой главе:

- ✓ Как правильно задать значение промежутка времени
- ✓ Понимание потенциально возможных утечек памяти при использовании `time.After`
- ✓ Как избежать распространенных ошибок при обработке JSON и SQL
- ✓ Закрытие временных ресурсов
- ✓ Важность оператора `return` в HTTP-обработчиках
- ✓ Почему в production-grade приложениях не должны использоваться HTTP-клиенты и серверы по умолчанию

Стандартная библиотека Go представляет собой набор основных пакетов, расширяющих возможность языка. Например, на Go можно разрабатывать HTTP-клиенты или серверы, обрабатывать данные JSON или писать программы взаимодействия с базами данных SQL. Все эти возможности предоставляются стандартной библиотекой. Но при ее использовании легко можно допускать

ошибки. Кроме того, разработчики могут не до конца понимать ее поведение, что ведет к ошибкам и написанию приложений, которые нельзя считать production-grade. Рассмотрим некоторые типичные ошибки при использовании стандартной библиотеки.

10.1. ОШИБКА #75: НЕПРАВИЛЬНО ЗАДАВАТЬ ПРОМЕЖУТОК ВРЕМЕНИ

Стандартная библиотека предоставляет общие функции и методы, которые принимают `time.Duration` (продолжительность времени). Но поскольку `time.Duration` — это псевдоним для типа `int64`, новички в языке Go могут запутаться и указывать неправильную продолжительность. Например, программисты с опытом работы на Java или JavaScript привыкли передавать числовые типы.

Создадим новый `time.Ticker`, который будет каждую секунду передавать тиканье часов:

```
ticker := time.NewTicker(1000)
   for {
      select {
      case <-ticker.C:
         // Какие-то действия
   }
}
```

Если мы запустим этот код, то заметим, что тики будут выдаваться не каждую секунду, а каждую микросекунду.

Поскольку `time.Duration` основан на типе `int64`, предыдущий код является допустимым, так как для `int64` число 1000 является также допустимым значением. Но функция `time.Duration` возвращает время, прошедшее между какими-то двумя моментами, в *наносекундах*. Поэтому в вышеприведенном примере `NewTicker` была задана равной длительности в 1000 наносекунд = 1 микросекунде.

Эта ошибка встречается очень часто. Стандартные библиотеки для Java или JavaScript зачастую требуют указывать соответствующую продолжительность в миллисекундах.

Более того, если мы хотим специально создать `time.Ticker` с интервалом в 1 микросекунду, то не должны передавать `int64` напрямую. Чтобы избежать возможной путаницы, всегда используйте `time.Duration` API:

```
ticker = time.NewTicker(time.Microsecond)
// Или
ticker = time.NewTicker(1000 * time.Nanosecond)
```

Это не самая сложная ошибка в этой книге, но разработчики, знакомые с другими языками, могут легко попасть в ловушку, полагая, что для функций и методов в пакете `time` требуются миллисекунды. Мы должны помнить об использовании API `time.Duration` и указывать типом единицы времени `int64`.

Теперь обсудим ошибку при использовании пакета `time` с `time.After`.

10.2. ОШИБКА #76: TIME.AFTER И УТЕЧКИ ПАМЯТИ

`time.After(time.Duration)` — это удобная функция, которая возвращает канал и ждет, пока истечет указанное время, прежде чем отправить в этот канал некоторое сообщение. Обычно она используется в конкурентном коде. В противном случае, если мы хотим приостановить выполнение программы на протяжении заданного промежутка времени, можно использовать `time.Sleep(time.Duration)`. Преимущество `time.After` заключается в том, что ее можно использовать для реализации таких сценариев, как: «Если я не получу никакого сообщения из этого канала в течение 5 секунд, то я...». Но в кодовых базах часто встречаются вызовы `time.After` в циклах, что может быть основной причиной утечек памяти.

Рассмотрим пример, где реализуем функцию, которая многократно получает сообщения из канала. Мы также хотим занести в журнал какую-то предупреждающую запись, если в течение более 1 часа не получено никаких сообщений. Вот возможная реализация:

```
func consumer(ch <-chan Event) {
   for {
      select {
      case event := <-ch:  ◄—— Обработка события
         handle(event)
      case <-time.After(time.Hour):  ◄—— Увеличение значения счетчика времени простоя
         log.Println("warning: no messages received")
      }
   }
}
```

Здесь `select` используется в двух случаях: при получении сообщения из `ch` и через 1 час, если за этот промежуток не пришло никаких сообщений (`time.After` оценивается во время каждой итерации, поэтому показатель тайм-аута каждый

раз *сбрасывается*). На первый взгляд код выглядит нормально. Но он может вызвать проблемы с использованием памяти.

Как мы уже говорили, `time.After` возвращает канал. Кто-то может ожидать, что этот канал будет закрываться во время каждой итерации цикла, но нет. Ресурсы, созданные `time.After` (включая канал), освобождаются по истечении времени ожидания (тайм-аута) и занимают место в памяти до тех пор, пока это освобождение не произойдет. Каков объем этого места? В Go 1.15 при каждом вызове `time.After` используется около 200 байт памяти. Если мы получаем значительный объем сообщений, например 5 миллионов в час, то приложение будет потреблять 1 Гбайт памяти только для хранения ресурсов `time.After`.

Можно ли решить эту проблему, программно закрывая канал во время каждой итерации? Нет. Возвращаемый канал представляет собой `<-chan time.Time`, это означает, что он работает только на прием и не может быть закрыт.

Есть несколько вариантов действий. Первый: использовать контекст вместо `time.After`:

```
func consumer(ch <-chan Event) {
   for {                                                    Основной цикл
      ctx, cancel := context.WithTimeout(context.Background(), time.Hour)
      select {                                                          Создание контекста
      case event := <-ch:                                               с тайм-аутом
         cancel()                  Отмена контекста, если мы
         handle(event)             получаем какое-то сообщение
      case <-ctx.Done():                                    Отмена контекста
         log.Println("warning: no messages received")
      }
   }
}
```

Недостаток этого подхода в том, что приходится заново создавать контекст во время каждой итерации цикла. Создание контекста — не самая легковесная операция в Go: для этого требуется целый канал. Как же улучшить код?

В основе второго варианта — использование функции `time.NewTimer` из пакета `time`. Эта функция создает структуру `time.Timer`, которая экспортирует следующее:

- Поле `C`, которое является внутренним каналом таймера.
- Метод `Reset(time.Duration)` для сброса отсчета продолжительности периода времени.
- Метод `Stop()` для остановки таймера.

Внутри time.After

Следует отметить, что `time.After` также зависит от `time.Timer`. Однако она возвращает только поле `C`, поэтому здесь нет доступа к методу `Reset`:

```
package time
func After(d Duration) <-chan Time {
   return NewTimer(d).C  <-- Создание нового time.Timer и возврат поля канала
}
```

Посмотрим на реализацию новой версии, в которой используется `time.NewTimer`:

```
func consumer(ch <-chan Event) {
   timerDuration := 1 * time.Hour
   timer := time.NewTimer(timerDuration)  <-- Создание нового таймера
   for {  <-- Основной цикл
      timer.Reset(timerDuration)  <-- Сброс отсчета таймера
      select {
      case event := <-ch:
         handle(event)
      case <-timer.C:  <-- Таймер отсчитал все заданное время
         log.Println("warning: no messages received")
      }
   }
}
```

В этой реализации сохраняется ситуация с повторением одного действия во время каждой итерации цикла — вызов метода `Reset`. Этот метод менее громоздкий, чем необходимость каждый раз создавать новый контекст. Он быстрее и оказывает меньшее давление на сборщик мусора, потому что не требует выделения нового места для кучи. Таким образом, использование `time.Timer` — наилучшее возможное решение исходной проблемы.

ПРИМЕЧАНИЕ В рассмотренном примере для простоты предыдущая горутина не останавливается. Как говорилось в описании ошибки #62 (запускать горутину и не знать, когда ее остановить), это не лучшая практика. В коде production-grade приложений нужно найти какое-то условие выхода — такое, как контекст, который можно отменить. В этом случае мы также должны помнить о том, что надо остановить `time.Timer` с помощью `defer timer.Stop()`, например, сразу после создания `timer`.

Использование `time.After` в цикле — не единственный случай, который может привести к пиковому потреблению памяти. Проблема связана с многократно вызываемым кодом. Цикл — это один из случаев, но использование `time.After`

в функции HTTP-обработчика может привести к тем же последствиям, поскольку эта функция будет вызываться несколько раз.

В общем, использовать `time.After` следует очень осторожно. Помните, что созданные при этом ресурсы будут освобождены только по истечении заданного в таймере времени. Когда вызов `time.After` повторяется (например, в цикле, в функции-потребителе Kafka или в обработчике HTTP), это может привести к пику в потреблении памяти. В таком случае используйте `time.NewTimer`.

В следующем разделе обсуждаются наиболее распространенные ошибки, совершаемые при обработке JSON.

10.3. ОШИБКА #77: ТИПИЧНЫЕ ОШИБКИ ПРИ ОБРАБОТКЕ JSON

В Go реализована отличная поддержка JSON с помощью пакета `encoding/json`. В этом разделе рассмотрим три распространенные ошибки, связанные с кодированием (маршалингом) и декодированием (демаршалингом) данных JSON.

10.3.1. Неожиданное поведение из-за встраивания типов

В разделе, посвященном разбору ошибки #10 (не знать о возможных проблемах со встраиванием типов) мы подробно рассмотрели эти проблемы. В контексте обработки формата JSON обсудим еще одно потенциальное влияние встраивания типов, которое может привести к неожиданным результатам маршалинга/демаршалинга.

В следующем примере создадим структуру `Event`, содержащую идентификатор (ID) и встроенную метку времени:

```
type Event struct {
   ID int
   time.Time   <-- Встроенное поле
}
```

Так как `time.Time` встроен, мы можем получить доступ к его методам непосредственно на уровне события, например, `event.Second()`.

Каково возможное влияние встроенных полей при маршалинге JSON? Узнаем это на следующем примере. Создадим экземпляр `Event` и маршалируем его в JSON. Что выдаст этот код?

```
event := Event{
   ID: 1234,
   Time: time.Now(),
}
b, err := json.Marshal(event)
if err != nil {
   return err
}
fmt.Println(string(b))
```

Имя анонимного поля во время создания экземпляра структуры — это имя структуры (Time)

Мы можем ожидать, что код выведет что-то подобное этой строке:

```
{"ID":1234,"Time":"2021-05-18T21:15:08.381652+02:00"}
```

Но на самом деле мы получим:

```
"2021-05-18T21:15:08.381652+02:00"
```

Как это объяснить? Что случилось с полем `ID` и значением `1234`? Поскольку это поле экспортируется, оно должно было быть маршалировано. Здесь нужно обратить внимание на два момента.

Во-первых, как обсуждалось при разборе ошибки #10, если встроенный тип поля реализует интерфейс, структура, содержащая встроенное поле, также будет реализовывать этот интерфейс. Во-вторых, мы можем изменить поведение маршалинга по умолчанию, заставив тип реализовывать интерфейс `json.Marshaler`. Этот интерфейс содержит единственную функцию `MarshalJSON`:

```
type Marshaler interface {
MarshalJSON() ([]byte, error)
}
```

Вот пример пользовательского маршалинга:

```
type foo struct{}
func (foo) MarshalJSON() ([]byte, error) {
   return []byte(`"foo"`), nil
}
func main() {
   b, err := json.Marshal(foo{})
   if err != nil {
      panic(err)
   }
   fmt.Println(string(b))
}
```

Определение структуры

Реализация метода MarshalJSON

Возврат статического ответа

json.Marshal определяется пользовательской реализацией MarshalJSON

Поскольку мы изменили поведение маршалинга JSON по умолчанию, реализовав интерфейс `Marshaler`, этот код выводит `foo`.

Прояснив эти два момента, вернемся к исходной проблеме со структурой `Event`:

```
type Event struct {
   ID int
   time.Time
}
```

`time.Time` реализует интерфейс `json.Marshaler`. Поскольку `time.Time` является встроенным полем `Event`, компилятор продвигает его методы. Поэтому `Event` также реализует `json.Marshaler`.

Следовательно, при передаче `Event` в `json.Marshal` используется поведение маршалинга, предоставляемое `time.Time`, вместо поведения по умолчанию. Поэтому маршалинг `Event` приводит к игнорированию поля `ID`.

> **ПРИМЕЧАНИЕ** Мы столкнулись бы с подобной проблемой и в противоположном случае, то есть если бы демаршалировали `Event` с помощью `json.Unmarshal`.

Есть два варианта решения этой проблемы. Первый — добавить имя, чтобы поле `time.Time` больше не было встроенным:

```
type Event struct {
   ID int
   Time time.Time  <-- time.Time более не является встроенным типом
}
```

Если мы так маршалируем версию этой структуры `Event`, то она выведет что-то вроде этого:

```
{"ID":1234,"Time":"2021-05-18T21:15:08.381652+02:00"}
```

Если же нужно сохранить поле `time.Time` встроенным, то другой вариант — заставить `Event` реализовать интерфейс `json.Marshaler`:

```
func (e Event) MarshalJSON() ([]byte, error) {
   return json.Marshal(
      struct {  <-- Создание анонимной структуры
         ID int
         Time time.Time
      }{
         ID: e.ID,
         Time: e.Time,
      },
   )
}
```

В этом решении мы реализуем собственный метод `MarshalJSON` при определении анонимной структуры, отражающей структуру `Event`. Но это решение более громоздкое и требует, чтобы метод `MarshalJSON` всегда соответствовал структуре `Event`.

Обращаться со встроенными полями следует с большой осторожностью. Хотя продвижение полей и методов встроенного типа поля иногда может быть удобным, это также может привести к малозаметным ошибкам, когда родительская структура будет реализовывать интерфейсы без явного сигнала. Кроме того, при использовании встроенных полей важно знать о побочных эффектах.

В следующем разделе рассмотрим еще одну распространенную ошибку, связанную с использованием `time.Time`.

10.3.2. JSON и монотонные часы

При маршалинге или демаршалинге структуры, содержащей тип `time.Time`, иногда можно столкнуться с непредвиденными ошибками сравнения. Поэтому будет полезно подробнее изучить `time.Time`, чтобы уметь их предотвращать.

Операционная система поддерживает два разных типа часов: настенные (wall clock) и монотонные (monotonic clock). Сначала рассмотрим оба этих типа, а затем возможные последствия при работе с JSON и `time.Time`.

Настенные часы используются для определения текущего времени суток. Эти часы могут быть изменены. Например, если часы синхронизированы с использованием протокола сетевого времени (NTP), они могут быть переведены по времени вперед или назад. Не следует измерять продолжительность какого-то промежутка времени с помощью настенных часов, иначе можно столкнуться с весьма странным поведением системы, например с отрицательной продолжительностью. Вот почему в ОС есть второй тип — монотонные часы. Монотонные часы гарантируют, что время всегда движется вперед и не подвержено влиянию скачков во времени. На них могут влиять корректировки частоты (например, если сервер обнаруживает, что локальные кварцевые часы идут в другом темпе, чем NTP-сервер), но никогда — скачки времени.

В следующем примере рассмотрим структуру `Event`, содержащую одно поле `time.Time` (невстроенное):

```
type Event struct {
   Time time.Time
}
```

Создаем экземпляр структуры `Event`, маршалируем его в JSON и демаршалируем в другую структуру. Затем сравниваем обе структуры. Выясним, всегда ли процесс маршалинга/демаршалинга симметричен:

```
t := time.Now()  ◄——  Получение текущего значения локального времени
event1 := Event{  ◄——  Создание экземпляра структуры Event
   Time: t,
}
b, err := json.Marshal(event1)  ◄——  Маршалирование в JSON
if err != nil {
   return err
}
var event2 Event
err = json.Unmarshal(b, &event2)  ◄——  Демаршалирование JSON
if err != nil {
   return err
}
fmt.Println(event1 == event2)
```

Что выдаст этот код? `false`, а не `true`. Почему?

Для начала выведем содержимое `event1` и `event2`:

```
fmt.Println(event1.Time)
fmt.Println(event2.Time)

2021-01-10 17:13:08.852061 +0100 CET m=+0.000338660
2021-01-10 17:13:08.852061 +0100 CET
```

Код выводит разное содержимое для `event1` и `event2`. Вывод почти одинаков, за исключением части `m=+0,000338660`. Что это значит?

В Go вместо разделения двух часов на два разных API `time.Time` может содержать как настенные, так и монотонные часы. Когда мы получаем местное время с помощью `time.Now()`, то этот оператор возвращает `time.Time` с показаниями времени обоих часов:

```
2021-01-10 17:13:08.852061 +0100 CET m=+0.000338660
---------------------------------- --------------
              Wall time              Monotonic time
```

И наоборот, когда мы демаршалируем JSON, поле `time.Time` не содержит времени монотонных часов — только настенных. Следовательно, когда мы сравниваем структуры, результат оказывается равен `false` из-за разницы в монотонном времени. По этой же причине мы видим разницу, когда выводим обе структуры. Как решить эту проблему? Есть два варианта.

time.Time и местоположение

Каждое поле `time.Time` связано с `time.Location`, представляющим часовой пояс. Например:

```
t := time.Now() // 2021-01-10 17:13:08.852061 +0100 CET
```

Здесь в качестве местоположения установлен пояс среднеевропейского времени, потому что использовано `time.Now()`, которое возвращает текущее местное время. Результат маршалинга JSON зависит от местоположения. Если этого следует избежать, можно зафиксировать определенное местоположение:

```
location, err := time.LoadLocation("America/New_York")
if err != nil {
   return err
}
t := time.Now().In(location) // 2021-05-18 22:47:04.155755 -0500 EST
```

Задаем «Америка/Нью-Йорк» в качестве текущего места

В качестве альтернативы можно получить текущее время в формате UTC:

```
t := time.Now().UTC() // 2021-05-18 22:47:04.155755 +0000 UTC
```

Когда мы используем оператор `==` для сравнения обоих полей `time.Time`, он сравнивает все поля структуры, включая и часть, соответствующую монотонным часам. Чтобы избежать этого, мы можем использовать метод `Equal`:

```
fmt.Println(event1.Time.Equal(event2.Time))
```

```
true
```

Метод `Equal` не учитывает время монотонных часов, поэтому код выводит `true`. Но в этом случае мы сравниваем только поля `time.Time`, а не родительские структуры `Event`.

Второй вариант — сохранить `==` для сравнения двух структур, но убрать время монотонных часов с помощью метода `Truncate`. Этот метод возвращает округленный в меньшую сторону до кратного заданной длительности результат значения `time.Time`. Можно использовать его, указав нулевую продолжительность, например:

```
t := time.Now()
event1 := Event{
   Time: t.Truncate(0),
}
b, err := json.Marshal(event1)
```

Очистка времени от «монотонной части»

```
if err != nil {
   return err
}
var event2 Event
err = json.Unmarshal(b, &event2)
if err != nil {
   return err
}
fmt.Println(event1 == event2)  ◄—— Сравнение с помощью оператора ==
```

В этой версии кода два поля `time.Time` оказываются равны. Поэтому данный код выводит `true`.

Процесс маршалинга/демаршалинга не всегда симметричен, и в данном случае мы столкнулись со структурой, содержащей `time.Time`. Помните об этом, чтобы не писать ошибочные тесты.

10.3.3. Карта типа any

При демаршалинге данных мы можем иметь дело с картой вместо структуры. Когда ключи и значения не определены, работа с картой, а не со статической структурой дает некоторую гибкость. Но есть правило, о котором следует помнить, чтобы избежать неверных предположений и возможной паники горутины.

Создадим код, который демаршалирует сообщение в карту:

```
b := getMessage()
var m map[string]any
err := json.Unmarshal(b, &m)  ◄—— Задание указателя на карту
if err != nil {
   return err
}
```

Добавим к предыдущему коду следующий JSON:

```
{
   "id": 32,
   "name": "foo"
}
```

Поскольку мы используем общую карту `map[string]any`, она автоматически парсит все различные поля:

```
map[id:32 name:foo]
```

При использовании карты типа `any` важно помнить вот о чем: любое числовое значение, независимо от того, содержит ли оно десятичное число или нет, преобразуется в тип `float64`. Выведем тип `m["id"]` и убедимся в этом:

```
fmt.Printf("%T\n", m["id"])
```

```
float64
```

Убедитесь, что не делаете ошибочных предположений и не ожидаете, что числовые значения без десятичных знаков будут по умолчанию преобразованы в целые числа. Неверные предположения относительно преобразования типов могут привести к панике горутин.

В следующем разделе обсудим распространенные ошибки при написании приложений, взаимодействующих с базами данных SQL.

10.4. ОШИБКА #78: ТИПИЧНЫЕ ОШИБКИ, СВЯЗАННЫЕ С SQL

Пакет `database/sql` предлагает общий интерфейс для SQL-баз данных (или SQL-подобных). При этом довольно часто можно встретиться с некоторыми шаблонными подходами и ошибками при использовании этого пакета. Рассмотрим пять типичных ошибок.

10.4.1. Не знать, что sql.Open не всегда устанавливает соединение с базой данных

При использовании `sql.Open` одно из заблуждений состоит в том, что эта функция должна устанавливать соединения с базой данных:

```
db, err := sql.Open("mysql", dsn)
if err != nil {
   return err
}
```

Но это не всегда так. Согласно документации (https://pkg.go.dev/database/sql),

> *Open может просто проверять правильность и действительность своих аргументов, не создавая соединение с базой данных.*

На самом деле поведение зависит от используемого драйвера SQL. Для некоторых драйверов `sql.Open` не устанавливает соединение: оператор является только подготовкой к этому (например, с `db.Query`). Поэтому первое подключение к базе данных может быть установлено методом ленивого подключения.

Что дает такое знание? Например, в каких-то случаях мы захотим сделать некоторый сервис готовым только после того, как будем знать, что все зависимости правильно настроены и доступны. Если мы этого не знаем, сервис может принимать трафик, несмотря на конфигурацию, содержащую ошибки. Если нужно убедиться, что функция, использующая `sql.Open`, обеспечивает доступность основной базы данных, применяйте метод `Ping`:

```
db, err := sql.Open("mysql", dsn)
if err != nil {
   return err
}
if err := db.Ping(); err != nil {   ◄── Вызов метода Ping вслед за sql.Open
   return err
}
```

`Ping` заставляет код установить соединение, которое гарантирует, что имя источника данных действительно и база данных доступна. Обратите внимание, что альтернативой `Ping` является `PingContext`, который запрашивает дополнительный контекст, сообщающий, когда пинг должен быть отменен или истекает время его ожидания.

Несмотря на возможную контринтуитивность, помните, что `sql.Open` не обязательно устанавливает соединение, и первое соединение может быть открыто лениво. Если нужно протестировать конфигурацию и убедиться, что база данных доступна, то после `sql.Open` нужно вызвать метод `Ping` или `PingContext`.

10.4.2. Забывать о пуле соединений

Подобно тому, как важно понимать, что стандартное поведение клиента и сервера в пакете HTTP может быть неэффективно в продакшене (см. ошибку #81), необходимо хорошо представлять себе, как обрабатываются подключения к базе данных в Go. Функция `sql.Open` возвращает структуру `*sql.DB`. Она представляет собой не какое-то одно соединение с базой данных, а пул таких соединений. Это стоит отметить, чтобы не было соблазна реализовать это вручную. Соединение в пуле может иметь два состояния:

- уже используется (например, другой горутиной, запускающей запрос);
- простаивает (уже создано, но пока не используется).

Важно помнить, что создание пула приводит к четырем доступным параметрам конфигурации, которые мы можем переопределить. Каждый из этих параметров является экспортированным методом `*sql.DB`:

- `SetMaxOpenConns` — максимальное количество открытых подключений к базе данных (значение по умолчанию `unlimited`, неограниченно).
- `SetMaxIdleConns` — максимальное количество неактивных подключений (значение по умолчанию 2).
- `SetConnMaxIdleTime` — максимальное количество времени, в течение которого соединение может быть бездействующим, прежде чем будет закрыто (значение по умолчанию `unlimited`, неограниченно).
- `SetConnMaxLifetime` — максимальное количество времени, в течение которого соединение может оставаться открытым, прежде чем будет закрыто (значение по умолчанию `unlimited`, неограниченно).

На рис. 10.1 показан пример с максимум пятью соединениями. Он имеет четыре текущих соединения: три неактивных и одно используемое. Таким образом, один слот остается доступным для дополнительного подключения. Если приходит новый запрос, он выберет одно из свободных соединений (если оно все еще будет доступно). Если свободных соединений больше нет, пул создаст новое соединение, если доступен дополнительный слот. В противном случае он будет ждать, пока какое-то из соединений не станет доступным.

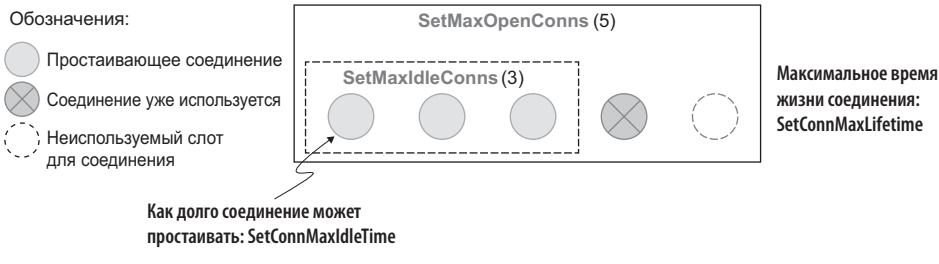


Рис. 10.1. Пул соединений с пятью соединениями

Итак, почему мы должны настраивать эти параметры конфигурации?

- Задание `SetMaxOpenConns` важно для production-grade-приложений. Поскольку значение по умолчанию unlimited, мы должны его задать, чтобы убедиться, что оно соответствует возможностям основной базы данных.

- Значение `SetMaxIdleConns` (по умолчанию: 2) следует увеличить, если приложение генерирует значительное количество одновременных запросов. В противном случае это приложение может сталкиваться с частыми повторными подключениями.
- Задание `SetConnMaxIdleTime` важно, если приложение может столкнуться со всплеском запросов. Когда приложение возвращается в более спокойное состояние, нужно убедиться, что созданные соединения в итоге освобождаются.
- Установка `SetConnMaxLifetime` может быть полезной, например, если мы подключаемся к базе данных, работающей в режиме балансировки нагрузки. Тогда мы убеждаемся, что приложение никогда не будет использовать соединение слишком долго.

Для production-grade-приложений нужно учитывать эти четыре параметра. Можно использовать несколько пулов соединений, если приложение имеет различные сценарии использования.

10.4.3. Не использовать подготовленные операторы

Подготовленный оператор — это функция, реализованная во многих базах данных SQL для выполнения повторяющегося оператора SQL. По сути, оператор SQL отличается тем, что предварительно компилируется и отделяется от данных, которые обрабатывает. У него есть два основных преимущества:

- *Эффективность* — оператор не нужно перекомпилировать (компиляция означает синтаксический анализ + оптимизация + трансляция).
- *Безопасность* — этот подход снижает риск атак путем внедрения кода SQL.

Если какой-то оператор применяется многократно, то следует использовать возможности, предоставляемые подготовленными операторами. Мы также должны их использовать в ненадежных контекстах (например, при открытии доступа к эндпоинту в интернете, где запрос отображается на оператор SQL).

Чтобы использовать подготовленные операторы, вызываем метод `Prepare`, а не `Query` из `*sql.DB`:

```
stmt, err := db.Prepare("SELECT * FROM ORDER WHERE ID = ?")
if err != nil {
   return err
}
rows, err := stmt.Query(id)
// ...
```

Подготовка оператора

Выполнение подготовленного запроса

Мы подготавливаем оператор, а затем выполняем его, передавая ему аргументы. Первым результатом выполнения метода `Prepare` является файл `*sql.Stmt`, который можно будет еще как-то использовать и одновременно запускать. Когда этот оператор становится ненужным, его надо закрыть при помощи метода `Close()`.

> **ПРИМЕЧАНИЕ** Методы `Prepare` и `Query` имеют альтернативы, если требуется предоставление какого-то дополнительного контекста: `PrepareContext` и `QueryContext`.

Из соображений эффективности и безопасности помните о возможности использования подготовленных операторов, когда в этом есть смысл.

10.4.4. Неправильная обработка нулевых значений

Следующая ошибка заключается в неправильной обработке нулевых значений в запросах. Создадим программу, с помощью которой требуется получить данные о возрасте сотрудника и его отделе:

```
rows, err := db.Query("SELECT DEP, AGE FROM EMP WHERE ID = ?", id)
if err != nil {
   return err
}
// Отложить закрытие строк
var (
   department string
   age int
)
for rows.Next() {
   err := rows.Scan(&department, &age)
   if err != nil {
      return err
   }
   // ...
}
```

Выполнение запроса

Сканируем каждую строку

Для выполнения запроса используется `Query`. Затем мы проводим итерацию по строкам и используем `Scan`, чтобы скопировать столбец в переменные в соответствии с указателями `department` и `age`. Если запустить этот код, то при вызове `Scan` получим ошибку:

```
2021/10/29 17:58:05 sql: Scan error on column index 0, name "DEPARTMENT":
converting NULL to string is unsupported
```

Драйвер SQL выдает ошибку, поскольку значение отдела равно `NULL`. Если столбец может принимать значение `NULL`, то есть два варианта предотвращения возврата ошибки оператором `Scan`.

Первый подход — сделать `department` указателем на строку:

```
var (
   department *string  ◄──  Изменение типа со string на *string
   age int
)
for rows.Next() {
   err := rows.Scan(&department, &age)
   // ...
}
```

Мы передаем в `scan` адрес указателя, а не напрямую адрес строкового типа. При этом если это значение равно `NULL`, то `department` будет равен нулю.

Другой подход заключается в использовании одного из типов `sql.NullXXX`, например `sql.Null-String`:

```
var (
   department sql.NullString  ◄──  Изменение типа на sql.NullString
   age int
)
for rows.Next() {
   err := rows.Scan(&department, &age)
   // ...
}
```

`sql.NullString` — это обертка поверх строки. Она содержит два экспортируемых поля: `String` содержит строковое значение, а `Valid` сообщает, не является ли значение строки `NULL`. Доступны следующие обертки:

- `sql.NullString`
- `sql.NullBool`
- `sql.NullInt32`
- `sql.NullInt64`
- `sql.NullFloat64`
- `sql.NullTime`

Оба подхода вполне работоспособны, причем `sql.NullXXX` более явно выражает намерение, как упомянул Расс Кокс, основной мейнтейнер Go (http://mng.bz/rJNX):

*Какой-либо разницы нет. Мы думали, что люди могут чаще хотеть использовать NullString, потому что он очень распространен и, возможно, более явно выражает намерение, чем *string. Но работать будут оба.*

Лучшей практикой для работы со столбцом, значение которого может быть равно `NULL`, является либо обработка его как указателя, либо использование типа `sql.NullXXX`.

10.4.5. Не обрабатывать ошибки итерации строк

Еще один типичный промах разработчиков заключается в пропуске возможных ошибок при итерациях по строкам. Рассмотрим функцию, где неправильно реализуется обработка таких ошибок:

```
func get(ctx context.Context, db *sql.DB, id string) (string, int, error) {
   rows, err := db.QueryContext(ctx,
      "SELECT DEP, AGE FROM EMP WHERE ID = ?", id)
   if err != nil {
      return "", 0, err
   }
   defer func() {
      err := rows.Close()
      if err != nil {
         log.Printf("failed to close rows: %v\n", err)
      }
   }()
   var (
      department string
      age int
   )
   for rows.Next() {
      err := rows.Scan(&department, &age)
      if err != nil {
         return "", 0, err
      }
   }
   return department, age, nil
}
```

Обработка ошибок во время выполнения запроса

Обработка ошибок во время закрытия строк

Обработка ошибок во время сканирования строки

В этой функции мы обрабатываем три ошибки: при выполнении запроса, закрытии строк и сканировании строки. Но этого недостаточно. Нужно знать и помнить, что цикл `for rows.Next() {}` может прерваться, когда больше нет никаких строк или когда возникает ошибка при подготовке следующей строки. После выполнения каждой итерации по строке мы должны вызывать `rows.Err`, чтобы различать эти два случая:

```
func get(ctx context.Context, db *sql.DB, id string) (string, int, error) {
   // ...
   for rows.Next() {
      // ...
   }
   if err := rows.Err(); err != nil {
      return "", 0, err
   }
   return department, age, nil
}
```

Проверка rows.Err для определения того, что цикл на предыдущем шаге остановился из-за какой-то ошибки

Это лучшая практика, о которой нужно помнить. Поскольку `rows.Next` может останавливаться либо когда в цикле проводятся итерации по всем строкам, либо когда возникает ошибка при подготовке следующей строки, нужно проводить проверку `rows.Err` после каждой итерации.

Давайте обсудим следующую частую ошибку: когда разработчики забывают закрывать временные ресурсы.

10.5. ОШИБКА #79: НЕ ЗАКРЫВАТЬ ВРЕМЕННЫЕ РЕСУРСЫ

Довольно часто программисты работают с переходными (или временными) ресурсами, которые нужно в какой-то момент закрывать, например, чтобы избежать утечек на диске или в памяти. Структуры обычно могут реализовывать интерфейс `io.Closer`, чтобы сообщить, что временный ресурс должен быть закрыт. Рассмотрим три типичных примера того, что происходит, когда ресурсы остаются незакрытыми.

10.5.1. Тело HTTP

Для начала обсудим эту проблему в контексте HTTP. В качестве примера рассмотрим собственный метод `getBody`, который отправляет запрос HTTP GET и возвращает тело ответа HTTP. Вот его первая реализация:

```
type handler struct {
   client http.Client
   url string
}
func (h handler) getBody() (string, error) {
   resp, err := h.client.Get(h.url)
   if err != nil {
      return "", err
```

Запрос HTTP GET

```
    }
    body, err := io.ReadAll(resp.Body)  <-- Чтение resp.Body и получение тела как []byte
    if err != nil {
        return "", err
    }
    return string(body), nil
}
```

Мы используем `http.Get` и анализируем ответ с помощью `io.ReadAll`. Реализация этого метода выглядит вполне нормально: он корректно возвращает тело ответа HTTP. Но при этом есть утечка ресурсов. Разберемся где.

`resp` — это тип `*http.Response`. Он содержит поле `Body io.ReadCloser` (`io.ReadCloser` реализует как `io.Reader`, так и `io.Closer`). Это тело должно быть закрыто, если `http.Get` не возвращает ошибку, иначе возникнет утечка ресурсов. В этом случае приложение сохранит ситуацию, когда часть ранее выделенной памяти, которая больше не нужна, но не может быть освобождена сборщиком мусора, останется зарезервированной, и в худших случаях помешает клиентам повторно использовать TCP-соединение.

Самый простой и удобный способ закрыть тела — использовать оператор `defer`:

```
defer func() {
    err := resp.Body.Close()
    if err != nil {
        log.Printf("failed to close response: %v\n", err)
    }
}()
```

В этой реализации закрытие ресурса тела обрабатывается корректно как функция `defer`, которая будет выполняться после получения возврата `getBody`.

> **ПРИМЕЧАНИЕ** На стороне сервера при реализации обработчика HTTP не требуется закрывать тело запроса, поскольку сервер делает это автоматически.

Тело ответа должно быть закрыто независимо от того, читаем ли мы его. Например, если нас интересует только код состояния HTTP, а не его тело, то, чтобы избежать утечки, последнее нужно закрыть, несмотря ни на что:

```
func (h handler) getStatusCode(body io.Reader) (int, error) {
    resp, err := h.client.Post(h.url, "application/json", body)
    if err != nil {
        return 0, err
    }
    defer func() {  <-- Закрытие тела ответа, даже если мы его не читаем
        err := resp.Body.Close()
```

```
      if err != nil {
         log.Printf("failed to close response: %v\n", err)
      }
   }()
   return resp.StatusCode, nil
}
```

Эта функция закрывает тело ответа, даже если оно не прочитано.

Еще одна важная вещь: когда мы закрываем тело, поведение различается в зависимости от того, прочитано что-то из него или нет:

- Если мы закроем тело без чтения, HTTP-транспорт, используемый по умолчанию, может закрыть соединение.
- Если мы закроем тело после чтения, используемый по умолчанию HTTP-транспорт не закроет соединение; следовательно, его можно будет использовать и далее.

Если `getStatusCode` вызывается повторно и мы хотим использовать остающиеся неразорванные соединения, то нужно прочитать тело, даже если его содержание нас не интересует:

```
func (h handler) getStatusCode(body io.Reader) (int, error) {
   resp, err := h.client.Post(h.url, "application/json", body)
   if err != nil {
      return 0, err
   }
   // Закрытие тела ответа
   _, _ = io.Copy(io.Discard, resp.Body)  <-- Чтение тела ответа
   return resp.StatusCode, nil
}
```

В этом примере мы читаем тело исключительно для того, чтобы поддерживать соединение в открытом состоянии. Обратите внимание, что вместо использования `io.ReadAll` мы применяли `io.Copy` к `io.Discard` реализации `io.Writer`. Этот код читает тело, но никак не копирует прочитанное, фактически игнорируя его, что делает такую реализацию более эффективной, чем `io.ReadAll`.

