O'REILLY

Облачный Go

Создание надежных сервисов в ненадежных окружениях

Мэтью А. Титмус

ДМК ПРЕСС
издательство

Мэтью А. Титмус

Облачный Go

Cloud Native Go

Building Reliable Services in Unreliable Environments

Matthew A. Titmus

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Облачный Go

Создание надежных служб в ненадежных окружениях

Мэтью А. Титмус

ДМК Пресс
издательство
Москва, 2022

УДК 004.432
ББК **32.972.1**
Э98

Титмус М. А.
Э98 Облачный Go / пер. с англ. А. Н. Киселева. – М.: ДМК Пресс, 2022. – 418 с.: ил.

ISBN 978-5-97060-965-1

Go – первый язык программирования, спроектированный специально для разработки облачных приложений. В настоящее время он занял лидирующие позиции в облачной разработке и используется повсюду: от Docker до Harbour, от Kubernetes до Consul, от InfluxDB до CockroachDB.

Требования к масштабированию вынуждают разработчиков размещать свои сервисы на десятках и сотнях серверов – IT-отрасль постепенно становится «облачной». Но как разрабатывать и поддерживать такой сервис? В этой книге описывается практическая реализация сложных принципов проектирования облачных вычислений с помощью Go. Издание адресовано опытным разработчикам, особенно инженерам веб-приложений и инженерам по надежности, которые решают задачи управления и развертывания облачных приложений.

УДК 004.432
ББК 32.972.1

ISBN 978-1-492-07633-9 (англ.)
ISBN 978-5-97060-965-1 (рус.)

Тебе, папа.

Нам очень не хватает твоей мягкости, мудрости и смирения.

Кроме того, это ты научил меня программировать,
поэтому любые ошибки в этой книге
технически считаются твоими ошибками!

Отзывы о книге «Облачный Go»

Автор книги проделал большую работу, подробно описав высокоуровневую концепцию «облачных приложений» и приемы ее реализации с использованием современного языка программирования Go. В результате получилась захватывающая и вдохновляющая книга.

– Ли Атчисон (*Lee Atchison*)
Владелец Atchison Technology LLC

Это первая книга, из встречавшихся мне, которая с такой широтой и глубиной освещает современные приемы реализации облачных вычислений. Представленные здесь шаблоны сопровождаются наглядными примерами решения реальных задач, с которыми инженеры сталкиваются ежедневно.

– Альваро Атьенза (*Alvaro Atienza*)
Инженер по надежности, Flatiron Health

На страницах этой книги ясно (и с юмором) отражен богатый опыт Мэтта в искусстве и науке построения надежных систем в принципиально ненадежном мире. Присоединяйтесь к нему, и он познакомит вас с фундаментальными строительными блоками и приемами конструирования систем, позволяющими создавать масштабные и надежные системы из эфемерных и ненадежных компонентов современной облачной инфраструктуры.

– Дэвид Никпонски (*David Nicponski*)
Главный инженер, Robinhood

За последние несколько лет наметились две важные тенденции: язык Go все чаще используется для разработки не только серверных компонентов, но и инфраструктуры; а инфраструктура перемещается в облако. В этой книге кратко описывается современное состояние сочетания этих двух факторов.

– Натали Пистунович (*Natalie Pistunovich*)
Ведущий пропагандист передовых методов разработки, Aerospike

Я начал читать эту книгу, почти ничего не зная о Go, и закончил, чувствуя себя экспертом. Я бы даже сказал, что, просто прочитав эту книгу, я стал намного более опытным инженером.

– Джеймс Куигли (*James Quigley*)
Инженер по надежности систем, Bloomberg

Содержание

От издательства

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв на нашем сайте www.dmkpress.com, зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com; при этом укажите название книги в теме письма.

Если вы являетесь экспертом в какой-либо области и заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу dmkpress@gmail.com.

Скачивание исходного кода примеров

Скачать файлы с дополнительной информацией для книг издательства «ДМК Пресс» можно на сайте www.dmkpress.com на странице с описанием соответствующей книги.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы обеспечить высокое качество наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг, мы будем очень благодарны, если вы сообщите о ней главному редактору по адресу dmkpress@gmail.com. Сделав это, вы избавите других читателей от недопонимания и поможете нам улучшить последующие издания этой книги.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и O'Reilly очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконной публикацией какой-либо из наших книг, пожалуйста, пришлите нам ссылку на интернет-ресурс, чтобы мы могли применить санкции.

Ссылку на подозрительные материалы можно прислать по адресу электронной почты dmkpress@gmail.com.

Мы высоко ценим любую помощь по защите наших авторов, благодаря которой мы можем предоставлять вам качественные материалы.

Об авторе

Мэтью А. Титмус – ветеран индустрии разработки программного обеспечения. Научившись создавать виртуальные миры в LPC, он получил удивительно востребованное образование в области молекулярной биологии, создал инструменты анализа терабайтных наборов данных для лаборатории физики высоких энергий, с нуля написал фреймворк для разработки веб-приложений, применил методы распределенных вычислений для анализа ракового генома, а также в числе первых разрабатывал методы машинного обучения на связанных данных.

Он был одним из первых сторонников облачных технологий в целом и языка Go в частности. Последние четыре года специализируется на переносе монолитных приложений в контейнерный облачный мир, помогая компаниям осваивать новые способы разработки, развертывания и управления своими службами. Он увлечен решением задач повышения качества промышленных систем и потратил много времени на обдумывание и реализацию стратегий наблюдения за распределенными системами и управления ими.

Мэтью живет на Лонг-Айленде с самой терпеливой женщиной в мире, на которой ему посчастливилось жениться, и самым очаровательным мальчиком в мире, от которого ему посчастливилось услышать «папа».

Об иллюстрации на обложке

На обложке «Облачный Go» изображено животное из семейства туко-туко (*Ctenomyidae*). Эти неотропические грызуны обитают в южной части Южной Америки.

Название «туко-туко» относится к широкому кругу видов. Эти грызуны имеют плотное тело с мощными короткими лапами и хорошо развитыми когтями. У них большая голова, но маленькие уши, и хотя до 90 % времени они проводят под землей, они имеют относительно большие глаза, по сравнению с другими норными грызунами. Цвет и текстура шерсти туко-туко варьируются в зависимости от вида, но в целом шерсть довольно густая. Хвост короткий и почти без шерсти.

Туко-туко роют системы туннелей, часто весьма обширные и сложные, в песчаной и/или суглинистой почве. В этих системах туннелей имеются отдельные камеры для гнездования и хранения пищи. В ходе эволюции туко-туко претерпели различные морфологические изменения, которые помогают им прекрасно чувствовать себя под землей, и развили хорошее обоняние, помогающее им ориентироваться в туннелях. При рытье нор используют как когти, так и рыло.

Рацион туко-туко состоит в основном из корней, стеблей и трав. В настоящее время считаются сельскохозяйственными вредителями, но во времена до появления европейцев в Южной Америке они были важным источником пищи для коренных народов, особенно на Огненной Земле. Современный охранный статус туко-туко зависит от вида и географического региона. Одним видам присвоена категория «вызывающий наименьшие опасения», тогда как другие считаются «находящимися под угрозой исчезновения». Многие животные, изображенные на обложках книг издательства O'Reilly, находятся под угрозой вымирания; все они очень важны для биосферы.

Иллюстрацию для обложки нарисовал Карен Монтгомери (Karen Montgomery) на основе черно-белой гравюры из энциклопедии «English Cyclopedia Natural History». Текст на обложке набран шрифтами Gilroy Semibold и Guardian Sans. Текст книги набран шрифтом Adobe Minion Pro; текст заголовков – шрифтом Adobe Myriad Condensed; а фрагменты программного кода – шрифтом Ubuntu Mono, созданным Далтоном Маагом (Dalton Maag).

Предисловие

Это волшебное время для инженеров

У нас есть Docker для создания контейнеров и Kubernetes для управления ими. Prometheus помогает нам следить за ними. Consul позволяет обнаруживать их. Jaeger дает возможность организовать взаимодействия между ними. Это лишь несколько примеров, в действительности круг возможностей гораздо шире, и все эти возможности поддерживают новое поколение технологий: все они «облачные», и все они написаны на Go.

Термин «облачный» кажется двусмысленным и отдает рекламной шумихой, но на самом деле он имеет довольно конкретное определение. Согласно Cloud Native Computing Foundation, подразделению известного фонда Linux Foundation, облачное приложение – это приложение, способное масштабироваться синхронно с изменением нагрузки, устойчивое к неопределенности окружения и управляемое в условиях нестабильности и постоянно меняющихся требований. Иначе говоря, облачные приложения создаются для работы в жесткой и неопределенной вселенной.

На основе опыта, накопленного за годы разработки облачного программного обеспечения, около десяти лет назад был создан Go – первый язык программирования, спроектированный специально для разработки облачных приложений. Во многом его появление было обусловлено тем, что типичные серверные языки, использовавшиеся в то время, просто не подходили для создания распределенных приложений, которые производит Google.

С тех пор Go занял лидирующие позиции в облачной разработке и используется повсюду: от Docker до Harbour, от Kubernetes до Consul, от InfluxDB до CockroachDB. Десять из пятнадцати сертифицированных проектов Cloud Native Computing Foundation и 42 из 62[1] его проектов в целом написаны в основном или полностью на Go. И с каждым днем их становится все больше.

Кому адресована эта книга

Эта книга адресована опытным разработчикам, особенно инженерам веб-приложений и инженерам по надежности. Многие из них уже использовали Go для создания веб-сервисов, но не знали некоторых тонкостей разработки в облачных окружениях или не имели четкого представления о том, что такое «облачные приложения», и впоследствии обнаруживали, что их сервисы сложно развертывать, ими сложно управлять или наблюдать за ними. Таким

[1] Включая проекты CNCF из категорий Sandbox, Incubating и Graduated, по состоянию на февраль 2021 года.

читателям эта книга не только поможет заложить прочный фундамент для создания собственных облачных сервисов, но также покажет, в чем преимущества этих методов, и представит конкретные примеры, способствующие пониманию этой довольно абстрактной темы.

Предполагается, что многие читатели хорошо знакомы с другими языками программирования, но их привлекает репутация Go как языка облачной разработки. Таким читателям эта книга предложит передовой опыт использования Go в качестве специализированного языка разработки для облачных окружений и поможет им решить собственные проблемы управления и развертывания облачных приложений.

Почему я написал эту книгу

Способы проектирования, конструирования и развертывания приложений меняются со временем. Требования к масштабированию вынуждают разработчиков размещать свои сервисы на десятках и сотнях серверов: отрасль постепенно становится «облачной». Но при этом возникает множество новых проблем: как разрабатывать, развертывать или управлять сервисом, действующим на десятках, сотнях или даже тысячах серверов? К сожалению, существующие книги об облачных вычислениях сосредоточены на абстрактных принципах проектирования и содержат лишь элементарные примеры, если вообще содержат. Эта книга призвана удовлетворить потребность в демонстрации практической реализации сложных принципов проектирования облачных вычислений.

Соглашения

В этой книге используются следующие соглашения по оформлению:

Курсив
: Используется для обозначения новых терминов, имен файлов и расширений.

`Моноширинный шрифт`
: Применяется для оформления листингов программ и программных элементов внутри обычного текста, таких как имена переменных и функций, баз данных, типов данных, переменных окружения, инструкций и ключевых слов.

`Моноширинный жирный`
: Обозначает команды или другой текст, который должен вводиться пользователем.

`Моноширинный курсив`
: Обозначает текст, который должен замещаться фактическими значениями, вводимыми пользователем или определяемыми из контекста.

 Так выделяются советы и предложения.

 Так обозначаются примечания общего характера.

 Так обозначаются предупреждения и предостережения.

Использование программного кода примеров

Вспомогательные материалы (примеры кода, упражнения и т. д.) доступны для загрузки по адресу: https://github.com/cloud-native-go/examples.

Данная книга призвана оказать вам помощь в решении ваших задач. В общем случае все примеры кода из этой книги вы можете использовать в своих программах и в документации. Вам не нужно обращаться в издательство за разрешением, если вы не собираетесь воспроизводить существенные части программного кода. Например, если вы разрабатываете программу и используете в ней несколько отрывков программного кода из книги, вам не нужно обращаться за разрешением. Однако в случае продажи или распространения примеров из этой книги вам необходимо получить разрешение от издательства O'Reilly. Если вы отвечаете на вопросы, цитируя данную книгу или примеры из нее, получение разрешения не требуется. Но при включении существенных объемов программного кода примеров из этой книги в вашу документацию вам необходимо будет получить разрешение издательства.