Закрытие ресурса во избежание утечек связано не только с управлением HTTP-телом. Все структуры, реализующие интерфейс `io.Closer`, в какой-то момент должны быть закрыты. Этот интерфейс содержит единственный метод `Close`:

```
type Closer interface {
   Close() error
}
```

Когда закрывать тело ответа

Довольно часто в реализациях тело закрывается не когда ошибка равна `nil`, а когда получен непустой ответ:

```
resp, err := http.Get(url)
if resp != nil {  <-- Если ответ не равен nil…
   defer resp.Body.Close()  <-- …закрыть тело ответа как функцию defer
}
if err != nil {
   return "", err
}
```

Эта реализация необязательна. Она основана на том, что при некоторых условиях (например, при сбое перенаправления) ни `resp`, ни `err` не будут равны `nil`. Но согласно документации Go (https://pkg.go.dev/net/http):

В случае ошибки любой ответ может быть проигнорирован. Ненулевой ответ с ненулевой ошибкой возникает только в случае сбоя CheckRedirect, но даже в этом случае возвращаемый Response.Body уже будет закрыт.

Поэтому проверка `if resp != nil {}` не требуется. Нужно придерживаться первоначального решения, которое закрывает тело в функции `defer`, только если нет никаких ошибок.

Рассмотрим, на что влияет `sql.Rows`.

10.5.2. sql.Rows

`sql.Rows` — это структура, используемая в качестве результата SQL-запроса. Поскольку эта структура реализует `io.Closer`, ее нужно закрывать. В следующем примере закрытие строк не выполняется:

```
db, err := sql.Open("postgres", dataSourceName)
if err != nil {
   return err
}
rows, err := db.Query("SELECT * FROM CUSTOMERS")  <-- Осуществление SQL-запроса
if err != nil {
   return err
}
// Использование строк.
return nil
```

Если вы забудете закрыть строки, произойдет утечка соединения, которая не позволит вернуть это соединение с базой данных обратно в пул соединений.

Мы можем обрабатывать закрытие как функцию `defer`, следующую за блоком `if err != nil`:

```
// Открытие соединения
rows, err := db.Query("SELECT * FROM CUSTOMERS")
if err != nil {
   return err
}
defer func() {
   if err := rows.Close(); err != nil {
      log.Printf("failed to close rows: %v\n", err)
   }
}()
// Использование строк.
```

После вызова `Query` мы должны в конечном итоге закрыть `rows`, чтобы предотвратить утечку соединения, если при этом не возвращается ошибка.

> **ПРИМЕЧАНИЕ** Как обсуждалось в предыдущем разделе, переменная `db` (тип `*sql.DB`) соответствует пулу соединений. Она также реализует интерфейс `io.Closer`. Но как следует из документации, структуру `sql.DB` закрывают редко: считается, что ее предназначение подразумевает открытое состояние в течение долгого времени и совместное ее использование многими горутинами.

Обсудим закрытие ресурсов при работе с файлами.

10.5.3. os.File

`os.File` представляет собой дескриптор открытого файла. Как и `sql.Rows`, в конце концов он должен быть закрыт:

```
f, err := os.OpenFile(filename, os.O_APPEND|os.O_WRONLY, os.ModeAppend)
if err != nil {
   return err
}
defer func() {
   if err := f.Close(); err != nil {
      log.Printf("failed to close file: %v\n", err)
   }
}()
```

В этом примере мы используем `defer`, чтобы отложить вызов метода `Close`. Если в итоге `os.File` не закроется, это не приведет к утечке: файл будет закрыт автоматически, когда `os.File` будет обработан сборщиком мусора. Но лучше

вызывать `Close` явно, потому что мы не знаем, когда будет запущен сборщик мусора в следующий раз (если только мы не запустим его вручную).

У явного вызова `Close` есть еще одно преимущество: активное отслеживание возвращаемой ошибки. Например, этот вызов должен выполняться в случае с файлами, открытыми для записи.

Запись в дескриптор файла не является синхронной операцией. Из соображений производительности данные буферизуются. На странице руководства BSD, где говорится о `close(2)`, упоминается, что закрытие может привести к ошибке в ранее незафиксированной записи (все еще находящейся в буфере), возникшей во время ошибки ввода/вывода. Поэтому если мы хотим записать данные в файл, мы должны передавать любые ошибки, которые возникают при закрытии файла:

```
func writeToFile(filename string, content []byte) (err error) {
   // Открытие файла
   defer func() {  <-- Возврат ошибки close, даже если запись проходит успешно
      closeErr := f.Close()
      if err == nil {
         err = closeErr
      }
   }()
   _, err = f.Write(content)
   return
}
```

В этом примере мы используем именованные аргументы и устанавливаем ошибку в ответе `f.Close`, если запись прошла успешно. И если что-то с этой функцией пойдет не так, то клиенты узнают об этом и смогут отреагировать.

Более того, успешное закрытие доступного для записи файла `os.File` не гарантирует, что он будет записан на диск. Запись может по-прежнему находиться в буфере файловой системы и не сбрасываться на диск. Если сохранение файла является критическим фактором, мы можем использовать метод `Sync()` для фиксации изменения. В этом случае ошибки, берущие свое начало в `Close`, можно смело игнорировать:

```
func writeToFile(filename string, content []byte) error {
   // Открытие файла
   defer func() {
      _ = f.Close()  <-- Игнорирование возможных ошибок
   }()
   _, err = f.Write(content)
   if err != nil {
      return err
   }
   return f.Sync()  <-- Фиксирует запись на диск
}
```

В этом примере содержится функция синхронной записи. Она обеспечивает запись содержимого (контента) на диск перед возвратом. Но тут недостатком является негативное влияние на производительность.

Мы увидели, насколько важным является требование закрывать временные ресурсы и таким образом избегать утечек. Эфемерные ресурсы необходимо закрывать в нужное время и в конкретных ситуациях. Но не всегда сразу становится понятным, что именно нужно будет закрыть. Мы можем получить эту информацию, только внимательно прочитав документацию по API или руководствуясь опытом. Помните, что если структура реализует интерфейс `io.Closer`, в итоге нужно вызвать метод `Close`. И последнее, но не менее важное — надо понимать, что делать, если закрытие не удалось: достаточно ли будет зарегистрировать сообщение и занести его в журнал или нужно передать его дальше? Все зависит от реализации, как было показано на трех примерах выше.

Давайте поговорим о типичных ошибках, связанных с обработкой HTTP, когда разработчики забывают об операторе `return`.

10.6. ОШИБКА #80: ЗАБЫВАТЬ ОБ ОПЕРАТОРЕ RETURN ПОСЛЕ ОТВЕТА НА HTTP-ЗАПРОС

При написании обработчика HTTP легко забыть оператор `return` после ответа на HTTP-запрос. Это может привести к странной ситуации, когда мы должны были бы остановить обработчик после ошибки, но не сделали этого.

Рассмотрим пример:

```
func handler(w http.ResponseWriter, req *http.Request) {
   err := foo(req)
   if err != nil {
      http.Error(w, "foo", http.StatusInternalServerError)  <-- Обработка ошибки
   }
   // ...
}
```

Если `foo` возвращает ошибку, то она будет обработана с помощью `http.Error`, который отвечает на запрос сообщением об ошибке `foo` и внутренней ошибкой сервера 500. Проблема в том, что если мы войдем в ветку `if err != nil`, выполнение приложения продолжится, потому что `http.Error` не останавливает обработчик.

Каковы реальные последствия такой ошибки? Для начала обсудим это на уровне HTTP. Предположим, что мы завершили работу предыдущего

HTTP-обработчика, добавив шаг для вывода тела успешного HTTP-ответа и кода состояния:

```
func handler(w http.ResponseWriter, req *http.Request) {
   err := foo(req)
   if err != nil {
      http.Error(w, "foo", http.StatusInternalServerError)
   }
   _, _ = w.Write([]byte("all good"))
   w.WriteHeader(http.StatusCreated)
}
```

В случае, если `err != nil`, HTTP-ответ будет выглядеть так:

```
foo
all good
```

Этот ответ содержит как сообщение об ошибке, так и сообщение об успешном выполнении.

Мы возвращаем только первый код состояния HTTP: в предыдущем примере это 500. Однако Go также запишет в журнал предупреждение:

```
2021/10/29 16:45:33 http: superfluous response.WriteHeader call
from main.handler (main.go:20)
```

Оно означает, что мы пытались записать код состояния несколько раз, что избыточно.

С точки зрения выполнения основное воздействие будет заключаться в продолжении выполнения функции, которая должна была быть остановлена. Например, если `foo` в дополнение к ошибке возвращает еще и указатель, продолжение выполнения будет означать использование этого указателя, что может привести к разыменованию нулевого указателя (и, следовательно, к панике горутины).

Чтобы исправить эту ошибку, нужно подумать о добавлении оператора `return` после `http.Error`:

```
func handler(w http.ResponseWriter, req *http.Request) {
   err := foo(req)
   if err != nil {
      http.Error(w, "foo", http.StatusInternalServerError)
      return  <-- Добавление оператора return
   }
   // ...
}
```

Благодаря оператору `return` функция остановит свое выполнение, если мы завершим ее в ветви `if err != nil`.

Эта ошибка, вероятно, не самая сложная из разбираемых в этой книге. Но о ней легко забыть, поэтому она встречается довольно часто. Всегда нужно помнить, что `http.Error` не останавливает выполнение обработчика и должен быть добавлен вручную. Такая проблема может и должна быть выловлена во время тестирования, если у нас достаточное покрытие тестами.

В последнем разделе этой главы обсудим вопросы, связанные с HTTP. Разберем, почему в production-grade-приложениях не нужно полагаться на стандартные реализации HTTP-клиента и сервера.

10.7. ОШИБКА #81: ИСПОЛЬЗОВАТЬ СТАНДАРТНЫЕ HTTP-КЛИЕНТ И СЕРВЕР

Пакет `http` предоставляет реализации HTTP-клиента и сервера. Но разработчики часто совершают такую ошибку — полагаются на их стандартные (по умолчанию) реализации в контексте приложений, которые потом развертываются в рабочей среде. Рассмотрим связанные с этим проблемы и способы их решения.

10.7.1. HTTP-клиент

Определим, что означает *клиент по умолчанию* (default client, или *стандартный клиент*). В качестве примера рассмотрим запрос GET. Используем нулевое значение структуры `http.Client` так:

```
client := &http.Client{}
resp, err := client.Get("https://golang.org/")
```

Или используем функцию `http.Get`:

```
resp, err := http.Get("https://golang.org/")
```

Оба подхода одинаковы. Функция `http.Get` использует `http.DefaultClient`, который также основан на нулевом значении `http.Client`:

```
// DefaultClient — это стандартный Client, который используется в Get,
// Head и Post.
var DefaultClient = &Client{}
```

В чем проблема с использованием стандартного HTTP-клиента?

Прежде всего в стандартном клиенте не определяются никакие тайм-ауты. Их отсутствие — это не то, что нужно для production-grade-систем: такая ситуация может привести ко многим проблемам, например бесконечно повторяющимся запросам, которые истощают системные ресурсы.

Перед тем как углубиться в доступные при выполнении запроса тайм-ауты, рассмотрим пять шагов, связанных с HTTP-запросом:

1. Запрос на установление TCP-соединения.
2. TLS-рукопожатие (TLS handshake — если оно включено).
3. Отправка запроса.
4. Чтение заголовков ответа.
5. Чтение тела ответа.

На рис. 10.2 показано, как эти шаги соотносятся с тайм-аутами основного клиента.

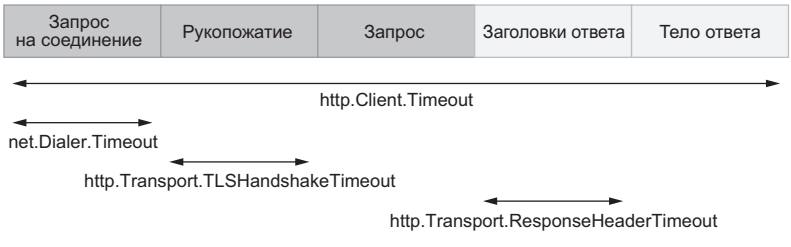


Рис. 10.2. Пять шагов HTTP-запроса и соответствующие тайм-ауты

Есть четыре тайм-аута:

- `net.Dialer.Timeout` — указывает максимальный отрезок времени, в течение которого запрос на соединение будет ожидать установления соединения.
- `http.Transport.TLSHandshakeTimeout` — указывает максимальное время ожидания подтверждения (рукопожатия) TLS.
- `http.Transport.ResponseHeaderTimeout` — указывает время, в течение которого происходит ожидание получения заголовков ответа сервера.
- `http.Client.Timeout` — указывает ограничение по времени для всего запроса. Он включает в себя все шаги, от шага 1 (Запрос на установление TCP-соединения) до шага 5 (Чтение тела ответа).

Тайм-аут HTTP-клиента

Возможно, вы сталкивались со следующей ошибкой при указании `http.Client.Timeout`:

```
net/http: request canceled (Client.Timeout exceeded while awaiting headers)
```

Эта ошибка означает, что эндпоинт не ответил в отведенное время. Мы получаем эту ошибку об ожидании заголовков, потому что их чтение — первый шаг при ожидании ответа.

Вот пример HTTP-клиента, который переопределяет эти тайм-ауты:

```
client := &http.Client{
   Timeout: 5 * time.Second,                    <-- Тайм-аут для глобального запроса
   Transport: &http.Transport{
      DialContext: (&net.Dialer{
         Timeout: time.Second,                  <-- Тайм-аут для запроса и ожидания соединения
      }).DialContext,
      TLSHandshakeTimeout: time.Second,         <-- Тайм-аут для TLS-рукопожатия
      ResponseHeaderTimeout: time.Second,       <-- Тайм-аут для ожидания получения заголовков
   },
}
```

Мы создаем клиент с 1-секундным тайм-аутом для запроса на соединение, TLS-рукопожатия и чтения заголовка ответа. Между тем каждый запрос имеет глобальный 5-секундный тайм-аут.

Второй аспект, который следует учитывать при работе со стандартным HTTP-клиентом, — это то, как обрабатываются соединения. По умолчанию HTTP-клиент создает пул соединений. Стандартный клиент повторно использует подключения (это можно запретить, установив для `http.Transport.DisableKeepAlives` значение `true`). Есть дополнительный тайм-аут, чтобы указать, как долго бездействующее соединение будет сохраняться в пуле: `http.Transport.IdleConnTimeout`. Его значение по умолчанию — 90 секунд, это означает, что соединение можно повторно использовать для других запросов в течение этого времени. После этого, если соединение не использовалось повторно, оно будет закрыто.

Чтобы настроить количество соединений в пуле, нужно переопределить `http.Transport.MaxIdleConns`. По умолчанию это значение равно 100. Но обратите внимание на нечто важное — ограничение `http.Transport.MaxIdleConnsPerHost` для каждого хоста, значение которого по умолчанию устанавливается равным 2.

Например, если мы инициируем 100 запросов к одному и тому же хосту, то после этого в пуле останутся только 2 соединения. Поэтому если мы еще раз инициируем 100 запросов, придется повторно открывать как минимум 98 соединений. Эта конфигурация может повлиять и на среднюю задержку, если приходится иметь дело со значительным количеством параллельных запросов к одному и тому же хосту.

Для production-grade-систем мы, вероятно, захотим переопределить тайм-ауты, которые используются по умолчанию. И настройка параметров, связанных с пулом соединений, также может существенно повлиять на задержку.

10.7.2. HTTP-сервер

К реализации HTTP-сервера следует подходить с осторожностью. Стандартный сервер можно создать, используя нулевое значение `http.Server`:

```
server := &http.Server{}
server.Serve(listener)
```

Можно использовать такие функции, как `http.Serve`, `http.ListenAndServe` или `http.ListenAndServeTLS`, которые также используются в стандартном `http.Server`.

Как только запрос на соединение принят, HTTP-ответ можно разделить на пять шагов:

1. Ожидания запроса клиента.
2. TLS-рукопожатие (TLS handshake — если оно включено).
3. Чтение заголовков запроса.
4. Чтение тела запроса.
5. Отправка ответа.

> **ПРИМЕЧАНИЕ** TLS-рукопожатие не нужно повторять при уже установленном соединении.

На рис. 10.3 показано, как эти шаги соотносятся с тайм-аутами главного сервера. Три основных тайм-аута:

- `http.Server.ReadHeaderTimeout` — поле, указывающее максимальное время, отводимое на чтение заголовков запроса;

- `http.Server.ReadTimeout` — поле, в котором указывается максимальное время, отводимое на чтение всего запроса;
- `http.TimeoutHandler` — обертка функции, указывающей максимальное время, отводимое обработчику на завершение.

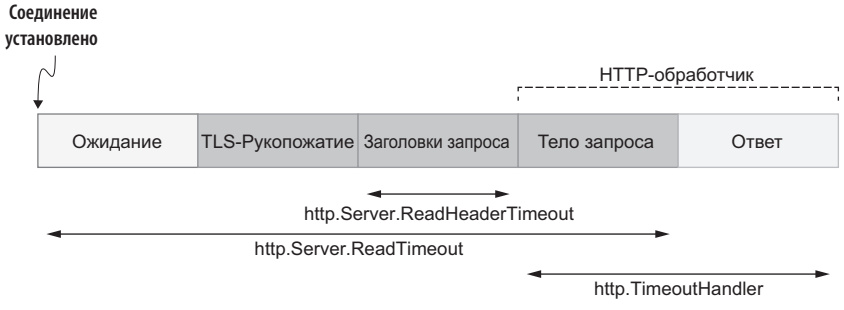


Рис. 10.3. Пять шагов HTTP-ответа и соответствующие тайм-ауты

Последний параметр является не параметром сервера, а оберткой поверх обработчика, ограничивающей время его работы. Если обработчик не ответит вовремя, то сервер отправит: «503 Service Unavailable» («503 сервис недоступен») с каким-то специфичным сообщением, а контекст, переданный обработчику, будет отменен.

> **ПРИМЕЧАНИЕ** Мы намеренно опустили `http.Server.WriteTimeout`, в котором нет необходимости, поскольку был выпущен (в Go 1.8) `http.TimeoutHandler`. У `http.Server.Write-Timeout` есть несколько проблем. Прежде всего, его поведение зависит от того, включен ли TLS или нет, что усложняет понимание и использование. Он также закрывает TCP-соединение без возврата правильного HTTP-кода, если время ожидания истекло. И он не передает отмену в контекст обработчика, поэтому тот может продолжить выполнение, не учитывая, что TCP-соединение уже закрыто.

При предоставлении конечной точки ненадежным клиентам рекомендуется хотя бы задать поле `http.Server.ReadHeaderTimeout` и использовать функцию-обертку `http.TimeoutHandler`. В противном случае клиенты могут использовать эту уязвимость и, например, бесконечно создавать соединения, что приведет к истощению системных ресурсов.

Вот так можно настроить сервер с этими тайм-аутами:

```
s := &http.Server{
   Addr:              ":8080",
   ReadHeaderTimeout: 500 * time.Millisecond,
   ReadTimeout:       500 * time.Millisecond,
   Handler:           http.TimeoutHandler(handler, time.Second, "foo"),
}
```

Обертка HTTP-обработчика

`http.TimeoutHandler` оборачивает обработчик. Если этот обработчик не отвечает в течение 1 секунды, сервер вернет код состояния 503 с `foo` в качестве HTTP-ответа.

Как говорилось для HTTP-клиентов, мы можем настроить и на стороне сервера максимальное время для обработки следующего запроса, когда разрешены keep-alive-соединения. Это делается с помощью `http.Server.IdleTimeout`:

```
s := &http.Server{
   // ...
   IdleTimeout: time.Second,
}
```

Обратите внимание, что если `http.Server.IdleTimeout` не задан, для тайм-аута простоя используется значение `http.Server.ReadTimeout`. Если ни один из этих параметров не задан, то тайм-аутов не будет вообще и соединения будут оставаться открытыми до тех пор, пока не окажутся закрыты клиентами.

Важно, чтобы в production-grade-приложениях не использовались стандартные HTTP-клиенты и серверы. В противном случае из-за отсутствия тайм-аутов или даже из-за вредоносных действий клиентов запросы могут зависнуть навсегда.

ИТОГИ

- Будьте осторожны с функциями, принимающими `time.Duration`. Несмотря на то что передача целого числа разрешена, чтобы предотвратить возможную путаницу, старайтесь использовать `time` API.
- Отказ от вызовов `time.After` в повторяющихся функциях (циклы или HTTP-обработчики) позволяет избежать пикового потребления памяти. Ресурсы, созданные `time.After`, освобождаются только по истечении установленного значения таймера.
- Подходите с осторожностью к использованию встроенных полей в структурах Go. Оно может привести к неочевидным ошибкам, например переопределению поведения маршалинга по умолчанию встроенным полем `time.Time`, реализующим интерфейс `json.Marshaler`.

- При сравнении двух структур `time.Time` помните, что эта структура содержит как настенные, так и монотонные часы и сравнение с использованием оператора `==` выполняется для них обоих.
- Чтобы не делать неверные допущения при предоставлении карты во время демаршалинга данных JSON, помните, что числовые значения по умолчанию преобразуются в тип `float64`.
- Вызывайте методы `Ping` или `PingContext`, если нужно протестировать конфигурацию и убедиться, что база данных доступна.
- Настройте параметры подключения к базе данных для production-grade-приложений.
- Использование подготовленных операторов SQL делает запросы более эффективными и безопасными.
- Работайте в таблицах со столбцами, значение которых может быть равно `NULL`, используя указатели или типы `sql.NullXXX`.
- После итерации по строкам вызывайте метод `Err` для `*sql.Rows` для того, чтобы убедиться, что вы не пропустили ошибку при подготовке следующей строки.
- Закрывайте все структуры, реализующие `io.Closer`, чтобы избежать возможных утечек.
- Чтобы избежать неожиданного поведения кода при реализации HTTP-обработчика, убедитесь, что не забыли использовать оператор `return`, если хотите, чтобы обработчик останавливался после `http.Error`.
- Для production-grade-приложений не пользуйтесь стандартными реализациями HTTP-клиента и сервера. По умолчанию в этих реализациях нет тайм-аутов, которые обязательны в продакшене.

11 Тестирование

В этой главе:

- ✓ Категоризация тестов и повышение их надежности
- ✓ Как сделать тесты Go детерминированными
- ✓ Работа с пакетами `httptest` и `iotest`
- ✓ Как избежать совершения типичных ошибок в бенчмарках
- ✓ Совершенствование процесса тестирования

Тестирование — важнейший аспект жизненного цикла проекта. Оно дает бесчисленные преимущества: укрепляет доверие к приложению, документирует код, облегчает рефакторинг. По сравнению с другими языками Go предоставляет продвинутые и полезные примитивы для написания тестов. В этой главе рассмотрим типичные ошибки, делающие тестирование хрупким, менее эффективным и менее точным.

11.1. ОШИБКА #82: НЕ РАСПРЕДЕЛЯТЬ ТЕСТЫ ПО КАТЕГОРИЯМ

Пирамида тестирования — это модель, которая группирует тесты по разным категориям (рис. 11.1). Юнит-тесты (или модульные тесты) находятся в основании пирамиды. Большинство тестов должны быть как модульные: они пишутся относительно просто, быстро выполняются и высокодетерминированны. По мере продвижения вверх по пирамиде тесты становятся более сложными для написания и более медленными при выполнении, и их детерминированность труднее гарантировать.

Обычный подход заключается в том, чтобы явно указать, какие из тестов следует проводить. Например, в зависимости от этапа жизненного цикла проекта требуется запустить все возможные тесты либо только юнит-тесты. Отсутствие классификации тестов означает потенциальную потерю времени и усилий, а также потерю точности в определении объема теста. В этом разделе обсудим три основных способа классификации тестов в Go.

Рис. 11.1. Пример пирамиды тестирования

11.1.1. Теги сборки

Наиболее распространенный способ классификации тестов — использование тегов сборки. Тег сборки — это специальный комментарий в начале файла Go, за которым следует пустая строка.

Посмотрите на такой файл bar.go:

```
//go:build foo

package bar
```

Этот файл содержит тег `foo`. Обратите внимание, что один пакет может содержать несколько файлов с разными тегами сборки.

> **ПРИМЕЧАНИЕ** Начиная с версии Go 1.17, синтаксис `// +build foo` был заменен на `//go:build foo`. Сейчас (в Go 1.18) `gofmt` синхронизирует эти две формы, чтобы упростить миграцию.

Теги сборки используются в двух случаях. Во-первых, в качестве условной опции для сборки приложения. Например, если нужно, чтобы исходный файл был включен, только если разрешен `cgo` (`cgo` позволяет пакетам Go вызывать код C), добавьте тег `//go:build cgo build`. Во-вторых, если нужно классифицировать тест как интеграционный, тогда мы добавляем специальный флаг сборки, например `integration`.

Вот пример файла db_test.go:

```
//go:build integration
package db
import (
   "testing"
)
func TestInsert(t *testing.T) {
   // ...
}
```

Здесь есть `integration` в качестве тега сборки. Он указывает, что файл содержит интеграционные тесты. Преимущество использования тегов сборки заключается в том, что можно выбирать, какие виды тестов выполнять. Предположим, что пакет содержит два тестовых файла:

- только что созданный файл: db_test.go;
- другой файл, не содержащий тег сборки: contract_test.go.

Если запустить внутри этого пакета `go test` без каких-либо параметров, он запустит только тестовые файлы без тегов сборки (contract_test.go):

```
$ go test -v .
=== RUN TestContract
--- PASS: TestContract (0.01s)
PASS
```

Но если задать тег `integration`, выполнение `go test` также будет включать db_test.go:

```
$ go test --tags=integration -v .
=== RUN TestInsert
--- PASS: TestInsert (0.01s)
=== RUN TestContract
--- PASS: TestContract (2.89s)
PASS
```

Таким образом, запуск тестов с каким-то тегом включает в себя исполнение как файлов без тегов, так и файлов, соответствующих тегу. Но что, если мы хотим запускать *только* интеграционные тесты? Возможный способ — добавить тег отрицания в файлы юнит-тестов. Например, использование `!integration` означает, что мы хотим включить тестовый файл, только если флаг `integration` *не* включен (contract_test.go):

```
//go:build !integration
package db
import (
   "testing"
)
func TestContract(t *testing.T) {
   // ...
}
```

Используя такой подход, мы добиваемся того, что:

- запуск `go test` с флагом `integration` запускает только интеграционные тесты;
- запуск `go test` без флага `integration` запускает только юнит-тесты.

Обсудим вариант, работающий на уровне одного теста, а не файла.

11.1.2. Переменные среды

Как сказал Питер Бургон (Peter Bourgon), член сообщества Go, теги сборки имеют один главный недостаток: отсутствие сигналов о том, что тест проигнорирован (см. http://mng.bz/qYlr). В первом примере, когда мы выполнили `go test` без флагов сборки, он показал только те тесты, которые были выполнены:

```
$ go test -v .
=== RUN TestUnit
--- PASS: TestUnit (0.01s)
PASS
ok db 0.319s
```

Если мы не будем внимательны к тому, как обрабатываются теги, мы можем забыть о существующих тестах. По этой причине некоторые проекты предпочитают проверять категорию тестов с помощью переменных среды.

Например, можно реализовать интеграционный тест `TestInsert`, проверив некоторую определенную переменную среды и, возможно, пропустив какой-то подобный тест:

```
func TestInsert(t *testing.T) {
   if os.Getenv("INTEGRATION") != "true" {
      t.Skip("skipping integration test")
   }
   // ...
}
```

Если значение переменной среды `INTEGRATION` не установлено `true`, то тест будет пропускаться с выдачей сообщения:

```
$ go test -v .
=== RUN TestInsert
    db_integration_test.go:12: skipping integration test
--- SKIP: TestInsert (0.00s)
=== RUN TestUnit
--- PASS: TestUnit (0.00s)
PASS
ok db 0.319s
```

Показывает сообщение о пропущенном тесте

Одно из преимуществ использования этого подхода — явное указание на то, какие тесты пропускаются и почему. Этот метод используется не так часто, как теги сборки, но о нем стоит знать, поскольку он предоставляет некоторые преимущества.

Рассмотрим еще один способ категоризации тестов: короткий режим.

11.1.3. Короткий режим

Другой подход к классификации тестов связан с их скоростью. Возможно, придется отделить тесты с коротким временем выполнения от тестов с длительным временем выполнения.

Допустим, есть набор юнит-тестов, и известно, что один из них медленный. Нужно отнести его к отдельной группе, чтобы не приходилось его запускать каждый раз (особенно если он запускается, например, после сохранения файла). Короткий режим позволяет сделать это:

```
func TestLongRunning(t *testing.T) {
   if testing.Short() {
      t.Skip("skipping long-running test")
   }
   // ...
}
```

Пометка теста как длительного

Используя `testing.Short`, можно узнать, был ли во время выполнения теста включен короткий режим. Затем мы используем `Skip`, чтобы пропустить тест.

Чтобы запустить тесты в коротком режиме, нужно послать `-short`:

```
% go test -short -v .
=== RUN TestLongRunning
   foo_test.go:9: skipping long-running test
--- SKIP: TestLongRunning (0.00s)
PASS
ok foo 0.174s
```

`TestLongRunning` явно пропускается при выполнении тестов. Обратите внимание, что, в отличие от тегов сборки, эта опция работает для каждого теста, а не для каждого файла.

Таким образом, категоризация тестов — это лучшая практика в успешной стратегии тестирования. В этом разделе мы рассмотрели три способа классификации тестов:

- с использованием тегов сборки на уровне тестового файла;
- с использованием переменных среды для пометки определенного теста;
- на основе длительности их выполнения с использованием короткого режима.

Можно комбинировать эти подходы: например, использовать теги сборки или переменные среды для классификации теста (например, юнит- или интеграционного) и короткий режим, если проект содержит долго выполняющиеся юнит-тесты.

В следующем разделе обсудим, почему важно включать флаг `-race`.

11.2. ОШИБКА #83: НЕ ВКЛЮЧАТЬ ФЛАГ -RACE

В разделе, посвященном разбору ошибки #58 (не понимать проблем гонки), мы определили гонку данных как ситуацию, когда две горутины одновременно обращаются к одной и той же переменной, при этом хотя бы одна из горутин производит запись в эту переменную. В Go есть стандартный инструмент, помогающий обнаруживать гонку данных. Одна из распространенных ошибок разработчиков состоит в том, что они забывают о важности этого инструмента и не активируют его. В этом разделе узнаем, какие нежелательные ситуации отлавливает детектор гонки, как его использовать и какие у него ограничения.

В Go детектор гонки — это не инструмент статического анализа, использующийся во время компиляции. Он предназначен для поиска гонок данных, которые

происходят во время выполнения. Чтобы его активировать, установите флаг `-race` во время компиляции или запуска теста. Например:

```
$ go test -race ./...
```

Как только детектор гонки оказывается активирован, компилятор инструментирует код для обнаружения гонок данных. Понятие *инструментация* относится к действиям компилятора, добавляющим дополнительные инструкции: отслеживание всех обращений к памяти и запись, когда и как они происходят. Во время выполнения детектор следит за тем, не возникают ли гонки данных. Но помните о том, что это достигается ценой дополнительного расхода ресурсов из-за включенного состояния детектора гонки:

- уровень использования памяти может увеличиться в 5–10 раз;
- время выполнения может увеличиться от 2 до 20 раз.

Из-за такого оверхеда обычно рекомендуется включать детектор гонки только во время локального тестирования или непрерывной интеграции. В рабочем продукте его следует избегать (или использовать, например, только в случае канареечных релизов).

Если обнаружена гонка, то Go выдает предупреждение. Например, следующий код содержит гонку данных, потому что к `i` можно обращаться одновременно и для чтения, и для записи:

```
package main
import (
   "fmt"
)
func main() {
   i := 0
   go func() { i++ }()
   fmt.Println(i)
}
```

Запуск этого приложения с флагом `-race` регистрирует следующее предупреждение о гонке данных:

```
==================
WARNING: DATA RACE
Write at 0x00c000026078 by goroutine 7:
   main.main.func1()
      /tmp/app/main.go:9 +0x4e
Previous read at 0x00c000026078 by main goroutine:
   main.main()
      /tmp/app/main.go:10 +0x88
```

Указывается на то, что горутина 7 что-то записывала

Указывается на то, что основная горутина что-то читала

```
Goroutine 7 (running) created at:
   main.main()
      /tmp/app/main.go:9 +0x7a
==================
```

Указывается на то, когда была создана горутина 7

Убедимся, что такие сообщения удобно читать. Go всегда регистрирует следующее:

- Причастные к гонке конкурирующие горутины: здесь — основная горутина и горутина 7.
- Где в коде происходят обращения: в данном случае это строки 9 и 10.
- Когда были созданы эти горутины: горутина 7 была создана в `main()`.

ПРИМЕЧАНИЕ Внутри себя детектор гонки использует векторные часы — структуру данных, которую применяют для определения частичного упорядочивания событий (а также в распределенных системах, таких как базы данных). Каждое создание горутины приводит к созданию векторных часов. Инструментация обновляет векторные часы при каждом действии по доступу к памяти и акте синхронизации. Затем она сравнивает векторные часы, чтобы обнаружить потенциальную гонку данных.

Детектор гонки не может отлавливать ложноположительные срабатывания (то есть ситуации, выглядящие как гонки данных, но не являющиеся ими). Если мы получим от него какое-то предупреждение, то это означает, что код действительно содержит гонку данных. И наоборот, иногда он дает ложноотрицательные результаты (пропуск фактически имеющих место гонок данных).

Отмечу две вещи, касающиеся тестирования. Во-первых, детектор гонки может быть хорош лишь настолько, насколько хороши наши тесты. Важно убедиться, что конкурентный код тщательно тестируется на предмет гонок данных. Во-вторых, учитывая возможные ложноотрицательные результаты, логику теста для проверки гонки данных можно поместить в цикл. Это увеличивает шансы на отлавливание возможных гонок данных:

```
func TestDataRace(t *testing.T) {
   for i := 0; i < 100; i++ {
      // Существующая логика.
   }
}
```

Кроме того, если какой-то конкретный файл содержит тесты, которые приводят к гонке данных, мы можем исключить его из обнаружения гонок с помощью тега сборки `!race`:

```
//go:build !race
package main
import (
   "testing"
)
func TestFoo(t *testing.T) {
   // ...
}
func TestBar(t *testing.T) {
   // ...
}
```

Этот файл подлежит сборке только в том случае, если отключен детектор гонки. В противном случае весь файл не будет собран и тесты выполняться не будут.

Настоятельно рекомендуется запуск тестов с флагом `-race` для приложений, в которых используется конкурентность. Иногда это даже обязательно. Это позволяет активировать детектор гонки, который инструментирует код для обнаружения потенциальных гонок данных. Его активация сильно влияет на производительность и значительно нагружает память, поэтому этот подход необходимо использовать в специфических обстоятельствах, например при локальном тестировании или при CI.

В следующем разделе обсудим два флага, относящиеся к режиму выполнения: `parallel` и `shuffle`.

11.3. ОШИБКА #84: НЕ ИСПОЛЬЗОВАТЬ РЕЖИМЫ ВЫПОЛНЕНИЯ ТЕСТОВ

Во время выполнения тестов команда `go` может воспринимать набор флагов, влияющих на выполнение тестов. Типичная недоработка — не знать эти флаги и упускать при этом возможности для более быстрого выполнения кода или более эффективного обнаружения ошибок. Рассмотрим флаги `parallel` и `shuffle`.

11.3.1. Флаг parallel

Режим параллельного выполнения позволяет запускать определенные тесты параллельно, что может быть очень полезно, например, для ускорения проведения длительных тестов. Можно поставить метку о том, что тест должен выполняться параллельно, вызвав `t.Parallel`:

```
func TestFoo(t *testing.T) {
   t.Parallel()
   // ...
}
```

Когда мы помечаем тест с помощью `t.Parallel`, он выполняется одновременно со всеми другими параллельными тестами. Но с точки зрения организации исполнения кода Go сначала запускает один за другим все последовательные тесты. После их завершения выполняются параллельные тесты.

Следующий код содержит три теста, но только два из них отмечены для параллельного выполнения:

```
func TestA(t *testing.T) {
   t.Parallel()
   // ...
}
func TestB(t *testing.T) {
   t.Parallel()
   // ...
}
func TestC(t *testing.T) {
   // ...
}
```

Запуск тестов для этого файла выводит следующие записи:

```
=== RUN      TestA
=== PAUSE    TestA  <-- TestA приостанавливается
=== RUN      TestB
=== PAUSE    TestB  <-- TestB приостанавливается
=== RUN      TestC  <-- Запускается TestC
--- PASS:    TestC (0.00s)
=== CONT     TestA  <-- TestA и TestB возобновляются
--- PASS:    TestA (0.00s)
=== CONT     TestB
--- PASS:    TestB (0.00s)
PASS
```

`TestC` выполняется первым. `TestA` и `TestB` зарегистрированы первыми, но их выполнение приостанавливается до завершения `TestC`. Затем исполнение возобновляется, и они выполняются параллельно.

По умолчанию максимальное количество одновременно выполняемых тестов равно значению `GOMAXPROCS`. Для сериализации тестов или, например, увеличения этого числа в контексте длительных тестов, выполняющих много операций ввода/вывода, можно изменить это значение с помощью флага `-parallel`:

```
$ go test -parallel 16 .
```

Здесь максимальное количество параллельно выполняемых тестов равно 16. Теперь рассмотрим другой режим при выполнении тестов Go: `shuffle`.

11.3.2. Флаг -shuffle

Начиная с версии Go 1.17, можно задать случайный порядок выполнения тестов и бенчмарков. Когда это нужно? Лучшая практика при написании тестов — делать их изолированными. Например, они не должны зависеть от порядка их выполнения или от каких-либо общих для них переменных. Наличие скрытых зависимостей может приводить к ошибкам теста или, что еще хуже, к ошибкам, которые не будут обнаружены во время тестирования. Используем флаг `-shuffle` для рандомизации тестов, то есть их выполнения в случайном порядке. Можно задать состояние этого флага `on` или `off`, чтобы включить или отключить режим перетасовки тестов (по умолчанию он отключен):

```
$ go test -shuffle=on -v .
```

Иногда нужно повторно запустить тесты в том же порядке. Например, если тесты не проходят во время CI, потребуется воспроизвести ошибку локально. В этом случае вместо установки флага `-shuffle` в состояние `on` требуется передать значение стартового числа (seed) для рандомизации тестов. Мы можем получить это значение как один из результатов выполнения тестов в случайном порядке, включив подробный режим (`-v`):

```
$ go test -shuffle=on -v .
-test.shuffle 1636399552801504000  ◄——  Значение seed
=== RUN TestBar
--- PASS: TestBar (0.00s)
=== RUN TestFoo
--- PASS: TestFoo (0.00s)
PASS
ok teivah 0.129s
```

Здесь мы выполнили тесты случайным образом, но при этом `go test` вывел значение стартового числа: `1636399552801504000`. Чтобы тесты выполнялись в том же порядке, мы передаем в `shuffle` это значение:

```
$ go test -shuffle=1636399552801504000 -v .
-test.shuffle 1636399552801504000
=== RUN      TestBar
--- PASS: TestBar (0.00s)
=== RUN      TestFoo
--- PASS: TestFoo (0.00s)
PASS
ok teivah 0.129s
```

Тесты были выполнены в том же порядке: сначала `TestBar`, а затем `TestFoo`.

С существующими тестовыми флагами следует работать осторожно. Также нужно знать, какие новые функции имеются в последних версиях Go. Параллельное выполнение тестов может быть отличным способом сократить общее время полного тестирования. А режим `shuffle` поможет обнаружить скрытые зависимости, которые приводят к ошибкам тестирования или даже к пропуску ошибок при выполнении тестов в одном и том же порядке.

11.4. ОШИБКА #85: НЕ ИСПОЛЬЗОВАТЬ ТАБЛИЧНЫЕ ТЕСТЫ

Табличные тесты (table-driven tests) — это эффективный метод написания компактных тестов, позволяющий сократить избыточный код и сосредоточиться на самой логике тестирования, то есть на том, что действительно важно. В этом разделе рассмотрим конкретный пример и поймем, почему при работе в Go полезно знать табличные тесты.

Рассмотрим следующую функцию, которая удаляет из какой-то строки все символы новой строки (`\n` или `\r\n`):

```
func removeNewLineSuffixes(s string) string {
   if s == "" {
      return s
   }
   if strings.HasSuffix(s, "\r\n") {
      return removeNewLineSuffixes(s[:len(s)-2])
   }
   if strings.HasSuffix(s, "\n") {
      return removeNewLineSuffixes(s[:len(s)-1])
   }
   return s
}
```

Эта функция рекурсивно удаляет символы `\r\n` и `\n`. Теперь предположим, что надо очень тщательно протестировать эту функцию. Как минимум нужно покрыть следующие случаи:

- вход пуст;
- вход заканчивается `\n`;
- вход заканчивается `\r\n`;

- вход заканчивается несколькими \n;
- вход заканчивается без новых строк.

В следующем коде для каждого случая создается юнит-тест:

```
func TestRemoveNewLineSuffix_Empty(t *testing.T) {
   got := removeNewLineSuffixes("")
   expected := ""
   if got != expected {
      t.Errorf("got: %s", got)
   }
}

func TestRemoveNewLineSuffix_EndingWithCarriageReturnNewLine(t *testing.T) {
   got := removeNewLineSuffixes("a\r\n")
   expected := "a"
   if got != expected {
      t.Errorf("got: %s", got)
   }
}

func TestRemoveNewLineSuffix_EndingWithNewLine(t *testing.T) {
   got := removeNewLineSuffixes("a\n")
   expected := "a"
   if got != expected {
      t.Errorf("got: %s", got)
   }
}

func TestRemoveNewLineSuffix_EndingWithMultipleNewLines(t *testing.T) {
   got := removeNewLineSuffixes("a\n\n\n")
   expected := "a"
   if got != expected {
      t.Errorf("got: %s", got)
   }
}

func TestRemoveNewLineSuffix_EndingWithoutNewLine(t *testing.T) {
   got := removeNewLineSuffixes("a\n")
   expected := "a"
   if got != expected {
      t.Errorf("got: %s", got)
   }
}
```

Каждая функция соответствует конкретному случаю, который надо покрыть. Но в этом коде два недостатка. Во-первых, все имена функций здесь очень сложные (например, `TestRemoveNewLine-Suffix_EndingWithCarriageReturnNewLine` длиной в 55 символов). Это может снизить понимание того, что функция должна тестировать. Второй недостаток — большая степень дублирования кода этих функций, которые имеют одинаковую структуру:

1. Вызов `removeNewLineSuffixes`.
2. Определение ожидаемого значения.
3. Сравнение значений.
4. Запись сообщения об ошибке.

Если потребуется изменить какой-то из этих шагов, например включить ожидаемое значение в сообщение об ошибке, то придется повторить это изменение во всех тестах. И чем больше тестов, тем сложнее поддержка кода.

В таких случаях можно использовать табличные тесты: их логика пишется только один раз. Табличные тесты основаны на подтестах, а одна тестовая функция может включать в себя несколько таких подтестов. Например, следующий тест содержит два подтеста:

```
func TestFoo(t *testing.T) {
   t.Run("subtest 1", func(t *testing.T) {   <-- Выполнение первого подтеста (subtest 1)
      if false {
         t.Error()
      }
   })
   t.Run("subtest 2", func(t *testing.T) {   <-- Выполнение второго подтеста (subtest 2)
      if 2 != 2 {
         t.Error()
   }
   })
}
```

Функция `TestFoo` включает в себя два подтеста. Если мы запустим этот тест, он покажет результаты как для `subtest 1`, так и для `subtest 2`:

```
--- PASS: TestFoo (0.00s)
    --- PASS: TestFoo/subtest_1 (0.00s)
    --- PASS: TestFoo/subtest_2 (0.00s)
PASS
```

Мы также можем запустить только один тест, используя для этого флаг `-run` и объединив имя родительского теста с подтестом. Например, запустить только `subtest 1`:

```
$ go test -run=TestFoo/subtest_1 -v   <-- Использование флага -run для запуска только subtest 1
=== RUN TestFoo
=== RUN TestFoo/subtest_1
--- PASS: TestFoo (0.00s)
    --- PASS: TestFoo/subtest_1 (0.00s)
```

Вернемся к нашему примеру и посмотрим, как использовать подтесты, чтобы предотвратить дублирование кода, который содержит в себе логику тестирования. Основная идея состоит в том, чтобы создавать для каждого случая свой подтест. Есть разные варианты, но мы обсудим структуру данных в виде карты, где ключ соответствует имени теста, а значение представляет собой тестовые данные (входные, ожидаемые).

В табличных тестах мы избегаем использования шаблонного кода при помощи структуры данных, содержащей тестовые данные вместе с подтестами. Вот возможная реализация с использованием карты:

```
func TestRemoveNewLineSuffix(t *testing.T) {
   tests := map[string]struct {   <-- Определение тестовых данных
      input string
      expected string
   }{
      `empty`: {   <-- Каждая запись в карте представляет собой подтест
         input: "",
         expected: "",
      },
      `ending with \r\n`: {
         input: "a\r\n",
         expected: "a",
      },
      `ending with \n`: {
         input: "a\n",
         expected: "a",
      },
      `ending with multiple \n`: {
         input: "a\n\n\n",
         expected: "a",
      },
      `ending without newline`: {
         input: "a",
         expected: "a",
      },
   }
   for name, tt := range tests {   <-- Итерации по карте
      t.Run(name, func(t *testing.T) {   <-- Выполнение нового подтеста
         got := removeNewLineSuffixes(tt.input)   для каждой записи в карте
         if got != tt.expected {
            t.Errorf("got: %s, expected: %s", got, tt.expected)
         }
      })
   }
}
```

Переменная `test` — это карта. Ее ключ — это имя теста, а значение — данные для теста, в нашем случае входные данные и ожидаемая строка. Каждая запись

в карте — это новый случай, который мы хотим покрыть. Поэтому для каждой записи карты запускается новый подтест.

Такой тест устраняет два недостатка:

- Имя каждого теста теперь представляет собой строку, а не имя функции в стиле PascalCase, что упрощает чтение.
- Логика теста записывается только один раз и используется для всех разнообразных случаев. Изменение структуры тестирования или добавление нового теста требуют минимальных усилий.

В табличных тестах может быть еще один источник ошибок: как уже ранее говорилось, мы можем пометить какой-то тест как выполняющийся параллельно, вызвав `t.Parallel`. Мы также можем сделать это в подтестах внутри замыкания, предоставляемого `t.Run`:

```
for name, tt := range tests {
   t.Run(name, func(t *testing.T) {
      t.Parallel()  ◄——  Пометка подтеста как выполняющегося параллельно
      // Использование tt
   })
}
```

Но это замыкание использует переменную цикла. Чтобы предотвратить проблему, подобную описанной при разборе ошибки # 63 (неосторожно обращаться с горутинами и переменными цикла), которая может привести к тому, что замыкания будут использовать неправильное значение переменной `tt`, мы должны создать другую переменную или теневую копию (shadow copy) `tt`:

```
for name, tt := range tests {
   tt := tt  ◄——  Создание теневой копии tt, что делает переменную локальной для цикла
   t.Run(name, func(t *testing.T) {
      t.Parallel()
      // Использование tt
   })
}
```

Таким образом, каждое замыкание будет обращаться к своей собственной переменной `tt`.

Если у нескольких юнит-тестов схожая структура, мы можем объединить их, используя возможности табличных тестов. Поскольку этот метод предотвращает дублирование, он упрощает изменение логики тестирования и добавление новых опций.

Обсудим теперь, как предотвратить появление нестабильных тестов.

11.5. ОШИБКА #86: ЗАДЕРЖКИ В ЮНИТ-ТЕСТАХ

Нестабильный (flaky) тест может как пройти, так и не пройти без каких-либо изменений в коде. Наличие нестабильных тестов — это одна из самых больших проблем в тестировании, поскольку они дорогие в отладке и подрывают уверенность в точности тестирования. Вызов `time.Sleep` в тесте может сигнализировать о возможной нестабильности. Например, конкурентный код часто тестируется с помощью задержек (sleeps). В этом разделе представлены конкретные методы удаления задержек из тестов и, таким образом, предотвращения создания нестабильных тестов.

Покажу это на примере анализа функции, которая возвращает значение и запускает горутину, выполняющую свои действия в фоновом режиме. Вызовем некоторую функцию, чтобы получить срез структур `Foo` и вернуть наилучший элемент (первый). Тем временем другая горутина будет отвечать за вызов метода `Publish` с первыми `n` элементами `Foo`:

```
type Handler struct {
   n         int
   publisher publisher
}
type publisher interface {
   Publish([]Foo)
}
func (h Handler) getBestFoo(someInputs int) Foo {
   foos := getFoos(someInputs)  <-- Получение среза Foo
   best := foos[0]  <-- Сохранение первого элемента (проверка
                        длины foos опущена для простоты)
   go func() {
      if len(foos) > h.n {  <-- Сохранение только первых n структур Foo
         foos = foos[:h.n]
      }
      h.publisher.Publish(foos)  <-- Вызов метода Publish
   }()
   return best
}
```

Структура `Handler` содержит два поля: поле `n` и зависимость `publisher`, используемую для публикации первых `n` структур `Foo`. Сначала мы получаем срез `Foo`. Но прежде чем вернуть первый элемент, запускаем новую горутину, фильтруем срез `foos` и вызываем `Publish`.

Как протестировать эту функцию? Написать какую-то часть кода, чтобы подтвердить отклик, просто. Но что, если нужно проверить, что передается в `Publish`?

Можно сымитировать интерфейс `publisher`, чтобы записать аргументы, передаваемые при вызове метода `Publish`. Затем прерваться на несколько миллисекунд перед проверкой записанных аргументов:

```
type publisherMock struct {
   mu sync.RWMutex
   got []Foo
}
func (p *publisherMock) Publish(got []Foo) {
   p.mu.Lock()
   defer p.mu.Unlock()
   p.got = got
}
func (p *publisherMock) Get() []Foo {
   p.mu.RLock()
   defer p.mu.RUnlock()
   return p.got
}
func TestGetBestFoo(t *testing.T) {
   mock := publisherMock{}
   h := Handler{
      publisher: &mock,
      n:         2,
   }
   foo := h.getBestFoo(42)
   // Проверка foo
   time.Sleep(10 * time.Millisecond)
   published := mock.Get()
   // Проверка published
}
```

10-миллисекундная пауза перед проверкой аргументов, переданных в Publish

Мы пишем мок (имитацию) `publisher`, который использует мьютекс для защиты доступа к полю `published`. В этом юнит-тесте мы вызываем `time.Sleep`, чтобы оставить некоторое время перед проверкой аргументов, переданных в `Publish`.

Этот тест нестабилен по своей сути. Строгой гарантии, что задержки именно в 10 миллисекунд будет достаточно, нет (в данном примере это весьма вероятно, но неточно).

Как улучшить такой юнит-тест? Прежде всего периодически проверять заданное условие, используя повторные попытки. Например, написать функцию, которая принимает утверждения в качестве аргумента, а также максимальное количество попыток и время ожидания, и вызывается периодически, чтобы избежать занятого цикла:

```
func assert(t *testing.T, assertion func() bool,
   maxRetry int, waitTime time.Duration) {
```

```
   for i := 0; i < maxRetry; i++ {
      if assertion() {  ◄—— Проверка assertion
         return
      }
      time.Sleep(waitTime)  ◄—— Пауза перед повторной попыткой
   }
   t.Fail()  ◄—— В итоге проваливается после нескольких попыток
}
```

Эта функция проверяет предоставленное утверждение (assertion) и выдает сбой после совершения определенного количества попыток. Мы также используем `time.Sleep`, но в этом коде могли бы использовать и более короткую задержку.

Например, вернемся к `TestGetBestFoo`:

```
assert(t, func() bool {
   return len(mock.Get()) == 2
}, 30, time.Millisecond)
```

Вместо задержки в 10 миллисекунд мы делаем задержку каждую миллисекунду и задаем максимальное количество повторных попыток. Такой подход в случае успешного прохождения теста сокращает общее время его выполнения, потому что сокращается интервал ожидания. Таким образом, «стратегия повторных попыток» — это предпочтительный подход по сравнению с использованием пассивных задержек.

> **ПРИМЕЧАНИЕ** Некоторые библиотеки тестирования, например `testify`, предлагают функции, использующие «повторные попытки». В `testify` есть функция `Eventually`, реализующая утверждения, которые в конечном итоге должны оказаться правильными, а также другие функции, например настройку сообщения об ошибке.

Другая стратегия заключается в использовании каналов для синхронизации горутины, публикующей структуры `Foo`, и горутины тестирования. Например, в реализации моков вместо копирования полученного среза в поле можно отправить это значение в канал:

```
type publisherMock struct {
   ch chan []Foo
}
func (p *publisherMock) Publish(got []Foo) {
   p.ch <- got  ◄—— Отправка полученного аргумента
}
func TestGetBestFoo(t *testing.T) {
```

```
   mock := publisherMock{
      ch: make(chan []Foo),
   }
   defer close(mock.ch)
   h := Handler{
      publisher: &mock,
      n:         2,
   }
   foo := h.getBestFoo(42)
   // Проверка foo
   if v := len(<-mock.ch); v != 2 {  <-- Сравнение аргументов
      t.Fatalf("expected 2, got %d", v)
   }
}
```

Происходит отправка полученного аргумента в канал. Тем временем горутина тестирования настраивает мок и создает утверждение на основе полученного значения. Мы также можем реализовать стратегию тайм-аута, чтобы убедиться, что не придется ждать `mock.ch` вечно, если что-то пойдет не так. Например, можно использовать `select` со случаем `time.After`.

Что выбрать: повторы или синхронизацию? Синхронизация сокращает время ожидания до минимума и делает тест полностью детерминированным, если он хорошо спроектирован.

Но если использовать синхронизацию по какой-то причине нельзя, то следует пересмотреть дизайн кода, поскольку в нем могут таиться проблемы. Если синхронизация действительно невозможна, мы должны — для устранения недетерминированности результатов тестов — использовать опцию «повторных попыток», которая является лучшим выбором, чем использование пассивных задержек.

Теперь обсудим, как предотвратить нестабильность тестов при использовании API времени.

11.6. ОШИБКА #87: НЕЭФФЕКТИВНАЯ РАБОТА С API ВРЕМЕНИ

Некоторые функции должны полагаться на API времени, например, для получения текущего времени. В таком случае легко получить хрупкие (brittle) юнит-тесты, которые в какой-то момент могут провалиться. В этом разделе рассмотрим пример и обсудим связанные с ним варианты. Цель этого раздела — не охватить

все возможные сценарии и методы, а дать рекомендации по написанию более надежных тестов функций с использованием API времени.

Допустим, в приложении происходят какие-то события, которые нужно сохранить в кэше памяти. Создаем структуру `Cache` для хранения самых последних событий. Эта структура будет предоставлять для использования три метода, которые делают следующее:

- добавляют событий;
- получают все события;
- обрезают события до заданной продолжительности (сосредоточимся на этом методе).

Каждый из методов должен иметь доступ к данным о текущем времени. Рассмотрим первую реализацию третьего метода с использованием `time.Now()` (будем считать, что все события отсортированы по времени):

```
type Cache struct {
   mu     sync.RWMutex
   events []Event
}
type Event struct {
   Timestamp time.Time
   Data string
}
func (c *Cache) TrimOlderThan(since time.Duration) {
   c.mu.RLock()
   defer c.mu.RUnlock()
   t := time.Now().Add(-since)
   for i := 0; i < len(c.events); i++ {
      if c.events[i].Timestamp.After(t) {
         c.events = c.events[i:]
         return
      }
   }
}
```

Вычитание значения заданной продолжительности из текущего времени

Обрезка событий

Вычисляем переменную `t`, которая должна быть равна текущему времени за вычетом заданной продолжительности. Поскольку события сортируются по времени, мы обновляем внутренний срез `events`, как только достигаем события, время которого наступает после `t`.

Как протестировать этот метод? Мы могли бы отталкиваться от значения текущего времени, используя `time.Now` для создания событий:

```
func TestCache_TrimOlderThan(t *testing.T) {
   events := []Event{   <-- Создание событий с помощью time.Now()
      {Timestamp: time.Now().Add(-20 * time.Millisecond)},
      {Timestamp: time.Now().Add(-10 * time.Millisecond)},
      {Timestamp: time.Now().Add(10 * time.Millisecond)},
   }
   cache := &Cache{}
   cache.Add(events)   <-- Добавление этих событий в кэш
   cache.TrimOlderThan(15 * time.Millisecond)   <-- Обрезка событий продолжительностью более 15 миллисекунд
   got := cache.GetAll()   <-- Получение всех событий
   expected := 2
   if len(got) != expected {
      t.Fatalf("expected %d, got %d", expected, len(got))
   }
}
```

Добавляем срез событий в кэш с помощью `time.Now()` и добавляем или вычитаем небольшие длительности. Затем обрезаем эти события на уровне 15 миллисекунд и выполняем утверждение (assertion).

У этого подхода есть один главный недостаток: если машина, на которой выполняется тест, внезапно оказывается занятой, может обрезаться меньше событий, чем ожидалось. Можно увеличить продолжительность, чтобы уменьшить вероятность проваленного теста, но это не всегда реально. Например, что, если бы поле метки времени было неэкспортированным полем, сгенерированным при добавлении события? В таком случае нельзя было бы передать конкретную метку времени, то может привести к добавлению в юнит-тест задержек.

Проблема связана с тем, как внутренне устроен `TrimOlderThan`: поскольку он вызывает `time.Now()`, то надежные юнит-тесты становится сложнее реализовывать. Обсудим два подхода, направленные на то, чтобы сделать тест более надежным.

Первый подход заключается в том, чтобы сделать способ получения текущего времени зависимостью от структуры `Cache`. В продакшене мы внедрили бы *настоящую* реализацию, а в юнит-тестах, например, передали бы заглушку (стаб).

Есть различные методы создания такой зависимости, например интерфейс или тип функции. Поскольку здесь используется только один метод (`time.Now()`), можно определить тип функции:

```
type now func() time.Time
type Cache struct {
   mu sync.RWMutex
   events []Event
   now now
}
```

Тип now — это функция, которая возвращает time.Time. В фабричной функции можно передать актуальную функцию time.Now так:

```
func NewCache() *Cache {
   return &Cache{
      events: make([]Event, 0),
      now: time.Now,
   }
}
```

Поскольку зависимость now остается неэкспортированной, то она недоступна для внешних клиентов. Кроме того, в нашем юнит-тесте можно создать структуру Cache, внедрив *поддельную* (fake) реализацию func() time.Time на основе заранее определенного времени:

```
func TestCache_TrimOlderThan(t *testing.T) {
   events := []Event{   <-- Создание события на основе определенных временных меток
      {Timestamp: parseTime(t, "2020-01-01T12:00:00.04Z")},
      {Timestamp: parseTime(t, "2020-01-01T12:00:00.05Z")},
      {Timestamp: parseTime(t, "2020-01-01T12:00:00.06Z")},
   }
   cache := &Cache{now: func() time.Time {          <-- Внедрение статической функции для фиксации времени
      return parseTime(t, "2020-01-01T12:00:00.06Z")
   }}
   cache.Add(events)
   cache.TrimOlderThan(15 * time.Millisecond)
   // ...
}
func parseTime(t *testing.T, timestamp string) time.Time {
   // ...
}
```

Использование глобальной переменной

Вместо использования поля можно получить данные о времени через глобальную переменную:

```
var now = time.Now   <-- Определение новой глобальной переменной
```

В целом мы должны стараться избегать такого состояния, в котором могут изменяться какие-то общие параметры и структуры. В нашем случае это привело бы как минимум к одной конкретной проблеме: тесты больше не были бы изолированы, поскольку зависели бы от одной общей переменной. В частности, поэтому тесты нельзя было запускать параллельно. При возможности следует обрабатывать такие случаи как часть структурных зависимостей, способствуя изоляции тестов.

При создании новой структуры `Cache` мы внедряем зависимость `now` на основе заданного времени. Благодаря такому подходу тест надежный. Даже в наихудших условиях результат этого теста будет детерминированным.

Это решение также расширяемое. Например, функция вызывает `time.After`. Тогда можно либо добавить другую зависимость `after`, либо создать один интерфейс, объединяющий два метода: `Now` и `After`. Но у этого подхода есть важный недостаток: зависимость `now` недоступна, если мы, например, создаем юнит-тест из внешнего пакета. Мы рассмотрим это в разделе, посвященном ошибке #90 (не изучать все возможности тестирования в Go).

В этом случае используем другую методику. Вместо обработки времени как неэкспортируемой зависимости запросим текущее время у клиентов:

```
func (c *Cache) TrimOlderThan(now time.Time, since time.Duration) {
   // ...
}
```

Чтобы пойти еще дальше, можно *объединить* два аргумента функции в одном `time.Time`, что будет представлять собой определенный момент времени, до которого мы хотим обрезать события:

```
func (c *Cache) TrimOlderThan(t time.Time) {
   // ...
}
```

Вычислить этот момент следует на вызывающей стороне:

```
cache.TrimOlderThan(time.Now().Add(time.Second))
```

И в тесте мы также должны передать соответствующее время:

```
func TestCache_TrimOlderThan(t *testing.T) {
   // ...
   cache.TrimOlderThan(parseTime(t, "2020-01-01T12:00:00.06Z").
      Add(-15 * time.Millisecond))
   // ...
}
```

Это самый простой подход, поскольку он не требует создания другого типа и заглушки.

К тестированию кода, использующего API времени, следует подходить с большой осторожностью, так как оно может быть причиной нестабильных тестов. Мы рассмотрели два способа справиться с этим. Первый — сохранить взаимодействия `time`

как часть зависимости, которую можно сымитировать в юнит-тестах, используя наши собственные реализации или полагаясь на внешние библиотеки. Второй — переработать API и запрашивать у клиентов необходимую информацию, например текущее время (этот метод проще, но подразумевает большие ограничения).

Теперь обсудим два полезных пакета Go, связанных с тестированием: `httptest` и `iotest`.

11.7. ОШИБКА #88: НЕ ИСПОЛЬЗОВАТЬ ПАКЕТЫ УТИЛИТ ДЛЯ ТЕСТИРОВАНИЯ

Стандартная библиотека предоставляет пакеты утилит для тестирования. И когда разработчик не знает о них, то может полагаться либо на другие решения, которые не так удобны, либо вообще заниматься велосипедостроением. В этом разделе рассмотрим два таких пакета: один поможет при работе с HTTP, а другой при вводе/выводе и использовании ридеров и райтеров.

11.7.1. Пакет httptest

Пакет `httptest` (https://pkg.go.dev/net/http/httptest) предлагает утилиты для тестирования как HTTP-клиентов, так и HTTP-серверов. Рассмотрим оба сценария.

Обсудим то, как `httptest` помогает при создании HTTP-сервера. Реализуем обработчик, который выполняет некоторые базовые действия: записывает заголовок и тело, а также возвращает определенный код состояния. Для простоты и большей ясности опустим обработку ошибок:

```
func Handler(w http.ResponseWriter, r *http.Request) {
   w.Header().Add("X-API-VERSION", "1.0")
   b, _ := io.ReadAll(r.Body)
   _, _ = w.Write(append([]byte("hello "), b...))
   w.WriteHeader(http.StatusCreated)
}
```

Объединение приветствия (hello) с телом запроса

HTTP-обработчик принимает два аргумента: запрос и способ написания ответа. Пакет `httptest` предоставляет утилиты для них обоих. Для запроса мы можем применить `httptest.NewRequest`, который создает `*http.Request` с использованием метода HTTP, URL-адреса и тела. Для ответа возьмем `httptest.NewRecorder`, чтобы записать изменения, сделанные в обработчике. Напишем юнит-тест для этого обработчика:

```
func TestHandler(t *testing.T) {
   req := httptest.NewRequest(http.MethodGet, "http://localhost",
      strings.NewReader("foo"))
   w := httptest.NewRecorder()
   Handler(w, req)

   if got := w.Result().Header.Get("X-API-VERSION"); got != "1.0" {
      t.Errorf("api version: expected 1.0, got %s", got)
   }

   body, _ := ioutil.ReadAll(wordy)
   if got := string(body); got != "hello foo" {
      t.Errorf("body: expected hello foo, got %s", got)
   }

   if http.StatusOK != w.Result().StatusCode {
      t.FailNow()
   }
}
```

Создание запроса

Создание регистратора ответов

Вызов обработчика

Проверка заголовка HTTP

Проверка тела HTTP

Проверка кода статуса HTTP

Тестирование обработчика с помощью `httptest` не проверяет транспорт (часть HTTP). В центре внимания теста находится прямой вызов обработчика с запросом и способ записи ответа. Затем, используя регистратор ответов, мы записываем утверждения для проверки HTTP-заголовка, тела и кода состояния.

Посмотрим с другой стороны: тестирование HTTP-клиента. Для этого создадим клиент, ответственный за запрос конечной точки HTTP, который вычисляет, сколько времени требуется для перехода от одной координаты к другой. Этот клиент выглядит так:

```
func (c DurationClient) GetDuration(url string,
   lat1, lng1, lat2, lng2 float64) (
   time.Duration, error) {
   resp, err := c.client.Post(
      url, "application/json",
      buildRequestBody(lat1, lng1, lat2, lng2),
   )
   if err != nil {
      return 0, err
   }
   return parseResponseBody(resp.Body)
}
```

Код выполняет HTTP-запрос POST по указанному URL-адресу и возвращает разобранный ответ (скажем, какой-то JSON).

А если надо протестировать этот клиент? Один из вариантов — использовать Docker и запустить фиктивный сервер для возврата некоторых предварительно

зарегистрированных ответов. Но такой подход замедляет выполнение теста. Другой вариант — использовать `httptest.NewServer` для создания локального HTTP-сервера на основе обработчика, который мы предоставим. Как только сервер запущен и выполняется, мы можем передать его URL в `GetDuration`:

```
func TestDurationClientGet(t *testing.T) {
   srv := httptest.NewServer(  <-- Запуск HTTP-сервера
      http.HandlerFunc(
         func(w http.ResponseWriter, r *http.Request) {
            _, _ = w.Write([]byte(`{"duration": 314}`))  <-- Регистрация обработчика для обслуживания ответа
         },
      ),
   )
   defer srv.Close()  <-- Отключение сервера

   client := NewDurationClient()
   duration, err :=
      client.GetDuration(srv.URL, 51.551261, -0.1221146, 51.57, -0.13)  <-- Указание URL-сервера

   if err != nil {
      t.Fatal(err)
   }
   if duration != 314*time.Second {  <-- Проверка ответа
      t.Errorf("expected 314 seconds, got %v", duration)
   }
}
```

Здесь мы создаем сервер со статическим обработчиком, возвращающим 314 секунд. Мы также можем делать утверждения на основе отправленного запроса. Кроме того, когда мы вызываем `Get-Duration`, то указываем URL-адрес запущенного сервера. По сравнению с тестированием обработчика этот тест выполняет фактический HTTP-вызов, но делает это всего за несколько миллисекунд.

Можно также запустить новый сервер, применив TLS с помощью `httptest.NewTLSServer`, и создать, но пока не запускать сервер с помощью `httptest.NewUnstartedServer`, чтобы его можно было запустить лениво.

Помните, что `httptest` очень полезен при работе в контексте HTTP-приложений. Создаем мы сервер или клиент, `httptest` поможет разработать эффективные тесты.

11.7.2. Пакет iotest

В пакете `iotest` (https://pkg.go.dev/testing/iotest) реализованы утилиты для тестирования ридеров и райтеров. Это удобный пакет, о котором разработчики часто забывают.

При реализации собственного `io.Reader` важно не забыть протестировать его с помощью `iotest.TestReader`. Эта вспомогательная функция проверяет, правильно ли ведет себя ридер: возвращает ли он число прочитанных байтов точно, заполняет ли заданный срез и т. д. Она тестирует также различные варианты поведения, если предоставленный модуль чтения реализует такие интерфейсы, как `io.ReaderAt`.

Предположим, есть пользовательский `LowerCaseReader`, который передает поток строчных букв из заданного источника ввода `io.Reader`. Вот как протестировать поведение ридера и убедиться, что оно правильное:

```
func TestLowerCaseReader(t *testing.T) {
   err := iotest.TestReader(
      &LowerCaseReader{reader: strings.NewReader("aBcDeFgHiJ")},
       []byte("acegi"),
   )
   if err != nil {
      t.Fatal(err)
   }
}
```

Предоставление io.Reader

Что ожидается

Мы вызываем `iotest.TestReader`, предоставляя пользовательский `LowerCaseReader` и задавая то, что ожидаем на входе/при вводе, — строчные буквы `acegi`.

Еще один вариант использования пакета `iotest` — убедиться, что приложение, использующее ридеры и райтеры, устойчиво к ошибкам:

- `iotest.ErrReader` создает `io.Reader`, который возвращает указанную ошибку.
- `iotest.HalfReader` создает `io.Reader`, который читает только половину запрошенного количества байтов из другого `io.Reader`.
- `iotest.OneByteReader` создает `io.Reader`, который читает по одному байту для каждого непустого чтения из другого `io.Reader`.
- `iotest.TimeoutReader` создает `io.Reader`, который возвращает ошибку при втором чтении без данных. Последующие вызовы будут успешными.
- `iotest.TruncateWriter` создает `io.Reader`, который записывает в другой io.Writer, но молча прекращает работу после записи `n` байт.

Например, мы реализуем следующую функцию, которая начинается с чтения всех байтов из ридера:

```
func foo(r io.Reader) error {
   b, err := io.ReadAll(r)
   if err != nil {
      return err
   }
   // ...
}
```

Мы хотим убедиться, что функция устойчива, если, например, ридер во время чтения выдаст сбой (симуляция сетевой ошибки):

```
func TestFoo(t *testing.T) {
   err := foo(iotest.TimeoutReader(
      strings.NewReader(randomString(1024)),
   ))
   if err != nil {
      t.Fatal(err)
   }
}
```

Упаковка предоставленного io.Reader с использованием io.TimeoutReader

Мы оборачиваем `io.Reader` с помощью `io.TimeoutReader`. Как уже говорилось, второе чтение завершится неудачей. Если мы запустим этот тест, чтобы убедиться, что функция толерантна к ошибкам, то получим провал теста. Действительно, `io.ReadAll` возвращает все ошибки, которые встречает.

Зная это, реализуем пользовательскую функцию `ReadAll`, которая терпима к `n` ошибкам:

```
func readAll(r io.Reader, retries int) ([]byte, error) {
   b := make([]byte, 0, 512)
   for {
      if len(b) == cap(b) {
         b = append(b, 0)[:len(b)]
      }
      n, err := r.Read(b[len(b):cap(b)])
      b = b[:len(b)+n]
      if err != nil {
         if err == io.EOF {
            return b, nil
         }
         retries--
         if retries < 0 {
            return b, err
         }
      }
   }
}
```

Допускает повторные попытки

Эта реализация похожа на `io.ReadAll`, но также обрабатывает настраиваемые повторные попытки. Если изменить реализацию исходной функции, чтобы использовать собственный `readAll` вместо `io.ReadAll`, тест больше не будет заканчиваться неудачей:

```
func foo(r io.Reader) error {
   b, err := readAll(r, 3)  <-- До трех повторных попыток
   if err != nil {
      return err
   }
   // ...
}
```

Мы рассмотрели пример того, как проверить, что функция толерантна к ошибкам при чтении из `io.Reader`, выполнив тест с использованием пакета `iotest`.

При выполнении ввода/вывода и при работе с `io.Reader` и `io.Writer` помните, насколько удобен пакет `iotest`. Как мы видели, он предоставляет утилиты для проверки поведения пользовательского `io.Reader` и проверки нашего приложения на наличие ошибок, возникающих при чтении или записи данных.

В следующем разделе обсудим, как не попадать в распространенные ловушки, которые могут привести к созданию неточных бенчмарков.

11.8. ОШИБКА #89: ПИСАТЬ НЕТОЧНЫЕ БЕНЧМАРКИ

О производительности никогда не стоит гадать. Занимаясь оптимизацией, приходится иметь дело с таким множеством факторов, что даже если мы и уверены в неких результатах, нелишне будет их протестировать. Но писать бенчмарки не так-то просто, и написав неточные, легко сделать на их основе неправильные выводы. Цель этого раздела — обсудить типичные ловушки, ведущие к неточным результатам.

Рассмотрим, как работают бенчмарки в Go. Скелет любого бенчмарка выглядит так:

```
func BenchmarkFoo(b *testing.B) {
   for i := 0; i < b.N; i++ {
      foo()
   }
}
```

Имя функции начинается с префикса `Benchmark`. Тестируемая функция (`foo`) вызывается внутри цикла `for`. Величина `b.N` представляет собой переменное количество итераций. При запуске теста Go пытается уложиться в требуемое

время, которое по умолчанию задано равным 1 секунде и может быть изменено с помощью флага `-benchtime`. `b.N` начинается с 1. Если бенчмарк завершается менее чем за 1 секунду, значение `b.N` увеличивается и бенчмарк запускается снова, пока `b.N` не станет примерно равным времени бенчмарка:

```
$ go test -bench=.
cpu: Intel(R) Core(TM) i5-7360U CPU @ 2.30GHz
BenchmarkFoo-4                73          16511228 ns/op
```

Здесь бенчмарк занял около 1 секунды, а `foo` был выполнен 73 раза, при среднем времени выполнения 16 511 228 наносекунд. Изменим время бенчмарка, используя `-benchtime`:

```
$ go test -bench=. -benchtime=2s
BenchmarkFoo-4               150          15832169 ns/op
```

В этом бенчмарке `foo` выполнялся 150 раз.

Теперь рассмотрим некоторые типичные ловушки.

11.8.1. Не сбрасывать или не ставить на паузу таймер

В некоторых случаях перед циклом бенчмарка требуется выполнить какие-либо операции. Они могут занять много времени (например, для создания большого среза данных) и сильно повлиять на результаты тестов:

```
func BenchmarkFoo(b *testing.B) {
   expensiveSetup()
   for i := 0; i < b.N; i++ {
      functionUnderTest()
   }
}
```

В этом случае перед входом в цикл используем метод `ResetTimer`:

```
func BenchmarkFoo(b *testing.B) {
   expensiveSetup()
   b.ResetTimer()  <-- Сброс таймера бенчмарка
   for i := 0; i < b.N; i++ {
      functionUnderTest()
   }
}
```

Вызов `ResetTimer` обнуляет счетчики прошедшего с начала бенчмарка времени и выделения памяти. Таким образом, дорогостоящая настройка может быть исключена из результатов теста.