Мы приветствуем, но не требуем добавлять ссылку на первоисточник при цитировании. Под ссылкой на первоисточник мы подразумеваем указание авторов, издательства и ISBN. Например: «*Мэтью А. Титмус*. Облачный Go. М.: ДМК Пресс, 2021. 978-5-97060-965-1» или «*Cloud Native Go* by Matthew A. Titmus (O'Reilly). Copyright 2021 Matthew A. Titmus, 978-1-492-07633-9».

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

Благодарности

В первую очередь я хочу поблагодарить жену и сына. Вы – главная причина всех моих успехов, достигнутых после того, как вы вошли в мою жизнь. Вы – мои путеводные звезды, которые помогают мне не сбиться с пути и заставляют смотреть в небо.

Моему отцу, которого мы недавно потеряли. Ты был ближе всех к людям эпохи Возрождения и при этом умудрялся быть самым добрым и скромным человеком, которого я когда-либо знал. Я все еще хочу быть похожим на тебя, когда вырасту.

Мэри. Ты чувствуешь его отсутствие острее других. Мы – одна семья, и мы всегда будем семьей, даже если я буду звонить тебе не так часто, как следовало бы. Папа гордился бы твоей силой и грацией.

Саре. Меня всегда поражала твоя сила духа и стойкость. Твой острый ум сделал тебя моим самым верным союзником и самым жестким противником с тех самых пор, как только научились говорить. Не говори Натану, но ты моя любимая сестра.

Натану. Если каждый из нас унаследовал что-то одно от отца, то ты, безусловно, получил его сердце. Я нечасто говорю это, но я очень горжусь тобою и твоими достижениями. Не говори Саре, но ты мой любимый брат.

Маме. Ты сильная, умная, яркая и необычная. Спасибо, что научила меня всегда делать то, что действительно нужно делать, независимо от того, что думают люди. Оставайся необычной и не забывай кормить цыплят.

Альберту. У тебя огромное сердце и бездонное терпение. Спасибо, что присоединился к нашей семье; мы все выиграли от этого.

Всем другим членам нашей семьи. Я не могу видеться с вами так часто, как хотелось бы, и я очень скучаю по вам, но я всегда ощущаю вас рядом, когда вы мне нужны. Спасибо, что праздновали со мной победы и поддерживали меня в поражениях.

Уолту и Альваро, с которыми я не смогу расстаться, даже поменяв работу. Спасибо за вашу восторженную поддержку в моих начинаниях и за ваш абсолютный реализм. Вы оба делаете меня лучше. Кроме того, спасибо за то, что познакомили меня с серией книг «Gradle» Уилла Уайта (Will Wight) и за последовавшую за этим пагубную зависимость.

Моим друзьям «Jeff Classic», «New Jeff», Алексу (Alex), Маркану (Markan), Приянке (Priyanka), Сэму (Sam), Оуэну (Owen), Мэтту М., Мариусу (Matt M., Marius), Питеру (Peter), Рохиту (Rohit) и коллегам из Flatiron Health. Спасибо, что позволили мне отвлечься на эту книгу, и за поддержку, что выступили в качестве советчиков, первых читателей рукописи и критиков, а также за то, что воодушевили меня и были моими помощниками.

Всем моим друзьям из CoffeeOps в Нью-Йорке и во всем мире. Вы любезно позволили мне отразить ваши мысли и бросить вам вызов, а вы приняли этот вызов. Эта книга определенно выиграла от вашего участия.

Лиз Фон-Джонс (Liz Fong-Jones), известному эксперту в области наблюдений и оракулу. Ваши указания, замечания и образцы кода были неоценимы, и без вашей щедрости написать эту книгу было бы намного труднее, а результат был бы намного хуже.

Моим техническим обозревателям Ли Атчисону (Lee Atchison), Альваро Атьензе (Alvaro Atienza), Дэвиду Никпонски (David Nicponski), Натали Пистунович (Natalie Pistunovich) и Джеймсу Куигли (James Quigley). Спасибо за терпение, которое вы проявили, чтобы прочитать каждое написанное мной слово (даже сноски). Эта книга получилась намного лучше благодаря вашей зоркости и упорному труду.

Наконец, спасибо всей команде редакторов и художников O'Reilly Media, с которыми мне посчастливилось работать, особенно Амелии Блевинс (Amelia Blevins), Дэнни Эльфанбаум (Danny Elfanbaum) и Зану Маккуэйду (Zan McQuade). 2020 год был сложным, но ваши доброта, терпение и поддержка помогли мне пройти через него.

Часть I

ОБЛАЧНОЕ ОКРУЖЕНИЕ

Глава 1

Что такое «облачное» приложение?

> Самая опасная фраза в этом языке звучит так: «Мы всегда так поступали»[1].
> – Грейс Хоппер (Grace Hopper), *Computerworld* (январь 1976)

Если вы читаете эту книгу, значит, вы, по крайней мере, слышали раньше термин *облачный*. Вероятно, вы даже читали некоторые из множества статей, написанных восторженными авторами, соблазнившимися хрустом купюр. Если ваше знание термина ограничивается только этим опытом, то вас можно простить за то, что сочли его неоднозначным, модным и просто еще одним из ряда рекламных выражений, которые могли начинаться как что-то полезное, но затем были использованы людьми, пытающимися что-то вам продать, как, например, «гибкая разработка» или «DevOps».

По схожим причинам поиск в интернете по запросу «определение термина облачный» может привести вас к мысли, что любое приложение, предназначенное для работы в облаке, должно быть написано на «правильном» языке[2] или с использованием «правильного» фреймворка или «правильной» технологии. Конечно, выбор языка может значительно упростить или усложнить вашу жизнь, но этот выбор не является ни необходимым, ни достаточным для создания облачного приложения.

Облачность зависит лишь от того, *где* запускается приложение. Термин *облачный* явно предполагает это. Все, что от вас требуется, – это «залить» свое старое, сляпанное кое-как приложение в контейнер и запустить его под управлением Kubernetes, после чего оно автоматически станет облачным, верно? Нет, потому что в этом случае вы лишь усложнили развертывание и управление приложением[3].

Итак, *что такое* облачное приложение? В этой главе мы ответим на этот вопрос. Для начала мы познакомимся с историей парадигм вычислительных служб до (и особенно) настоящего времени и обсудим, как неумолимое тре-

---

[1] Surden, Esther. «Privacy Laws May Usher in Defensive DP: Hopper». *Computerworld*, 26 Jan. 1976, p. 9.

[2] Это Go. Не поймите меня неправильно, но, в конце концов, эта книга о Go.

[3] Вы когда-нибудь задумывались, почему так много миграций в Kubernetes терпят неудачу?

бование к масштабируемости стимулировало (и продолжает стимулировать) разработку и внедрение технологий, которые обеспечивают высокий уровень надежности. Наконец, мы определим конкретные атрибуты, характерные для таких приложений.

История развития до настоящего времени

История сетевых приложений – это история все усиливающегося требования масштабируемости.

В конце 1950-х появилась большая ЭВМ – мейнфрейм. В то время каждая программа и каждый фрагмент данных хранились в одной гигантской машине, к которой пользователи могли обращаться с помощью простых терминалов, не имевших собственных вычислительных возможностей. Вся логика и все данные жили вместе как одна большая счастливая семья. Это было простое время.

Все изменилось в 1980-х с появлением недорогих персональных компьютеров, подключаемых к сети. В отличие от простых терминалов, персональные компьютеры (ПК) могли выполнять некоторые вычисления самостоятельно, что позволяло переносить на них часть логики приложения. Эта новая многоуровневая архитектура, разделяющая логику представления, бизнес-логику и данные (рис. 1.1), сделала возможным изменение или замену компонентов сетевого приложения независимо друг от друга.

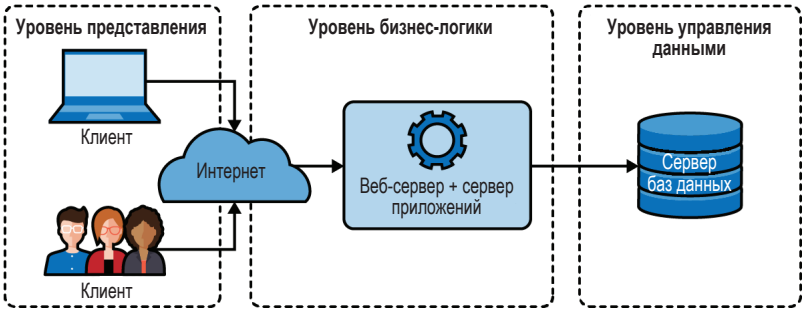


Рис. 1.1 ❖ Традиционная трехуровневая архитектура, в которой четко разделялись представление, бизнес-логика и данные

В 1990-х популяризация Всемирной паутины и последовавшая за ней «золотая лихорадка доткомов» породили технологию «программное обеспечение как услуга» (Software as a Service, SaaS). Целые отрасли были основаны на модели SaaS, что привело к созданию более сложных и ресурсоемких приложений, которые, в свою очередь, было труднее разрабатывать, поддерживать и развертывать. Внезапно классической многоуровневой архитектуры стало недостаточно. В результате бизнес-логика начала дробиться на подкомпоненты, которые можно было разрабатывать, поддерживать и развертывать независимо, и наступила эра микросервисов.

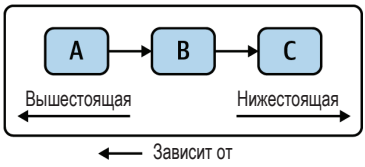
В 2006 году Amazon запустила облачную платформу Amazon Web Services (AWS), включавшую службу Elastic Compute Cloud (EC2). AWS была *не первым* предложением инфраструктуры как услуги (Infrastructure as a Service, IaaS), но именно она произвела революцию в отношении доступности хранилищ данных и вычислительных ресурсов, сделав облачные вычисления – и возможность быстрого масштабирования – доступными для масс, что ускорило массовую миграцию ресурсов в «облако».

К сожалению, организации вскоре поняли, что масштабирование – непростая задача. Проблемы неизбежны, и когда вы работаете с сотнями или тысячами ресурсов, проблемы возникают *очень часто*. Трафик может резко увеличиваться или уменьшаться, основное оборудование может выходить из строя, вышестоящие зависимости могут внезапно и необъяснимо стать недоступными. Но даже притом, что что-то может пойти не так, вам все равно придется развернуть все эти ресурсы и управлять ими. При таких масштабах люди не могут (или, по крайней мере, могут с большими затруднениями) справляться со всеми этими проблемами вручную.

Вышестоящие и нижестоящие зависимости

В этой книге я иногда буду использовать термины *вышестоящая зависимость* и *нижестоящая зависимость* для описания относительного положения двух ресурсов. В отрасли нет полного согласия относительно этих терминов, поэтому в данной книге я буду вкладывать в них следующий смысл.

Представьте, что у нас есть три службы: A, B и C, как показано на следующем рисунке:



В этом сценарии служба A посылает запросы службе B (и, следовательно, зависит от нее), которая, в свою очередь, зависит от службы C.

Поскольку служба B зависит от службы C, можно сказать, что служба C является *нижестоящей зависимостью* службы B. В более широком смысле, поскольку служба A зависит от службы B, которая зависит от службы C, служба C также является *транзитивной*[1] *нижестоящей зависимостью* службы A.

И наоборот, поскольку служба C используется службой B, можно сказать, что служба B является *вышестоящей зависимостью* службы C, а служба A – *транзитивной вышестоящей зависимостью* службы C.

[1] То есть непрямой, косвенной. – *Прим. перев.*

Что значит быть «облачным»?

Вообще говоря, по-настоящему облачное приложение включает в себя все, что мы узнали о масштабировании сетевых приложений за последние 60 лет. Они масштабируются в условиях резко меняющейся нагрузки, устойчивы в условиях неопределенности окружения и управляемы в условиях постоянно меняющихся требований. Иначе говоря, облачное приложение создано для жизни в жесткой и неопределенной вселенной.

Но как *определить* термин *облачный*? К счастью для всех нас[1], в этом нет необходимости. Cloud Native Computing Foundation (https://oreil.ly/621yd) – подразделение известного фонда Linux Foundation и в некотором роде признанный авторитет в этой области – уже сделал это за нас:

> Облачные технологии позволяют организациям создавать и запускать масштабируемые приложения в современных динамических окружениях, таких как общедоступные, частные и гибридные облака...
>
> Они делают слабосвязанные системы устойчивыми, управляемыми и наблюдаемыми. В сочетании с надежной автоматизацией они позволяют инженерам часто и предсказуемо вносить важные изменения с минимальными усилиями[2].
>
> – Cloud Native Computing Foundation, CNCF Cloud Native Definition v1.0

Согласно этому определению, облачные приложения – больше, чем просто приложения, которые действуют в облаке. Они также *масштабируемы*, *слабо связаны*, *устойчивы*, *управляемы* и *доступны для наблюдения*. Можно сказать, что наличие этих «облачных атрибутов» позволяет называть системы облачными.