Но что, если приходится выполнять длительные подготовительные шаги не один раз, а в каждой итерации цикла?

```
func BenchmarkFoo(b *testing.B) {
   for i := 0; i < b.N; i++ {
      expensiveSetup()
      functionUnderTest()
   }
}
```

Тут мы не можем сбрасывать таймер, потому что такой сброс будет выполняться внутри цикла во время каждой итерации. Но можно остановить и возобновить работу таймера бенчмарка, окружив таким образом вызов `expensiveSetup`:

```
func BenchmarkFoo(b *testing.B) {
   for i := 0; i < b.N; i++ {
      b.StopTimer()   <-- Приостановка таймера бенчмарка
      expensiveSetup()
      b.StartTimer()  <-- Возобновление работы таймера бенчмарка
      functionUnderTest()
   }
}
```

Здесь мы приостанавливаем работу бенчмарка, чтобы выполнить дорогостоящую настройку, а затем возобновляем работу таймера.

> **ПРИМЕЧАНИЕ** С этим подходом связана некая загвоздка: если тестируемая функция выполняется слишком быстро по сравнению с функцией настройки, общее время выполнения бенчмарка может быть слишком большим. Причина в том, что для достижения значения, равного `benchtime`, потребуется гораздо больше времени, чем 1 секунда. Вычисление контрольного времени основано исключительно на времени выполнения `functionUnderTest`. Если в каждой итерации цикла мы будем находиться в состоянии ожидания долгое время, бенчмарк будет намного медленнее, чем 1 секунда. Одним из способов смягчения последствий будет уменьшение `benchtime`.

Важно убедиться, что мы используем методы таймера, чтобы сохранить точность бенчмарка.

11.8.2. Делать неверные предположения о микробенчмарках

Микробенчмарк измеряет крошечную вычислительную единицу, и в связи с этим очень легко сделать неверное предположение. Например, мы изначально не

уверены, следует ли использовать `atomic.StoreInt32` или `atomic.StoreInt64` (при условии, что обрабатываемые значения всегда умещаются в 32 бита). Напишем бенчмарк для сравнения обеих функций:

```
func BenchmarkAtomicStoreInt32(b *testing.B) {
   var v int32
   for i := 0; i < b.N; i++ {
      atomic.StoreInt32(&v, 1)
   }
}

func BenchmarkAtomicStoreInt64(b *testing.B) {
   var v int64
   for i := 0; i < b.N; i++ {
      atomic.StoreInt64(&v, 1)
   }
}
```

Допустим, при запуске мы получаем следующее:

```
cpu: Intel(R) Core(TM) i5-7360U CPU @ 2.30GHz
BenchmarkAtomicStoreInt32
BenchmarkAtomicStoreInt32-4 197107742 5.682 ns/op
BenchmarkAtomicStoreInt64
BenchmarkAtomicStoreInt64-4 213917528 5.134 ns/op
```

Мы могли бы легко принять эти результаты как должное и решить использовать `atomic.Store-Int64`, поскольку кажется, что он быстрее. Но теперь, чтобы провести *честный* бенчмарк, изменим порядок и сначала запустим `atomic.StoreInt64`, а затем `atomic.StoreInt32`. Вот пример результата:

```
BenchmarkAtomicStoreInt64
BenchmarkAtomicStoreInt64-4 224900722 5.434 ns/op
BenchmarkAtomicStoreInt32
BenchmarkAtomicStoreInt32-4 230253900 5.159 ns/op
```

В этот раз `atomic.StoreInt32` показал лучшие результаты. Что же произошло?

В случае микробенчмарков на их результаты могут влиять многие факторы: занятость процессора во время выполнения бенчмарков, режимы управления питанием, лучшее выравнивание последовательности инструкций в кэше и многие другие факторы, возможно, даже выходящие за рамки проекта на Go.

> **ПРИМЕЧАНИЕ** Мы должны убедиться, что машина, на которой выполняется бенчмарк, простаивает. Но в фоновом режиме могут выполняться какие-либо внешние процессы, что может повлиять на результаты. По этой причине такие инструменты, как `perflock`, могут ограничивать ресурсы CPU, которые

может потреблять бенчмарк. Например, можно запустить бенчмарк, указав, что на него должно расходоваться не более 70 % всех доступных ресурсов CPU, отдав 30 % операционной системе и другим процессам и снизив тем самым влияние фактора занятости процессора на результаты.

Один из вариантов — увеличить время бенчмарка с помощью параметра `-benchtime`. Подобно закону больших чисел в теории вероятностей, если мы запускаем бенчмарк многократно, его результат должен стремиться к ожидаемому значению (при условии, что мы опускаем возможные преимущества от кэширования инструкций и от других техник).

Другой вариант — использовать внешние инструменты поверх классического инструментария бенчмаркинга. Например, инструментарий `benchstat`, являющийся частью репозитория `golang.org/x`, позволяет получать и сравнивать статистику выполнения бенчмарков.

Запустим бенчмарк с параметром `-count` 10 раз и запишем результат в файл:

```
$ go test -bench=. -count=10 | tee stats.txt
cpu: Intel(R) Core(TM) i5-7360U CPU @ 2.30GHz
BenchmarkAtomicStoreInt32-4 234935682 5.124 ns/op
BenchmarkAtomicStoreInt32-4 235307204 5.112 ns/op
// ...
BenchmarkAtomicStoreInt64-4 235548591 5.107 ns/op
BenchmarkAtomicStoreInt64-4 235210292 5.090 ns/op
// ...
```

Затем запустим `benchstat` для этого файла:

```
$ benchstat stats.txt
name time/op
AtomicStoreInt32-4 5.10ns ± 1%
AtomicStoreInt64-4 5.10ns ± 1%
```

Результаты одинаковы: выполнение обеих функций занимает в среднем 5.10 наносекунды. Мы видим и процентное отклонение в результатах выполнения: ± 1 %. Эта метрика говорит нам, что оба бенчмарка стабильны, что дает бóльшую степень уверенности в рассчитанных средних результатах. Вместо того чтобы делать вывод о том, что `atomic.StoreInt32` быстрее или медленнее, следует заключить, что время его выполнения аналогично времени выполнения `atomic.StoreInt64` для протестированного нами варианта использования (в конкретной версии Go на конкретной машине).

К микробенчмаркам следует подходить с большой осторожностью. На их результаты могут сильно влиять многие факторы, что может привести к ошибочным

предположениям и выводам. Увеличение времени бенчмарков, повторение отдельных и получение статистики с помощью `benchstat` — все это поможет ограничить влияние внешних факторов и получать более точные результаты, ведущие к справедливым выводам.

Следует критически относиться к результатам какого-либо микробенчмарка, выполненного на каком-то одном процессоре, если оказывается, что приложение будет запускаться на другой системе, которая может вести себя совершенно иначе, чем та, на которой мы запускали микробенчмарки.

11.8.3. Небрежное отношение к оптимизациям компилятора

Оптимизации компилятора также могут приводить к некорректным бенчмаркам, что, в свою очередь, приводит к неверным выводам. В этом разделе рассмотрим проблему Go 14813 (https://github.com/golang/go/issues/14813, также обсуждается членом проекта GO Дейвом Чейни) с функцией, подсчитывающей количество битов, установленных в 1:

```
const m1 = 0x5555555555555555
const m2 = 0x3333333333333333
const m4 = 0x0f0f0f0f0f0f0f0f
const h01 = 0x0101010101010101

func popcnt(x uint64) uint64 {
   x -= (x >> 1) & m1
   x = (x & m2) + ((x >> 2) & m2)
   x = (x + (x >> 4)) & m4
   return (x * h01) >> 56
}
```

Эта функция принимает и возвращает `uint64`. Для бенчмаркинга этой функции сделаем следующее:

```
func BenchmarkPopcnt1(b *testing.B) {
   for i := 0; i < b.N; i++ {
      popcnt(uint64(i))
   }
}
```

Но после выполнения бенчмарка мы получим удивительно низкий результат:

```
cpu: Intel(R) Core(TM) i5-7360U CPU @ 2.30GHz
BenchmarkPopcnt1-4      1000000000               0.2858 ns/op
```

Промежуток времени в 0.28 наносекунды примерно равен одному тактовому циклу, поэтому полученное число неправдоподобно низкое. Проблема в том, что разработчик был недостаточно внимателен к оптимизациям компилятора. В этом случае тестируемая функция достаточно проста, чтобы компилятор воспринял ее в качестве кандидата для *встраивания* (inlining): оптимизации, в результате которой вызов функции заменяется непосредственно ее телом и позволяет предотвратить вызов функции, которая имеет небольшой след. Как только функция оказывается встроена, компилятор замечает, что ее вызов не приводит ни к каким побочным эффектам, и заменяет ее следующим бенчмарком:

```
func BenchmarkPopcnt1(b *testing.B) {
   for i := 0; i < b.N; i++ {
   // Пустое место
   }
}
```

Бенчмарк теперь пуст, поэтому мы получили результат, близкий к продолжительности одного тактового цикла. Чтобы это не произошло, лучшей практикой будет вот что:

- Во время каждой итерации в цикле присваивайте полученный результат какой-то локальной переменной (локально в контексте функции бенчмарка).
- При последнем выполнении цикла результат присваивайте какой-либо глобальной переменной.

В нашем случае мы напишем такой бенчмарк:

```
var global uint64  <-- Определение глобальной переменной
func BenchmarkPopcnt2(b *testing.B) {
   var v uint64  <-- Определение локальной переменной
   for i := 0; i < b.N; i++ {
      v = popcnt(uint64(i))  <-- Присвоение результата локальной переменной
   }
   global = v  <-- Присвоение результата глобальной переменной
}
```

ПРИМЕЧАНИЕ А почему бы не назначить результат вызова `popcnt` непосредственно `global`, чтобы упростить тест? Запись в глобальную переменную медленнее, чем в локальную. Мы обсуждаем эти вопросы в разделе, посвященном разбору ошибки #95 (не понимать различий между стеком и кучей). Поэтому следует записывать каждый результат в локальную переменную, чтобы ограничить использование памяти на каждой итерации цикла.

`global` — глобальная, а `v` — локальная переменная, область видимости которой ограничена функцией бенчмарка. Во время каждой итерации цикла мы присваиваем результат `popcnt` этой локальной переменной, а результат последней итерации — глобальной переменной `global`.

Если запустить эти два бенчмарка, мы получим значительную разницу в их результатах:

```
cpu: Intel(R) Core(TM) i5-7360U CPU @ 2.30GHz
BenchmarkPopcnt1-4      1000000000               0.2858 ns/op
BenchmarkPopcnt2-4      606402058                1.993 ns/op
```

`BenchmarkPopcnt2` — это точная версия бенчмарка. Она гарантирует, что мы избегаем влияния оптимизации, связанной со встраиванием функций, которая может искусственно снижать время выполнения или даже убирать вызов тестируемой функции. Если же полагаться на результаты `BendmarkPopcnt1`, можно сделать неверные предположения.

Избегайте влияния тех приемов, которые применяются в компиляторе для оптимизации и которые могут заметно повлиять на результаты. Это также позволит избежать неверных предположений.

11.8.4. Эффект наблюдателя

В физике эффект наблюдателя заключается в том, что в результате наблюдения за какой-то системой происходит изменение ее состояния. Этот эффект может также проявляться в бенчмарках и приводить к неправильным выводам о результатах. Посмотрим на пример, где возникает этот эффект, и попробуем смягчить его.

Требуется реализовать функцию, получающую матрицу элементов типа `int64`. Эта матрица имеет фиксированное число столбцов — 512, и нужно вычислить общую сумму первых восьми столбцов, как показано на рис. 11.2.

В целях оптимизации нужно определить, оказывает ли изменение количества столбцов какое-то влияние на работу функции, поэтому мы реализуем вторую функцию с 513 столбцами. Она выглядит так:

```
func calculateSum512(s [][512]int64) int64 {
   var sum int64
   for i := 0; i < len(s); i++ {   <-- Итерации по каждой строке
      for j := 0; j < 8; j++ {   <-- Итерации по первым восьми колонкам
         sum += s[i][j]   <-- Увеличение суммы
```

```
      }
   }
   return sum
}

func calculateSum513(s [][513]int64) int64 {
   // Тот же код, что и для calculateSum512
}
```

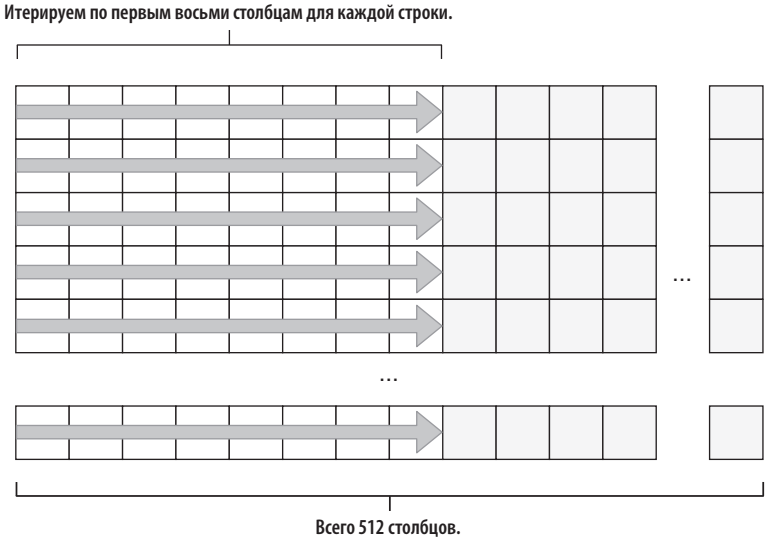


Рис. 11.2. Вычисление суммы первых восьми столбцов

Мы проводим итерации по каждой строке, а затем по первым восьми столбцам и увеличиваем вычисляемую сумму, которую возвращаем. Реализация в `CalculatesUm513` будет точно такой же.

Нужно сравнить эти функции, чтобы решить, какая из них наиболее эффективна, учитывая, что количество строк фиксированно:

```
const rows = 1000

var res int64

func BenchmarkCalculateSum512(b *testing.B) {
   var sum int64
   s := createMatrix512(rows)  <-- Создание матрицы из 512 колонок
   b.ResetTimer()
   for i := 0; i < b.N; i++ {
      sum = calculateSum512(s)  <-- Вычисление суммы
```

```
    }
    res = sum
}

func BenchmarkCalculateSum513(b *testing.B) {
    var sum int64
    s := createMatrix513(rows)  <-- Создание матрицы из 513 колонок
    b.ResetTimer()
    for i := 0; i < b.N; i++ {
        sum = calculateSum513(s)  <-- Вычисление суммы
    }
    res = sum
}
```

Мы создаем матрицу только один раз, чтобы ограничить влияние на результаты, и поэтому вызываем `CreateMatrix512` и `CreateMatrix513` вне цикла. Ожидается, что результаты будут похожи, так как итерации проводятся только по первым восьми столбцам. Но это не так (на моей машине):

```
cpu: Intel(R) Core(TM) i5-7360U CPU @ 2.30GHz
BenchmarkCalculateSum512-4        81854             15073 ns/op
BenchmarkCalculateSum513-4       161479              7358 ns/op
```

Результаты второго бенчмарка с 513 столбцами показывают, что он примерно на 50 % быстрее первого. Но поскольку мы проводим итерации только по первым восьми столбцам, этот результат несколько удивителен.

Чтобы понять эту разницу, нужно понять основы того, как устроен кэш процессора. В нем есть разные кэши (обычно L1, L2 и L3). Они снижают среднее время доступа к данным из основной памяти. При некоторых условиях процессор может извлекать данные из основной памяти и копировать их в L1. В этом случае CPU пытается получить в L1 подмножество матрицы, которое используется в функции `Calculatesum` (то есть первые восемь столбцов каждой строки). Но матрица соответствует объему памяти в одном случае (513 столбцов), а в другом (512 столбцов) — нет.

> **ПРИМЕЧАНИЕ** В этой главе я не буду вдаваться в объяснение причин. Рассмотрим этот вопрос в разделе, посвященном ошибке #91 (не понимать устройство кэша CPU).

Возвращаясь к бенчмарку, можно сказать, что главная проблема заключается в том, что мы продолжаем повторно использовать одну и ту же матрицу в обоих случаях. Поскольку функция выполняется тысячи раз, мы не измеряем параметры того, как происходит ее выполнение, когда она получает простую новую матрицу. Вместо этого мы измеряем функцию, которая получает

матрицу, подмножество ячеек которой уже есть в кэше. Поскольку выполнение `CalculateSum513` сопровождается меньшим количеством промахов кэша, у нее оказывается лучшее время этого выполнения.

Это пример проявления эффекта наблюдателя. Поскольку мы продолжаем наблюдать за неоднократно вызываемой функцией, которая интенсивно потребляет ресурсы процессора, то в игру может вступить кэширование процессора и повлиять на результаты. Чтобы предотвратить этот эффект, в примере следует создавать матрицу во время каждого теста вместо ее переиспользования:

```
func BenchmarkCalculateSum512(b *testing.B) {
   var sum int64
   for i := 0; i < b.N; i++ {
      b.StopTimer()
      s := createMatrix512(rows)
      b.StartTimer()
      sum = calculateSum512(s)
   }
   res = sum
}
```

Создание новой матрицы во время каждой итерации цикла

Новая матрица теперь создается во время каждой итерации. Если снова запустить бенчмарк (и, соответственно, сделать поправку в `benchtime`, а иначе для выполнения потребуется слишком много времени), то результаты окажутся уже ближе друг к другу:

```
cpu: Intel(R) Core(TM) i5-7360U CPU @ 2.30GHz
BenchmarkCalculateSum512-4         1116              33547 ns/op
BenchmarkCalculateSum513-4          998              35507 ns/op
```

Теперь мы не делаем неправильный вывод, что `CalculatesUm513` быстрее, а видим, что при получении новой матрицы оба бенчмарка приводят к близким результатам.

Из-за переиспользования одной и той же матрицы кэш CPU значительно повлиял на результаты. Чтобы предотвратить это, пришлось создавать новую матрицу во время каждой итерации. Помните, что наблюдение за тестируемой функцией может привести к значительным различиям в результатах, особенно в контексте микробенчмарков функций, привязанных к CPU, где низкоуровневые оптимизации имеют значение. Заставляя бенчмарк воссоздавать данные во время каждой итерации, можно предотвращать этот эффект.

В последнем разделе главы обсудим несколько общих рекомендаций по тестированию в Go.

11.9. ОШИБКА #90: НЕ ИЗУЧАТЬ ВСЕ ВОЗМОЖНОСТИ ТЕСТИРОВАНИЯ В GO

Разработчики должны знать о конкретных возможностях, функциях и опциях тестирования в Go, иначе оно станет менее точным и даже менее эффективным. В этом разделе обсуждаются темы, которые облегчат написание тестов.

11.9.1. Покрытие тестами

При разработке полезно видеть, для каких частей кода есть соответствующие тесты. Что получить эту информацию, используйте флаг `-coverprofile`:

```
$ go test -coverprofile=coverage.out ./...
```

Эта команда создает файл coverage.out, который можно открыть с помощью `go tool cover`:

```
$ go tool cover -html=coverage.out
```

Эта команда открывает браузер и показывает покрытие для каждой строки кода.

По умолчанию такой анализ проводится только для текущего тестируемого пакета. Предположим, что есть структура:

```
/myapp
   |_ foo
      |_ foo.go
      |_ foo_test.go
   |_ bar
      |_ bar.go
      |_ bar_test.go
```

Если какая-то часть foo.go тестируется только в bar_test.go, по умолчанию она не будет отображаться в отчете о покрытии каким-либо тестом. Чтобы получить эту информацию, мы должны находиться в папке myapp и использовать флаг `-coverpkg`:

```
go test -coverpkg=./... -coverprofile=coverage.out ./...
```

Помните о такой возможности, чтобы увидеть текущее покрытие кода и решить, какие части заслуживают большего числа тестов.

ПРИМЕЧАНИЕ Будьте бдительны, когда речь идет о погоне за покрытием кода. Стопроцентное покрытие тестами не означает, что приложение не содержит ошибок. Правильное понимание того, что покрывают тесты, важнее любого статического порогового значения.

11.9.2. Тестирование из другого пакета

Один из подходов к написанию юнит-тестов состоит в том, чтобы сосредоточиться на поведении, а не на внутреннем устройстве. Предположим, мы предоставляем клиентам какой-то API и хотим, чтобы тесты были направлены на те его аспекты, которые видны извне, а не на детали его реализации. Таким образом, если реализация изменится (например, если мы разделим какую-то одну функцию на две), тесты останутся прежними. Их будет легче понимать, потому что они показывают, как используется наш API. Чтобы применить такой подход, нужно использовать другой пакет.

В Go все файлы в папке должны принадлежать одному и тому же пакету, за одним исключением: тестовый файл может принадлежать пакету `_test`. Допустим, следующий исходный файл counter.go относится к пакету `counter`:

```
package counter

import "sync/atomic"

var count uint64

func Inc() uint64 {
   atomic.AddUint64(&count, 1)
   return count
}
```

Тестовый файл может находиться в том же пакете и иметь доступ к внутренним компонентам, таким как переменная `count`. Или он может быть в пакете `counter_test`, как этот файл counter_test.go:

```
package counter_test

import (
   "testing"
   "myapp/counter"
)

func TestCount(t *testing.T) {
   if counter.Inc() != 1 {
      t.Errorf("expected 1")
   }
}
```

В этом случае тест реализован во внешнем пакете и не может получить доступ к внутренним компонентам — переменной `count`. Используя подобный подход, мы гарантируем, что в тесте не будут использоваться неэкспортированные элементы. Следовательно, он сосредоточится на тестировании открытого поведения.

11.9.3. Вспомогательные функции

При написании тестов мы можем обрабатывать ошибки иначе, чем в окончательном варианте кода, который используется в продакшене. Например, нужно протестировать функцию, которая принимает в качестве аргумента структуру `Customer`. Поскольку действия по созданию `Customer` будут переиспользоваться, создадим для целей тестирования специальную функцию `createCustomer`. Она вернет возможную ошибку вместе с `Customer`:

```
func TestCustomer(t *testing.T) {
   customer, err := createCustomer("foo")
   if err != nil {
      t.Fatal(err)
   }
   // ...
}
func createCustomer(someArg string) (Customer, error) {
   // Создание customer
   if err != nil {
      return Customer{}, err
   }
   return customer, nil
}
```

Создание customer и проверка на предмет ошибок

Мы создаем клиент с помощью вспомогательной функции `createCustomer`, а затем выполняем оставшуюся часть теста. Но в контексте функций тестирования можно упростить обработку ошибок, передав переменную `*testing.T` вспомогательной функции:

```
func TestCustomer(t *testing.T) {
   customer := createCustomer(t, "foo")
   // ...
}
func createCustomer(t *testing.T, someArg string) Customer {
   // Создание customer
   if err != nil {
      t.Fatal(err)
   }
   return customer
}
```

Вызов вспомогательной функции и передача ей t

Тест не проходит напрямую, если мы не можем создать клиент

Чтобы не возвращать ошибку, `createCustomer` прямо останавливает тест как проваленный, если не может создать `Customer`. Это делает `TestCustomer` короче и легче для чтения.

Помните об этом, чтобы улучшить свои тесты.

11.9.4. Настройка и демонтаж

В некоторых случаях может потребоваться подготовить тестовую среду. Например, в интеграционных тестах мы запускаем определенный контейнер Docker, а затем останавливаем его. Мы можем вызывать функции настройки и демонтажа для каждого теста или пакета. К счастью, в Go возможны оба варианта.

Чтобы делать это для каждого теста, вызываем функцию настройки как предварительное действие, а функцию демонтажа с помощью `defer`:

```
func TestMySQLIntegration(t *testing.T) {
   setupMySQL()
   defer teardownMySQL()
   // ...
}
```

Также можно зарегистрировать функцию, которая будет выполняться в конце теста. Предположим, что функции `TestMySQLIntegration` нужно вызвать `createConnection` для установления подключения к базе данных. Если мы хотим, чтобы эта функция также включала часть, относящуюся к демонтажу, мы можем использовать `t.Cleanup` для регистрации функции очистки:

```
func TestMySQLIntegration(t *testing.T) {
   // ...
   db := createConnection(t, "tcp(localhost:3306)/db")
   // ...
}

func createConnection(t *testing.T, dsn string) *sql.DB {
   db, err := sql.Open("mysql", dsn)
   if err != nil {
      t.FailNow()
   }
   t.Cleanup(   <-- Регистрация функции, которая будет выполняться в конце теста
      func() {
         _ = db.Close()
      })
   return db
}
```

В конце теста выполняется замыкание из `t.Cleanup`. Это упрощает написание будущих юнит-тестов, потому что они не будут отвечать за закрытие переменной `db`.

Обратите внимание, что мы можем зарегистрировать несколько функций очистки. В этом случае они будут выполняться так же, как если бы мы использовали `defer`: пришедший последним выходит первым.

Чтобы выполнять настройку и демонтаж каждого пакета, используйте функцию `TestMain`. Самая простая реализация `TestMain` выглядит так:

```
func TestMain(m *testing.M) {
   os.Exit(m.Run())
}
```

Эта функция принимает аргумент `*testing.M`, который предоставляет единственный метод `Run` для запуска всех тестов. Поэтому мы можем окружить этот вызов функциями настройки и демонтажа:

```
func TestMain(m *testing.M) {
   setupMySQL()   <-- Установка MySQL
   code := m.Run()   <-- Проведение тестов
   teardownMySQL()   <-- Демонтаж MySQL
   os.Exit(code)
}
```

Этот код запускает MySQL один раз перед всеми тестами, а затем демонтирует его.

Используя эти методы для добавления функций настройки и демонтажа, можно настраивать сложную среду для проведения тестов.

ИТОГИ

- Категоризация тестов с помощью флагов сборки, переменных среды или короткого режима делает процесс тестирования более эффективным. Создавайте категории тестов, используя флаги сборки или переменные среды (например, категории интеграционных и юнит-тестов), и разграничивайте короткие и длительные тесты, чтобы решить, какие выполнять.
- Включение флага `-race` настоятельно рекомендуется при написании конкурентных приложений. Это позволит выявлять потенциальные гонки данных, которые могут приводить к ошибкам в программах.

- Использование флага `-parallel` — эффективный способ ускорить тесты, особенно длительные.
- Используйте флаг `-shuffle`, чтобы убедиться, что набор тестов не опирается на неверные предположения, которые скрывают ошибки.
- Табличные тесты — эффективный способ сгруппировать набор похожих тестов. Так вы предотвратите дублирование кода и упростите работу с будущими обновлениями.
- Избегайте задержек с помощью синхронизации — это сделает тест более стабильным и надежным. Если синхронизация невозможна, рассмотрите подход с повторными попытками.
- Понимание того, как работать с функциями с помощью API времени, — это еще один способ сделать тест более надежным. Используйте стандартные методы: работу со временем как часть скрытой зависимости — или запрашивайте его у клиентов.
- Пакет `httptest` полезен для работы с HTTP-приложениями. Он предоставляет набор утилит для тестирования как клиентов, так и серверов.
- Пакет `iotest` помогает написать `io.Reader` и проверить, что приложение устойчиво к ошибкам.
- Бенчмарки:
  - Используйте методы `time`, чтобы обеспечить точность бенчмарка.
  - Увеличение `benchtime` или использование таких инструментов, как `benchstat`, может быть полезным при работе с микробенчмарками.
  - Следует с осторожностью подходить к результатам микробенчмарков, если система, где должно работать приложение, отличается от той, на которой выполняются микробенчмарки.
  - Убедитесь, что у тестируемой функции нет каких-либо побочных эффектов, чтобы оптимизации компилятора не ввели вас в заблуждение относительно результатов бенчмарка.
  - Чтобы предотвратить эффект наблюдателя, сделайте так, чтобы бенчмарк повторно создавал данные, используемые функцией, которая интенсивно потребляет ресурсы процессора.
- Используйте флаг `-coverprofile`, чтобы быстро увидеть, какая часть кода требует большего внимания.

- Соберите юнит-тесты в другом пакете, чтобы они фокусировались на анализе и измерении параметров поведения приложения, а не на его внутреннем устройстве.
- Обработка ошибок с помощью переменной `*testing.T` вместо классического `if err!= nil` делает код более удобочитаемым.
- Для настройки сложной тестовой среды используйте функции настройки и демонтажа, например, в случае интеграционных тестов.

12 Оптимизация

В этой главе:

- ✓ Концепция «mechanical sympathy» («любовь к железу»)
- ✓ Различия между кучей и стеком и сокращение выделения памяти
- ✓ Использование стандартных инструментов диагностики Go
- ✓ Работа сборщика мусора
- ✓ Запуск Go внутри Docker и Kubernetes

Прежде чем начать, я сделаю одно замечание: в большинстве случаев читаемый и понятный код будет лучшим решением, чем оптимизированный, но более сложный и трудный для понимания. Оптимизация, как правило, имеет свою цену, и я рекомендую следовать знаменитой цитате инженера-программиста Уэса Дайера (Wes Dyer):

> *Сделайте его правильным, сделайте его чистым, сделайте его кратким, сделайте его быстрым — именно в таком порядке.*

Это не означает, что оптимизация приложения для повышения скорости и эффективности запрещается. Например, можно определить те пути кода, которые

следует оптимизировать, потому что в этом есть необходимость, например, чтобы удовлетворить запросы наших клиентов или сократить оверхед. В этой главе обсудим общие методы оптимизации: некоторые специфичны для Go, некоторые нет. Обсудим и методы выявления узких мест, чтобы не работать вслепую.

12.1. ОШИБКА #91: НЕ ПОНИМАТЬ УСТРОЙСТВО КЭША CPU

Термин «mechanical sympathy» ввел в обиход Джеки Стюарт (Jackie Stewart) — трехкратный чемпион гонок «Формулы 1»:

> *Вам не надо быть инженером, чтобы быть хорошим гонщиком, достаточно просто чувствовать машину.*[1]

Когда мы понимаем, как какая-то система устроена и для чего, будь то автомобиль F1, самолет или компьютер, мы можем сделать дизайн своего приложения согласованным с этим устройством и достичь оптимальной производительности. В этом разделе рассмотрим конкретные примеры, когда понимание того, как работают кэши CPU, поможет оптимизировать приложения Go.

12.1.1. Архитектура CPU

Разберемся с основами архитектуры центральных процессоров и с тем, почему их кэши так важны. Возьмем в качестве примера CPU Intel Core i5-7300.

Современные процессоры используют кэширование для ускорения доступа к памяти. В большинстве случаев уровней кэширования три: L1, L2 и L3. У i5-7300 размеры кэшей следующие:

- L1: 64 Кбайт;
- L2: 256 Кбайт;
- L3: 4 Мбайт.

---

[1] В оригинале цитата Стюарта звучит как «You don't have to be an engineer to be a racing driver, but you do have to have mechanical sympathy». Mechanical sympathy можно перевести как «любовь к железу» или «чувствовать железо», хотя иногда можно встретить буквальный (и неграмотный) перевод «механическая симпатия». — *Примеч. ред.*

i5-7300 имеет два физических и четыре логических ядра; последние также называются *виртуальными ядрами*, или *потоками* (threads). В семействе Intel разделение физического ядра на несколько логических ядер называется Hyper-Threading («гиперпоточность»).

На рис. 12.1 представлено внутреннее устройство процессора Intel Core i5-7300 (`Tn` означает *n-й поток*). Каждое физическое ядро (ядро 0 и ядро 1) разделено на два логических ядра (поток 0 и поток 1). Кэш L1 разделен на два подкэша: L1D для данных и L1I для инструкций (каждый по 32 Кбайт). Кэширование относится не только к данным: когда CPU выполняет какое-то приложение, он также может кэшировать некоторые инструкции все с той же целью: ускорить выполнение в целом.

Чем ближе какой-то участок памяти к логическому ядру, тем быстрее осуществляется к нему доступ (см. http://mng.bz/o29v):

- L1 — порядка 1 наносекунды.
- L2 — примерно в 4 раза медленнее, чем для L1.
- L3 — примерно в 10 раз медленнее, чем для L1.

Эти различия помогают объяснить также и физическое расположение кэшей CPU. L1 и L2 называются *on-die* — это означает, что они расположены на том же кристалле кремния, что и остальная часть процессора. И наоборот, L3 находится вне кристалла (*off-die*), что частично объясняет разницу в задержке по сравнению с L1 и L2.

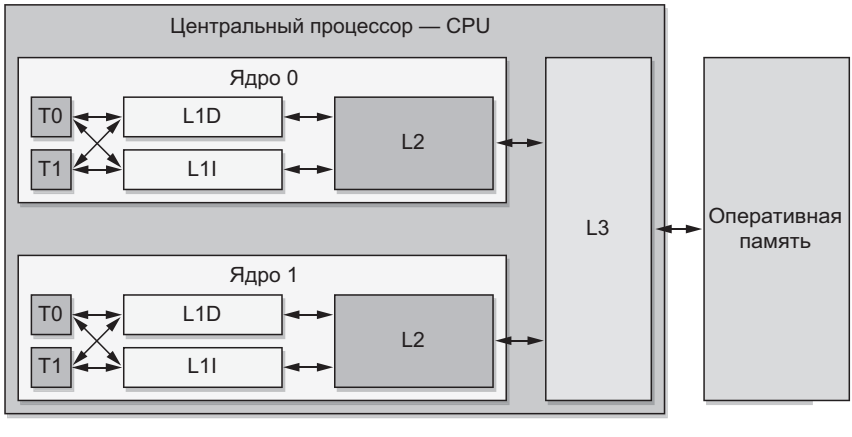


Рис. 12.1. Процессор i5-7300 имеет три уровня кэшей, два физических и четыре логических ядра

Для оперативной памяти (основной памяти, или ОЗУ) средняя скорость доступа в 50–100 раз медленнее, чем для L1. За время, необходимое для получения доступа к одной переменной, хранящейся в ОЗУ, мы можем получить доступ к 100 переменным, хранящимся в L1. Поэтому Go-разработчикам так важно заботиться о том, чтобы в приложениях использовались кэши CPU.

12.1.2. Кэш-линия

Понимание концепции кэш-линий (cache lines) критически важно. Но прежде чем объяснить, что они собой представляют, разберемся, зачем они вообще нужны.

При доступе к какому-либо конкретному месту в памяти (например, при чтении переменной) сразу после этого может произойти одно из следующих событий:

- К этому же месту будет снова сделано какое-то обращение.
- Будет сделано какое-то обращение к близлежащим ячейкам памяти.

Первая возможность относится к временной локализации, а вторая — к пространственной. Обе — части так называемого *принципа локальности ссылки* (locality of reference).

Рассмотрим следующую функцию, которая вычисляет сумму среза `int64`:

```
func sum(s []int64) int64 {
   var total int64
   length := len(s)
   for i := 0; i < length; i++ {
      total += s[i]
   }
   return total
}
```

В этом примере принцип временной локальности применяется к нескольким переменным: `i`, `length` и `total`. На протяжении всех итераций мы продолжаем обращаться к этим переменным. Принцип пространственной локальности применяется к инструкциям кода и срезу `s`. Поскольку за этим срезом стоит резервный массив, все элементы которого расположены в памяти рядом друг с другом непрерывно, то доступ к `s[0]` означает также доступ к `s[1]`, `s[2]` и т. д.

Временная локальность — одна из причин того, зачем нужны кэши CPU: чтобы ускорять повторный доступ к одним и тем же переменным. Но из-за пространственной локальности CPU копирует из ОЗУ в кэш не только одну переменную, но и *кэш-линию*.

Кэш-линия — это непрерывный сегмент памяти фиксированного размера, обычно 64 байта (8 переменных `int64`). Всякий раз, когда CPU решает кэшировать какой-либо блок памяти из ОЗУ, он копирует это блок в строку кэша. Поскольку память иерархична, то когда процессор хочет получить доступ к какой-то конкретной ячейке в памяти, он сначала проверяет ее наличие в L1-кэше, затем в L2-кэше, затем в L3-кэше, и наконец, если в этих кэшах она не найдена, то в основной памяти.

Рассмотрим на конкретном примере получение процессором блока памяти. Мы вызываем функцию `sum` для среза из 16 элементов типа `int64` в первый раз. Когда `sum` обращается к `s[0]`, этот адрес памяти еще не находится в кэше. Если процессор решит кэшировать эту переменную (мы также обсудим такое решение позже в этой главе), то он скопирует весь блок памяти (рис. 12.2).

Рис. 12.2. Обращение к `s[0]` заставляет CPU копировать блок памяти `0x000`

Сначала обращение к `s[0]` приводит к промаху кэша, потому что адрес памяти еще не находится в кэше. Это называется *принудительным промахом* (compulsory miss). Но если CPU получает доступ к блоку памяти `0x000`, обращение к элементам с 1-го по 7-й приведет к их чтению уже из кэша (попаданию в кэш). Та же логика применяется, когда `sum` обращается к `s[8]` (рис. 12.3).

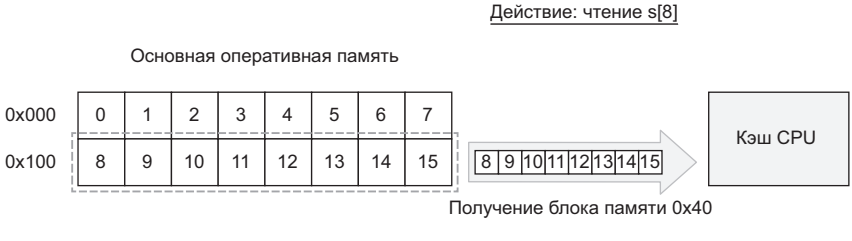


Рис. 12.3. Обращение к `s[8]` заставляет CPU копировать блок памяти `0x100`

Обращение к `s8` приводит к принудительному промаху. Но если блок памяти `0x100` скопировать в строку кэша, то это также ускорит доступ к элементам с 9-го

по 15-й. В конце концов перебор 16 элементов приводит к двум принудительным промахам кэша и к 14 попаданиям в кэш.

Стратегии кэширования CPU

Вы можете задаться вопросом о точной стратегии копирования процессором какого-то блока памяти. Например, будет ли блок копироваться на все уровни? Или только на L1? Что в этом случае происходит с L2 и L3?

Есть разные стратегии. Иногда кэши являются инклюзивными (например, данные, находящиеся в L2, также есть и в L3), а иногда — эксклюзивными (например, L3 называется *кэшем жертвы, victim cache*, поскольку он содержит только данные, вытесненные из L2).

Как правило, эти стратегии производителями CPU не разглашаются, и знать их особо не нужно. Поэтому не будем углубляться в эти вопросы.

Рассмотрим пример, чтобы проиллюстрировать, насколько быстры кэши CPU. Реализуем две функции, которые вычисляют итог при итерациях по срезу элементов типа `int64`. В одном случае итерации будут делаться по каждому второму элементу, а в другом — по каждому восьмому:

```
func sum2(s []int64) int64 {
   var total int64
   for i := 0; i < len(s); i+=2 {  ◄── Итерации по каждому второму элементу
      total += s[i]
   }
   return total
}

func sum8(s []int64) int64 {
   var total int64
   for i := 0; i < len(s); i += 8 {  ◄── Итерации по каждому восьмому элементу
      total += s[i]
   }
   return total
}
```

Обе функции одинаковы, за исключением итераций. Если мы их сравним, то интуиция может подсказать, что вторая версия будет примерно в 4 раза быстрее, поскольку нужно будет перебрать в 4 раза меньше элементов. Но запуск бенчмарка показывает, что `sum8` на моем компьютере быстрее всего на 10 %.

Причина в кэш-линиях. Обычно они имеют размер 64 байта и содержат до восьми переменных типа `int64`. В нашем случае время выполнения этих циклов

определяется параметрами доступа к памяти, а не инструкцией цикла. В первом случае три из четырех обращений приводят к попаданию в кэш. Поэтому разница во времени выполнения этих двух функций несущественна. Этот пример показывает, почему кэш-линии важны и то, что интуиция может легко обмануть, если у нас нет «mechanical sympathy» — в данном случае понимания того, как процессор кэширует данные.

Продолжим обсуждать принцип локальности ссылки и рассмотрим пример использования пространственной локальности.

12.1.3. Срез структур и структура срезов

В этом разделе рассмотрим пример, где сравним время выполнения двух функций. Первая в качестве своего аргумента принимает фрагмент структур и суммирует все поля a:

```
type Foo struct {
   a int64
   b int64
}

func sumFoo(foos []Foo) int64 {  <-- Получение среза Foo
   var total int64
   for i := 0; i < len(foos); i++ {  <-- Циклы по каждому Foo с суммированием всех полей
      total += foos[i].a
   }
   return total
}
```

sumFoo получает срез Foo и увеличивает итоговую сумму при чтении каждого поля.

Вторая функция также вычисляет сумму. Но на этот раз аргумент представляет собой структуру, содержащую срезы:

```
type Bar struct {
   a []int64  <-- a и b теперь срезы
   b []int64
}

func sumBar(bar Bar) int64 {  <-- Получение одной структуры
   var total int64
   for i := 0; i < len(bar.a); i++ {  <-- Циклы по каждому элементу a
      total += bar.a[i]  <-- Увеличение итоговой суммы
   }
   return total
}
```

`sumBar` получает одну структуру `Bar`, содержащую два среза: `a` и `b`. Она выполняет итерацию по каждому элементу `a` для увеличения `total`.

Ожидаем ли мы какой-либо разницы в скорости выполнения этих двух функций? Перед запуском бенчмарка визуально посмотрим на различия в памяти, обратившись к рис. 12.4. Оба случая имеют одинаковое количество данных: 16 элементов `Foo` в срезе и 16 элементов в срезах `Bar`. Каждая черная полоса представляет собой `int64`, который считывается и прибавляется к сумме, а каждая серая полоса обозначает `int64`, который пропускается.

Рис. 12.4. Структура срезов более компактна и поэтому требует меньше кэш-линий для итераций

В случае с `sumFoo` мы получаем срез структур, содержащий два поля — `a` и `b`. Следовательно, у нас в памяти есть последовательность из `a` и `b`. И наоборот, в случае `sumBar` мы получаем структуру, содержащую два среза, `a` и `b`. Следовательно, все элементы `a` размещаются в смежных ячейках памяти.

Эта разница не приводит к какой-либо оптимизации уплотнения памяти. Но цель обеих функций — перебрать каждый элемент `a`, и для этого требуется 4 кэш-линии в одном случае и всего 4 кэш-линии в другом.

Если мы сравним эти две функции, то `sumBar` окажется быстрее (около 20 % на моем компьютере). Основная причина — лучшая пространственная локализация, благодаря которой CPU извлекает из памяти меньше кэш-линий.

Этот пример показывает, как пространственная локализация может сильно повлиять на производительность. Чтобы оптимизировать приложение, нужно организовать данные так, чтобы получить максимальную отдачу от каждой отдельной кэш-линии.

Но достаточно ли использования пространственной локализации для помощи процессору? Все еще не хватает одной критической характеристики: предсказуемости.

12.1.4. Предсказуемость

Предсказуемость понимается как способность CPU предвидеть, что будет делать приложение, чтобы ускорить его выполнение. Рассмотрим пример, когда отсутствие предсказуемости негативно влияет на производительность приложения.

Еще раз возьмем две функции, которые суммируют список элементов. Первая перебирает связный (linked) список и суммирует все значения:

```
type node struct {   <-- Структура данных связного списка
   value int64
   next *node
}

func linkedList(n *node) int64 {
   var total int64
   for n != nil {   <-- Итерации по каждому узлу
      total += n.value   <-- Увеличение total
      n = n.next
   }
   return total
}
```

Эта функция получает связный список, итерирует по нему и увеличивает `total`.

Вторая функция `sum2` итерирует по каждому второму элементу среза:

```
func sum2(s []int64) int64 {
   var total int64
   for i := 0; i < len(s); i+=2 {   <-- Итерации по каждому второму элементу
      total += s[i]
   }
   return total
}
```

Допустим, что память для связного списка выделяется непрерывно: например, в одной функции. В 64-битной архитектуре слово имеет длину 64 бита. На рис. 12.5 сравниваются две структуры данных, которые получают функции (связный список или срез). Более темные полосы обозначают элементы типа `int64`, которые мы используем для увеличения общей суммы.

В обоих примерах мы сталкиваемся с одинаково плотным размещением в памяти. Поскольку связный список представляет собой последовательность значений и 64-битных элементов-указателей, мы увеличиваем сумму, используя для этого один элемент из такой пары. Между тем `sum2` считывает только каждый второй элемент.

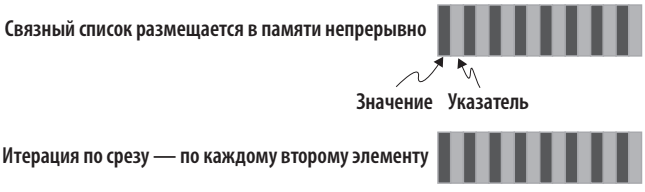


Рис. 12.5. Связные списки и срезы упаковываются в памяти аналогично

Эти две структуры данных имеют одинаковую пространственную локализацию, поэтому мы можем ожидать одинаковое время выполнения двух функций. Но функция, включающая в себя итерации по срезу, оказывается значительно быстрее (около 70 % на моем компьютере). Почему?

Чтобы понять это, обсудим концепцию striding (шагов), относящуюся к тому, как процессоры работают с данными. Есть три разных типа шагов (рис. 12.6).

- Шаг единичного размера (Unit stride) — все значения, к которым мы хотим получить доступ, располагаются последовательно, например срез элементов типа `int64`. Этот шаг предсказуем для процессора и наиболее эффективен, так как требует минимального количества кэш-линий для обхода элементов.
- Шаг постоянного размера (Constant stride) — по-прежнему предсказуем для процессора, например срез, который перебирает каждые два элемента. Этот шаг требует большего количества кэш-линий для обхода данных, поэтому он менее эффективен, чем шаг единичного размера.
- Шаг с непредсказуемым размером, или неединичный (Non-unit stride), — шаг, который процессор не может предсказать, например связный список или срез указателей. Поскольку процессор не знает, выделены ли данные последовательно, он не будет загружать кэш-линии.

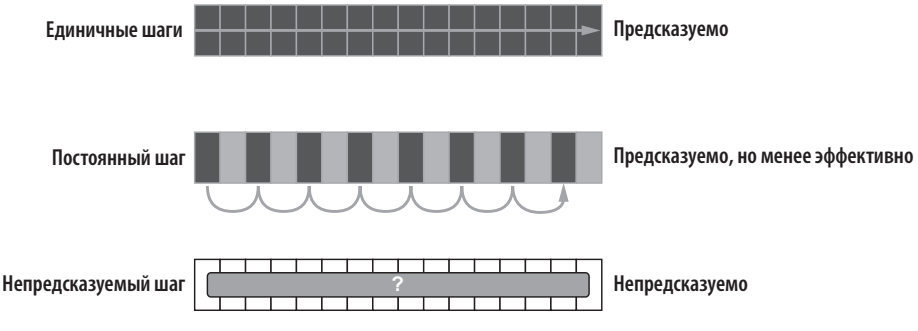


Рис. 12.6. Три типа шагов

В случае с sum2 мы имеем дело с постоянным шагом. Но для связного списка мы сталкиваемся с неединичным шагом. Даже если мы сами знаем, что данные расположены непрерывно, но об этом не знает CPU. Следовательно, он не может предсказать, как надо будет проходить по связному списку.

Из-за разного шага и схожей пространственной локализации итерация по связному списку выполняется значительно медленнее, чем по срезу значений. Обычно мы должны отдавать предпочтение единичным шагам, а не постоянным из-за лучшей пространственной локализации. Но CPU не может предсказывать шаг, отличный от единицы, независимо от того, как распределены данные, что негативно влияет на производительность.

До сих пор мы обсудили, что кэши процессора быстрые, но значительно меньшие по размеру, чем основная память. Поэтому процессору необходима стратегия для загрузки блока памяти в кэш-линию. Эта стратегия называется политикой размещения кэша и может сильно влиять на производительность.

12.1.5. Стратегия размещения кэша

При разборе ошибки #89 (писать неточные бенчмарки) мы обсуждали пример с матрицей, где нужно было вычислить общую сумму содержимого первых восьми ее столбцов. На тот момент мы не объяснили, почему изменение общего количества столбцов повлияло на результаты бенчмарков. Это может показаться нелогичным: нужно прочитать только первые восемь столбцов, почему же изменение общего количества столбцов влияет на время выполнения? Вернемся к этому примеру.

Вспомним, что там обсуждалась следующая реализация:

```
func calculateSum512(s [][512]int64) int64 {   <-- Получение матрицы из 512 столбцов
   var sum int64
   for i := 0; i < len(s); i++ {
      for j := 0; j < 8; j++ {
         sum += s[i][j]
      }
   }
   return sum
}

func calculateSum513(s [][513]int64) int64 {   <-- Получение матрицы из 513 столбцов
   // Тот же код, что и для calculateSum512
}
```

Мы проводим итерации по каждой строке, каждый раз суммируя первые восемь столбцов. Если мы проводим бенчмаркинг этих двух функций при условии, что

им всякий раз задается в качестве аргумента новая матрица, мы не наблюдаем никакой разницы. Но если продолжить переиспользовать одну и ту же матрицу, `calculateSum513` на моем компьютере будет выполняться примерно на 50 % быстрее. Причина кроется в кэше CPU и в том, как блок памяти копируется в кэш-линию. Разберемся, откуда появляется разница.

Когда CPU решает скопировать блок памяти и поместить его в кэш, он должен следовать определенной стратегии. Предположим, что кэш L1D имеет размер 32 Кбайт, а кэш-линия — объем 64 байта. Если блок размещается в L1D случайным образом, то в худшем случае процессору придется выполнить итерацию по 512 кэш-линиям, чтобы прочитать переменную. Такой вид кэша называется *полностью ассоциативным* (fully associative).

Для улучшения скорости доступа к адресу из кэша процессора разработаны различные стратегии размещения кэша. Пропустим историю и обсудим наиболее широко используемый на сегодняшний день вариант: *наборно-ассоциативный* кэш (set-associative cache), принцип действия которого основан на секционировании кэша.

Для простоты изложения на последующих рисунках мы будем работать с несколько урезанной версией этой задачи. Предположим, что:

- кэш L1D имеет размер 512 байт (8 кэш-линий);
- матрица состоит из 4 строк и 32 столбцов, и мы будем читать только первые 8 столбцов.

На рис. 12.7 показано, как эта матрица может храниться в памяти. Мы будем использовать двоичное представление адресов блоков памяти. Кроме того, примем, что серые блоки представляют первые 8 элементов типа `int64`, по которым мы хотим провести итерацию. Остальные блоки при этом пропускаются.

Каждый блок памяти содержит 64 байта и, следовательно, 8 элементов `int64`. Первый блок памяти начинается с адреса 0x0000000000000, второй — с адреса 0001000000000 (512 в двоичном формате) и т. д. Мы также показываем кэш, который может содержать 8 строк/линий (lines).

> **ПРИМЕЧАНИЕ** В разделе, посвященном разбору ошибки #94 (не знать о выравнивании данных), мы увидим, что срез не обязательно начинается в начале блока.

При использовании политики наборно-ассоциативного кэширования кэш разбивается на сектора. Мы предполагаем, что кэш является двусторонним наборно-ассоциативным, то есть каждый сектор содержит две строки/линии.

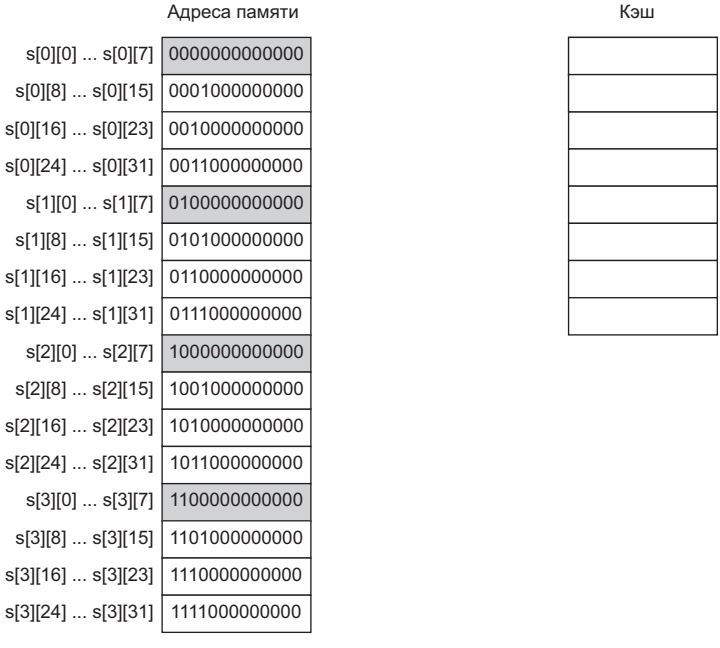


Рис. 12.7. Хранящаяся в памяти матрица и пустой кэш, готовый к работе

Блок памяти может принадлежать только одному разделу, и его размещение определяется его адресом в памяти. Чтобы понять это, мы должны разделить адрес блока памяти на три части:

- *Смещение блока* (block offset) зависит от размера блока. В данном случае этот размер равен 512 байт, а 512 равно 2^9. Следовательно, первые 9 бит адреса представляют собой смещение блока (bo — block offset).
- *Индекс сектора* (set index) указывает на сектор, к которому относится адрес. Так как кэш является двусторонним секторно-ассоциативным и содержит 8 строк/линий, то имеется всего 8/2 = 4 сектора. Кроме того, 4 равно 2^2, поэтому следующие два бита представляют индекс набора (si — set index).
- Остальная часть адреса состоит из битов тега (tb — tag bits). На рис. 12.7 для простоты мы указываем адрес в 13-битном представлении. Для вычисления tb мы используем 13 – bo – si = 2. Это означает, что два оставшихся бита представляют собой биты тега.

Допустим, мы запускаем функцию, которая пытается прочитать элемент `s[0][0]`, который соответствует адресу 0000000000000. Поскольку этого элемента еще нет

в кэше, CPU вычисляет индекс его сектора и копирует его в соответствующий раздел кэша (рис. 12.8).

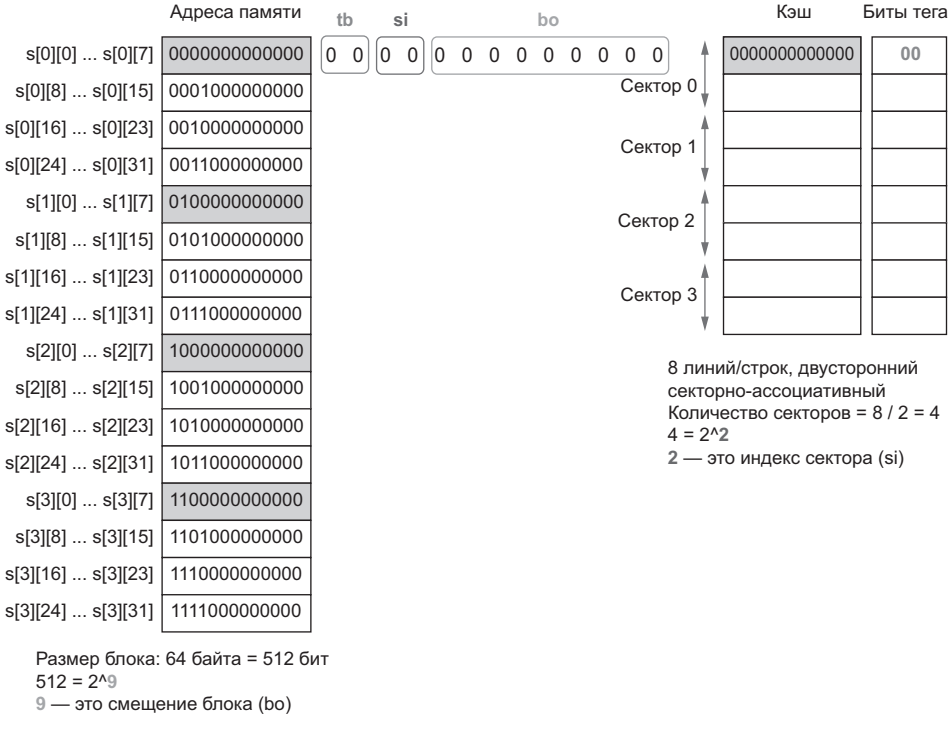


Рис. 12.8. Адрес памяти 0000000000000 копируется в сектор 0

Как уже говорилось, 9 бит отводится на смещение блока: это минимальный общий префикс для каждого адреса блока памяти. Затем 2 бита представляют индекс сектора. С адресом 0000000000000 si равно 00. Следовательно, этот блок памяти копируется в сектор 0.

Когда функция читает элементы от `s[0][1]` до `s[0][7]`, данные уже находятся в кэше. Как CPU узнает об этом? CPU вычисляет начальный адрес блока памяти, вычисляет индекс сектора и биты тега, а затем проверяет, есть ли 00 в секторе 0.

Далее функция считывает элемент `s[0][8]`, но он еще не кэширован. Поэтому происходит аналогичная операция копирования блока памяти 0100000000000 (рис. 12.9).

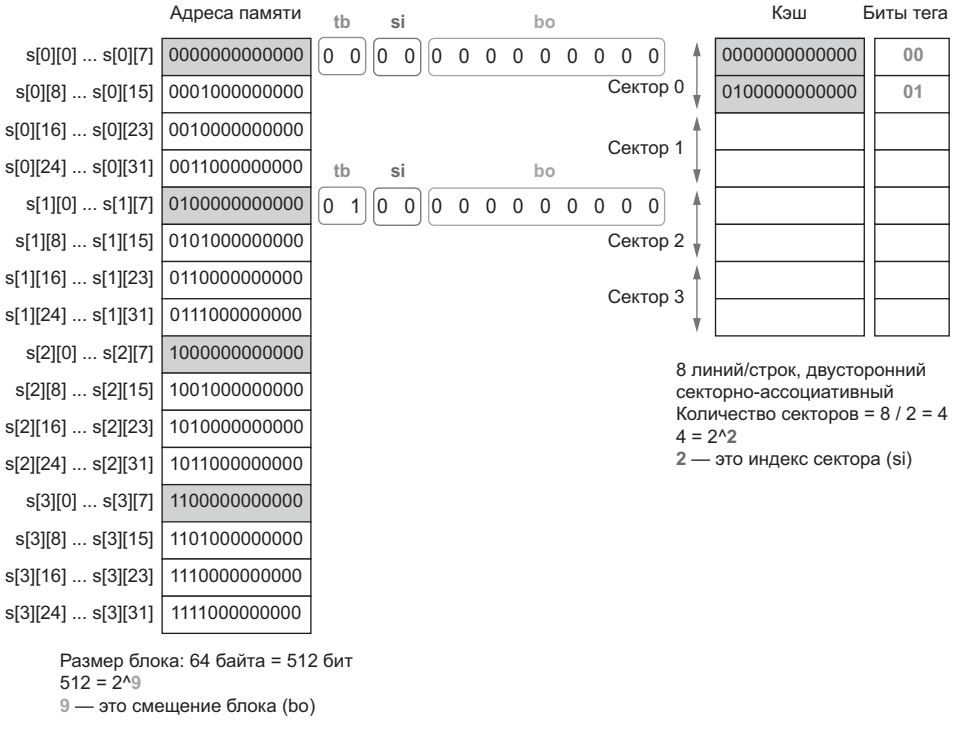


Рис. 12.9. Адрес памяти 0100000000000 копируется в сектор 0

Этот блок памяти имеет индекс сектора, равный 00, поэтому он также принадлежит сектору 0. Кэш-линия копируется в следующую доступную строку в секторе 0. Затем чтение элементов от `s[1][1]` до `s[1][7]` приводит к попаданию в кэш.

Дальше все становится интереснее. Функция читает `s[2][0]`, а этого адреса в кэше нет. Проводится такая же операция (рис. 12.10).

Индекс сектора снова равен 00. Однако сектор 0 уже заполнен — что тогда делать процессору? Скопировать блок памяти в другой сектор? Нет. CPU заменяет одну из существующих кэш-линий, чтобы получить возможность скопировать блок памяти 1000000000000.

Политика вытеснения из кэша зависит от конкретного CPU, но обычно является псевдо-LRU политикой. Настоящая LRU (Least Recently Used — вытеснение давно не используемых) была бы слишком сложной для обработки. Допустим, что в нашем случае заменяется первая кэш-линия: 0000000000000. Эта ситуация повторяется при итерации по строке 3: индекс сектора для адреса памяти

1100000000000 также равен 00, что приводит к вытеснению существующей кэш-линии.

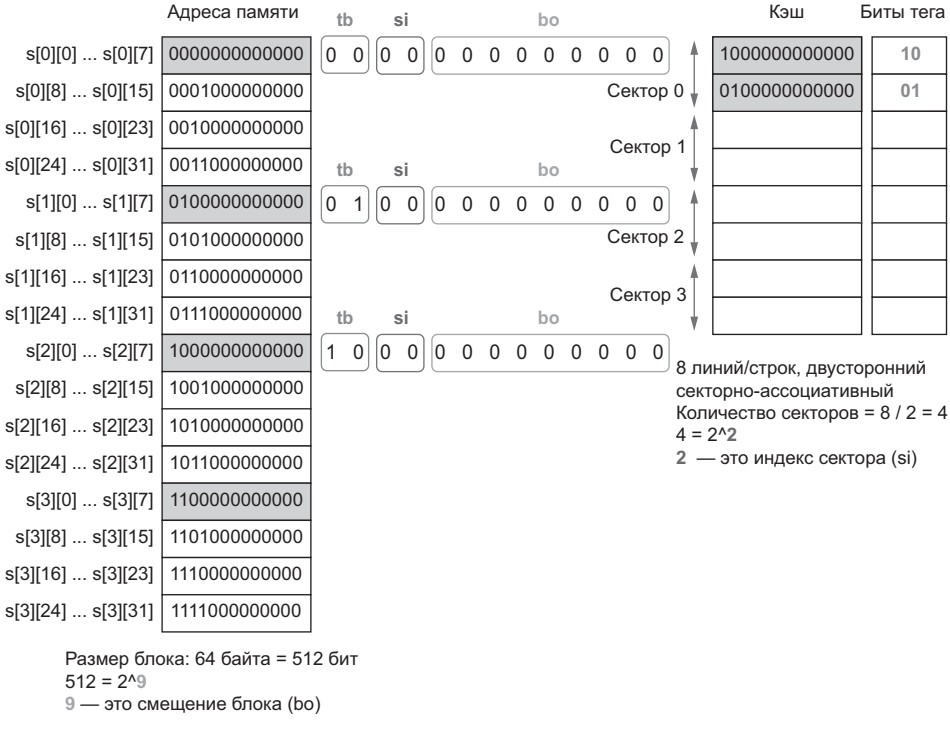


Рис. 12.10. Адрес памяти 1000000000000 заменяет существующую в секторе 0 кэш-линию

Теперь предположим, что тест выполняет функцию со срезом, указывающим на ту же матрицу, начиная с адреса 00000000000000. Когда функция считывает `s[0][0]`, то оказывается, что содержимого ячейки по этому адресу в кэше нет. Этот блок уже вытеснен.

Вместо использования кэшей процессора между выполнениями этот бенчмарк приведет к большему количеству промахов кэша. Этот тип промаха называется *конфликтным промахом* (conflict miss): промах кэша, который не произошел бы, если бы кэш не был разбит на сектора. Все переменные, по которым мы итерируем, принадлежат блоку памяти, индекс сектора которого равен 00. Поэтому мы используем только один сектор кэша вместо распределения по всему кэшу.

Ранее мы обсуждали концепцию шагов (striding), которую описали как способ обращения CPU к данным. В этом примере такой шаг называется *критическим шагом* (critical stride): он приводит к доступу к адресам памяти с тем же индексом набора, которые, таким образом, сохраняются в одном и том же наборе кэша.

Вернемся к нашему примеру из реального мира — с двумя функциями, `calculateSum512` и `calculateSum513`. Бенчмарк выполнялся с восьмиканальным секторно-ассоциативным кэшем L1D объемом 32 Кбайт — всего 64 сектора. Поскольку длина кэш-линии составляет 64 байта, критический шаг равен 64 × 64 байта = 4 Кбайт. Четыре килобайта для типа `int64` соответствуют 512 элементам. Таким образом, мы достигаем критического шага при матрице, состоящей из 512 столбцов, поэтому использование кэша неэффективно. Но если матрица содержит 513 столбцов, то это не приводит к критическому шагу. Вот почему мы наблюдали такую огромную разницу между результатами двух бенчмарков.

Помните, что современные кэши разбиты на сектора. В зависимости от шагов в некоторых случаях может оказаться так, что будет использоваться только один сектор. Это может снизить производительность приложения и привести к конфликтным промахам кэша. Шаг такого типа называется критическим. Для приложений, требующих достижения высокой производительности, нужно избегать критических шагов, чтобы получить от использования возможностей кэша CPU максимальную отдачу.

> **ПРИМЕЧАНИЕ** Наш пример также показывает, почему к результатам микробенчмарка нужно подходить с осторожностью, если он выполнялся в системе, отличной от той, где приложение будет реально работать. Если продакшен-система имеет другую архитектуру кэша, то производительность может быть существенно иной.

Продолжим обсуждать влияние кэша CPU. На этот раз увидим его конкретное влияние при написании конкурентного кода.

12.2. ОШИБКА #92: ПИСАТЬ КОНКУРЕНТНЫЙ КОД, КОТОРЫЙ ПРИВОДИТ К ЛОЖНОМУ СОВМЕСТНОМУ ИСПОЛЬЗОВАНИЮ

До сих пор мы обсуждали фундаментальные концепции кэширования CPU. Мы видели, что некоторые специфические кэши (обычно L1 и L2) не являются

общими для всех логических ядер, а относятся только к физическому ядру. Эту специфику необходимо учитывать при конкурентном программировании, так как она приводит к так называемому ложному совместному использованию, а это может вызвать значительное снижение производительности. Посмотрим, что такое ложное совместное использование на примере, а затем обсудим, как его предотвращать.

В этом примере мы используем две структуры, `Input` и `Result`:

```
type Input struct {
   a int64
   b int64
}

type Result struct {
   sumA int64
   sumB int64
}
```

Цель в том, чтобы реализовать функцию `count`, которая получает срез `Input` и вычисляет следующее:

- сумму всех полей `Input.a`, записывая ее в `Result.sumA`;
- сумму всех полей `Input.b`, записывая ее в `Result.sumB`.

Реализуем конкурентное решение при помощи двух горутин, одна из которых вычисляет `sumA`, а другая `sumB`:

```
func count(inputs []Input) Result {
   wg := sync.WaitGroup{}
   wg.Add(2)
   result := Result{}
   go func() {
      for i := 0; i < len(inputs); i++ {
         result.sumA += inputs[i].a  <-- Вычисление sumA
      }
      wg.Done()
   }()

   go func() {
      for i := 0; i < len(inputs); i++ {
         result.sumB += inputs[i].b  <-- Вычисление sumB
      }
      wg.Done()
   }()
   wg.Wait()
   return result
}
```

Мы запускаем две горутины: одну, которая проводит итерации по всем полям `a`, и другую, которая проводит итерации по всем полям `b`. Этот пример хорош с точки зрения конкурентности. Например, в нем нет гонки данных, потому что в каждой горутине происходит увеличение своих собственных переменных. Но пример иллюстрирует концепцию ложного совместного использования, которая снижает ожидаемую производительность.

Посмотрим, что происходит в основной оперативной памяти (рис. 12.11). Поскольку `sumA` и `sumB` выделяются непрерывно, в большинстве случаев (в семи из восьми возможных) обе переменные размещаются в одном и том же блоке памяти.

Рис. 12.11. В этом примере `sumA` и `sumB` — это части одного и того же блока памяти

Предположим, что процессор содержит два ядра. В большинстве случаев нужно иметь два потока, выполнение которых происходит на разных ядрах. Таким образом, если CPU решит скопировать этот блок памяти в кэш-линию, то копирование будет происходить дважды (рис. 12.12).

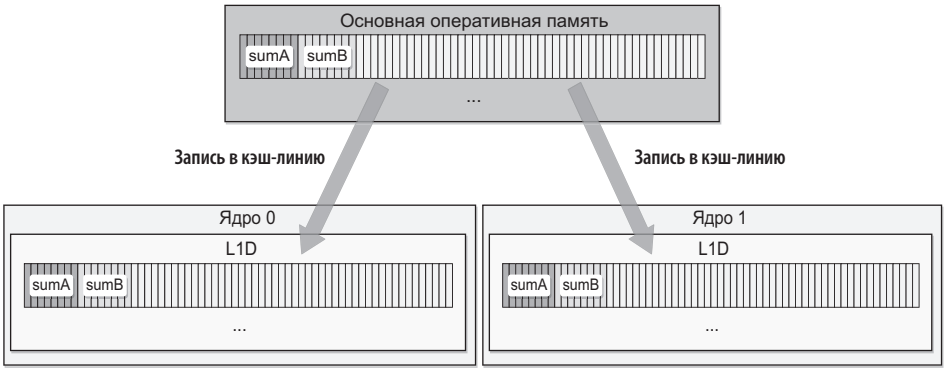


Рис. 12.12. Каждый блок копируется в кэш-линию как на ядре 0, так и на ядре 1

Обе кэш-линии реплицируются, поскольку L1D (данные L1) относятся к каждому ядру. Напомним, что в нашем примере каждая горутина обновляет свою собственную переменную: одна `sumA`, а другая `sumB` (рис. 12.13).

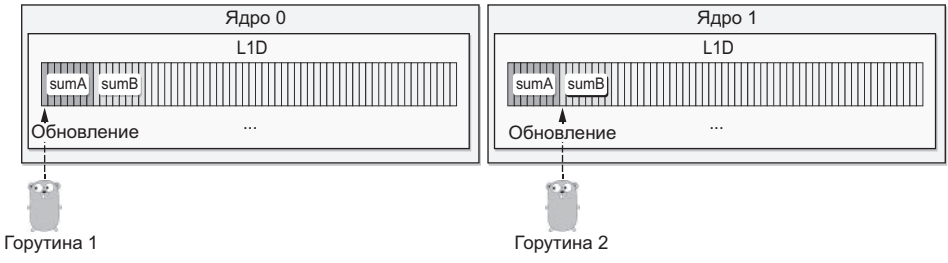


Рис. 12.13. Каждая горутина обновляет свою переменную

Поскольку эти кэш-линии реплицируются, одной из целей CPU является обеспечение когерентности кэша. Например, если одна горутина обновляет `sumA`, а другая читает `sumA` (после проведения какой-то синхронизации), мы ожидаем, что приложение получит самое последнее значение.

Однако в нашем примере такого не происходит. Обе горутины обращаются к своим собственным переменным, а не к общей. Мы могли бы ожидать, что процессор узнает об этом и поймет, что это не конфликт, но это не так. Когда мы записываем переменную, находящуюся в кэше, гранулярность, отслеживаемая CPU, является не переменной, а кэш-линией.

Когда кэш-линия используется несколькими ядрами и хотя бы одна горутина что-то записывает, то содержимое всей кэш-линии становится невалидным. Это происходит, даже если обновления логически независимы (например, `sumA` и `sumB`). Это и есть проблема ложного совместного использования, снижающего производительность.

> **ПРИМЕЧАНИЕ** Внутри CPU используется протокол MESI, гарантирующий когерентность кэша. Он отслеживает каждую кэш-линию, помечая ее как измененную, эксклюзивную, совместно используемую или недействительную (MESI).

Одна из наиболее важных особенностей внутреннего устройства памяти и процессов кэширования, которую важно понять, — совместное использование памяти ядрами является не реальностью, а иллюзией. Осознать это помогает «mechanical sympathy»: мы не рассматриваем процессор как черный ящик, а пытаемся «прочувствовать машину» на базовых уровнях.

Как решить проблему ложного совместного использования? Есть два способа.

Первое решение — использовать тот же подход, который мы только что рассмотрели, но при этом сделать так, чтобы `sumA` и `sumB` не были частью одной и той же кэш-линии. Например, мы можем обновить структуру `Result`, добавив *заполнение* (padding) между полями. Заполнение — это техника выделения дополнительной памяти. Поскольку `int64` требует выделения 8 байт и кэш-линии длиной 64 байта, нужно 64 – 8 = 56 байт заполнения:

```
type Result struct {
   sumA int64
   _ [56]byte  ◄——  Заполнение
   sumB int64
}
```

На рис. 12.14 показано возможное распределение памяти. Используя заполнения, `sumA` и `sumB` всегда будут частями разных блоков памяти и, следовательно, разных кэш-линий.

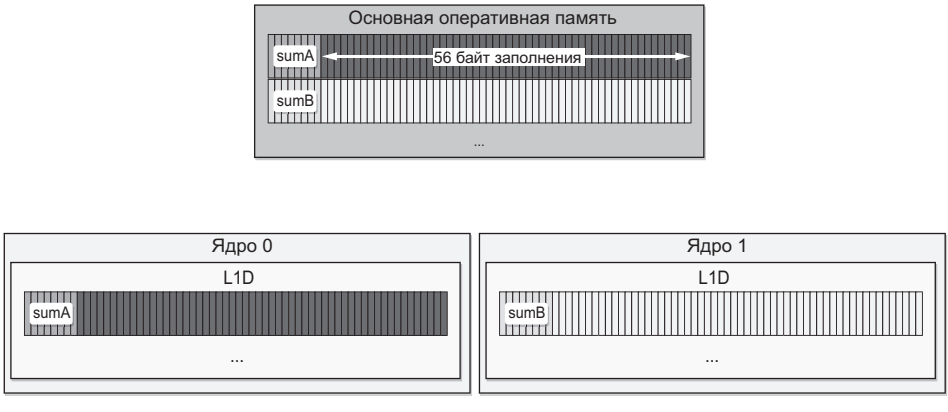


Рис. 12.14. `sumA` и `sumB` являются частями разных блоков памяти

Если мы сравним два решения (с заполнением и без него), то увидим, что решение с заполнением значительно быстрее (около 40 % на моем компьютере). Такое улучшение производительности — это результат добавления заполнения между двумя полями для предотвращения ложного совместного использования.