Как оказалось, каждый из этих атрибутов имеет довольно специфическое значение, поэтому рассмотрим их поближе.

Масштабируемость

В контексте облачных вычислений *масштабируемость* можно определить как способность показывать ожидаемое поведение в условиях значительных колебаний спроса вверх и вниз. Систему можно считать масштабируемой, если ее не нужно реорганизовывать для решения намеченной задачи во время или после резкого увеличения спроса.

Немасштабируемые службы могут отлично функционировать в начальных условиях, поэтому масштабируемость не всегда является первоочередной задачей при проектировании. Однако службы, не способные масштабироваться за пределы первоначальных ожиданий, имеют ограниченный срок жизни. Более того, реорганизовать службу для поддержки масштабируемости зачастую чрезвычайно сложно, поэтому проектирование с прицелом на

---

[1] И особенно для меня, пишущего эту классную книгу.

[2] Cloud Native Computing Foundation. «CNCF Cloud Native Definition v1.0», GitHub, 7 Dec. 2020. https://oreil.ly/KJuTr.

такую поддержку может сэкономить время и деньги в долгосрочной перспективе.

Существует два способа масштабирования, каждый из которых имеет свои плюсы и минусы.

Вертикальное масштабирование

Под *вертикальным масштабированием* подразумевается увеличение (или уменьшение) аппаратных ресурсов, доступных системе. Например, можно выделить дополнительную память или процессоры базе данных, которая работает на выделенном экземпляре. Преимущество вертикального масштабирования – в технической простоте реализации, но ресурсы любого конкретного экземпляра невозможно наращивать до бесконечности.

Горизонтальное масштабирование

Под *горизонтальным масштабированием* подразумевается увеличение (или уменьшение) количества действующих экземпляров службы. Например, можно увеличить количество узлов за балансировщиком нагрузки или контейнеров в Kubernetes либо в другой системе управления контейнерами. Эта стратегия имеет свои преимущества, включая избыточность и свободу от ограничений размеров экземпляров. Однако чем больше экземпляров, тем сложнее проектирование и управление, а кроме того, не все службы можно масштабировать по горизонтали.

Итак, у нас есть два способа масштабирования – по вертикали и по горизонтали. Но если служба поддерживает вертикальное масштабирование аппаратных ресурсов (и способна извлечь выгоду из этого), то можно ли назвать ее «масштабируемой»? И насколько она масштабируема? Вертикальное масштабирование по своей природе ограничено объемом доступных вычислительных ресурсов, поэтому служба, которую можно масштабировать только по вертикали, вообще не очень масштабируема. Если может понадобиться масштабирование в десять, сто или тысячу раз, то служба должна поддерживать горизонтальное масштабирование.

Так в чем же разница между службами, поддерживающими и не поддерживающими горизонтальное масштабирование? Все сводится к одному: состояние. Службу, которая не поддерживает состояния или была спроектирована для распределения своего состояния между экземплярами, будет довольно легко масштабировать. Для любого другого приложения это будет сложно.

Более подробно идеи масштабируемости, состояния и избыточности будут рассмотрены в главе 7.

Слабая связанность

Слабая связанность – это свойство системы и стратегия проектирования, согласно которой компоненты системы знают лишь самый минимум о любых других компонентах. Можно сказать, что две системы *слабо связаны*, если изменение одного компонента не требует изменения другого.

Например, веб-серверы и веб-браузеры можно считать слабо связанными: серверы можно обновлять и даже полностью заменять, не опасаясь влияния на наши браузеры. Это возможно потому, что стандартные веб-серверы обмениваются данными с использованием набора стандартных протоколов[1]. Иначе говоря, они предоставляют *контракт на обслуживание*. Представьте, какой был бы хаос, если бы все веб-браузеры в мире приходилось обновлять после выхода новой версии NGINX или httpd[2]!

Можно сказать, что «слабая связанность» – это один из базовых принципов архитектуры микросервисов: разделение компонентов таким образом, чтобы изменения в одном не влияли на другой. Однако этим принципом часто пренебрегают, и его стоит повторить. Преимущества слабой связанности – и последствия пренебрежения ею – нельзя недооценивать. Очень легко создать систему, «худшую из возможных», которая сочетает в себе накладные расходы на управление и сложность, связанные с наличием нескольких служб, с зависимостями и связями монолитной системы: жуткий *распределенный монолит*.

К сожалению, не существует волшебной технологии или протокола, которые могли бы предотвратить тесную связанность ваших служб. Любой формат обмена данными может использоваться неправильно. Однако есть несколько приемов, помогающих добиться желаемого и – в сочетании с такими практиками, как декларативные API и методы управления версиями, – создавать слабо связанные службы, которые могут изменяться независимо друг от друга.

Эти приемы и практики будут подробно обсуждаться и демонстрироваться в главе 8.

Устойчивость

Устойчивость (близкий синоним к *отказоустойчивости*) – это мера способности системы восстанавливаться после ошибок и сбоев. Систему можно считать *устойчивой*, если она может продолжать работать правильно – возможно, с меньшей эффективностью – вместо полного отказа после выхода из строя какой-либо ее части.

При обсуждении устойчивости (а также других «облачных атрибутов», но особенно при обсуждении устойчивости) мы довольно часто используем слово «система». Термин *система*, в зависимости от контекста, может относиться к чему угодно, от сложной сети взаимосвязанных служб (например, целого распределенного приложения) до набора тесно связанных компонентов (таких как реплики одной функции или экземпляра службы), и даже к единственному процессу, выполняющемуся на единственной машине. Всякая система состоит из нескольких подсистем, которые, в свою очередь, состоят из более мелких подсистем, которые сами состоят из еще более мелких подсистем. И так до самого конца.

---

[1] Те, кто помнит браузерные войны 1990-х годов, знают, что так было не всегда.

[2] Или если бы для каждого веб-сайта нужно было использовать другой браузер. Это было бы крайне неудобно, не так ли?

Выражаясь языком системотехники, любая система может содержать дефекты, или *нарушения*, которые мы, программисты, любовно называем *жучками* (*bugs*). Мы все слишком хорошо знаем, что при определенных условиях любая неисправность может привести к *ошибке* – так мы называем любое несоответствие между предполагаемым и фактическим поведением системы. Ошибки могут привести к тому, что система не сможет выполнить свою функцию, то есть *откажет*. Но это еще не все: отказ в подсистеме или компоненте приводит к нарушению работы более крупной системы; любое нарушение, не устраненное должным образом, может вызывать нарушения во все более вышестоящих подсистемах и, в конце концов, привести к полному отказу системы.

В идеальном мире каждая система должна тщательно проектироваться, чтобы предотвратить появление нарушений, но в реальной жизни это невозможно. Невозможно предотвратить все возможные нарушения – пытаться сделать это бессмысленно и непродуктивно. Однако если предположить, что все компоненты системы могут выходить из строя – а это так и есть, – и спроектировать их так, чтобы они правильно реагировали на возможные нарушения и ограничивали распространение их последствий, то можно создать систему, которая продолжает исправно функционировать, даже если некоторые из ее компонентов выйдут из строя.

Существует множество подходов к проектированию отказоустойчивых систем. Наиболее распространенным из них является, пожалуй, развертывание избыточных компонентов, но при этом также предполагается, что нарушение не повлияет на другие компоненты того же типа. Можно добавить автоматическое размыкание цепи и логику повтора, чтобы предотвратить распространение нарушений между компонентами. Неисправные компоненты можно даже вывести из строя намеренно, чтобы принести пользу всей системе.

Мы обсудим все эти подходы (и многие другие) в главе 9.

Устойчивость – это не безотказность

Термины *устойчивость* и *безотказность* описывают похожие понятия, которые часто путают. Но, как мы обсудим в главе 9, это не совсем одно и то же[1]:

- устойчивость системы – это степень, в которой она может продолжать правильно функционировать, столкнувшись с ошибками и неполадками. Устойчивость, наряду с другими четырьмя облачными свойствами, является лишь одним из факторов, влияющих на надежность;
- безотказность системы – это ее способность сохранять ожидаемое поведение в течение заданного интервала времени. Безотказность в сочетании с такими атрибутами, как доступность и ремонтопригодность, способствует общей надежности системы.

[1] Если вам интересно узнать академическую трактовку, то я настоятельно рекомендую книгу Кишора С. Триведи (Kishor S. Trivedi) и Андреа Боббио (Andrea Bobbio) «Reliability and Availability Engineering» (https://oreil.ly/80wGT).

Управляемость

Управляемость системы – это простота (или ее отсутствие), с которой можно изменить поведение системы для обеспечения безопасности, бесперебойной работы и соответствия меняющимся требованиям. Систему можно считать *управляемой*, если она позволяет изменить ее поведение без изменения кода.

Управляемость как свойство системы привлекает гораздо меньше внимания, чем другие атрибуты, такие как масштабируемость или наблюдаемость. Однако она играет не менее важную роль, особенно в сложных распределенных системах.

Например, представьте гипотетическую систему, которая включает службу и базу данных, и служба обращается к базе данных по URL. Что, если вам потребуется изменить эту службу так, чтобы она обращалась к другой базе данных? Если URL жестко запрограммирован, то вам может понадобиться изменить код и повторно развернуть систему, что иногда может быть неудобно по некоторым причинам. Конечно, можно обновить запись в системе DNS, чтобы она возвращала другой IP-адрес для заданного URL, но как быть, если вам потребуется повторно развернуть разрабатываемую версию службы, которая будет ссылаться на базу данных в среде разработки?

Управляемая система может, например, извлекать требуемый URL из легко изменяемой переменной окружения; если служба, которая ее использует, развернута в Kubernetes, то для корректировки ее поведения достаточно будет обновить значение в конфигурации. Более сложная система может даже предоставлять декларативный API, с помощью которого разработчик сможет сообщить системе, какого поведения он ожидает. Нет единственного правильного ответа[1].

Управляемость не ограничивается поддержкой изменений в конфигурации. Она охватывает все возможные аспекты поведения системы, будь то активация функций или ротация учетных данных и сертификатов TLS, или даже (и, возможно, особенно) развертывание либо обновление компонентов системы.

Управляемые системы предполагают адаптируемость и могут легко приспосабливаться к изменяющимся функциональным требованиям, а также к требованиям окружения или безопасности. С другой стороны, неуправляемые системы, как правило, более хрупкие, часто требуют специальных изменений, нередко выполняемых вручную. Накладные расходы, связанные с управлением такими системами, вводят фундаментальные ограничения на их масштабируемость, доступность и надежность.

Идея управляемости и некоторые предпочтительные практики ее реализации в Go будут обсуждаться в главе 10.

---

[1] И есть много неправильных.

Управляемость – это не удобство сопровождения

Можно сказать, что управляемость и удобство сопровождения (ремонтопригодность) в чем-то «перекликаются», потому что оба понятия связаны с простотой изменения системы[1], но на самом деле они совершенно разные:

- управляемость описывает простоту изменения поведения работающей системы, вплоть до развертывания (и повторного развертывания) ее компонентов. Управляемость – это простота внесения изменений *извне*;
- удобство сопровождения описывает простоту изменения базовых функций системы, чаще всего ее кода. Удобство сопровождения – это простота внесения изменений *изнутри*.

Наблюдаемость

Наблюдаемость системы – это мера простоты определения ее внутреннего состояния по наблюдаемым результатам. Система считается *наблюдаемой*, если можно быстро и последовательно получать ответы на все новые вопросы о ней с минимальными предварительными знаниями, без необходимости внедряться в существующий код или писать новый.

На первый взгляд в этом нет ничего сложного: достаточно добавить журналирование, включить пару панелей мониторинга (дашбордов) – и система станет наблюдаемой, не так ли? Почти наверняка нет, особенно в современных сложных системах, где почти любая проблема так или иначе связана с сетью, в которой одновременно может обнаружиться несколько проблем. Эпоха стека LAMP закончилась; сейчас дело обстоит намного сложнее.

Это не означает, что метрики, журналы и трассировка стали неважны. Напротив, они так и остались основными строительными блоками наблюдаемости. Но одного их существования недостаточно: данные – это не информация. Их важно правильно интерпретировать. Они должны иметься в достаточном количестве. Они должны подсказывать ответы на вопросы, о которых вы даже не думали раньше.

Способность обнаруживать и отлаживать проблемы является фундаментальным требованием для сопровождения и развития надежной системы. Но в распределенной системе бывает сложно выяснить причины проблемы. Сложные системы слишком… сложны. Количество возможных состояний отказа в любой системе пропорционально произведению количества возможных состояний частичного и полного отказа каждого из ее компонентов, и невозможно предсказать их все заранее. Традиционного подхода, заключающегося в фокусировке внимания на вещах, которые по нашему мнению могут потерпеть сбой, недостаточно.