Другой подход — переработать структуру алгоритма. Например, чтобы не использовать одну и ту же структуру для обеих горутин, можно сделать так, чтобы они коммуницировали свои локальные результаты через каналы. Результат будет примерно таким же, как и при подходе с заполнением.

Помните, что совместное использование памяти разными горутинами на самых низких уровнях памяти иллюзорно. Ложное совместное использование происходит, когда какая-то кэш-линия совместно используется двумя ядрами и при этом хотя бы одна горутина что-то записывает в память. Если нужно оптимизировать конкурентное приложение, проверьте, нет ли в нем ложного совместного использования, поскольку известно, что это снижает производительность приложения. Можно предотвратить его либо с помощью заполнения (padding), либо с помощью коммуницирования.

В следующем разделе поговорим, как процессоры выполняют инструкции параллельно и как использовать эту возможность.

12.3. ОШИБКА #93: НЕ УЧИТЫВАТЬ ПАРАЛЛЕЛИЗМ НА УРОВНЕ ИНСТРУКЦИЙ

Параллелизм на уровне инструкций — это еще один фактор, который может значительно повлиять на общую производительность. Прежде чем дать определение этой концепции, обсудим конкретный пример и способы оптимизации, которые здесь применимы.

Создадим функцию, которая получает массив из двух элементов типа `int64`. Она будет исполняться циклически некоторое (постоянное) число раз. Во время каждой итерации цикла она будет делать следующее:

- увеличивать значение первого элемента массива;
- увеличивать значение второго элемента массива, если первый элемент четный.

Вот как эту функцию можно написать на Go:

```
const n = 1_000_000
func add(s [2]int64) [2]int64 {
   for i := 0; i < n; i++ {   <-- Цикл из n итераций
      s[0]++   <-- Увеличение значения s[0]
      if s[0]%2 == 0 {   <-- Увеличение значения s[1], если s[0] четное
         s[1]++
      }
   }
   return s
}
```

Инструкции, выполняемые в цикле, показаны на рис. 12.15 (увеличение значения переменной требует операций как чтения, так и записи). Инструкции

исполняются последовательно: сначала мы увеличиваем `s[0]`, затем перед увеличением `s[1]` нужно снова прочитать `s[0]`.

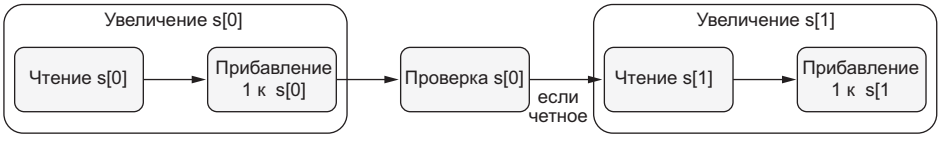


Рис. 12.15. Три основных шага: увеличение значения, проверка, увеличение значения

> **ПРИМЕЧАНИЕ** Эта последовательность инструкций не соответствует гранулярности инструкций ассемблера. Но для ясности в этом разделе мы используем упрощенную схему.

Теперь обсудим теорию, лежащую в основе параллелизма на уровне инструкций (ILP — instruction-level parallelism). Несколько десятилетий назад разработчики CPU в гонке за повышением производительности процессоров перестали фокусироваться исключительно на увеличении их тактовой частоты. Они разработали несколько схем оптимизации, в том числе ILP, которые позволяют распараллеливать выполнение отдельных последовательностей инструкций. Процессор, реализующий ILP на одном виртуальном ядре, называется *суперскалярным процессором*. Например, на рис. 12.16 показано, как процессор выполняет приложение, состоящее из трех инструкций: `I1`, `I2` и `I3`.

Выполнение какой-либо последовательности инструкций требует прохождения различных этапов. CPU должен декодировать инструкции, а потом выполнить их. За последний этап отвечает исполнительный блок, который выполняет операции и вычисления.

На рис. 12.16 CPU решил выполнить три указанные инструкции параллельно. Обратите внимание, что не все они обязательно завершаются за один такт. Например, инструкция, считывающая значение, которое уже есть в регистре, завершится за один процессорный такт, а инструкция, считывающая адрес, который должен быть извлечен из основной памяти, может потратить десятки тактовых циклов.

При последовательном выполнении эти инструкции потребовали бы следующее общее время (функция `t(x)` обозначает время, необходимое CPU для выполнения инструкции `x`):

```
total time = t(I1) + t(I2) + t(I3)
```

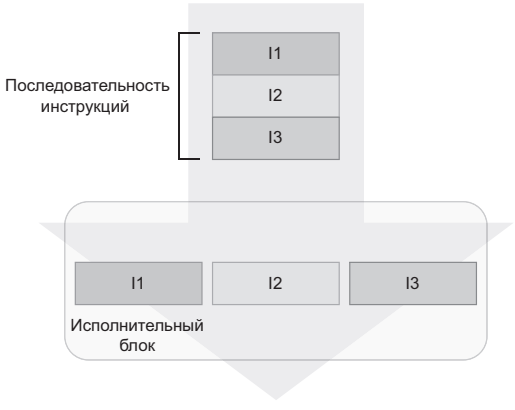


Рис. 12.16. Несмотря на то что эти три инструкции были написаны последовательно, исполняются они параллельно

Но благодаря ILP это общее время будет определяться как:

```
total time = max(t(I1), t(I2), t(I3))
```

В теории ILP выглядит магией, но на практике может привести к сложностям, называемым *конфликтами* (hazards).

Например, что будет, если `I3` устанавливает для переменной значение 42, а `I2` является условной инструкцией (например, `if foo == 1`)? Теоретически этот сценарий должен предотвращать параллельное выполнение `I2` и `I3`. Это называется *конфликтом управления* (control hazard) или *конфликтом ветвления* (branching hazard.). На практике разработчики CPU решали такие проблемы с помощью прогнозирования ветвлений.

Например, CPU может отследить, что условие [`I2`] оказалось истинным в девяносто девяти из последних ста раз, поскольку будет выполнять `I2` и `I3` параллельно. В случае неправильного прогноза (то есть если `I2` оказывается ложным) CPU очищает свой текущий конвейер выполнения, обеспечивая тем самым отсутствие несоответствий и ошибок. Но такой сброс приводит к потерям от 10 до 20 тактов, что сказывается на производительности.

Другие типы конфликтов могут помешать параллельному выполнению инструкций. Как разработчики, мы должны об этом знать. Например, давайте рассмотрим еще две инструкции, которые обновляют регистры (области временного хранения, используемые для выполнения операций):

- `I1` добавляет числа, содержащиеся в регистрах A и B, к C;
- `I2` складывает числа, содержащиеся в регистрах C и D, и записывает результат в D.

Поскольку `I2` зависит от результата `I1` в том плане, что `I2` использует значение регистра C, эти две инструкции не могут выполняться одновременно. `I1` должен завершиться до начала выполнения `I2`. Это называется *конфликтом данных* (data hazard). Чтобы справиться с такими конфликтами, разработчики CPU придумали трюк, называемый *переадресацией* (forwarding), суть которого состоит в обходе записи в регистр. Этот метод не решает проблему, а скорее пытается смягчить последствия.

> **ПРИМЕЧАНИЕ** Есть также *структурные конфликты* (structural hazards), связанные с ситуациями, когда по крайней мере две инструкции в конвейере требуют одного и того же ресурса. Как Go-разработчики, мы не можем как-либо повлиять на такого рода конфликты, поэтому и не обсуждаем их здесь.

Теперь, когда у нас есть достаточное понимание теоретических аспектов ILP, вернемся к исходной проблеме и сосредоточимся на том, что происходит при выполнении цикла:

```
s[0]++
if s[0]%2 == 0 {
   s[1]++
}
```

Как мы уже говорили, ситуация конфликта данных препятствует одновременному выполнению инструкций. Посмотрим на последовательность инструкций на рис. 12.17. Фокусируемся на различных конфликтах между инструкциями.

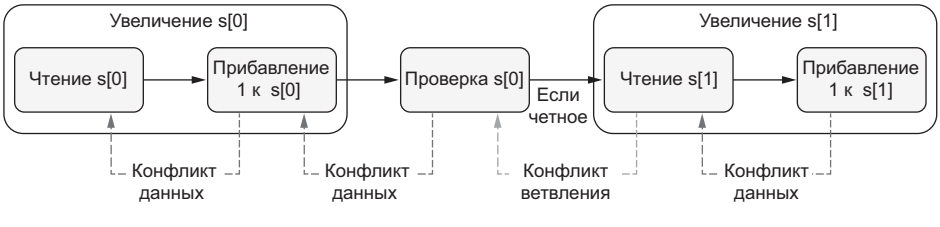


Рис. 12.17. Типы конфликтов между инструкциями

Эта последовательность содержит один конфликт ветвления из-за наличия оператора `if`. Но как уже говорилось, на CPU возлагается задача по оптимизации выполнения и прогнозу того, какую ветвь выбирать. В нашем примере есть несколько конфликтов данных. Как я говорил, они не позволяют

ILP выполнять инструкции параллельно. На рис. 12.18 показана последовательность инструкций с точки зрения ILP: единственные независимые инструкции — это проверка `s[0]` и инкремент `s[1]`, поэтому эти два набора инструкций могут выполняться параллельно благодаря возможности прогнозирования ветвлений.

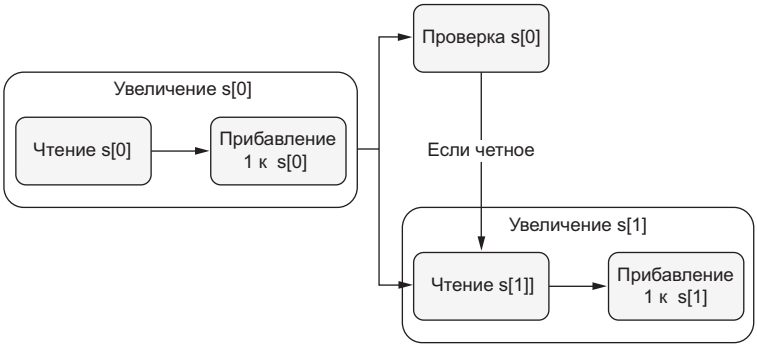


Рис. 12.18. Обе операции по прибавлению единицы выполняются последовательно

А что насчет операций прибавления единицы? Можем ли мы как-либо улучшить код, чтобы свести к минимуму количество конфликтов данных?

Напишем другую версию этой функции (`add2`), которая вводит временную переменную:

```
func add(s [2]int64) [2]int64 {   <-- Первая версия
   for i := 0; i < n; i++ {
      s[0]++
      if s[0]%2 == 0 {
         s[1]++
      }
   }
   return s
}

func add2(s [2]int64) [2]int64 {   <-- Вторая версия
   for i := 0; i < n; i++ {
      v := s[0]   <-- Введение новой переменной для фиксации значения s[0]
      s[0] = v + 1
      if v%2 != 0 {
         s[1]++
      }
   }
   return s
}
```

Во второй версии мы фиксируем значение `s[0]` в новой переменной `v`. Ранее мы увеличивали `s[0]` и проверяли, четное оно или нет. Чтобы воспроизвести это поведение, то — поскольку `v` определяется значением `s[0]` — для увеличения `s[1]` мы теперь проверяем, является ли нечетным `v`.

На рис. 12.19 сравниваются две версии с точки зрения конфликтов. Количество шагов остается таким же. Существенная разница касается конфликтов данных: шаг по прибавлению единицы к `s[0]` и шаг проверки `v` теперь зависят от одной и той же инструкции (чтение `s[0]` и запись в `v`).

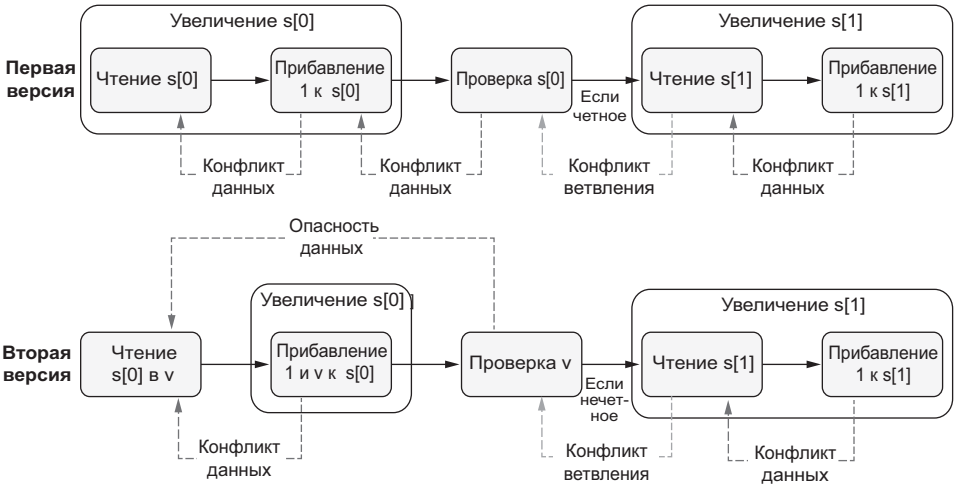


Рис. 12.19. Существенная разница: конфликт данных теперь относится к шагу чтения v

Почему это важно? Потому что позволяет центральному процессору увеличить степень параллельности (рис. 12.20).

Хотя число шагов в обеих версиях одинаково, во второй версии увеличивается количество шагов, которые могут быть выполнены параллельно: есть три параллельных маршрута вместо двух. Время выполнения должно стать более оптимальным, так как самый длинный путь был сокращен. Если мы сравним две функции, то увидим значительное улучшение скорости в случае второй версии (около 20 % на моем компьютере), в основном это произошло из-за использования ILP.

Подведем итог этого раздела. Мы узнали, как современные процессоры используют параллелизм для оптимизации времени выполнения набора инструкций.

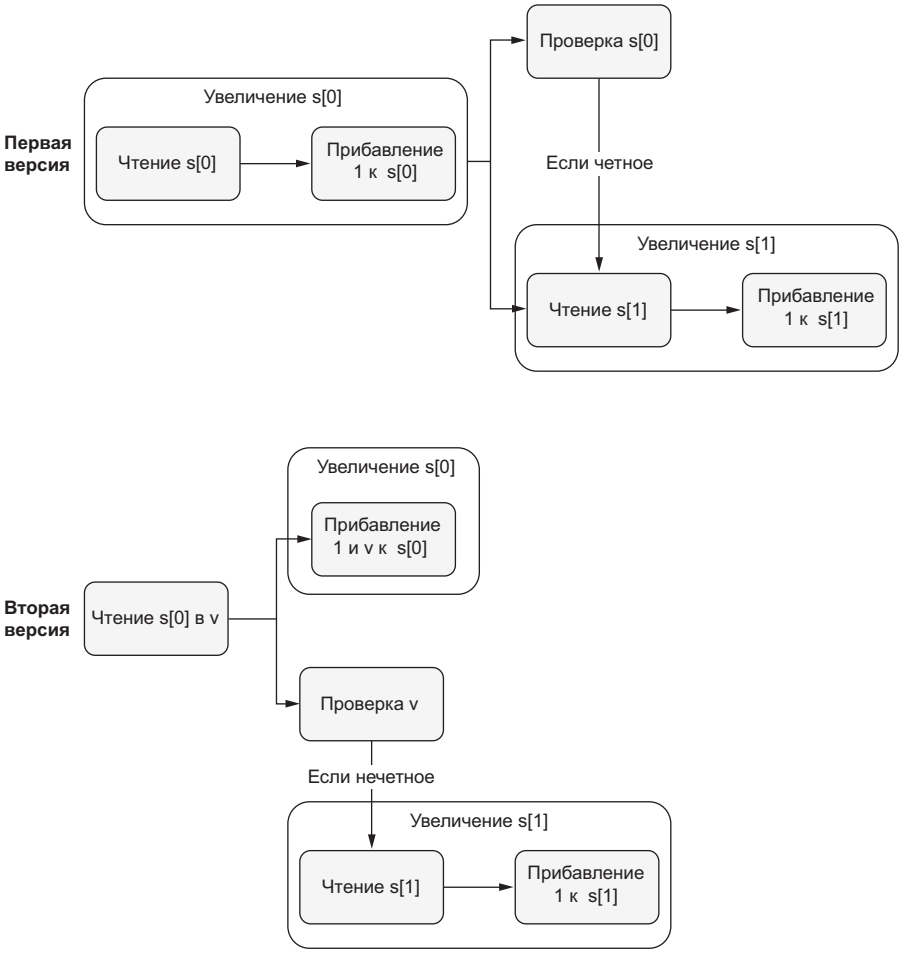


Рис. 12.20. Во второй версии оба шага по прибавлению единицы могут выполняться параллельно

Далее рассмотрели проблему конфликтов данных, которые могут помешать параллельному выполнению инструкций. Затем оптимизировали наш пример, уменьшив количество конфликтов данных, чтобы увеличить количество инструкций, которые можно выполнять параллельно.

Понимание того, как Go компилирует код в ассемблер и как использовать приемы оптимизации производительности CPU (такие, как ILP), — это еще один подход к улучшению работы кода. В рассмотренном примере введение временной переменной привело к значительному улучшению общей производительности.

Этот пример показал, как «mechanical sympathy» поможет оптимизировать Go-приложение.

Не забывайте, что надо с осторожностью подходить к таким микрооптимизациям. Поскольку компилятор Go продолжает развиваться, сгенерированный ассемблерный код приложения также может измениться при обновлении версии Go.

В следующем разделе обсудим эффекты выравнивания данных.

12.4. ОШИБКА #94: НЕ ЗНАТЬ О ВЫРАВНИВАНИИ ДАННЫХ

Выравнивание данных — это способ размещения данных в памяти для ускорения доступа. Незнание этой концепции может привести к лишнему расходу памяти и даже к снижению производительности. В этом разделе обсудим, где это можно применять, а также как предотвратить написание плохо оптимизированного кода.

Чтобы понять, как работает выравнивание данных, обсудим, что было бы без его использования. Допустим, в памяти выделяется место под две переменные: одна типа `int32` (32 байта), другая типа `int64` (64 байта):

```
var i int32
var j int64
```

Без выравнивания данных в системе с 64-битной архитектурой эти две переменные можно было бы разместить в памяти так, как показано на рис. 12.21. Переменная `j` может располагаться в ячейках, относящихся к двум словам. Тогда если CPU будет читать `j`, то ему потребуется два обращения к памяти вместо одного.

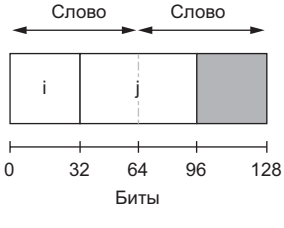


Рис. 12.21. Переменная `j` размещена по ячейкам, относящимся к двум словам

Чтобы этого не произошло, адрес переменной в памяти должен быть кратен ее размеру. Это и есть концепция выравнивания данных. В Go есть следующие варианты выравнивания:

- `byte`, `uint8`, `int8`: 1 байт;
- `uint16`, `int16`: 2 байта;
- `uint32`, `int32`, `float32`: 4 байта;
- `uint64`, `int64`, `float64`, `complex64`: 8 байт;
- `complex128`: 16 байт.

Все эти типы гарантированно выровнены: их адреса в памяти кратны их размеру. Например, адрес любой переменной типа `int32` кратен 4.

Вернемся в реальный мир. На рис. 12.22 показаны два разных случая размещения `i` и `j` в памяти.

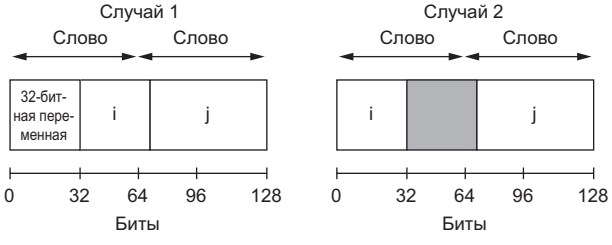


Рис. 12.22. В обоих случаях переменная `j` выровнена по своему размеру

В первом случае некая 32-битная переменная была размещена в памяти непосредственно перед `i`. Поэтому переменные `i` и `j` располагаются в смежных ячейках (непрерывно). Во втором случае перед `i` никакой 32-битной переменной в памяти нет (например, перед `i` была 64-битная переменная), поэтому `i` размещается в начале слова. Чтобы соблюдать принцип выравнивания данных (чтобы адрес был кратен 64), `j` не может располагаться непосредственно рядом с `i`, а только в следующей ближайшей ячейке, адрес которой кратен 64. Серый квадрат отображает 32 бита заполнения (padding).

Посмотрим, когда заполнения могут быть проблемой. Рассмотрим структуру, содержащую три поля:

```
type Foo struct {
   b1 byte
   i int64
   b2 byte
}
```

Есть переменные типа `byte` (1 байт), типа `int64` (8 байт) и еще одна типа `byte` (1 байт). В 64-битной архитектуре эта структура размещается в памяти так, как показано на рис. 12.23. Переменная `b1` размещается в памяти первой. Поскольку `i` типа `int64`, то ее адрес должен быть кратен 8. Следовательно, ее невозможно разместить рядом с `b1` по адресу `0x01`. Какой следующий адрес кратен 8? `0x08`. Для `b2` выделяется следующий доступный адрес, кратный 1: `0x10`.

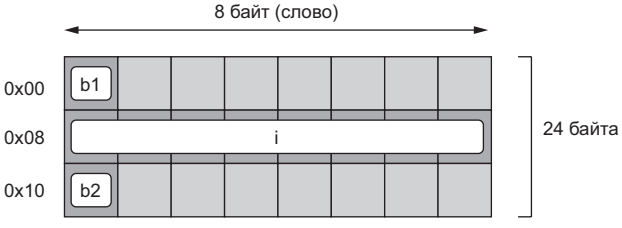


Рис. 12.23. Структура занимает в памяти всего 24 байта

Поскольку размер структуры должен быть кратным размеру слова (8 байт), ее адрес будет занимать не 17 байт, а в общей сложности 24 байта. Во время компиляции Go добавляет заполнение, чтобы обеспечить выравнивание данных:

```
type Foo struct {
   b1 byte
   _ [7]byte  <-- Добавлено компилятором
   i int64
   b2 byte
   _ [7]byte  <-- Добавлено компилятором
}
```

Каждый раз, когда создается структура `Foo`, ей требуется 24 байта в памяти, но из них только 10 байт содержат данные, остальные 14 байт используются для заполнения. Поскольку структура является атомарной единицей, она никогда не будет преобразована, даже после сборки мусора она всегда будет занимать 24 байта в памяти. Обратите внимание, что компилятор не переупорядочивает поля — он только добавляет заполнения, чтобы гарантировать выравнивание данных.

Как уменьшить объем резервируемой памяти? Эмпирическое правило: реорганизовать структуру так, чтобы ее поля сортировались по размеру типов в порядке убывания. В нашем случае первой должна идти переменная типа `int64`, за которой следуют две переменные типа `byte`:

```
type Foo struct {
   i int64
```

```
   b1 byte
   b2 byte
}
```

На рис. 12.24 показано, как размещается в памяти новая версия структуры `Foo`. Переменная `i` размещается первой и занимает целое слово. Основное отличие состоит в том, что теперь `b1` и `b2` могут располагаться рядом друг с другом в одном и том же слове.

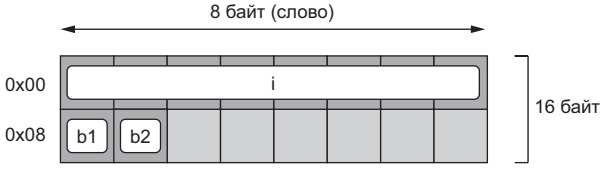


Рис. 12.24. Теперь структура занимает только 16 байт памяти

Размер структуры в памяти по-прежнему должен быть кратен размеру слова, но теперь структура занимает не 24 байта, а всего 16 байт. Мы сэкономили 33 % памяти, просто переместив в коде переменную `i` на первую позицию.

Какие были бы последствия, если бы мы использовали первую версию структуры `Foo` (24 байта) вместо более компактной? Если бы структуры `Foo` сохранялись (например, кэш `Foo` в памяти), приложение требовало бы для себя дополнительный объем памяти. Но даже если бы они и не сохранялись, то возникали бы другие эффекты. Например, если бы мы часто создавали переменные `Foo` и они размещались бы в куче (обсудим эту концепцию в следующем разделе), результатом было бы более частое выполнение сборки мусора, влияющее на общую производительность приложения.

Говоря о производительности, следует упомянуть еще один эффект пространственной локализации. Рассмотрим следующую функцию, `sum`, которая принимает в качестве аргумента срез структур `Foo`. Эта функция итерирует по срезу и суммирует все поля `i` (`int64`):

```
func sum(foos []Foo) int64 {
   var s int64
   for i := 0; i < len(foos); i++ {
      s += foos[i].i  ◄—— Суммирование всех полей i
   }
   return s
}
```

Поскольку за срезом стоит резервный массив, это означает размещение структур `Foo` в смежных ячейках памяти непрерывно.

Рассмотрим резервный массив для двух этих версий `Foo` и проверим две кэш-линии данных (128 байт). На рис. 12.25 каждая серая полоска представляет собой 8 байт данных, а темные полоски показывают, где размещены переменные `i` (то есть поля, которые мы хотим просуммировать).

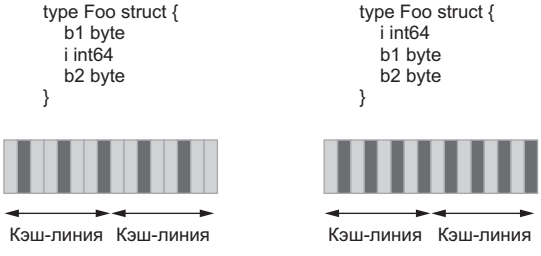


Рис. 12.25. Поскольку каждая кэш-линия содержит в себе большее число переменных `i`, итерация по срезу `Foo` требует меньшего общего количества кэш-линий

Как мы видим, в последней версии `Foo` каждая кэш-линия используется более эффективно, поскольку содержит в среднем на 33 % больше переменных `i`. Следовательно, итерация по срезу `Foo` для суммирования всех элементов типа `int64` более эффективна.

Подтвердим это наблюдение бенчмарком. Если мы запустим два бенчмарка с двумя версиями `Foo`, взяв срез из 10 000 элементов, то версия, использующая последнюю структуру `Foo`, будет примерно на 15 % быстрее (на моей машине). Это 15-процентное увеличение скорости достигнуто всего лишь за счет того, что мы изменили положение одного поля в структуре.

Учитывайте принцип выравнивания данных. Как мы видели в этом разделе, реорганизация полей структуры Go таким образом, что они оказываются отсортированными по размеру в порядке убывания, предотвращает появление заполнений. А это означает, что структуры оказываются более компактными, что может приводить к различным оптимизациям, например к уменьшению частоты обращения к GC и к лучшей пространственной локализации.

В следующем разделе обсудим фундаментальные различия между стеком и кучей, а также то, почему эти различия важны.

12.5. ОШИБКА #95: НЕ ПОНИМАТЬ РАЗЛИЧИЙ МЕЖДУ СТЕКОМ И КУЧЕЙ

В Go переменная может быть размещена в стеке или в куче. Эти два типа памяти принципиально различаются, что может сильно повлиять на работу приложений, интенсивно использующих данные. Рассмотрим концепции и правила, которым следует компилятор, чтобы решить, где размещать переменную.

12.5.1. Стек и куча

Для начала обсудим различия между стеком и кучей. Стек — это память, используемая по умолчанию. Это структура данных, организованная по принципу «последний пришел — первый ушел» (LIFO — Last-In, First-Out), в которой хранятся все локальные переменные для конкретной горутины. Когда горутина запускается, для нее резервируется 2 Кбайт памяти в качестве пространства стека (этот размер со временем менялся и может измениться снова) и в виде смежных друг с другом ячеек. Но этот размер во время выполнения не фиксирован: он может увеличиваться и уменьшаться по мере необходимости (хотя стек всегда остается в памяти в виде непрерывной последовательности ячеек, тем самым соблюдается принцип локальности данных).

Когда в Go определяется какая-то функция, то создается фрейм стека, представляющий собой непрерывную область в памяти, доступ к которой может получить только эта функция. Рассмотрим пример, чтобы понять эту концепцию. В примере функция `main` выводит результат функции `sumValue`:

```
func main() {
   a := 3
   b := 2
   c := sumValue(a, b)  <-- Вызов функции sumValue
   println(c)  <-- Вывод результата
}
//go:noinline  <-- Отключаем встраивание (inlining)
   func sumValue(x, y int) int {
   z := x + y
   return z
}
```

Отмечу две вещи. Во-первых, мы используем встроенную функцию `println` вместо `fmt.Println`, которая принудительно разместила бы переменную `c` в куче. Во-вторых, отключаем встраивание в случае с функцией `sumValue`; в противном

случае вызов функции не произойдет (обсудим особенности встраивания в разделе, посвященном разбору ошибки #97 (не полагаться на встраивание)).

На рис. 12.26 показано состояние стека на шаге резервирования в памяти места под a и b. Поскольку выполнялась функция main, то для нее был создан фрейм стека. Две переменные, a и b, были размещены в стеке внутри фрейма. Всем сохраненным переменным соответствуют реальные (допустимые) адреса, это означает, что на них можно ссылаться и к ним можно получить доступ.

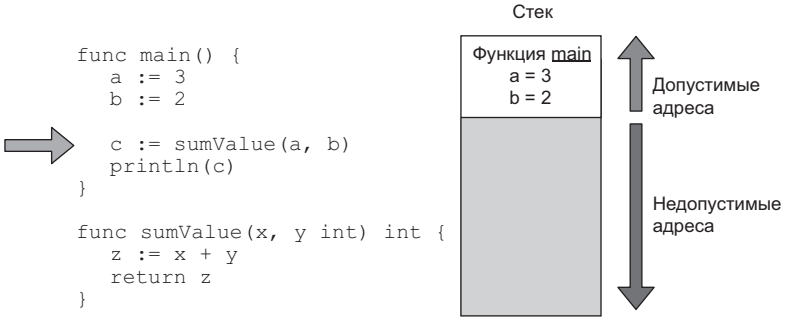


Рис. 12.26. Переменные a и b размещены в стеке

На рис. 12.27 показано, что происходит внутри функции sumValue вплоть до оператора return. Среда выполнения Go создает новый фрейм стека как часть текущего стека горутин. x и y размещаются вместе с z в этом фрейме.

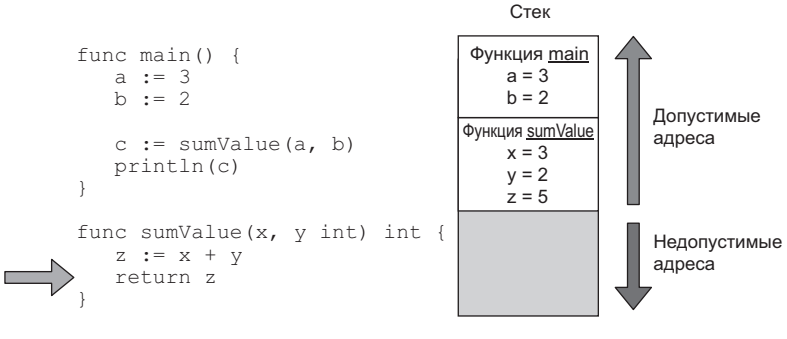


Рис. 12.27. Вызов sumValue приводит к созданию в стеке нового фрейма

В предыдущем фрейме стека (соответствующем функции main) содержатся адреса, которые по-прежнему считаются допустимыми. Мы не можем получить доступ к a и b напрямую; но если у нас будет указатель, например, на a, то он

тоже будет действительным. Мы вскоре обсудим некоторые вопросы, связанные с указателями.

Перейдем к последней инструкции функции `main`, то есть к шагу, на котором выполняется `println`. На этом шаге уже произошел выход из функции `sumValue` — и что же происходит с соответствующим фреймом стека (рис. 12.28)?

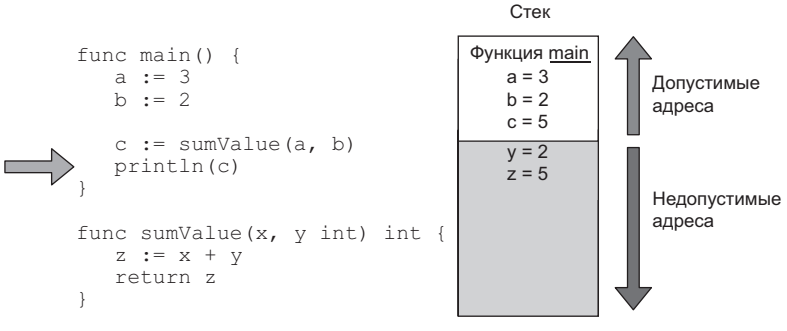


Рис. 12.28. Фрейм стека функции `sumValue` был удален, на его место были записаны переменные, относящиеся к `main`. Здесь переменная x стерта и поверх нее была записана переменная c, в то время как переменные y и z все еще сохраняются в памяти, но теперь стали недоступны

Фрейм стека функции `sumValue` не был полностью стерт из памяти. Когда происходит возврат из функции, Go не тратит время на стирание переменных для освобождения места. Но к этим ставшим уже ненужными переменным больше нельзя получить доступ, а когда в стеке выделяется место под новые переменные для родительской функции, происходит их запись поверх того, что находилось в соответствующих ячейках. В некотором смысле стек очищается сам — он не требует какого-то дополнительного механизма, например сборщика мусора.

Внесем небольшое изменение, чтобы понять ограничения, присущие стеку. Вместо того чтобы возвращать переменную типа `int`, пусть функция вернет указатель:

```
func main() {
   a := 3
   b := 2
   c := sumPtr (a, b)
   println(*c)
}
//go:noinline
   func sumPtr (x, y int) *int {   ◄—— Возврат указателя
   z := x + y
   return &z
}
```

Переменная `c` в функции `main` теперь имеет тип `*int`. Перейдем непосредственно к последней инструкции `println`, следующей за вызовом `sumPtr`. Что произойдет, если место под переменную `z` останется выделенным в стеке (чего быть не должно) (рис. 12.29)?

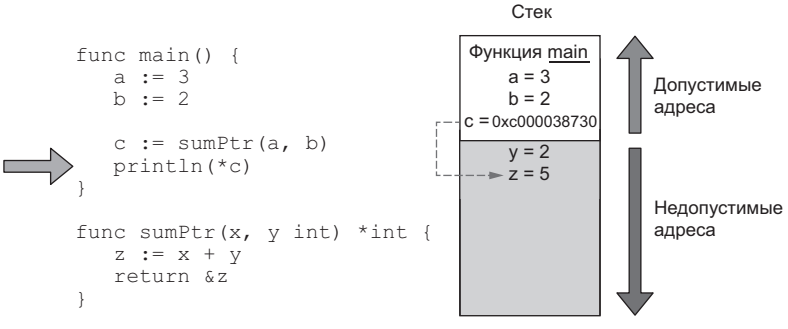


Рис. 12.29. Переменная c указывает на адрес памяти, который более не является допустимым

Если бы переменная `c` ссылалась на адрес переменной `z`, а `z` была размещена в стеке, то возникла бы серьезная проблема. Этот адрес был бы недопустим, фрейм стека `main` продолжал бы расти и стирать переменную `z`. По этой причине организации памяти по принципу стека недостаточно, и нужен другой тип памяти — куча.

Куча — это пул памяти, совместно используемый всеми горутинами. На рис. 12.30 каждая из трех горутин, `G1`, `G2` и `G3`, имеет собственный стек. Но они все пользуются одной и той же кучей.

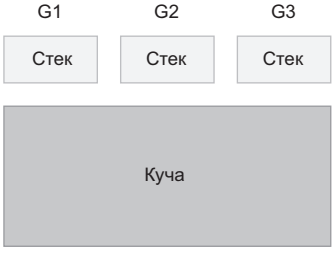


Рис. 12.30. Три горутины имеют собственные стеки, но совместно используют одну общую кучу

В предыдущем примере мы видели, что переменная `z` не могла быть размещена в стеке, поэтому она была помещена в кучу. Если компилятор не может доказать,

что переменная *не* используется после возврата из функции, то место для переменной выделяется в куче.

Почему это важно? Зачем нужно понимать разницу между стеком и кучей? Потому что это влияет на производительность.

Как мы уже говорили, стек очищается сам, и к нему обращается только одна горутина. И наоборот, куча должна очищаться внешней системой — сборщиком мусора. Чем больше происходит резервирований места в куче, тем большее давление оказывается на сборщик мусора. Когда он задействован, то использует 25 % доступной мощности CPU и может вызывать задержку в миллисекунды в фазе «stop the world» (фаза, когда приложение ставится на паузу).

Важно понимать, что выделение места в стеке происходит в среде выполнения Go быстрее, потому что оно весьма тривиально: указатель ссылается на следующий доступный адрес памяти. И наоборот, выделение места в куче требует больше усилий для поиска нужного места и, следовательно, занимает больше времени.

Чтобы проиллюстрировать эти различия, сравним производительность `sumValue` и `sumPtr`:

```
var globalValue int
var globalPtr *int

func BenchmarkSumValue(b *testing.B) {
   b.ReportAllocs()    <-- Сообщается о выделении места в куче
   var local int
   for i := 0; i < b.N; i++ {
      local = sumValue(i, i)  <-- Производится суммирование значений
   }
   globalValue = local
}

func BenchmarkSumPtr(b *testing.B) {
   b.ReportAllocs()  <-- Сообщается о выделении места в куче
   var local *int
   for i := 0; i < b.N; i++ {
      local = sumPtr(i, i)  <-- Производится суммирование с использованием указателей
   }
   globalValue = *local
}
```

Если мы запустим эти бенчмарки (и по-прежнему запретим встраивание), то получим следующие результаты:

```
BenchmarkSumValue-4 992800992 1.261 ns/op 0 B/op 0 allocs/op
BenchmarkSumPtr-4 82829653 14.84 ns/op 8 B/op 1 allocs/op
```

Функция `sumPtr` примерно на порядок медленнее, чем `sumValue`, и это прямое следствие использования кучи вместо стека.

> **ПРИМЕЧАНИЕ** Этот пример показывает, что использование указателей, чтобы избежать копирования, не всегда приводит к более быстрому выполнению; все зависит от контекста. До сих пор в этой книге мы обсуждали значения и указатели только через призму семантики: использование указателя, когда значение [какой-либо сущности] должно быть доступно из разных точек кода. В большинстве случаев этому правилу надо следовать. Также имейте в виду, что современные процессоры чрезвычайно эффективно совершают копирование данных, особенно в пределах одной кэш-линии. Избегайте преждевременной оптимизации и фокусируйтесь на удобочитаемости и семантике.

Следует также отметить, что в предыдущих бенчмарках мы вызывали функцию `b.ReportAllocs()`, которая отображает резервирование места в куче (резервирования в стеке не учитываются):

- `B/op`: сколько байтов резервируется при выполнении операции;
- `allocs/op`: сколько актов резервирования памяти происходит за операцию.

Обсудим теперь условия переноса переменной в кучу (размещения ее в куче).

12.5.2. Эскейп-анализ

Эскейп-анализ (или *escape-анализ*) — это работа компилятора, в процессе которой он решает, где должна быть выделена переменная: в стеке или в куче. Обсудим основные правила.

Когда резервирование места не может быть выполнено в стеке, оно выполняется в куче. Несмотря на то что это звучит как некое упрощенное правило, его важно помнить. Например, если у компилятора нет надежных данных о том, что после возврата из функции ссылки на переменную не будет, то эта переменная размещается в куче. В предыдущем разделе именно так было в случае с функцией `sumPtr`, возвращавшей указатель на переменную, созданную в рамках этой функции. Совместно используемые переменные, определенные в функции, которая вызывается из основного фрагмента кода (sharing up), отправляются в кучу.

Но как быть в ином случае? Что, если мы принимаем указатель, как в примере:

```
func main() {
   a := 3
   b := 2
   c := sum(&a, &b)
   println(c)
}
//go:noinline
func sum(x, y *int) int {   ◄—— Функция принимает в качестве своих аргументов указатели
   return *x + *y
}
```

`sum` принимает два указателя на переменные, созданные в функции main. На рис. 12.31 показано текущее состояние стека на шаге выполнения оператора `return` в функции `sum`.

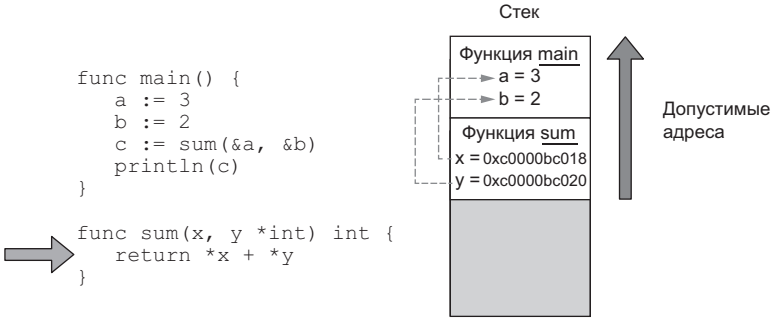


Рис. 12.31. Переменные x и y ссылаются на допустимые адреса

Несмотря на то что они являются частью другого фрейма стека, переменные x и y ссылаются на допустимые адреса. Следовательно, a и b не нужно отправлять в кучу, они могут оставаться в стеке. Как правило, совместно используемые переменные, определенные во фрагменте основного кода (sharing down), остаются в стеке.

Ниже приведены другие случаи, когда переменная может выделяться в куче:

- Глобальная переменная — потому что к ней могут обращаться несколько горутин.
- Указатель, отправляемый в канал:

```
type Foo struct{ s string }
ch := make(chan *Foo, 1)
foo := &Foo{s: "x"}
ch <- foo
```

Здесь `foo` отправляется в кучу.

- Переменная, на которую ссылается какое-то значение, отправляемая в канал:

```
type Foo struct{ s *string }
ch := make(chan Foo, 1)
s := "x"
bar := Foo{s: &s}
ch <- bar
```

Поскольку `Foo` ссылается на переменную `s` через ее адрес, в таких ситуациях она отправляется в кучу.

- Если локальная переменная слишком велика для размещения в стеке.
- Если размер локальной переменной неизвестен. Например, `s := make([]int, 10)` может не размещаться в куче, а вот `s := make([]int, n)` может размещаться, поскольку ее размер зависит от значения переменной.
- Если резервный массив среза переопределяется с помощью `append`.

Хотя этот список и дает некоторое понимание того, как компилятор принимает решения, он не исчерпывающий и может измениться в будущих версиях Go. Чтобы проверить какое-либо предположение по решению компилятора, используйте флаг `-gcflags`:

```
$ go build -gcflags "-m=2"
...
./main.go:12:2: z escapes to heap:
```

Здесь компилятор сообщает, что переменная `z` будет отправлена в кучу.

Понимание фундаментальных различий между кучей и стеком очень важно для оптимизации Go-приложений. Как мы видели, резервирование места в куче сложнее для среды выполнения Go. Кроме того, при этом для очистки данных нужна внешняя система со сборщиком мусора. На управление кучей в некоторых приложениях, интенсивно использующих данные, может тратиться до 20–30 % общего времени CPU. С другой стороны, стек очищается сам и он локальный для одной горутины, что ускоряет процессы выделения места в памяти. Поэтому затраты на оптимизацию этих процессов могут принести большую отдачу.

Правила эскейп-анализа важны и для написания более эффективного кода. «Sharing down» остается в стеке, тогда как «sharing up» размещается в куче. Это предотвращает некоторые ошибки, например преждевременную оптимизацию, когда мы хотим возвращать указатели, «чтобы избежать операций копирования».

Сначала сфокусируйтесь на удобочитаемости и семантике, а затем на оптимизации, если это нужно.

В следующем разделе поговорим, как уменьшить выделение памяти.

12.6. ОШИБКА #96: НЕ ЗНАТЬ, КАК СОКРАТИТЬ ЧИСЛО ВЫДЕЛЕНИЙ ПАМЯТИ

Сокращение выделений памяти (allocations) — распространенный метод оптимизации для ускорения приложений Go. В этой книге я уже рассмотрел несколько подходов, сокращающих число выделений памяти в куче:

- Неоптимизированное объединение строк (см. ошибку #39): использование `strings.Builder` вместо оператора `+`.
- Бесполезные преобразования строк (см. ошибку #40): по возможности избегайте преобразования `[]byte` в строки.
- Неэффективная инициализация срезов и карт (см. ошибки #21 и #27): предварительное резервирование места под срезы и карты, если их длина уже известна.
- Выравнивание структур данных для уменьшения их размера (см. ошибку #94).

Обсудим три распространенных подхода к сокращению числа выделений:

- изменять API;
- рассчитывать на оптимизации компилятора;
- использовать такие инструменты, как `sync.Pool`.

12.6.1. Изменения API

Первый вариант — тщательно поработать с API, который мы предоставляем. Возьмем в качестве примера интерфейс `io.Reader`:

```
type Reader interface {
   Read(p []byte) (n int, err error)
}
```

Метод `Read` принимает срез и возвращает количество прочитанных байтов. Теперь представьте, если бы интерфейс `io.Reader` делал обратное: передавал

значение `int`, которое бы задавало, сколько байтов нужно прочитать, и возвращал срез:

```
type Reader interface {
   Read(n int) (p []byte, err error)
}
```

С точки зрения семантики тут нет ничего плохого. Но тогда возвращенный срез автоматически будет размещен в куче. Это случай sharing up, описанный в предыдущем разделе.

Чтобы предотвратить автоматическое размещение среза в кучу, разработчики среды Go использовали подход sharing down. Таким образом, вызывающая сторона должна предоставлять срез. Это не обязательно означает, что он не будет размещен в куче: компилятор может решить, что этот срез не может оставаться в стеке. Но обрабатывать его должна вызывающая сторона, а не ограничение, обусловленное вызовом метода `Read`.

Иногда даже небольшое изменение в API может положительно повлиять на процесс резервирования места в памяти. При разработке API помните об описанных в предыдущем разделе правилах эскейп-анализа и, при необходимости, используйте `-gcflags` для понимания решений, принятых компилятором.

12.6.2. Приемы оптимизации компилятора

Одна из целей компилятора Go — по возможности оптимизировать код. Вот пример с картами.

В Go мы не можем определить карту, используя срез в качестве типа ключа. В некоторых случаях, особенно в приложениях, выполняющих ввод/вывод, можно получать данные типа `[]byte`, которые бы хотелось взять в качестве ключа. Но мы должны сначала преобразовать их в строку, поэтому напишем такой код:

```
type cache struct {
   m map[string]int  <-- Содержит карту строк
}
func (c *cache) get(bytes []byte) (v int, contains bool) {
   key := string(bytes)  <-- Преобразование из типа []byte в тип string
   v, contains = c.m[key]  <-- Запрос к карте по строковому значению
   return
}
```

Поскольку функция `get` получает срез `[]byte`, мы преобразуем его в строку `key` для запроса к карте.

Но компилятор Go проделывает определенную оптимизацию в случае запроса к карте с использованием `string(bytes)`:

```
func (c *cache) get(bytes []byte) (v int, contains bool) {
   v, contains = c.m[string(bytes)]  ◄—— Прямой запрос к карте с помощью string(bytes)
   return
}
```

Несмотря на то что код почти такой же (мы вызываем `string(bytes)` напрямую вместо вызова переменной), компилятор при этом будет избегать преобразования байтов в строку. Следовательно, вторая версия будет быстрее первой.

Этот пример показывает, что две версии функции, которые выглядят одинаково, могут привести к различным кодам ассемблера, получающимся в результате работы компилятора Go. Для оптимизации приложения следует знать о возможных приемах оптимизации, заложенных в компилятор. Важно следить за будущими релизами Go и проверять, добавляются ли в него новые особенности, направленные на оптимизацию.

12.6.3. sync.Pool

Еще один способ повысить производительность приложений, сократив выделения памяти, — использовать `sync.Pool`. Важно понимать, что `sync.Pool` — это не кэш: для него нет фиксированного размера или максимальной емкости, которые можно устанавливать. Это пул общих объектов, чтобы можно было их переиспользовать.

Допустим, мы хотим реализовать функцию `write`, которая получает `io.Writer`, вызывает какую-то функцию для получения среза `[]byte`, а затем записывает его в `io.Writer`. Код может выглядеть так (для наглядности опускаем обработку ошибок):

```
func write(w io.Writer) {
   b := getResponse()  ◄—— Получение ответа типа []byte
   _, _ = w.Write(b)   ◄—— Запись в io.Writer
}
```

Здесь `getResponse` при каждом вызове возвращает новый срез `[]byte`. Что, если мы хотим уменьшить число выделений памяти, переиспользовав этот срез? Предположим, что все ответы имеют максимальный размер 1024 байта. В этой ситуации можно применить `sync.Pool`.

Для создания `sync.Pool` требуется фабричная функция `func()any` (рис. 12.32). `sync.Pool` предоставляет два метода:

- `Get()any` — для получения объекта из пула;
- `Put(any)` — для возвращения объекта в пул.

```
func factory() any {
    return 
}
```

Рис. 12.32. Определение функции `factory`, которая создает новый объект при каждом новом обращении к себе

При выполнении `Get` либо создается новый объект, если пул пуст, либо объект используется повторно. После использования объекта его можно поместить обратно в пул с помощью `Put`. На рис. 12.33 показан пример с ранее определенной `factory` вместе с `Get`, когда пул пуст, а также с `Put` и `Get`, когда пул не пуст.

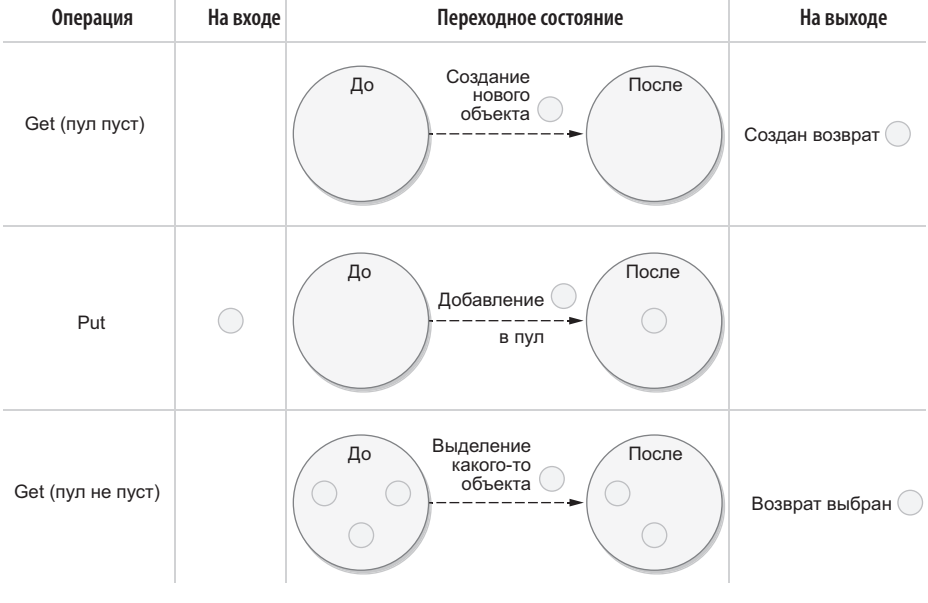


Рис. 12.33. `Get` либо создает новый объект, либо возвращает выбранный из пула. `Put` помещает объект в пул

Когда объекты удаляются из пула? Для этой операции не предусмотрено никакого специального метода — она передается сборщику мусора. После каждого обращения к нему объекты из пула уничтожаются.

Допустим, что мы обновим функцию `getResponse` для записи данных в заданный срез вместо создания их заново. Тогда реализуем другую версию метода `write`, которая использует пул:

```
var pool = sync.Pool{
   New: func() any {  <-- Создание пула и задание фабричной функции
      return make([]byte, 1024)
   },
}

func write(w io.Writer) {
   buffer := pool.Get().([]byte)  <-- Получение []byte из пула или создание его
   buffer = buffer[:0]  <-- Сброс буфера
   defer pool.Put(buffer)  <-- Занесение буфера обратно в пул
   getResponse(buffer)  <-- Запись ответа в заданный буфер
   _, _ = w.Write(buffer)
}
```

Мы определяем новый пул с помощью структуры `sync.Pool` и задаем фабричную функцию для создания нового `[]byte` длиной в 1024 элемента. В функции `write` делается попытка получить один буфер из пула. Если пул пуст, функция создает новый буфер. В противном случае она выбирает произвольный буфер из пула и возвращает его. Одним из важных шагов является сброс буфера с помощью `buffer[:0]`, так как этот срез, возможно, уже использовался. Затем мы откладываем вызов `Put`, чтобы поместить срез обратно в пул.

В этой новой версии вызов `write` не приводит к созданию нового среза `[]byte` для каждого вызова. Вместо этого можно переиспользовать существующие выделенные срезы. В худшем случае, например после сборки мусора, функция создаст новый буфер. При этом амортизированная стоимость выделения снижается.

Если приходится часто выделять место в памяти под большое число объектов одного типа, можно рассмотреть применение `sync.Pool`. Это набор временных объектов, которые помогут предотвратить многократное пересоздание данных одного и того же типа. Кроме того, `sync.Pool` безопасен для одновременного использования несколькими горутинами.

Далее обсудим концепцию встраивания.

12.7. ОШИБКА #97: НЕ ПОЛАГАТЬСЯ НА ВСТРАИВАНИЕ

Термин *встраивание* (inlining) означает замену вызова функции ее телом. Встраивание выполняется компиляторами автоматически. Встраивание также может быть способом оптимизации определенных путей кода приложения.

Рассмотрим пример встраивания с помощью простой функции `sum`, которая суммирует два значения типа `int`:

```
func main() {
   a := 3
   b := 2
   s := sum(a, b)
   println(s)
}

func sum(a int, b int) int {  <-- Встраивание функции
   return a + b
}
```

Если мы запустим `go build` с параметром `-gcflags`, то узнаем решение, принятое компилятором относительно обработки функции `sum`:

```
go build -gcflags "-m=2"
./main.go:10:6: can inline sum with cost 4 as:
   func(int, int) int { return a + b }
...
./main.go:6:10: inlining call to sum func(int, int) int { return a + b }
```

Компилятор решил встроить вызов `sum`, и предыдущий код был заменен следующим:

```
func main() {
   a := 3
   b := 2
   s := a + b  <-- Замена вызова функции sum ее телом
   println(s)
}
```

Встраивание работает только для функций с определенным уровнем сложности, также известным как *бюджет встраивания* (inlining budget). Иначе компилятор сообщит, что функция слишком сложна для того, чтобы оказаться встроенной:

```
./main.go:10:6: cannot inline foo: function too complex:
cost 84 exceeds budget 80
```

Встраивание имеет два преимущества. Во-первых, оно удаляет оверхеды на вызов функции (хотя эти оверхеды были уменьшены, начиная с версии Go 1.17, а также с помощью соглашений о вызовах на основе регистров). Во-вторых, это позволяет компилятору перейти к дальнейшим оптимизациям. Например, после встраивания функции компилятор может решить, что переменная, которую он изначально должен был разместить в куче, может остаться в стеке.

Возникает вопрос: если эта оптимизация применяется компилятором автоматически, почему нам вообще об этом нужно заботиться? Ответ кроется в концепции встраивания в середине стека (mid-stack inlining).

Встраивание в середине стека — это встраивание функций, которые вызывают другие функции. До версии Go 1.9 для встраивания рассматривались только листовые функции. Теперь, благодаря встраиванию в середине стека, можно встроить и следующую функцию `foo`:

```
func main() {
   foo()
}

func foo() {
   x := 1
   bar(x)
}
```

Поскольку функция `foo` не слишком сложная, компилятор может ее встроить:

```
func main() {
   x := 1   ◄——  Замена на тело функции foo
   bar(x)
}
```

Благодаря встраиванию в середине стека Go-разработчики могут оптимизировать приложение, используя концепцию встраивания быстрого пути (fast-path inlining), чтобы различать, какие пути быстрые, а какие медленные. Рассмотрим пример реализации `sync.Mutex`, чтобы понять, как это работает.

До появления встраивания в середине стека реализация метода `Lock` была следующей:

```
func (m *Mutex) Lock() {
   if atomic.CompareAndSwapInt32(&m.state, 0, mutexLocked) {
      // Мьютекс не заблокирован
      if race.Enabled {
         race.Acquire(unsafe.Pointer(m))
      }
      return
```

```
    }
    // Мьютекс уже заблокирован
    var waitStartTime int64
    starving := false
    awoke := false
    iter := 0
    old := m.state
    for {
        // ...  ◄── Какая-то сложная логика
    }
    if race.Enabled {
        race.Acquire(unsafe.Pointer(m))
    }
}
```

Мы можем выделить два основных пути:

- если мьютекс не заблокирован (`atomic.CompareAndSwapInt32` возвращает `true`), это быстрый путь;
- если мьютекс уже заблокирован (`atomic.CompareAndSwapInt32` возвращает `false`), это медленный путь.

Но независимо от выбранного пути, функция не может быть встроена из-за ее сложной структуры. Для встраивания в середине стека нужно сделать такой рефакторинг метода `Lock`, чтобы медленный путь находился внутри конкретной функции:

```
func (m *Mutex) Lock() {
    if atomic.CompareAndSwapInt32(&m.state, 0, mutexLocked) {
        if race.Enabled {
            race.Acquire(unsafe.Pointer(m))
        }
        return
    }
    m.lockSlow()  ◄── Путь, на котором мьютекс уже заблокирован
}

func (m *Mutex) lockSlow() {
    var waitStartTime int64
    starving := false
    awoke := false
    iter := 0
    old := m.state
    for {
        // ...
    }
    if race.Enabled {
    race.Acquire(unsafe.Pointer(m))
    }
}
```

Благодаря этому изменению метод `Lock` можно встроить. Преимущество заключается в том, что мьютекс, который еще не заблокирован, теперь блокируется без затрат времени на вызов функции (скорость повышается примерно на 5 %). Медленный путь, когда мьютекс уже заблокирован, не изменился. Ранее для выполнения этой логики требовался один вызов функции. Теперь же необходимость только в одном вызове функции остается, но на этот раз для `lockSlow`.

Эта техника оптимизации заключается в различии между быстрыми и медленными путями выполнения. Если быстрый путь может быть встроен, а медленный путь нет, можно выделить медленный путь в отдельную функцию. Если бюджет встраивания не превышен, функция является кандидатом для встраивания.

Встраивание — это не просто некая невидимая оптимизация кода, которую выполняет компилятор и о которой не нужно думать. Если мы понимаем, как работает встраивание и как получить доступ к решению компилятора на этот счет, то сможем оптимизировать код методом встраивания быстрого пути. Размещение медленного пути внутри отдельной функции предотвращает вызов функции, если выполнение идет по быстрому пути.

В следующем разделе обсудим общие инструменты диагностики, которые помогут понять, что нужно оптимизировать в Go-приложениях.

12.8. ОШИБКА #98: НЕ ИСПОЛЬЗОВАТЬ ДИАГНОСТИЧЕСКИЙ ИНСТРУМЕНТАРИЙ GO

В Go есть несколько отличных инструментов диагностики, которые помогут понять, как работает приложение. В этом разделе основное внимание уделяется двум самым важным: профилированию и трассировщику. Они должны входить в набор инструментов любого Go-разработчика, интересующегося оптимизацией. Сначала обсудим профилирование.

12.8.1. Профилирование

Профилирование дает возможность проанализировать, как выполняется приложение. А это, в свою очередь, позволяет решать проблемы с производительностью, обнаруживать конфликты, утечки памяти и многое другое. Информация может быть собрана с помощью нескольких профилей:

- `CPU` — определяет, на выполнение чего приложение затрачивает время.
- `Goroutine` — выдает информацию о трассировке стека текущих горутин.
- `Heap` — выдает информацию о выделении памяти в куче, чтобы отследить текущее использование памяти и проверить, нет ли возможных утечек памяти.
- `Mutex` — выдает информацию о конфликтах между блокировками, чтобы отслеживать поведение используемых в коде мьютексов и узнавать, не тратит ли приложение слишком много времени на блокировку вызовов.
- `Block` — показывает места, где горутины блокируются, ожидая синхронизационных примитивов.

Профилирование осуществляется через инструментацию с помощью инструмента, который называется профилировщиком: в Go это `pprof`. Разберемся, как и когда имеет смысл запускать `pprof`, и обсудим наиболее важные типы профилей.

Запуск pprof

Есть несколько способов запустить `pprof`. Например, использовать пакет `net/http/pprof` для передачи данных профилирования через HTTP:

```
package main
import (
   "fmt"
   "log"
   "net/http"
   _ „net/http/pprof"   ◄—— Пустой импорт в pprof
)

func main() {
   http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {   ◄——
                                                                          Предоставление
      fmt.Fprintf(w, "")                                                  конечной точки HTTP
   })
   log.Fatal(http.ListenAndServe(":80", nil))
}
```

Импорт `net/http/pprof` приводит к побочному эффекту, который позволяет получить URL-адрес `pprof` — http://host/debug/pprof. Обратите внимание, что запуск `pprof` безопасен даже в продакшене (https://go.dev/doc/diagnostics#profiling). Профили, влияющие на производительность, например профилирование CPU, по умолчанию не запущены и не работают постоянно: они активируются только на определенное время.

Теперь, когда мы увидели, как предоставить конечную точку `pprof`, обсудим наиболее распространенные профили.

Профилирование CPU

Работа профилировщика CPU основана на получении сигналов операционной системы. Когда он активирован, приложение указывает ОС на то, что его надо прерывать каждые 10 миллисекунд (это интервал по умолчанию) с помощью сигнала `SIGPROF`. Когда приложение получает `SIGPROF`, оно приостанавливает текущее выполнение и передает управление профилировщику. Тот собирает различные данные: информацию о текущем выполнении горутины и статистику выполнения, к которой мы можем получить доступ. Затем он останавливается, и выполнение возобновляется до следующего `SIGPROF`.

Мы можем получить доступ к конечной точке /debug/pprof/profile, чтобы активировать профилирование CPU. При этом по умолчанию выполняется профилирование CPU в течение 30 секунд: в это время приложение прерывается каждые 10 миллисекунд. Обратите внимание, что можно изменить эти два значения по умолчанию с помощью параметра `seconds`, передав конечной точке, как долго должно продолжаться профилирование (например, /debug/pprof/profile?seconds=15), и изменив частоту прерываний (сделать ее даже менее 10 миллисекунд). Но в большинстве случаев 10 миллисекунд должно быть достаточно, и к уменьшению этого значения (что означает увеличение частоты прерываний) следует подходить с осторожностью, чтобы не повлиять негативно на производительность. Через 30 секунд мы загружаем данные из профилировщика CPU.

Профилирование CPU во время бенчмаркинга

Можно включить профилировщик CPU, используя флаг `-cpuprofile`, например, при запуске бенчмарка:

```
$ go test -bench=. -cpuprofile profile.out
```

Эта команда создает файл того же типа, который можно загрузить через /debug/pprof/profile.

От этого файла можно перейти к результатам с помощью `go tool`:

```
$ go tool pprof -http=:8080 <file>
```

Эта команда открывает веб-интерфейс, показывающий график вызовов. На рис. 12.34 приведен пример, взятый из некоторого приложения. Жирные стрелки показывают «горячий путь» кода (hot path). Затем мы можем перейти к этому графику и получить данные о выполнении.

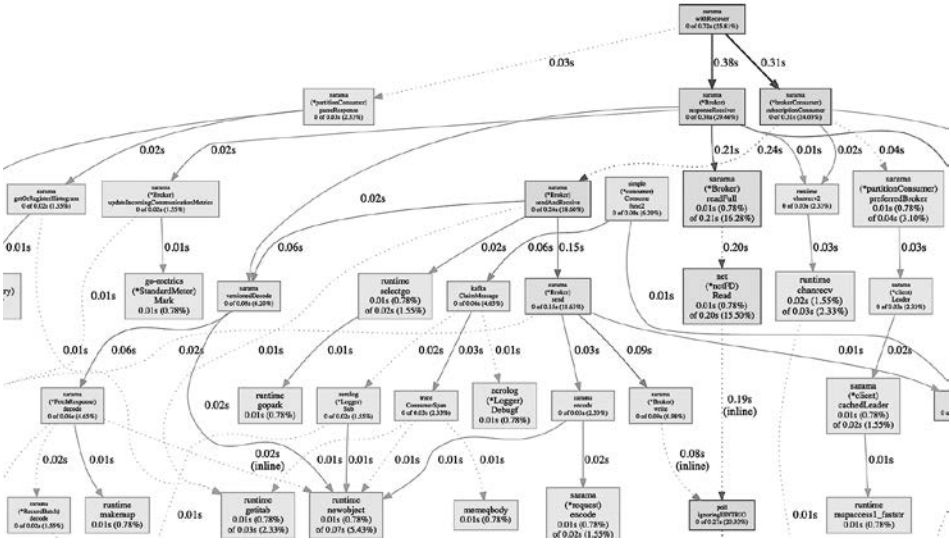


Рис. 12.34. График вызовов приложения за 30 секунд

Например, график на рис. 12.35 говорит о том, что за 30 секунд на метод `decode` (приемник `*FetchResponse`) было потрачено 0.06 секунды. Из них 0.02 ушло на `Record-Batch.decode` и 0.01 на `makemap` (создание карты).

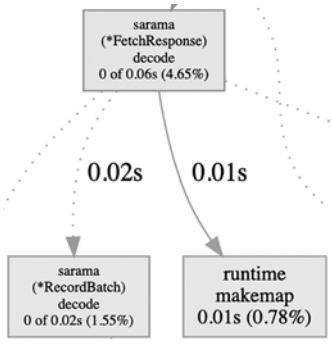


Рис. 12.35. Пример графика вызовов

При помощи веб-интерфейса можно получить доступ к различным представлениям такой информации. Например, в представлении Top view функции сортируются по времени их выполнения, а во Flame Graph визуализируется иерархия времени выполнения. Пользовательский интерфейс может даже отображать самые дорогостоящие части исходного кода построчно.

ПРИМЕЧАНИЕ Можно получить данные профилирования с помощью командной строки. Но в этом разделе сосредоточимся на веб-интерфейсе.

Благодаря наличию этих данных мы получаем общее представление о том, как ведет себя приложение:

- Слишком много вызовов `runtime.mallogc` может означать чрезмерное количество небольших выделений памяти в куче, которые можно попытаться свести к минимуму.
- Слишком большое количество времени, потраченное на операции с каналами или блокировку мьютексов, может указывать на чрезмерный уровень конкурентности, что негативно отражается на производительности приложения.
- Слишком большое количество времени, потраченное на `syscall.Read` или `syscall.Write`, означает, что приложение проводит значительное количество времени в режиме ядра (kernel mode). Тогда при должной работе над буферизацией ввода/вывода можно добиться некоторых улучшений.

Именно такие сведения позволяет получить профилировщик процессора. Он ценен для понимания горячих путей кода и выявления узких мест. Но он не может предоставить более детальную информацию, чем заданная частота, поскольку выполняется с фиксированным интервалом (по умолчанию 10 миллисекунд). Чтобы получить более детальную информацию, используйте трассировку, которую мы обсудим дальше.

ПРИМЕЧАНИЕ Можно прикреплять к различным функциям ярлыки. Например, представьте себе общую функцию, вызываемую из разных клиентов. Чтобы отслеживать время, потраченное на всех клиентов, используйте `pprof.Labels`.

Профилирование кучи

Профилирование кучи позволяет получить статистику о ее текущем использовании. Как и профилирование CPU, оно основано на принципе сэмплирования. Мы можем менять частоту сэмплирования, но не должны стремиться к слишком большой гранулярности, так как чем сильнее уменьшаем шаг, тем больше усилий требуется для сбора данных. По умолчанию сэмплы профилируются при одном распределении на каждые 512 Кбайт кучи.

Если мы получим /debug/pprof/heap/, то увидим необработанные данные, которые трудно прочитать. Но можно загрузить профиль кучи, используя debug/pprof/

heap/?debug=0, а затем открыть его с помощью `go tool` (это та же команда, что упоминалась в предыдущем разделе), чтобы просмотреть данные с помощью веб-интерфейса.

На рис. 12.36 показан пример графического отображения кучи. Вызов метода `MetadataResponse.decode` приводит к выделению 1536 Кбайт в куче (что составляет 6.32 % от общего объема кучи). Но из этих 1536 Кбайт напрямую были выделены 0 Кбайт, поэтому нужно сделать второй вызов. С помощью метода `TopicMetadata.decode` было выделено 512 Кбайт из 1536 Кбайт; остальные 1024 Кбайт были выделены другим способом.

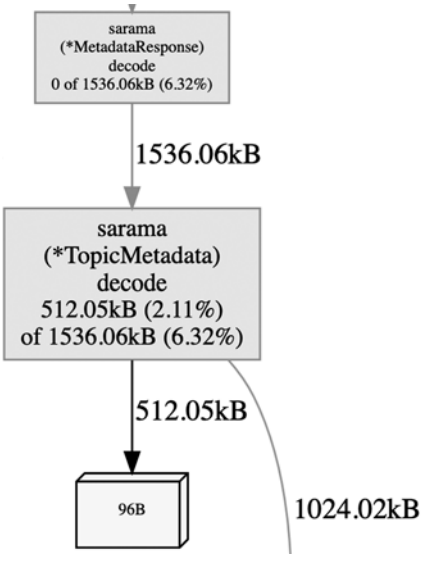


Рис. 12.36. Графическая схема кучи

Так мы можем двигаться по цепочке вызовов, чтобы понять, какая часть приложения отвечает за бóльшую часть выделений памяти в куче. Можно рассмотреть различные типы выборки:

- `alloc_objects` — общее количество объектов, под которые выделено место в памяти;
- `alloc_space` — общий объем распределенной памяти;
- `inuse_objects` — количество объектов, под которые выделено и еще не освобождено место в памяти;
- `inuse_space` — объем распределенной памяти, которая еще не освобождена.

Еще одна очень полезная возможность при профилировании кучи — отслеживание утечек памяти. Для языка, в который встроен сборщик мусора, обычная процедура выглядит так:

1. Запустите GC.
2. Загрузите данные кучи.
3. Подождите несколько секунд/минут.
4. Запустите другой GC.
5. Загрузите другие данные кучи.
6. Сравните.

Принудительная сборка мусора перед загрузкой данных — это способ предотвратить ложные выводы. Например, если мы увидим максимальное число сохраненных объектов без предварительного запуска GC, то не будем знать, является эта ситуация утечкой памяти или эти объекты будут обработаны при следующем запуске GC.

Используя `pprof`, можно загрузить профиль кучи и тем временем принудительно выполнить GC. Вот как это делается:

1. Перейдите к `/debug/pprof/heap?gc=1` (запустите GC и загрузите профиль кучи).
2. Подождите несколько секунд/минут.
3. Перейдите к `/debug/pprof/heap?gc=1` еще раз.
4. Используйте `go tool` для сравнения обоих профилей кучи:

```
$ go tool pprof -http=:8080 -diff_base <file2> <file1>
```

ПРИМЕЧАНИЕ Другой тип профилирования, связанный с кучей, — это `allocs`, который сообщает о выделении мест в памяти. Профилирование кучи показывает текущее состояние памяти кучи. Чтобы получить представление о прошлых распределениях памяти (с момента запуска приложения), используйте профилирование таких распределений. Как я говорил, поскольку выделение места в стеке происходит достаточно быстро, оно не учитывается в профилировании, которое собирает данные только по куче.

На рис. 12.37 показано, к каким данным можно получить доступ. Например, объем памяти в куче, необходимый для метода `newTopicProducer` (вверху слева), уменьшился (–513 Кбайт). Напротив, объем для `updateMetadata` (внизу

справа) увеличился (+512 Кбайт). Медленные увеличения нормальны. Второй профиль кучи мог быть получен, например, в середине сервисного вызова. Можно повторить этот процесс или подождать несколько дольше. Важно отслеживать, нет ли постоянного увеличения выделенного места в памяти для конкретного объекта.

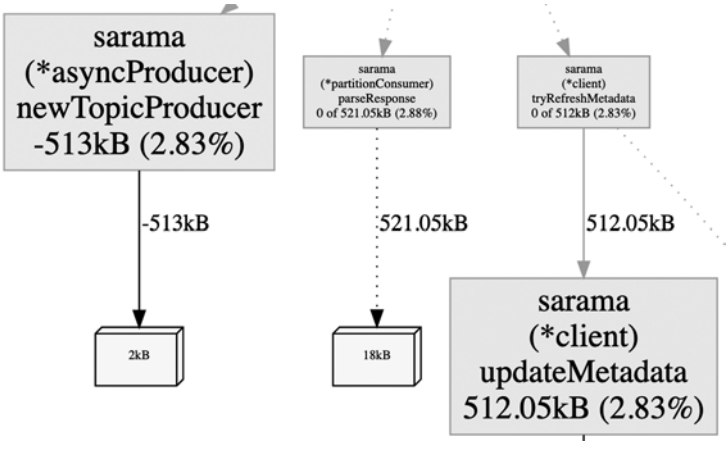


Рис. 12.37. Различия между двумя профилями кучи

Профилирование горутин

Профиль `goroutine` выдает отчет о трассировке стека всех текущих горутин в приложении. Скачайте соответствующий файл с помощью debug/pprof/goroutine/?debug=0 и снова используйте `go tool`. На рис. 12.38 показано, какую информацию можно получить.

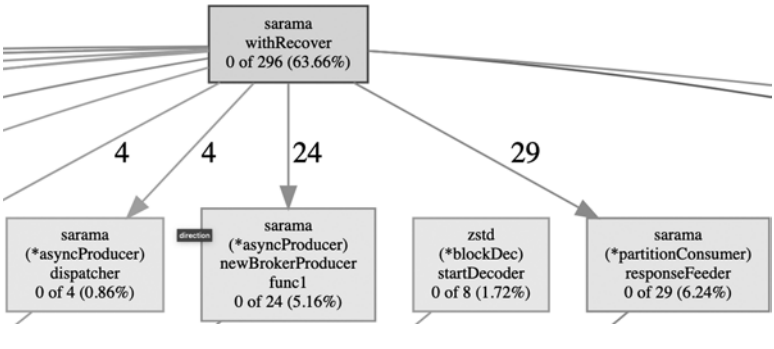


Рис. 12.38. Графическая схема для горутин

Мы видим текущее состояние приложения и количество горутин, созданных для каждой функции. В этом случае `withRecover` создал 296 выполняющихся горутин (63 %), а 29 были связаны с вызовом `responseFeeder`.