Появление новых методов наблюдения можно рассматривать как процесс эволюции мониторинга. Многолетний опыт проектирования, создания и сопровождения сложных систем научил нас, что традиционные методы

[1] К тому же оба термина начинаются на «У». Очень запутанно.

инструментирования, включая, помимо всего прочего, панели мониторинга, журналы или оповещения о различных «известных неизвестных», просто не справляются с проблемами, создаваемыми современными распределенными системами.

Наблюдаемость – сложная и тонкая тема, но, по сути, она сводится к следующему: подготовить систему к реальным условиям настолько хорошо, чтобы в будущем можно было отвечать на вопросы, о которых вы пока не задумывались.

Идея наблюдаемости и некоторые предложения по ее реализации будут обсуждаться в главе 11.

Что особенного в облачном окружении?

Миграция в облако является примером архитектурной и технической адаптации, движимой давлением окружающей среды. Это эволюция – выживание сильнейшего. Имейте в виду, что по образованию я – биолог.

В стародавние времена, когда все только начиналось[1], приложения создавались и развертывались (обычно вручную) на одном или нескольких серверах, где обслуживались и поддерживались со всем тщанием. Если они заболевали, то их заботливо лечили. Если служба выходила из строя, то ее исправляли простым перезапуском. Наблюдаемость заключалась во входе в командную оболочку сервера и просмотре журналов. В те времена все было проще.

В 1997 году только 11 % людей в промышленно развитых странах и 2 % во всем мире пользовались интернетом. Однако в последующие годы наблюдался экспоненциальный рост подключений к интернету, и к 2017 году это число выросло до 81 % в промышленно развитых странах и 48 % во всем мире[2] и продолжает расти.

Все эти пользователи – и их деньги – создавали давление на службы, требуя масштабирования. Более того, по мере роста сложности требований пользователей и их зависимости от веб-сервисов росли и ожидания, что их любимые веб-приложения будут многофункциональными и всегда доступными.

Результатом стало значительное эволюционное давление в сторону масштабирования, сложности и надежности. Однако эти три атрибута плохо сочетаются друг с другом, и традиционные подходы просто не могли и не могут угнаться за ними. Пришлось изобретать новые приемы и методы.

К счастью, с появлением общедоступных облаков и IaaS возможность масштабирования инфраструктуры существенно упростилась. Недостатки надежности часто можно было компенсировать количеством. Но это создало новые проблемы. Как обслуживать сотни, тысячи или даже десятки тысяч

---

[1] Это было в 1990-х.

[2] International Telecommunication Union (ITU). «Internet users per 100 inhabitants 1997 to 2007» и «Internet users per 100 inhabitants 2005 to 2017». *ICT Data and Statistics* (*IDS*).

серверов? Как установить на них свое приложение или обновлять его? Как отлаживать возникающие в нем проблемы? Как вообще узнать – здорово ли оно? Проблемы в небольшом масштабе, вызывающие лишь легкое раздражение, становятся очень сложными с увеличением масштаба.

Облачные технологии важны, потому что масштабирование является причиной (и решением) всех наших проблем. Это не волшебство. Здесь нет ничего особенного. Если отбросить причудливую терминологию, то облачные методы и технологии существуют только для того, чтобы дать возможность использовать преимущества «облака» (количество) и компенсировать его недостатки (отсутствие надежности).

Итоги

В этой главе мы познакомились с историей компьютерных вычислений и узнали, что то, что теперь мы называем «облачным окружением», не является чем-то новым – это неизбежный результат цикла технологического развития, стимулирующего инновации.

Однако все эти причудливые термины, с которыми мы познакомились, сводятся к одному: современные приложения должны надежно служить большому количеству людей. Методы и технологии, которые мы называем «облачными», – это лучшие современные практики создания масштабируемых, адаптируемых и достаточно устойчивых служб.

Но какое отношение все это имеет к языку Go? Как оказывается, для облачного окружения нужны свои облачные инструменты. В главе 2 мы начнем говорить о том, что это означает.

Глава 2

Почему Go правит облачным миром

> Каждый смышленый дурак сможет сделать любой предмет больше, сложнее и мощнее. Развитие в другом направлении требует присутствия гениальности и огромного мужества.
>
> – Э. Ф. Шумахер (E. F. Schumacher), *Small Is Beautiful*[1] (август 1973 г.)

Как появился Go

Идея создания языка Go возникла в Google в сентябре 2007 года, и это был неизбежный результат того, что нескольких умных парней заперли в одной комнате и чертовски расстроили их.

Речь идет о Роберте Гриземере (Robert Griesemer), Робе Пайке (Rob Pike) и Кене Томпсоне (Ken Thompson); все они уже имели богатый опыт разработки других языков. Источником их расстройства стал тот факт, что ни один из языков программирования, доступных в то время, просто не подходил для описания видов распределенных, масштабируемых, устойчивых служб, создававшихся в Google[2].

По сути, все основные языки программирования того времени были разработаны в другую эпоху, еще до того, как многопроцессорные системы стали обычным явлением, а сети – повсеместными. Поддержка многопроцессорной обработки и сетей – основных строительных блоков современных «облачных» служб[3] – была ограничена или требовала чрезвычайных усилий для использования. Проще говоря, языки программирования не соответствовали потребностям разработки современного программного обеспечения.

---

[1] *Шумахер Э. Ф.* Малое прекрасно. Экономика, в которой люди имеют значение. М.: Изд. дом Высшей школы экономики, 2012. ISBN 978-5-7598-0822-0. – *Прим. перев.*

[2] Это были «облачные» службы, созданные еще до появления термина «облачные».

[3] Конечно, в то время они не назывались «облачными»; для Google они были просто «службами».

Особенности облачного мира

Разочаровывающих факторов было много, но все они сводились к чрезмерной сложности языков, с которыми они работали, затруднявшей создание серверного программного обеспечения, в том числе[1]:

Замысловатость программ
: Код было трудно читать. Излишние проверки и повторяющиеся фрагменты усугублялись функционально перекрывающимися особенностями, которые помогали оттачивать ум, но не придавали ясности.

Медленная сборка
: Конструкция языка и многие годы постепенного развития привели к тому, что сборка приложений могла длиться часами, даже на больших кластерах.

Неэффективность
: Многие программисты отреагировали на вышеупомянутые проблемы, взяв на вооружение более гибкие и динамичные языки, фактически поменяв эффективность и безопасность типов на выразительность.

Высокая стоимость обновлений
: Несовместимость даже между младшими версиями языка, а также любые зависимости, которые он может иметь (и транзитивные зависимости!), часто вызывали большие сложности при обновлении.

За прошедшие годы было предложено множество – часто довольно остроумных – решений для преодоления некоторых из этих проблем, обычно вносящих дополнительные сложности в процесс. Ясно, что их нельзя исправить с помощью нового API или особенности языка. Поэтому проектировщики Go представили новый современный язык, созданный специально для облачных вычислений, который поддерживает сетевые и многопроцессорные вычисления и обладает большой выразительностью, но при этом понятный и позволяющий пользователям сосредоточиться на решении своих задач, а не на борьбе с языком.

В результате получился язык Go, отличающийся явными и неявными особенностями. Некоторые из этих особенностей (явные и неявные) и их мотивация обсуждаются в следующих разделах.

Композиция и структурная типизация

Объектно-ориентированное программирование, основанное на понятии «объектов» различных «типов», обладающих различными атрибутами, существует с 1960-х годов, но по-настоящему вошло в моду в начале-середине 1990-х с появлением Java и добавлением объектно-ориентированных воз-

[1] Pike, Rob. «Go at Google: Language Design in the Service of Software Engineering». Google, Inc., 2012. https://oreil.ly/6V9T1.

можностей в C++. С тех пор этот стиль превратился в доминирующую парадигму программирования и остается таковой по сей день.

Перспективы объектно-ориентированного программирования соблазнительны, а теория, лежащая в основе, даже имеет определенный интуитивный смысл. Данные и поведение могут быть связаны с *типами* вещей и наследоваться *подтипами* этих вещей. *Экземпляры* этих типов можно представить как материальные объекты со свойствами и поведением – компоненты более крупной системы, моделирующей конкретные понятия реального мира.

Однако на практике объектно-ориентированное программирование с наследованием часто требует тщательной проработки отношений между типами, а также точного соблюдения определенных шаблонов и практик проектирования. Таким образом, как показано на рис. 2.1, в объектно-ориентированном программировании наблюдается тенденция к смещению фокуса от разработки алгоритмов к разработке и поддержке классификации и объектных моделей.

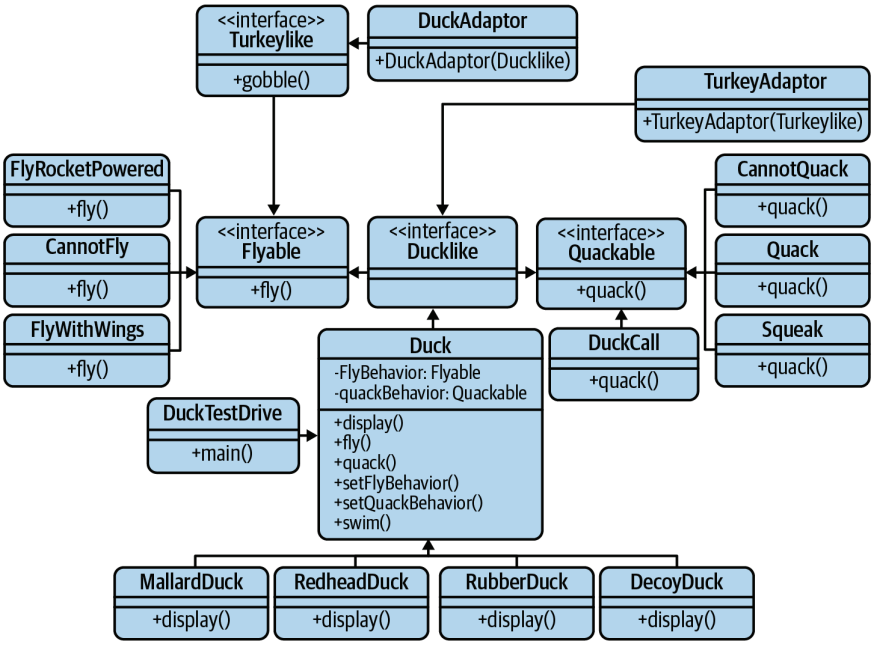


Рис. 2.1 ❖ Со временем объектно-ориентированное программирование стало все больше тяготеть к классификации

Это не означает, что Go не обладает объектно-ориентированными возможностями, которые поддерживают полиморфное поведение и повторное использование кода. В нем тоже есть понятие типов в виде *структур*, которые могут иметь свойства и поведение. Но он отвергает наследование и связанную с ним сложность оформления отношений, отдавая предпочтение сборке сложных типов путем *встраивания* более простых: этот подход известен как *композиция*.

В частности, там, где наследование крутится вокруг расширения отношения «является» между классами (например, автомобиль «является» транспортным средством с мотором), композиция позволяет создавать типы с использованием отношений «имеет», определяющих, что тот или иной тип может делать (например, автомобиль «имеет» мотор). На практике это обеспечивает большую гибкость проектирования, позволяя создавать код, менее подверженный проблемам из-за причуд «членов семьи».

В более широком смысле: в Go есть интерфейсы для описания поведенческих контрактов, но в нем нет понятия «является», поэтому эквивалентность экземпляров определяется путем изучения определения типа, а не его происхождения. Например, пусть есть интерфейс `Shape`, который определяет метод `Area`, тогда любой тип с методом `Area` будет неявно удовлетворять интерфейсу `Shape` и вам не потребуется явно объявлять его как `Shape`:

```
type Shape interface {              // Любой экземпляр Shape
    Area() float64                  // должен иметь метод Area
}

type Rectangle struct {             // Rectangle не заявляет явно
    width, height float64           // о поддержке Shape
}

func (Rectangle r) Area() float64 { // Rectangle имеет метод Area
    return r.width * r.height       // и удовлетворяет интерфейсу Shape
}
```

Этот механизм *структурной типизации*, который во время компиляции описывается как *утиная типизация*[1], в значительной степени избавляет от обременительного обслуживания классификации, от которой страдают традиционные объектно-ориентированные языки, такие как Java и C++, что позволяет программистам сосредоточиться на структурах данных и алгоритмах.

Понятность

Такие языки, как C++ и Java, часто критикуют за неуклюжесть, неудобство и излишнюю многословность. Они требуют большого количества шаблонного кода и тщательной проверки результатов, обременяя проекты избыточным кодом, который мешает программистам, отвлекая их внимание от решаемой задачи и ограничивая масштабируемость проектов под тяжестью итоговой сложности.

Go проектировался с прицелом на разработку больших проектов большим количеством программистов. Его минималистский дизайн (всего 25 ключевых слов и 1 вид циклов) и бескомпромиссный компилятор решительно отдают

[1] В языках, использующих утиную типизацию, тип объекта менее важен, чем методы, которые он определяет. То есть «если он ходит как утка и крякает как утка, значит, это утка».

предпочтение ясности перед остроумием[1]. Это, в свою очередь, способствует простоте кода и продуктивности программистов. Получившийся код легко читать, проверять и поддерживать, и в нем гораздо сложнее допустить ошибку.

Модель взаимодействия последовательных процессов

Большинство основных языков поддерживают возможность одновременного запуска нескольких конкурирующих процессов, позволяя составлять программы из процессов, выполняемых независимо. При правильном использовании конкуренция может быть невероятно полезным инструментом, но она также создает ряд проблем, особенно это касается упорядочивания событий, взаимодействий между процессами и координации доступа к общим ресурсам.

Программист неизбежно сталкивается с этими проблемами, позволяя процессам совместно использовать некоторую область памяти и затем с помощью блокировок и мьютексов упорядочивать доступ к ней, чтобы в каждый момент времени работать с этой областью мог только один процесс. Но даже при правильной реализации эта стратегия может привести к значительным накладным расходам на выполнение проверок. Также легко забыть заблокировать или разблокировать общую память, что может привести к появлению состояния гонки, взаимоблокировки или одновременному изменению одного и того же значения разными процессами. Этот класс ошибок чертовски труден для отладки.

Go, напротив, отдает предпочтение другой стратегии, основанной на формальном языке под названием «модель взаимодействия последовательных процессов» (Communicating Sequential Processes, CSP), впервые описанном Тони Хоаром (Tony Hoare) в статье с тем же названием[2], где иллюстрируются шаблоны взаимодействий в конкурентных системах с использованием каналов для передачи сообщений.

Получившаяся модель конкуренции, реализованная в Go в виде языковых примитивов, таких как *сопрограммы* и *каналы*, делает Go уникальным[3] и способным элегантно структурировать конкурентные вычисления вообще без использования блокировок. Это побуждает разработчиков ограничивать совместное использование памяти и разрешать процессам взаимодействовать друг с другом исключительно путем передачи сообщений. Эту идею часто выражают в виде афоризма:

> Не общайтесь, разделяя память. Разделяйте память, общаясь.
>
> – Афоризм Go

---

[1] Cheney, Dave. «Clear Is Better than Clever». *The Acme of Foolishness*, 19 July 2019. https://oreil.ly/vJs0X.

[2] Hoare, C. A. R. «Communicating Sequential Processes». *Communications of the ACM*, vol. 21, no. 8, Aug. 1978, pp. 666–77. https://oreil.ly/CHiLt.

[3] По крайней мере, среди «основных» языков, что бы это ни значило.

Конкуренция – это не параллелизм

Конкуренцию и параллелизм часто путают, что вполне понятно, учитывая, что оба понятия описывают выполнение нескольких процессов в течение некоторого периода времени. Однако это определенно не одно и то же[1]:

- *параллелизм* описывает одновременное выполнение нескольких независимых процессов;
- *конкуренция* описывает комплекс независимо выполняющихся процессов, но ничего не говорит о том, когда эти процессы будут выполняться.

Быстрая сборка

Одним из основных мотивов для создания языка Go было безумно долгое время сборки программ на некоторых языках того времени[2], на которую даже в больших кластерах Google часто уходило от нескольких минут до нескольких часов. Это отнимает время у разработчика и снижает его продуктивность. Учитывая, что основная цель Go заключалась в повышении продуктивности, а не в снижении, долгие сборки должны были уйти в прошлое.

Обсуждение особенностей компилятора Go выходит за рамки этой книги (и моей компетенции). Тем не менее вкратце отмечу, что язык Go проектировался с прицелом предоставить модель сборки программного обеспечения, свободную от сложных взаимосвязей, что значительно упрощает анализ зависимостей и устраняет необходимость в подключаемых файлах и библиотеках в стиле языка C, а также устраняет накладные расходы, связанные с ними. В результате сборка подавляющего большинства программного обеспечения на Go завершается за секунды, редко за минуты, даже на относительно скромном оборудовании. Например, для компиляции всех 1,8 миллиона строк[3] реализации Go в Kubernetes v1.20.2 на MacBook Pro с 8-ядерным процессором Intel i9 2,4 ГГц и 32 Гбайт ОЗУ потребовалось всего около 45 секунд:

```
mtitmus:~/workspace/kubernetes[MASTER]$ time make

real    0m45.309s
user    1m39.609s
sys     0m43.559s
```

Конечно же не обошлось без компромиссов. Любое предлагаемое изменение языка Go оценивается также с учетом его вероятного влияния на время сборки; некоторые многообещающие предложения были отклонены на том основании, что они увеличат это время.

---

1 Gerrand, Andrew. «Concurrency Is Not Parallelism». *The Go Blog*, 16 Jan. 2016. https://oreil.ly/WXf4g.

2 C++. Я говорю о C++.

3 Не считая комментариев; Openhub.net. «Kubernetes». *Open Hub*, Black Duck Software, Inc., 18 Jan. 2021. https://oreil.ly/y5Rty.

Стабильность языка

Go 1 был выпущен в марте 2012-го, включая спецификацию языка и спецификацию набора основных библиотек. Естественным следствием этого явилось явное обещание команды разработчиков Go пользователям Go, что программы, написанные на Go 1, будут продолжать компилироваться и работать правильно, без изменений, в течение всего срока действия спецификации Go 1. То есть можно было ожидать, что программы на Go, работающие сегодня, будут продолжать работать и в будущих выпусках Go 1 (Go 1.1, Go 1.2 и т. д.)[1].

Это резко контрастирует со многими другими языками, в которые иногда добавляются новые возможности, усложняющие сам язык и все, что на нем написано, и приводит к тому, что некогда элегантный язык превращается в обширную среду, богатую возможностями, которую зачастую чрезвычайно трудно освоить[2].

Команда разработчиков Go считает этот исключительный уровень стабильности языка жизненно важной особенностью; она позволяет пользователям доверять Go и полагаться на него, использовать имеющиеся и создавать новые библиотеки с минимальными усилиями и значительно снижает стоимость обновлений, особенно крупных проектов. Важно отметить, что высокая стабильность позволяет также сообществу учить Go и использовать его; писать на языке, а не его.

Это не означает, что Go не будет развиваться: и сам язык, и его библиотеки безусловно *будут* расширяться новыми пакетами и возможностями[3], и к настоящему времени накопилось уже много предложений по расширению[4], но без нарушения работоспособности существующего кода на Go 1.

При этом вполне возможно[5], что Go 2 никогда не появится. Скорее всего, совместимость с Go 1 будет сохраняться бесконечно; и в том маловероятном случае, если будет внесено критическое изменение, разработчики Go создадут утилиту преобразования, подобную команде `go fix`, которая использовалась при переходе на Go 1.

Безопасность памяти

Разработчики Go приложили огромные усилия, чтобы избавиться от различных ошибок и уязвимостей, не говоря уже о необходимости писать шаблонный код, связанных с прямым доступом к памяти. Указатели строго ти-

---

1 Разработчики Go. «Go 1 and the Future of Go Programs». *The Go Documentation*. https://oreil.ly/Mqn0I.

2 Кто-нибудь помнит Java 1.1? Я помню. Да, у нас не было ни дженериков, ни автоматического преобразования элементарных типов, ни расширенных циклов, но мы были счастливы. Счастливы, это я вам ответственно заявляю.

3 Я вхожу в команду, занимающуюся разработкой дженериков. Но Go против параметрического полиморфизма!

4 Разработчики Go. «Proposing Changes to Go». *GitHub*, 7 Aug. 2019. https://oreil.ly/folYF.

5 Pike, Rob. «Sydney Golang Meetup – Rob Pike – Go 2 Draft Specifications» (видео). *YouTube*, 13 Nov. 2018. https://oreil.ly/YmMAd.

пизированы и всегда инициализируются некоторым значением (даже если это значение `nil`), а арифметика указателей явно запрещена. Встроенные ссылочные типы, такие как ассоциативные массивы и каналы, которые внутренне представлены указателями на изменяемые структуры, инициализируются функцией `make`. Проще говоря, Go не требует и не допускает ручного управления памятью и манипуляций, которых поддерживают и требуют низкоуровневые языки, такие как C и C++, и полученный в результате выигрыш в отношении сложности и безопасности памяти невозможно переоценить.

Тот факт, что Go – это язык со сборкой мусора, избавляет программиста от необходимости тщательно следить за своевременным освобождением каждого выделенного байта и снимает с его плеч значительное бремя рутины. Жизнь без `malloc` дает свободу.

Более того, отказавшись от управления памятью вручную – даже от арифметики указателей, – разработчики Go сделали его практически не восприимчивым к целому классу ошибок, возникающих при работе с памятью и проблемам безопасности, которые они могут привнести. Никаких утечек памяти, никаких переполнений буфера, никакой рандомизации адресного пространства. Ничего.

Конечно, такая простота невозможна без определенных компромиссов, и, несмотря на невероятную интеллектуальность, сборщик мусора в Go все же вносит некоторые накладные расходы. По этой причине Go не может конкурировать с такими языками, как C++ и Rust, по скорости выполнения в чистом виде. Тем не менее, как будет показано в следующем разделе, Go занимает далеко не последнее место в этом состязании.

Производительность

Столкнувшись с необходимостью писать шаблонный код и медленной сборкой в статически типизированных компилируемых языках, таких как C++ и Java, многие программисты перешли на более динамичные и гибкие языки, такие как Python. Эти языки превосходно справляются со многими задачами, но они очень неэффективны по сравнению с компилирующими языками, такими как Go, C++ и Java.

Это можно заметить по результатам тестирования в табл. 2.1. Конечно, не следует слепо доверять тестам, но некоторые результаты особенно поразительны.

На первый взгляд, результаты можно сгруппировать в три категории, соответствующие типам языков:

- компилирующие, строго типизированные языки с ручным управлением памятью (C++, Rust);
- компилирующие, строго типизированные языки со сборкой мусора (Go, Java);
- интерпретирующие языки с динамической типизацией (Python, Ruby).

Эти результаты показывают, что языки со сборкой мусора, как правило, немного уступают в производительности языкам с ручным управлением

памятью, но различия не выглядят значительными, за исключением самых строгих требований.

Таблица 2.1. Тесты быстродействия некоторых распространенных языков программирования (время в секундах)[1]

1	C++	Go	Java	NodeJS	Python3	Ruby	Rust
Fannkuch-Redux	8.08	8.28	11.00	11.89	367.49	1255.50	7.28
FASTA	0.78	1.20	1.20	2.02	39.10	31.29	0.74
K-Nucleotide	1.95	8.29	5.00	15.48	46.37	72.19	2.76
Mandlebrot	0.84	3.75	4.11	4.03	172.58	259.25	0.93
N-Body	4.09	6.38	6.75	8.36	586.17	253.50	3.31
Spectral norm	0.72	1.43	4.09	1.84	118.40	113.92	0.71

Однако различия между интерпретирующими и компилирующими языками разительны. По крайней мере, в этих примерах Python – типичный представитель динамических языков – выполняет тесты примерно *в десять или в сто раз медленнее*, чем большинство компилирующих языков. Конечно, для многих (если не для большинства) задач и такого быстродействия более чем достаточно, но к облачным приложениям это относится в самой меньшей степени, потому что им часто приходится выдерживать значительные всплески спроса, и при этом было бы желательно обойтись без потенциально дорогостоящего вертикального масштабирования.

Статическая компоновка

По умолчанию программы на Go компилируются непосредственно в статически скомпонованные выполняемые файлы, куда копируются все необходимые библиотеки и сама среда выполнения. По этой причине файлы получаются довольно большими (для простейшей программы «привет мир» создается выполняемый файл с размером порядка 2 Мбайт), зато они не требуют установки внешней среды выполнения[2] или внешних библиотек, не подвержены конфликтам с имеющимся окружением[3] и могут распространяться между пользователями или развертываться на хосте без риска повредить или создать конфликты в существующем окружении.

Это особенно удобно при работе с контейнерами. Поскольку двоичные файлы Go не требуют внешней среды выполнения или даже дистрибутива, они могут встраиваться в «рабочие» образы, у которых нет родительских образов. В результате получается очень маленький (единицы мегабайт) образ с минимальным временем развертывания и небольшими накладными расходами на передачу данных. Это очень полезные черты в такой системе оркестровки, как Kubernetes, которой может потребоваться регулярно извлекать образы.

---

[1] Gouy, Isaac. «The Computer Language Benchmarks Game». 18 Jan. 2021. https://oreil.ly/bQFjc.