Эта информация также полезна, если подозревается утечка горутин. Можно просмотреть данные профилировщика горутин, чтобы узнать, какая часть системы является подозрительной.

Профилирование блокировок

Профиль `block` показывает, где горутины блокируются, ожидая синхронизационных примитивов. Можно отследить:

- отправку или получение по небуферизованному каналу;
- отправку в заполненный канал;
- прием из пустого канала;
- конфликт мьютекса;
- состояние ожидания сети или файловой системы.

Профилирование блокировок фиксирует также время ожидания горутины, что доступно в debug/pprof/block. Этот профиль будет чрезвычайно полезен, если мы подозреваем, что производительность оказывается сниженной из-за блокировки вызовов.

Полный дамп стека горутин

Если мы столкнулись с взаимоблокировкой или подозреваем, что горутины находятся в заблокированном состоянии, полный дамп стека горутин (debug/pprof/goroutine/?debug=2) создает дамп всех текущих трассировок стека горутин. Это может быть полезно в качестве первого шага для анализа ситуации. Например, в следующем дампе показана горутина `Sarama`, заблокированная на 1420 минут при операции получения из канала:

```
goroutine 2494290 [chan receive, 1420 minutes]:
github.com/Shopify/sarama.(*syncProducer).SendMessages(0xc00071a090,
➥ {0xc0009bb800, 0xfb, 0xfb})
     /app/vendor/github.com/Shopify/sarama/sync_producer.go:117 +0x149
```

По умолчанию профиль `block` неактивен. Для его запуска вызовите `runtime.SetBlockProfileRate`. Эта функция контролирует те события блокировок горутин, о которых выдаются сообщения. После запуска профилировщик будет

продолжать собирать данные в фоновом режиме, даже если мы не будем обращаться к конечной точке debug/pprof/block. При высокой частоте выборки будьте осторожны, чтобы сильно не снизить производительность.

Профилирование мьютексов

Последний тип профилей связан с блокировкой, но только в отношении мьютексов. Если вы подозреваете, что приложение тратит значительное время на ожидание блокировки мьютексов, что негативно сказывается на выполнении, используйте профилирование мьютексов. Оно доступно в /debug/pprof/mutex.

Этот профиль работает аналогично профилю блокировки. По умолчанию он отключен: его нужно запустить с помощью функции `runtime.SetMutexProfileFraction`, которая определяет долю событий конкуренции мьютексов, которые будут отражены в профиле.

Дополнительные замечания о профилировании:

- Мы не упоминали профиль `threadcreate`, потому что с 2013 года он перестал работать (https://github.com/golang/go/issues/6104).
- В каждый промежуток времени запускайте только один профилировщик, например, не включайте одновременно профилирование CPU и кучи. Это может привести к ошибочным данным наблюдений.
- `pprof` расширяемый, и мы можем создавать собственные профили, используя `pprof.Profile`.

Мы рассмотрели самые важные профили, которые можем запускать, чтобы понимать, как работает приложение и какие есть пути его оптимизации. Рекомендуется включать `pprof` даже в продакшене, поскольку в большинстве случаев он предлагает отличный баланс между затратами и объемом получаемой информации. Запуск некоторых профилей, например CPU, приводит к снижению общей производительности, но только в то время, когда он включен. Теперь рассмотрим трассировщик выполнения.

12.8.2. Трассировщик выполнения

Трассировщик выполнения — это инструмент, который фиксирует различные события во время выполнения приложения с помощью `go tool`, чтобы визуализировать их. Это полезно для:

- понимания событий, происходящих во время выполнения приложения, например работы сборщика мусора;

- понимания того, как выполняются горутины;
- выявления плохо распараллеленных процессов.

Попробуем увидеть это на примере, который приводили в описании ошибки #56 (полагать, что конкурентность быстрее). Мы тогда обсудили две параллельные версии алгоритма сортировки слиянием. Проблема с первой версией заключалась в плохом распараллеливании, что приводило к созданию слишком большого количества горутин. Посмотрим, как трассировщик поможет в проверке этого утверждения.

Создадим бенчмарк для первой версии и выполним его с флагом `-trace`, чтобы запустить трассировщик выполнения:

```
$ go test -bench=. -v -trace=trace.out
```

> **ПРИМЕЧАНИЕ** Можно загрузить удаленный файл трассировки, используя конечную точку /debug/pprof/trace?debug=0 pprof.

Эта команда создает файл trace.out, который открывается с помощью `go tool`:

```
$ go tool trace trace.out
2021/11/26 21:36:03 Parsing trace...
2021/11/26 21:36:31 Splitting trace...
2021/11/26 21:37:00 Opening browser. Trace viewer is listening on
   http://127.0.0.1:54518
```

Откроется браузер, и мы можем щелкнуть по View Trace, чтобы просмотреть все трассировки за определенный период времени, как показано на рис. 12.39.

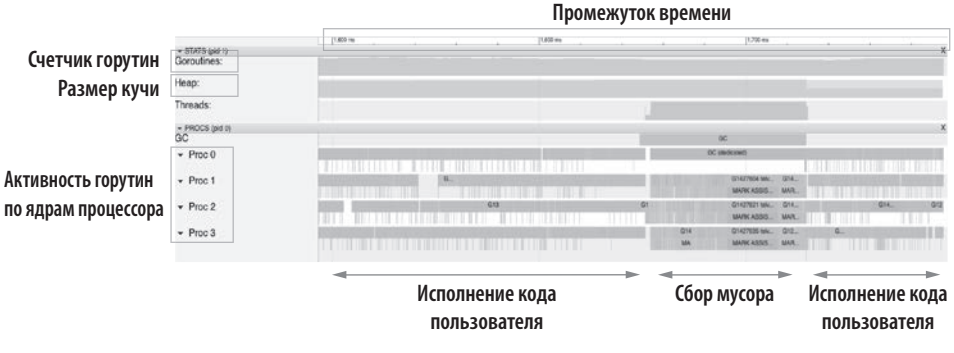


Рис. 12.39. Отображение активности горутин и событий во время выполнения приложения (например, фаза сборки мусора)

На скриншоте отражен промежуток примерно в 150 миллисекунд. Мы видим полезные метрики: счетчик горутин и размер кучи. Размер кучи неуклонно растет, пока не будет запущен сборщик мусора. Можно понаблюдать за активностью приложения на каждом ядре CPU. Временной интервал начинается с кода уровня пользователя, затем выполняется остановка всей системы (stop the world), что занимает все четыре ядра CPU примерно на 40 миллисекунд.

Что касается конкурентности, мы видим, что эта версия использует все доступные ядра CPU. Но посмотрим на рис. 12.40, на котором участок в 1 миллисекунду показан в увеличенном масштабе. Каждый столбик этой диаграммы соответствует одному выполнению горутины. Наличие слишком большого числа маленьких столбиков наводит на мысль о том, что тут что-то не так: это означает, что выполнение плохо распараллелено.

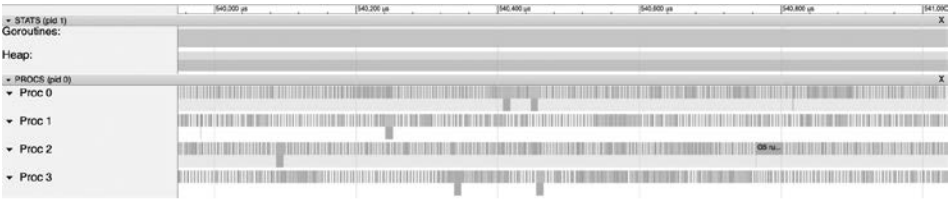


Рис. 12.40. Наличие слишком большого числа маленьких столбиков означает, что выполнение плохо распараллелено

На рис. 12.41 картина показана в большем увеличении, чтобы рассмотреть, как горутины оркестрованы. Примерно 50 % процессорного времени тратится не на выполнение кода приложения. Пробелы отображают время, необходимое среде выполнения Go для запуска и оркестрации новых горутин.

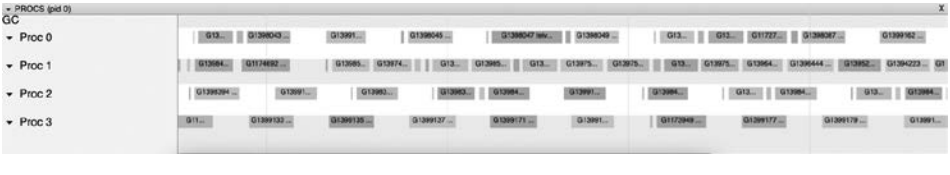


Рис. 12.41. Примерно 50 % процессорного времени тратится на обработку переключений между горутинами

Сравним это со второй параллельной реализацией, которая была примерно на порядок быстрее. Рисунок 12.42 снова отображает промежуток времени в 1 миллисекунду.

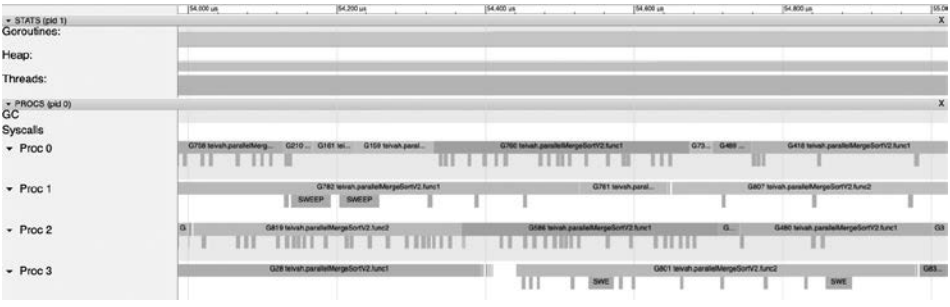


Рис. 12.42. Количество пробелов значительно меньше, что говорит о более полной загрузке CPU

Выполнение каждой горутины занимает больше времени, а количество пробелов значительно меньше. Следовательно, процессор в гораздо большей мере занят выполнением кода приложения, чем в первой версии. Каждая миллисекунда процессорного времени расходуется более эффективно, что объясняет различия в бенчмарках.

Обратите внимание, что степень гранулярности трассировок зависит от горутины, а не от функции, как при профилировании CPU. Но можно определить задачи уровня пользователя, чтобы получить представление о каждой функции или группе функций с помощью пакета `runtime/trace`.

Представьте себе функцию, которая вычисляет число Фибоначчи, а затем записывает его в глобальную переменную, используя `atomic`. Можно определить две разные задачи:

```
var v int64
ctx, fibTask := trace.NewTask(context.Background(), "fibonacci")
trace.WithRegion(ctx, "main", func() {
   v = fibonacci(10)
})
fibTask.End()
ctx, fibStore := trace.NewTask(ctx, "store")
trace.WithRegion(ctx, "main", func() {
   atomic.StoreInt64(&result, v)
})
fibStore.End()
```

Создание задачи Fibonacci

Создание задачи store

Используя `go tool`, мы получим более точную информацию о том, как выполняются эти две задачи. В предыдущем пользовательском интерфейсе трассировки (см. рис. 12.42) мы видим границы каждой задачи для каждой горутины.

В пользовательских задачах (User-Defined Tasks) можем следить за распределением продолжительности (рис. 12.43).

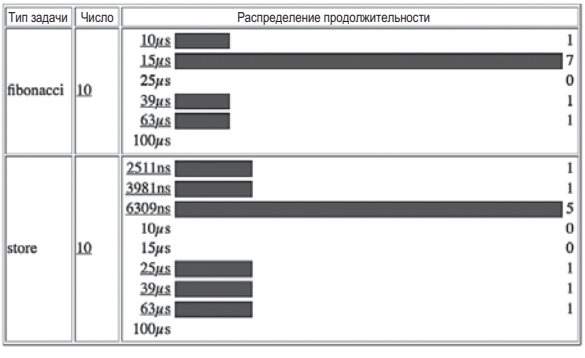


Рис. 12.43. Распределение задач пользовательского уровня

Мы видим, что в большинстве случаев задача `fibonacci` выполняется менее чем за 15 микросекунд, тогда как задача `store` требует менее 6309 наносекунд.

В предыдущем разделе мы обсудили виды информации, которую получаем в результате профилирования CPU. Каковы их основные отличия от данных, которые возможно получить из трассировки на уровне пользователя?

- Профилирование CPU:
  - основано на сэмплировании;
  - делается для каждой функции;
  - не опускается ниже частоты сэмплирования (по умолчанию 10 миллисекунд).
- Трассировки на уровне пользователя:
  - не основаны на сэмплировании;
  - исполняются для каждой горутины (если мы не используем пакет `runtime/trace`);
  - время выполнения не ограничено никакой частотой.

Таким образом, трассировщик выполнения — это мощный инструмент, позволяющий понять, как работает приложение. Как было показано на примере сортировки слиянием, мы можем идентифицировать плохо распараллеленное выполнение. Однако гранулярность трассировки остается на уровне горутин,

если мы не используем пакет `runtime/trace` вручную, в отличие, например, от профилирования CPU.

Используйте как профилирование, так и трассировщик выполнения, чтобы получить максимальную отдачу от стандартных инструментов диагностики Go и оптимизировать приложение.

В следующем разделе поговорим, как работает сборщик мусора и как его настроить.

12.9. ОШИБКА #99: НЕ ПОНИМАТЬ, КАК РАБОТАЕТ СБОРЩИК МУСОРА

Сборщик мусора (GC) — важная часть языка Go, упрощающая жизнь разработчикам. Он позволяет отслеживать и освобождать ресурсы кучи, которые больше не нужны. Поскольку мы не можем заменить каждое резервирование памяти в куче выделением места в стеке, то важно знать, как работает GC, чтобы оптимизировать приложения.

12.9.1. Концепции

Сборщик мусора хранит дерево ссылок на объекты. GC в Go основан на алгоритме пометок (mark-and-sweep-algorithm), который состоит из двух этапов:

- Этап пометки (mark) — просмотр всех объектов в куче и пометка тех, которые все еще используются.
- Этап очистки (sweep) — просмотр дерева ссылок от корня и освобождение места, ранее выделенного под блоки объектов, на которые больше нет никаких ссылок.

Когда запускается сборщик мусора, он сначала выполняет действия, которые приводят к остановке всей системы — stop the world (точнее, две остановки системы на каждый цикл GC). Весь доступный процессорный временной интервал используется для сборки мусора, и выполнение кода приложения приостанавливается. После завершения этих шагов система вновь запускается, приложение продолжает выполнение и одновременно запускается конкурентная фаза. Именно поэтому сборщик мусора Go называется конкурентной пометкой и освобождением (concurrent mark-and-sweep): его целью является уменьшение

количества stop-of-the-world-операций на каждом цикле GC и работа в основном конкурентно с приложением.

GC также включает в себя способ освобождения памяти после пика потребления. Представьте, что приложение основано на двух фазах:

- фаза инициализации, которая приводит к частым выделениям памяти и к большому размеру кучи;
- фаза выполнения с умеренным количеством выделений памяти и небольшому размеру кучи.

Как среда Go реагирует на тот факт, что большой размер кучи имеет смысл только при старте приложения, но не далее? Эта ситуация обрабатывается как часть операций по сборке мусора так называемым вспомогательным сборщиком мусора (periodic scavender). Через какое-то время GC обнаруживает, что такая большая куча больше не требуется, поэтому освобождает часть памяти и возвращает ее в распоряжение ОС.

> **ПРИМЕЧАНИЕ** Если вспомогательный сборщик мусора недостаточно быстр, можно принудительно вернуть память в распоряжение ОС, используя `debug.FreeOSMemory()`.

Важный вопрос: когда будет выполняться цикл сборки мусора? По сравнению с другими языками, например Java, конфигурация Go остается достаточно простой. Она задается одной переменной среды: `GOGC`. Эта переменная определяет тот процент роста кучи с момента последней сборки мусора, достижение которого должно запускать следующий цикл GC; ее значение по умолчанию — 100 %.

Посмотрим пример. Предположим, что сборщик мусора запустился только что, а текущий размер кучи составляет 128 Мбайт. Если `GOGC=100`, то следующая сборка мусора запустится, когда размер кучи достигает 256 Мбайт. По умолчанию GC выполняется каждый раз, когда размер кучи удваивается. Кроме того, если сборка мусора не выполнялась в течение последних двух минут, то Go запустит ее в принудительном порядке.

Если мы профилируем приложение с теми нагрузками, которые характерны для продакшена, то можем задать значение `GOGC` очень точно:

- его уменьшение приведет к замедлению роста кучи и увеличит нагрузку на GC;
- и наоборот, его увеличение приведет к ускорению роста кучи и снизит нагрузку на GC.

Трассировка GC

Выведем трассировку GC, задав значение переменной среды GODEBUG, например, при запуске бенчмарка:

```
$ GODEBUG=gctrace=1 go test -bench=. -v
```

Установка флага gctrace записывает трассировку в stderr каждый раз, когда запускается GC.

Рассмотрим несколько примеров, чтобы понять, как ведет себя сборщик мусора в случае увеличения нагрузки.

12.9.2. Примеры

Допустим, мы предоставляем пользователям какие-то публичные сервисы. В пиковое время, в 12:00 к ним подключается миллион пользователей. Но до этого был устойчивый плавный рост числа подключенных пользователей. На рис. 12.44 показан средний размер кучи и размер при запуске сборщика мусора, если значение GOGC оставить равным 100.

Рис. 12.44. Устойчивый и плавный рост числа подключенных пользователей

Поскольку значение GOGC установлено равным 100, сборщик мусора запускается каждый раз, когда размер кучи удваивается. Поскольку количество пользователей растет более-менее плавно, мы должны столкнуться с приемлемым количеством циклов GC в течение дня (рис. 12.45).

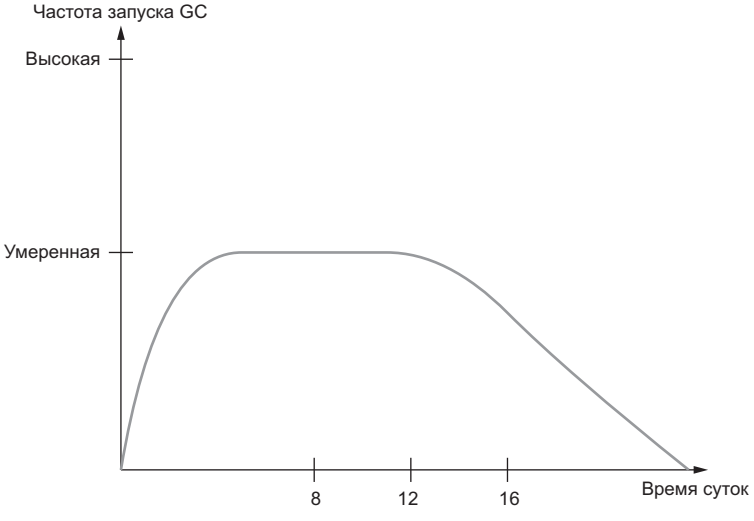


Рис. 12.45. Частота запуска GC никогда не достигает уровня выше умеренного

В начале дня должно быть умеренное количество циклов GC. После 12:00, когда количество пользователей начинает уменьшаться, количество циклов GC также должно плавно уменьшаться. В таком сценарии сохранение значения GOGC на уровне 100 должно быть адекватным решением.

Теперь рассмотрим второй сценарий, когда большинство из миллиона пользователей подключается менее чем за один час (рис. 12.46). В 8:00 средний размер кучи быстро растет и достигает своего пика примерно через час.

Как показано на рис. 12.47, частота запусков GC сильно меняется в течение этого часа. Из-за значительного и резкого увеличения размера кучи мы сталкиваемся с ситуацией, когда циклы сборки мусора запускаются очень часто и в течение короткого периода времени. Несмотря на то что сборка мусора в Go выполняется конкурентно, такая ситуация может привести к значительному числу периодов остановки всей системы и увеличить среднюю задержку, что будет заметно пользователям.

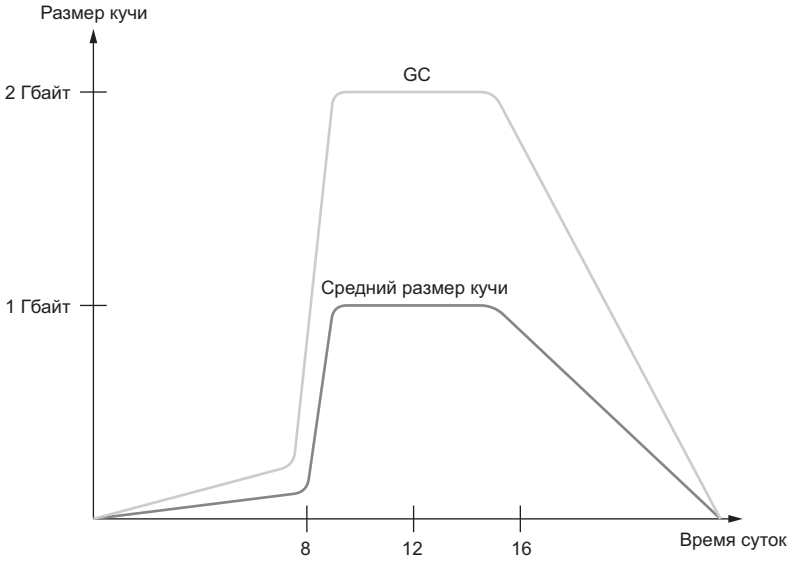


Рис. 12.46. Резкий рост числа подключенных пользователей

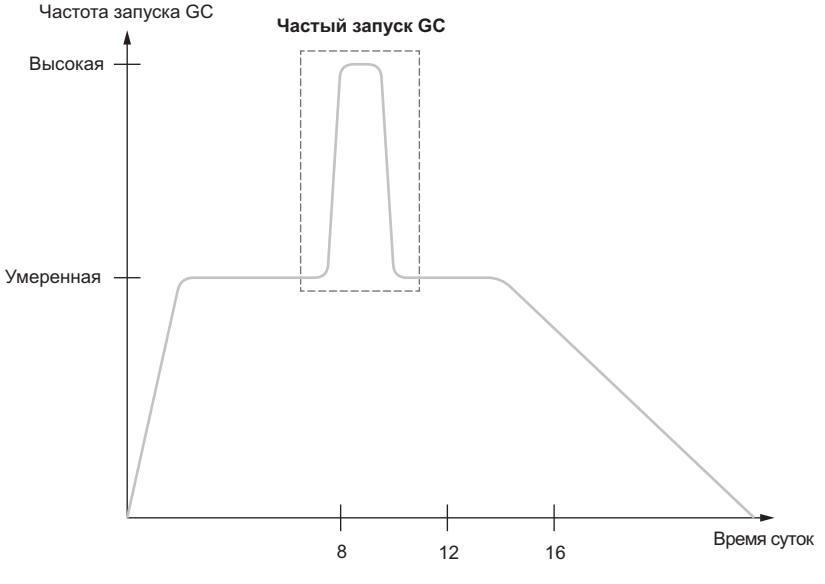


Рис. 12.47. В течение одного часа мы сталкиваемся с высокой частотой циклов запуска GC

В этом случае нужно рассмотреть возможность увеличения значения GOGC, чтобы уменьшить нагрузку на GC. Обратите внимание, что увеличение GOGC не приводит к линейному увеличению выгоды: ведь чем больше размер кучи, тем больше времени потребуется на ее очистку. Следовательно, в случае тех нагрузок, которые характерны для продакшена, нужно проявлять осторожность в тонкой настройке значения GOGC.

В совершенно исключительных условиях, при еще более резком изменении, подгонки значения GOGC может оказаться недостаточно. Допустим, что вместо перехода от 0 к 1 миллиону пользователей в течение часа, такой скачок происходит за несколько секунд. В течение этих секунд количество запусков сборщика мусора может достичь критического уровня, что приведет к низкой производительности приложения.

Если мы знаем о наличии пика в графике роста размера кучи, можно использовать какой-нибудь трюк, который заставляет выделять большое количество памяти для повышения стабильности кучи. Например, принудительно выделить 1 Гбайт с помощью глобальной переменной в main.go:

```
var min = make([]byte, 1_000_000_000) // 1 Гбайт
```

В чем смысл? Если GOGC оставить равным 100, то вместо того, чтобы запускать сборщик мусора каждый раз, когда размер кучи удваивается (что происходит очень часто в течение этих нескольких секунд), Go будет его запускать только тогда, когда куча достигает 2 Гбайт. Это должно уменьшить количество циклов сборки мусора, запускаемых в момент подключения всех пользователей, что снизит влияние GC на среднюю задержку.

На это можно возразить: когда размер кучи уменьшается, такой трюк приведет к большим и бесполезным затратам памяти. Но на самом деле это не так. В большинстве ОС выделение памяти для переменной min не ведет к тому, что приложение будет потреблять 1 Гбайт памяти. Вызов make влечет за собой системный вызов mmap(), что приводит к ленивому распределению памяти. Например, в Linux память виртуально адресуется и отображается через таблицы страниц. Использование mmap() резервирует 1 Гбайт памяти в виртуальном, а не в физическом адресном пространстве. Только чтение или запись вызовут ошибку страницы, что приведет к фактическому выделению физической памяти. Таким образом, даже если приложение запускается без подключенных клиентов, оно не будет потреблять 1 Гбайт физической памяти.

ПРИМЕЧАНИЕ Можно проверить такое поведение с помощью инструмента ps.

Для оптимизации сборки мусора важно понимать, как ведет себя GC. Используйте `GOGC` для настройки запуска следующего цикла GC. В большинстве случаев достаточно держать значение этой переменной на уровне `100`. Но если приложение может сталкиваться с пиками в числе запросов, приводящими к частому запуску GC и к задержкам, можно это значение увеличивать. Наконец, в случае исключительно резкого пика запросов подумайте о том, чтобы прибегнуть к трюку с сохранением минимального размера виртуальной кучи.

В последнем разделе этой главы поговорим о последствиях запуска Go в Docker и Kubernetes.

12.10. ОШИБКА #100: НЕ ПОНИМАТЬ ОСОБЕННОСТЕЙ ЗАПУСКА GO ВНУТРИ DOCKER И KUBERNETES

Опрос Go-разработчиков, проведенный в 2021 году (https://go.dev/blog/survey2021-results), выявил, что написание сервисов — это самое распространенное применение Go. А Kubernetes — это наиболее широко используемая платформа для развертывания таких сервисов. Важно понимать последствия запуска Go в Docker и Kubernetes, чтобы предотвращать появление нежелательных ситуаций типа троттлинга CPU.

В разделе, посвященном ошибке #56 (полагать, что конкурентность быстрее), я говорил, что переменная `GOMAXPROCS` определяет лимит потоков ОС, отвечающих за одновременное выполнение фрагментов кода пользовательского уровня. По умолчанию ее значение установлено равным количеству логических ядер CPU, видимых для ОС. Что это означает в контексте Docker и Kubernetes?

Предположим, что наш кластер Kubernetes состоит из восьмиядерных узлов. Когда контейнер развертывается в Kubernetes, мы можем определить лимит CPU, чтобы гарантировать, что приложение не будет потреблять все ресурсы хоста. Например, следующая конфигурация ограничивает использование CPU до 4000 millicpu (или миллиядер), то есть четырьмя ядрами CPU:

```
spec:
   containers:
   - name: myapp
      image: myapp
      resources:
         limits:
            cpu: 4000m
```

Мы можем предположить, что когда приложение будет развернуто, значение `GOMAXPROCS` будет основано на этих ограничениях и, следовательно, равно 4. Но это не так: оно устанавливается равным количеству логических ядер на хосте, то есть 8. Разберемся с этим.

Kubernetes использует Completely Fair Scheduler (CFS) в качестве планировщика процессов. CFS также используется для соблюдения ограничений CPU для ресурсов пода. При администрировании кластера Kubernetes администратор может настроить два параметра:

- `cpu.cfs_period_us` (глобальная установка);
- `cpu.cfs_quota_us` (установка для каждого пода).

Первый определяет период, а второй — квоту. По умолчанию период установлен равным 100 мс. Между тем значение квоты по умолчанию — это то, сколько процессорного времени приложение может потребить за 100 мс. Ограничение установлено на четыре ядра, это означает, что оно равно 400 мс (4 × 100 мс). Таким образом, CFS гарантирует, что наше приложение никогда не потребляет более 400 мс процессорного времени в течение 100 мс.

Представим сценарий, когда в некоторый момент времени несколько горутин выполняются в четырех разных потоках. Выполнение каждого потока запланировано на разных ядрах (1, 3, 4 и 8), как показано на рис. 12.48.

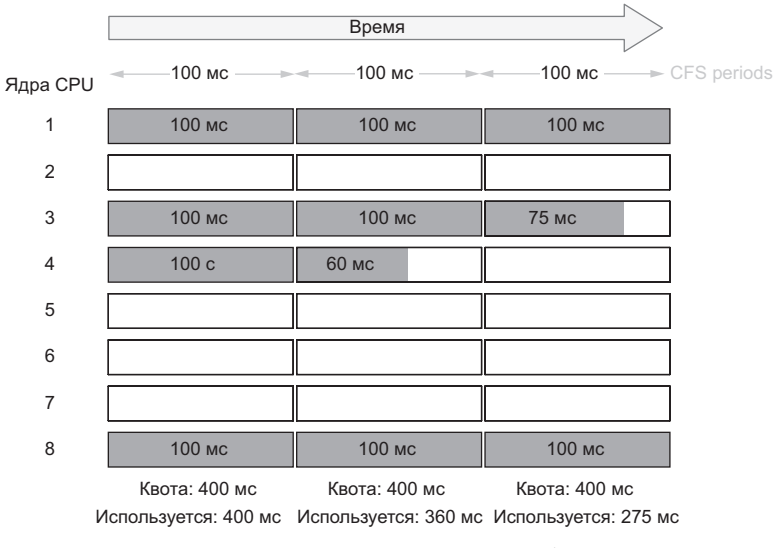


Рис. 12.48. На каждые 100 мс приложение потребляет менее 400 мс

В течение первого 100-миллисекундного периода заняты четыре потока, поэтому мы потребляем 400 из 400 мс: 100 % квоты. Во второй период мы потребляем 360 мс из 400 мс и т. д. Все нормально, потому что приложение потребляет ресурсы меньше квоты.

Теперь вспомним, что `GOMAXPROCS` равно 8. В худшем случае может быть восемь потоков, выполнение каждого из которых запланировано на отдельном ядре (рис. 12.49).

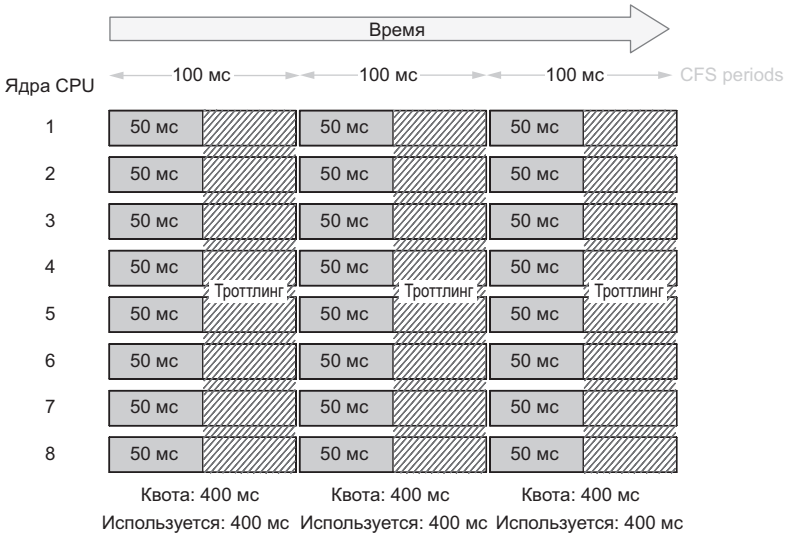


Рис. 12.49. В течение каждых 100 мс происходит троттлинг CPU после 50 мс

На каждые 100 мс установлена квота в 400 мс. Если восемь потоков заняты выполнением горутин, через 50 мс мы достигаем этой квоты (8 × 50 мс = 400 мс). Каковы последствия? CFS будет ограничивать ресурсы CPU. Следовательно, ресурсы CPU не будут выделяться до начала следующего периода. Другими словами, приложение будет приостановлено на 50 мс.

Например, время выполнения какого-то сервиса со средней задержкой в 50 мс может вырастать до 150 мс — это увеличение задержки на 300 %.

Как выйти из такой ситуации? Прежде всего, следите за проблемой Go 33803 (https://github.com/golang/go/issues/33803). Возможно, в будущей версии Go `GOMAXPROCS` будет учитывать CFS.

На сегодняшний день решением является библиотека `automaxprocs`, созданная Uber (github.com/uber-go/automaxprocs). Используйте эту библиотеку, добавив пустой импорт в go.uber.org/automaxprocs в main.go: он автоматически установит `GOMAXPROCS` в соответствии с квотой CPU контейнера Linux. В предыдущем примере для `GOMAXPROCS` было установлено значение 4 вместо 8, поэтому мы не смогли бы достичь состояния, когда происходит троттлинг CPU.

В настоящее время CFS в Go не поддерживается. `GOMAXPROCS` зависит от особенностей хост-машины, а не от определенных ограничений CPU. Следовательно, можно достичь состояния, когда происходит троттлинг CPU, что приводит к длительным паузам и негативным эффектам, например к значительным увеличениям задержек. До тех пор, пока Go не поддерживает CFS, одним из решений будет использование `automaxprocs` для автоматической установки `GOMAXPROCS` в соответствии с заданной квотой.

ИТОГИ

- Понимать, как использовать кэши CPU, важно для оптимизации приложений, которые интенсивно потребляют ресурсы процессора, поскольку кэш L1 примерно в 50–100 раз быстрее, чем основная память.
- Знание концепции кэш-линии поможет организовать данные в приложениях, интенсивно использующих данные. CPU не извлекает данные из памяти слово за словом, а копирует блок памяти в 64-байтовую кэш-линию. Чтобы получить максимальную отдачу от каждой отдельной кэш-линии, учитывайте принцип пространственной локальности и организуйте память в соответствии с ним.
- Предсказуемость поведения кода для CPU также может быть эффективным способом оптимизации определенных функций. Например, единичный или постоянный шаг предсказуем для CPU, а неединичный шаг (например, связный список) непредсказуем.
- Чтобы избежать критических шагов и, следовательно, использования только крошечной части кэша, имейте в виду, что кэши разбиваются на сектора.
- Знание того, что более низкие уровни кэша CPU не используются совместно всеми ядрами, помогает избежать снижающих производительность паттернов конкурентного кода, например ложного совместного использования. Совместное использование памяти — это иллюзия.
- Используйте параллелизм на уровне инструкций (ILP) для оптимизации определенных частей кода, чтобы CPU мог выполнять как можно больше

инструкций параллельно. Выявление опасностей данных — один из основных моментов, связанных с этим.

- Вы избежите типичных ошибок, если вспомните, что в Go основные типы выравниваются по своему размеру. Имейте в виду, что реорганизация полей структуры по размеру в порядке убывания может привести к более компактным структурам (выделение меньшего объема памяти и, возможно, лучшая пространственная локализация).
- Понимание фундаментальных различий между кучей и стеком поможет оптимизировать приложения. Выделение памяти в стеке почти ничего не стоит, в то время как такая же операция в куче выполняется медленнее и зависит от того, как память очищена сборщиком мусора.
- Сокращение числа выделений памяти также важно при оптимизации. Это можно сделать разными способами, например, тщательно проработать дизайн вашего API, чтобы предотвратить совместное использование переменных, определенных в функции, которая вызывается из основного фрагмента кода, применять встроенные в компилятор Go методы оптимизации, а также `sync.Pool`.
- Применяйте технику быстрого встраивания, чтобы эффективно сократить амортизированное время вызова функции.
- Используйте профилирование и трассировщик выполнения, чтобы понять, как работает приложение и какие его части нужно оптимизировать.
- Знание того, как настраивать сборщик мусора, даст множество преимуществ. Например, вы сможете более эффективно обрабатывать пиковые увеличения нагрузки.
- Чтобы избежать троттлинга CPU при развертывании в средах Docker и Kubernetes, помните, что Go не поддерживает CFS.

ЗАКЛЮЧЕНИЕ

Поздравляю! Вы дошли до конца книги, и я надеюсь, что она понравилась вам и поможет в ваших личных и/или профессиональных проектах.

Помните, что ошибки — это часть процесса обучения. И как я говорил в предисловии, именно это послужило важным источником вдохновения для написания этой книги. В конце концов, самое главное — это наша способность учиться на ошибках.

Подписывайтесь на меня в Твиттере, чтобы пообсуждать темы этой книги: @teivah.

Тейва Харшани

100 ошибок Go и как их избежать

Перевел с английского Д. Строганов

Руководитель дивизиона	*Ю. Сергиенко*
Ведущий редактор	*Е. Строганова*
Литературный редактор	*К. Тульцева*
Художественный редактор	*В. Мостипан*
Корректоры	*С. Беляева, Н. Викторова*
Верстка	*Л. Егорова*

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 09.2023. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 25.08.23. Формат 70×100/16. Бумага офсетная. Усл. п. л. 38,700. Тираж 1000. Заказ 0000.