[2] Как этого не хватает в Java.

[3] Как этого не хватает в Python.

Статическая типизация

Еще на заре разработки Go его авторам пришлось сделать выбор: будет ли он статически типизированным, как C++ или Java, и требовать явного определения переменных перед использованием, или динамически типизированным, как Python, что позволит программистам присваивать значения переменным без их определения и, следовательно, писать программы быстрее. Это было не очень сложное решение; на его принятие ушло совсем немного времени. Статическая типизация была очевидным выбором, но этот выбор не был произвольным или основанным на личных предпочтениях[1].

Во-первых, корректность типов в статически типизированных языках можно оценить во время компиляции, что делает их гораздо более производительными (см. табл. 2.1).

Разработчики Go понимали, что время, затрачиваемое на разработку, составляет лишь часть общего жизненного цикла проекта, и любой выигрыш в скорости программирования в динамически типизированных языках с лихвой компенсируется возрастающими трудностями при отладке и поддержке такого кода. В конце концов, какой программист на Python не допускал ошибку, пытаясь использовать строку как целое число?

Возьмем, к примеру, следующий фрагмент кода на Python:

```
my_variable = 0

while my_variable < 10:
    my_varaible = my_variable + 1 # Опечатка, порождающая бесконечный цикл!
```

Заметили? Прочитайте внимательно этот код, если нет. Это может занять несколько секунд.

Любой программист может допустить такую малозаметную опечатку, которая приводит к созданию совершенно корректного выполняемого кода на Python. Это всего лишь два тривиальных примера целого класса ошибок, которые Go будет обнаруживать во время компиляции, а не (не дай бог) после запуска в производство, и, как правило, по времени ближе к моменту их появления. В конце концов, всем понятно, что чем раньше в цикле разработки обнаружится ошибка, тем проще (читайте: дешевле) ее исправить.

Наконец, я даже выражу несколько спорное мнение: типизированные языки более читабельны. Python считается особенно удобочитаемым за его снисходительный характер и синтаксис, похожий на синтаксис английского языка[2], но что бы вы подумали, увидев следующую сигнатуру функции на Python?

```
def send(message, recipient):
```

Что такое `message` – это строка? И является ли `recipient` экземпляром какого-либо класса, описанного где-то еще? Да, этот код можно улучшить,

---

[1] Редко какие аргументы в программировании порождают столько жарких споров, как статическая или динамическая типизация, за исключением, пожалуй, дебатов «табуляции против пробелов», в которых неофициальная позиция Go звучит как: «прикуси язык, твое мнение нас не волнует».

[2] Меня тоже хвалили за мою снисходительную натуру и довольно хорошее знание синтаксиса английского языка.

добавив комментарии и разумные значения по умолчанию, но многим из нас достаточно долго приходилось поддерживать код, чтобы знать, что все это – далекая звезда, которую можно только желать. Явное определение типов может направлять разработку, облегчать умственную нагрузку за счет автоматической проверки информации, которую программист в противном случае должен был бы проверять сам, и служить документацией как для программиста, так и для всех, кому придется поддерживать этот код.

Итоги

В главе 1 основное внимание уделялось атрибутам, отличающим облачные *системы*, а в этой главе основное внимание уделяется характеристикам языка, в частности Go, делающим его пригодным для создания облачных служб.

Облачная система должна быть масштабируемой, слабо связанной, устойчивой, управляемой и наблюдаемой, а язык для облачной эпохи должен уметь делать больше, чем просто создавать системы с этими атрибутами. В конце концов, приложив немного усилий, облачные системы можно создавать практически на любом языке. Так что же делает язык Go таким особенным?

Можно утверждать, что все особенности, представленные в этой главе, прямо или косвенно влияют на облачные атрибуты, перечисленные в предыдущей главе. Что поддержка конкуренции и безопасность памяти способствуют масштабируемости служб, а структурная типизация обеспечивает слабую связанность. Go – единственный известный мне распространенный язык, который сочетает в себе все эти особенности, но настолько ли они новы?

Наиболее заметной из особенностей Go являются, пожалуй, встроенная, а не «прикрученная» поддержка конкуренции, позволяющая программисту безопасно и полностью использовать возможности современного сетевого и многопроцессорного оборудования. Сопрограммы и каналы, конечно, удивительны и значительно упрощают создание устойчивых сетевых служб, но технически они не уникальны, если вспомнить о некоторых менее распространенных языках, таких как Clojure или Crystal.

Я бы добавил, что главная сила Go состоит в его неуклонном следовании принципу ясности над умом, которое исходит из понимания, что исходный код пишется людьми для других людей[1]. А тот факт, что исходный код компилируется в машинный, почти не так важен.

Go проектировался с учетом возможности совместной работы людей в командах, состав которых иногда меняется, участники которых также могут работать над другими проектами. В этой ситуации критически важны ясность кода, минимизация «племенных знаний» и возможность быстрого выполнения итераций. Простоту Go часто неправильно понимают и недооценивают, а между тем он позволяет программистам сосредоточиться на решении задач, а не на борьбе с языком.

В главе 3 мы рассмотрим многие особенности языка Go и познакомимся с его простотой поближе.

---

[1] Или для самих себя, но спустя несколько месяцев размышлений о чем-то другом.

Часть II

ОБЛАЧНЫЕ КОНСТРУКЦИИ В GO

Глава 3

Основы языка Go

> Не стоит изучать язык, который не меняет вашего представления о программировании[1].
>
> – Алан Перлис (Alan Perlis), *ACM SIGPLAN Notices* (сентябрь 1982)

Ни одна книга по программированию не может считаться полной без хотя бы краткого обзора избранного языка, поэтому далее я немного расскажу о Go!

Эта глава будет чуть отличаться от типичных вводных глав, однако я предполагаю, что вы знакомы хотя бы с общими парадигмами программирования, но, возможно, немного подзабыли тонкости синтаксиса Go. Соответственно, основное внимание в данной главе будет уделено нюансам и тонкостям Go, а также его основам. Для более полного знакомства с основами я рекомендую книгу *Introducing Go*[2] (O'Reilly) Калеба Докси (Caleb Doxsey) или *The Go Programming Language*[3] (Addison-Wesley Professional) Алана А. А. Донована (Alan A. A. Donovan) и Брайана У. Кернигана (Brian W. Kernighan).

Если вы только начинаете осваивать этот язык, то вам обязательно стоит прочитать эту главу. Даже если вы достаточно уверенно программируете на Go, все равно просмотрите ее: здесь вы наверняка подметите пару новых интересных приемов. Если вы опытный программист на Go, то можете пропустить эту главу (или прочитать ее с ироничной улыбкой и подколоть меня).

Базовые типы данных

Базовые типы данных в Go, фундаментальные строительные блоки, из которых конструируются более сложные типы, можно разделить на три категории:

- логические значения, несущие только один бит информации, – `true` или `false`, – представляющий некоторое логическое заключение или состояние;

---

[1] Perlis, Alan. *ACM SIGPLAN Notices* 17(9), September 1982, pp. 7–13.

[2] Существует перевод на русский язык, выполненный сообществом: http://golang-book.ru. – *Прим. перев.*

[3] *Донован Алан А. А., Керниган Брайан У.* Язык программирования Go. М.: Вильямс, 2018. ISBN: 978-5-907114-21-0. – *Прим. перев.*

- числовые типы, представляющие простые (разного размера с плавающей запятой и целые со знаком или без знака) или комплексные числа;
- строки, представляющие неизменяемую последовательность кодовых пунктов Юникода.

Логические значения

Логический тип, описывающий состояние истинности, существует в той или иной форме[1] во всех языках программирования. В Go он называется `bool` и является особым 1-битным целочисленным типом и имеет два возможных значения:

- `true`;
- `false`.

Go поддерживает все обычные логические операции:

```
and := true && false
fmt.Println(and)       // "false"

or := true || false
fmt.Println(or)        // "true"

not := !true
fmt.Println(not)       // "false"
```

Интересно отметить, что в Go нет встроенного логического оператора XOR (ИСКЛЮЧАЮЩЕЕ ИЛИ). В нем есть оператор ^, но он выполняет поразрядную операцию XOR.

Простые числа

В Go есть небольшой набор числовых типов с именами, укладывающимися в определенную систему, которые представляют числа с плавающей запятой и целые со знаком и без знака:

Целые со знаком
`int8`, `int16`, `int32`, `int64`

Целые без знака
`uint8`, `uint16`, `uint32`, `uint64`

С плавающей запятой
`float32`, `float64`

---

[1] В ранних версиях C, C++ и Python отсутствовал истинный логический тип, а для представления логических значений использовались целые числа – `0` представлял `false` и `1` – `true`. Некоторые языки, такие как Perl, Lua и Tcl, по-прежнему используют эту стратегию.

Использование определенной системы для именования – это хорошо, но код пишут люди, которые думают по-разному, поэтому дизайнеры Go предусмотрели два дополнительных удобства.

Во-первых, существует два «машинно зависимых» типа с простыми именами – `int` и `uint`, – размер которых определяется доступным аппаратным окружением. Это удобно, если конкретная размерность чисел не играет большой роли. К сожалению, машинно зависимых чисел с плавающей запятой не существует.

Во-вторых, два целочисленных типа имеют мнемонические псевдонимы: `byte` – псевдоним для `uint8`; и `rune` – псевдоним для `int32`.

В большинстве случаев имеет смысл использовать просто `int` и `float64`.

Комплексные числа

Go предлагает два типа *комплексных чисел*, для работы с которыми нужно немного воображения[1]: `complex64` и `complex128`. Их можно выразить в виде *воображаемого литерала* – числа с плавающей запятой, за которым следует `i`:

```
var x complex64 = 3.1415i
fmt.Println(x)                  // "(0+3.1415i)"
```

Комплексные числа очень удобны, но редко используются на практике, поэтому я не буду подробно описывать их. Если вам интересно узнать больше, то полное представление этих чисел, какого они заслуживают, вы найдете в книге *The Go Programming Language* Донована и Кернигана.

Строки

Строка – это последовательность кодовых пунктов Юникода. Строки в Go неизменяемы: содержимое строки нельзя изменить после ее создания.

Go поддерживает два стиля строковых литералов: в двойных кавычках (интерпретируемые литералы) и в обратных апострофах (низкоуровневые строковые литералы). Например, следующие два строковых литерала эквивалентны:

```
// Интерпретируемая форма
"Hello\nworld!\n"

// Низкоуровневая форма
`Hello
world!`
```

В этом интерпретируемом строковом литерале каждая пара символов `\n` будет преобразована в один символ перевода строки, а каждая пара символов `\"` – в один символ двойной кавычки.

---

[1] Вы меня понимаете?

В действительности строковый тип является лишь тонкой оберткой вокруг срезов значений byte в кодировке UTF-8, поэтому любая операция, применимая к срезам и массивам, также может применяться к строкам. Если вы пока незнакомы со срезами, то можете воспользоваться удобным моментом и прочитать раздел «Срезы» ниже.

Переменные

Переменные можно объявлять с помощью ключевого слова var, чтобы связать имя с некоторым типизированным значением, и изменять в любой момент. Типовой синтаксис объявления переменной:

```
var имя тип = выражение
```

Однако объявление переменной обладает довольно большой гибкостью:

- с инициализацией: `var foo int = 42;`
- нескольких переменных: `var foo, bar int = 42, 1302;`
- с автоматическим определением типа: `var foo = 42;`
- нескольких переменных разных типов: `var b, f, s = true, 2.3, "four";`
- без инициализации (см. раздел «Нулевые значения» ниже): `var s string`.

Go очень строго относится к беспорядку: он его *ненавидит*. Если вы объявите переменную в функции, но не будете ее использовать, то программа просто не будет компилироваться.

Сокращенная форма объявления переменных

Go поддерживает синтаксический сахар, позволяющий одновременно объявлять переменные и присваивать им значения внутри функций: оператор := вместо объявления var с неявным типом.

В общем случае сокращенная форма объявления имеет вид:

```
имя := выражение
```

С его помощью можно объявить и одну, и сразу несколько переменных:

- с инициализацией: `percent := rand.Float64() * 100.0;`
- сразу несколько переменных: `x, y := 0, 2`.

На практике сокращенная форма является наиболее распространенным способом объявления и инициализации переменных; ключевое слово var обычно используется либо для объявления локальных переменных, когда требуется явно указать тип, либо для объявления переменных, которым значения будут присвоены позже.

Запомните, что := – это объявление, а = – присваивание. Попытка повторно использовать оператор := для присваивания нового значения существующей переменной завершится ошибкой во время компиляции.

Интересно отметить, что если краткая форма объявления содержит слева смесь новых и существующих переменных, то она действует как форма присваивания новых значений существующим переменным.

Нулевые значения

Когда переменная объявляется без явного значения, ей присваивается нулевое значение соответствующего типа:

- целые числа: `0`;
- числа с плавающей запятой: `0.0`;
- логические значения: `false`;
- строки: `""` (пустая строка).

Для иллюстрации определим четыре переменные разных типов без явной инициализации:

```
var i int
var f float64
var b bool
var s string
```

Если теперь обратиться к этим переменным, то можно обнаружить, что они инициализированы нулевыми значениями:

```
fmt.Printf("integer: %d\n", i) // integer: 0
fmt.Printf("float: %f\n", f)   // float: 0.000000
fmt.Printf("boolean: %t\n", b) // boolean: false
fmt.Printf("string: %q\n", s)  // string: ""
```

Обратите внимание, что в этом примере используется функция `fmt.Printf`, которая позволяет управлять форматом вывода. Если вы незнакомы с этой функцией или строками формата в Go, то прочитайте следующую врезку.

Форматирование ввода/вывода в Go

Пакет `fmt` для Go реализует несколько функций форматированного ввода/вывода. Чаще других (как мне кажется) используются `fmt.Printf` и `fmt.Scanf`, осуществляющие запись в стандартный вывод и чтение из стандартного ввода соответственно:

```
func Printf(format string, a ...interface{}) (n int, err error)
func Scanf(format string, a ...interface{}) (n int, err error)
```

Обратите внимание, что обе имеют параметр `format`. Это *строка формата*, то есть строка, содержащая один или несколько *спецификаторов* (или *глаголов* в терминологии Go), определяющих, как следует интерпретировать остальные параметры. Для функций вывода, таких как `fmt.Printf`, спецификаторы определяют формат вывода значений аргументов.

У каждой функции также есть параметр `a`. Оператор `...` (*произвольное число аргументов*) указывает, что функция принимает ноль или более аргументов в этом месте; `interface{}` сообщает, что тип параметра не указан. Функции с переменным числом

аргументов будут рассматриваться в разделе «Функции с переменным числом аргументов», а тип `interface{}` – в разделе «Интерфейсы».

Вот некоторые спецификаторы, наиболее часто используемые в строках формата:

```
%v   Значение в формате по умолчанию
%T   Представление типа значения
%%   Сам знак процента; не потребляет ни одного аргумента
%t   Для логических значений: выводит слово true или false
%b   Для целых чисел: выводит значение в двоичном виде
%d   Для целых чисел: выводит значение в десятичном виде
%f   Для чисел с плавающей запятой: выводит в формате с запятой
     без экспоненты, например 123.456
%s   Для строк: выводит байты из строки или среза без дополнительной интерпретации
%q   Для строк: интерпретирует как строку в двойных кавычках
     (экранированную с использованием синтаксиса Go)
```

Знакомые с языком C могут распознать эти спецификаторы как несколько упрощенные версии спецификаторов функций `printf` и `scanf`. Более полный список можно найти в документации с описанием пакета `fmt` (https://oreil.ly/Qajzp).

Пустой идентификатор

Пустой идентификатор, представленный оператором _ (подчеркивание), действует как анонимный заполнитель. Его можно использовать как любой другой идентификатор в объявлении, с той лишь разницей, что с ним не будет связано никакое значение.

Чаще всего он используется для выборочного игнорирования ненужных значений в операциях присваивания, что особенно полезно в языке, поддерживающем возврат нескольких значений и не терпящем неиспользуемых переменных. Например, вот как можно поступить, чтобы обработать любые потенциальные ошибки, возвращаемые `fmt.Printf`, не заботясь о количестве записанных байтов[1]:

```
str := "world"

_, err := fmt.Printf("Hello %s\n", str)
if err != nil {
    // обработать ошибку
}
```

Пустой идентификатор также можно использовать для импортирования пакета исключительно ради побочного эффекта:

```
import _ "github.com/lib/pq"
```

Пакеты, импортируемые таким способом, загружаются и инициализируются как обычно и запускают любые свои функции `init`, но на них нельзя сослаться и использовать напрямую.

---

[1] А какой интерес может представлять это количество?

Константы

Константы очень похожи на переменные: ключевое слово `const` связывает идентификатор с некоторым типизированным значением. Однако константы – это не переменные. Во-первых, и это наиболее очевидно, попытка изменить константу приведет к ошибке во время компиляции. Во-вторых, константы должны получать значения при объявлении: они не имеют нулевого значения.

Ключевые слова `var` и `const` можно использовать как на уровне пакета, так и на уровне функции:

```
const language string = "Go"

var favorite bool = true

func main() {
    const text = "Does %s rule? %t!"
    var output = fmt.Sprintf(text, language, favorite)

    fmt.Println(output) // "Does Go rule? true!"
}
```

Для демонстрации их поведенческого сходства предыдущий пример намеренно смешивает явные определения типов с автоматическим определением типа констант и переменных.

Наконец, выбор функции `fmt.Sprintf` здесь не играет особой роли, но если вам неясно назначение спецификаторов в строке формата, то вернитесь к врезке «Форматирование ввода/вывода в Go».

Контейнеры: массивы, срезы и ассоциативные массивы

Go поддерживает стандартные типы контейнеров, в которых можно хранить коллекции значений:

Массив
: Последовательность фиксированной длины элементов конкретного типа.

Срез
: Абстрактная обертка для массивов, размеры которых можно изменять во время выполнения.

Ассоциативный массив
: Структура ассоциативных данных, которая позволяет произвольно сопоставлять ключи со значениями (то есть отображать ключи в значения).

Все эти типы контейнеров имеют свойство `length`, возвращающее количество элементов в контейнере. Также для определения длины любого массива,

среза (включая строки) или ассоциативного массива можно использовать встроенную функцию `len`.

Массивы

В Go, как и в большинстве других основных языков, *массив* представляет последовательность фиксированной длины из нуля или более элементов определенного типа.

Массивы объявляются включением длины в объявление. Нулевое значение для массива – это массив указанной длины, содержащий элементы с нулевыми значениями. Элементы массива доступны по индексам от `0` до `N-1`, которые указываются знакомым способом с использованием квадратных скобок:

```
var a [3]int           // Массив типа [3]int, заполненный нулевыми значениями
fmt.Println(a)         // "[0 0 0]"
fmt.Println(a[1])      // "0"

a[1] = 42              // Изменить второй элемент
fmt.Println(a)         // "[0 42 0]"
fmt.Println(a[1])      // "42"

i := a[1]
fmt.Println(i)         // "42"
```

Массивы можно инициализировать с помощью литералов:

```
b := [3]int{2, 4, 6}
```

Также можно доверить компилятору самому подсчитать количество элементов в объявлении массива:

```
b := [...]int{2, 4, 6}
```

В обоих случаях будет создан массив `b` с типом `[3]int`.

Для определения длины любого представителя контейнерного типа можно использовать встроенную функцию `len`:

```
fmt.Println(len(b))        // "3"
fmt.Println(b[len(b)-1])   // "6"
```

На практике массивы редко используются напрямую. Вместо них гораздо чаще используются *срезы* – абстрактный тип массивов, действующий как массив с изменяемым размером.

Срезы

Срезы (slices) – это тип данных в Go, который обеспечивает мощную абстракцию для традиционных массивов. Операции со срезами выглядят для программиста очень похожими на операции с массивами. Как и массивы, срезы

обеспечивают доступ к последовательностям элементов определенного типа через знакомую форму записи индексов от `0` до `N-1` в квадратных скобках. Однако, в отличие от массивов, которые имеют фиксированную длину, размеры срезов можно изменять во время выполнения.

Как показано на рис. 3.1, в действительности срез является облегченной структурой данных с тремя компонентами:

- указатель на некоторый элемент массива, лежащего в основе среза, представляющий первый элемент среза (не обязательно первый элемент массива);
- длина, представляющая количество элементов в срезе;
- емкость, представляющая максимально возможное значение длины.

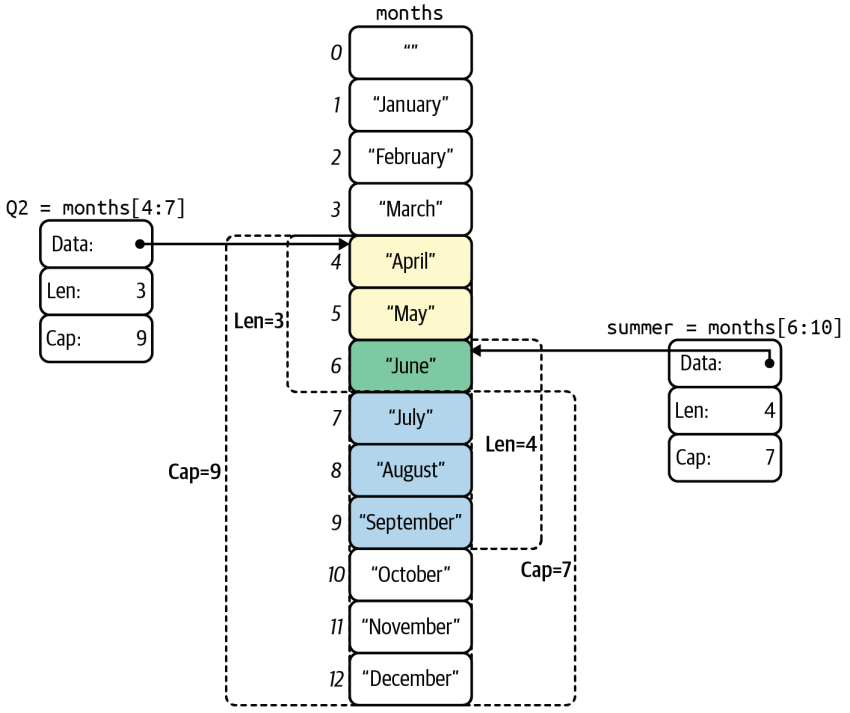


Рис. 3.1 ❖ Два среза, основанных на одном и том же массиве

Если не указано иное, значение емкости равно количеству элементов от начала среза до конца массива, лежащего в основе. Получить длину и емкость среза можно с помощью встроенных функций `len` и `cap` соответственно.

Работа со срезами

Создание среза несколько отличается от создания массива: срезы определяются только типами их элементов, но не их количеством. Для создания среза ненулевой длины можно использовать встроенную функцию `make`:

```
n := make([]int, 3)    // Создать срез с 3 элементами типа int

fmt.Println(n)         // "[0 0 0]"
fmt.Println(len(n))    // "3"; len можно применять и к срезам, и к массивам

n[0] = 8
n[1] = 16
n[2] = 32

fmt.Println(n)         // "[8 16 32]"
```

Как видите, работа со срезами во многом напоминает работу с массивами. Так же как массивы, нулевое значение среза – это срез указанной длины с нулевыми значениями в элементах, а элементы в срезе доступны по их индексам точно так же, как в массиве.

Литерал среза объявляется так же, как литерал массива, за исключением того, что количество элементов не указывается:

```
m := []int{1}          // Литерал объявления среза []int
fmt.Println(m)         // "[1]"
```

Длину среза можно увеличить с помощью встроенной функции `append`, которая возвращает срез увеличенной длины с дополнительными элементами в конце:

```
m = append(m, 2)       // Добавит 2 в конец m
fmt.Println(m)         // "[1 2]"
```

Встроенная функция `append` может принимать произвольное количество аргументов, помимо среза. Подробнее о функциях с переменным числом аргументов рассказывается в разделе «Функции с переменным числом аргументов»:

```
m = append(m, 2)       // Добавит 2 в m из предыдущего примера
fmt.Println(m)         // "[1 2]"

m = append(m, 3, 4)
fmt.Println(m)         // "[1 2 3 4]"

m = append(m, m...)    // Добавит в конец m элементы из самого себя
fmt.Println(m)         // "[1 2 3 4 1 2 3 4]"
```

Обратите внимание, что встроенная функция `append` возвращает расширенный срез, а не изменяет его на месте. Причина в том, что за кулисами, если в базовом массиве достаточно места для размещения новых элементов, новый срез создается на его основе. В противном случае автоматически создается новый базовый массив.

Функция `append` *возвращает* расширенный срез. Несохранение его – распространенная ошибка.

Оператор извлечения среза

Массивы и срезы (включая строки) поддерживают *оператор извлечения среза*, имеющий синтаксис `s[i: j]`, где значения `i` и `j` должны находиться в диапазоне `0 ≤ i ≤ j ≤ cap(s)`.

Например:

```
s0 := []int{0, 1, 2, 3, 4, 5, 6} // Литерал среза
fmt.Println(s0)                  // "[0 1 2 3 4 5 6]"
```

В предыдущем примере определяется литерал среза. Он очень похож на литерал массива, с той лишь разницей, что в литералах среза не указывается размер.

Если значение `i` или `j` опущено, то по умолчанию они принимаются равными `0` и `len(s)` соответственно:

```
s1 := s0[:4]
fmt.Println(s1)        // "[0 1 2 3]"

s2 := s0[3:]
fmt.Println(s2)        // "[3 4 5 6]"
```

Оператор извлечения среза создает новый срез, основанный на том же массиве, с длиной `j - i`. Изменения, произведенные в этом срезе, отразятся на содержимом базового массива и, соответственно, на всех срезах, базирующихся на том же самом массиве:

```
s0[3] = 42             // Это изменение отразится на всех 3 срезах
fmt.Println(s0)        // "[0 1 2 42 4 5 6]"
fmt.Println(s1)        // "[0 1 2 42]"
fmt.Println(s2)        // "[42 4 5 6]"
```

Более наглядно подобный эффект иллюстрирует рис. 3.1.

Строки и срезы

Внутренняя реализация строк в Go немного сложнее, чем вы могли бы ожидать, и опирается на множество тонкостей, таких как различия между байтами, символами и рунами (тип `rune`), кодировка UTF-8 Юникода и различия между строками и строковыми литералами.

На данный момент вам достаточно знать, что строки в Go – это просто срезы байтов, доступные только для чтения, которые обычно (но не обязательно) содержат последовательность символов UTF-8, представляющих кодовые пункты Юникода, называемые *рунами*. Go даже позволяет преобразовывать строки в массивы байтов или рун:

```
s := "foö"             // Юникод: f=0x66 o=0x6F ö=0xC3B6
r := []rune(s)
b := []byte(s)
```

Преобразовав строку `s` таким способом, можно раскрыть ее идентичность как среза байтов или рун. Это легко проиллюстрировать с помощью `fmt`.

Printf и спецификаторами %T (тип) и %v (значение), представленными во врезке «Форматирование ввода/вывода в Go» выше:

```
fmt.Printf("%7T %v\n", s, s)   // "string foö"
fmt.Printf("%7T %v\n", r, r)   // "[]int32 [102 111 246]"
fmt.Printf("%7T %v\n", b, b)   // "[]uint8 [102 111 195 182]"
```

Обратите внимание, что значение строкового литерала – foö – содержит смесь символов, часть которых может быть представлена одним байтом (f и o представлены кодами 102 и 111 соответственно), и один символ может быть представлен только двумя байтами (ö представлен парой кодов 195 182).

Напомню еще раз, что типы byte и rune – это мнемонические псевдонимы для uint8 и int32 соответственно.

Все три вызова fmt.Printf выводят тип и значение указанной переменной. Как и ожидалось, строковое значение foö выводится буквально. Однако следующие два вызова выглядят интереснее. Срез типа uint8 содержит четыре байта, представляющих символы в кодировке UTF-8 (два однобайтовых кодовых пункта и один двухбайтовый). Срез типа int32 содержит три значения, представляющих кодовые пункты отдельных символов.

В Go поддерживаются и другие способы кодирования строк, но у нас не так много места для их обсуждения. Чтобы узнать больше, загляните в статью Роба Пайка (Rob Pike) «Strings, Bytes, Runes and Characters in Go» в блоге Go (https://oreil.ly/mgku7).

Ассоциативные массивы

Ассоциативный массив в Go реализован в виде *хеш-таблицы* – невероятно полезной ассоциативной структуры данных, позволяющей «отображать» ключи в значения, то есть формировать пары ключ/значение. Эта структура данных имеется во всех основных современных языках программирования: если у вас есть опыт программирования на одном из них, то, вероятно, вы уже использовали их, например, в форме dict в Python, Hash в Ruby или HashMap в Java.

Ассоциативные массивы в Go объявляются как map[K]V, где K и V – это типы ключей и значений соответственно. В качестве типа ключа можно использовать любой тип, поддерживающий сравнение с использованием оператора ==, при этом не требуется, чтобы K и V были одним и тем же типом. Например, строковые ключи могут отображаться в значения типа float32.

Ассоциативный массив можно инициализировать с помощью встроенной функции make, а ссылаться на значения в нем – с использованием привычного синтаксиса name[key]. Уже знакомая нам функция len вернет для ассоциативного массива количество пар ключ/значение; удалять пары ключ/значение можно с помощью встроенной функции delete:

```
freezing := make(map[string]float32) // пустой ассоциативный массив,
                                     // отображающий string в float32
```

```
freezing["celsius"] = 0.0
freezing["fahrenheit"] = 32.0
freezing["kelvin"] = 273.2

fmt.Println(freezing["kelvin"])      // "273.2"
fmt.Println(len(freezing))           // "3"

delete(freezing, "kelvin")           // Удалить "kelvin"
fmt.Println(len(freezing))           // "2"
```

Инициализировать и заполнять ассоциативные массивы можно также с применением литералов:

```
freezing := map[string]float32{
    "celsius": 0.0,
    "fahrenheit": 32.0,
    "kelvin": 273.2,                 // Запятая в конце обязательна!
}
```

Обратите внимание на запятую в конце последней строки. Она обязательна: компилятор откажется компилировать код, если опустить эту запятую.

Проверка наличия в ассоциативном массиве

Попытка обратиться к ключу, отсутствующему в ассоциативном массиве, не вызовет исключения (в любом случае в языке Go их нет) или возврата какого-либо пустого (null) значения. Вместо этого будет возвращено нулевое значение для типа значения:

```
foo := freezing["no-such-key"] // Получить значение отсутствующего ключа
fmt.Println(foo)               // "0" (нулевое значение для float32)
```

Это очень полезная особенность, сокращающая количество шаблонных проверок наличия при работе с ассоциативными массивами, но иногда она может вызывать сложности, когда ассоциативный массив действительно содержит нулевые значения. К счастью, операция обращения к ассоциативному массиву возвращает также второе необязательное логическое значение – признак присутствия ключа в ассоциативном массиве:

```
newton, ok := freezing["newton"]   // Существует ли шкала Ньютона?
fmt.Println(newton)                // "0"
fmt.Println(ok)                    // "false"
```

В этом примере переменная `newton` получит значение `0.0`. Но как узнать, это действительное значение[1] или просто в массиве не оказалось искомого ключа? Благодаря значению `false` в `ok` мы можем смело утверждать, что верно последнее.

[1] И действительно, точка замерзания воды по шкале Ньютона равна 0,0 градуса, но это не суть важно.

Указатели

Итак, указатели. Проклятие и погибель студентов во всем мире. Если вы используете язык с динамической типизацией, то идея указателя может показаться вам чуждой. Мы не будем *слишком* вдаваться в подробности этого предмета, но тем не менее постараемся охватить его достаточно полно, чтобы у вас сложилось более или менее ясное представление о нем.

Вернемся к первому принципу: «переменная» – это область в памяти, где хранится какое-то значение. Обычно, когда вы ссылаетесь на переменную по ее имени (`foo = 10`) или используете ее в выражении (`s[i] = "foo"`), то фактически читаете или изменяете ее значение.

Указатель хранит адрес *переменной*: адрес места в памяти, где хранится значение. Каждая переменная имеет адрес, и использование указателей позволяет читать или изменять значения переменных косвенным способом (как показано на рис. 3.2):

Получение адреса переменной
: Адрес именованной переменной можно получить с помощью оператора `&`. Например, выражение `p := &a` получит адрес переменной `a` и присвоит его переменной-указателю `p`.

Типы указателей
: Переменная `p`, про которую можно сказать, что она «указывает на», имеет тип `*int`, где `*` сообщает, что это тип указателя на значение типа `int`.

Разыменование указателя
: Чтобы получить значение `a` с помощью `p`, указатель нужно *разыменовать*, добавив `*` перед именем переменной-указателя. Это позволит косвенно читать или изменять `a`.

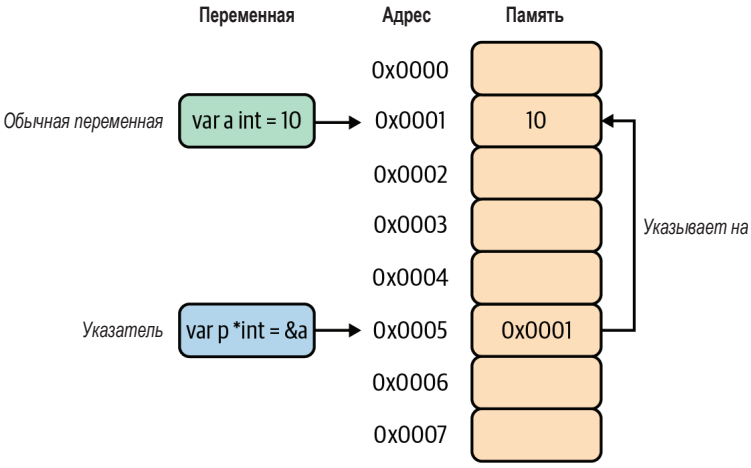


Рис. 3.2 ❖ Выражение p := &a извлекает адрес переменной a и присваивает его переменной p

Теперь соберем все вместе:

```
var a int = 10

var p *int = &a        // p типа *int указывает на a
fmt.Println(p)         // "0x0001"
fmt.Println(*p)        // "10"

*p = 20                // косвенно изменит a
fmt.Println(a)         // "20"
```

Как и любые другие переменные, указатели можно объявлять с нулевым значением `nil`, если их не требуется инициализировать заранее. Также их можно сравнивать: если указатели равны, значит, они содержат один и тот же адрес (то есть указывают на одну и ту же переменную или оба содержат `nil`):

```
var n *int
var x, y int

fmt.Println(n)         // "<nil>"
fmt.Println(n == nil)  // "true" (n содержит nil)

fmt.Println(x == y)    // "true" (x и y обе содержат ноль)
fmt.Println(&x == &x)  // "true" (адрес x равен самому себе)
fmt.Println(&x == &y)  // "false" (разные адреса переменных)
fmt.Println(&x == nil) // "false" (адрес x не равен nil)
```

Поскольку в этом примере переменная `n` нигде не инициализируется, она всегда имеет значение `nil`, поэтому сравнение ее с `nil` возвращает `true`. Переменные `x` и `y` хранят значение `0`, поэтому сравнение их значений дает `true`, но они являются разными переменными, поэтому сравнение их адресов дает `false`.

Управляющие структуры

Любой программист, пришедший в Go из другого языка, обнаружит, что набор управляющих структур, имеющихся в Go, в целом знаком и даже привычен (поначалу), особенно тем, кто имеет опыт программирования на языках, близких к языку C. Однако их реализации имеют некоторые довольно существенные отличия, которые первое время могут казаться странными.

Например, инструкции управляющих структур не требуют большого количества скобок. И это хорошо – чем меньше отвлекающих деталей синтаксиса, тем лучше.

Также существует только один тип циклов. В Go нет циклов `while` – только `for`. Серьезно! Хотя на самом деле это очень круто. Читайте дальше, и вы поймете, о чем я.

Забавный цикл for

Инструкция `for` – это единственный цикл в Go, но, несмотря на отсутствие явного цикла `while`, его можно реализовать с помощью `for`, эффективно унифицируя все способы управления входом в цикл, к которым вы привыкли.

Go не имеет эквивалента `do-while`.

Универсальная инструкция for

В общем случае инструкция цикла `for` в Go имеет почти такой же синтаксис, как и в других языках семейства C, она включает: оператор инициализации, условие продолжения и оператор, выполняемый в конце каждой итерации; все операторы традиционно разделяются точкой с запятой. Любые переменные, объявленные в операторе инициализации, будут доступны только в теле цикла `for`:

```
sum := 0

for i := 0; i < 10; i++ {
    sum += 1
}

fmt.Println(sum)       // "10"
```

В этом примере переменная `i` инициализируется значением 0. В конце каждой итерации `i` увеличивается на 1, и пока ее значение остается меньше 10, процесс повторяется.

В отличие от большинства C-подобных языков, инструкция `for` в Go не требует использования круглых скобок вокруг предложений в заголовке цикла, а фигурные скобки, ограничивающие тело цикла, являются обязательными.

В отличие от традиционных C-подобных языков, оператор инициализации и оператор, выполняемый в конце каждой итерации, в языке Go не являются обязательными. Как показано в следующем примере, это делает цикл `for` значительно более гибким:

```
sum, i := 0, 0

for i < 10 {           // Эквивалентно: for ; i < 10;
    sum += i
    i++
}

fmt.Println(i, sum)    // "10 45"
```

Инструкция `for` в предыдущем примере содержит только условие проверки продолжения итераций. На самом деле это очень важное усовершенствование, потому что позволяет циклу `for` выполнять функции, обычно возлагаемые на цикл `while`.

Наконец, исключение всех трех операторов из заголовка инструкции `for` создает блок, выполняющийся в цикле бесконечно, подобно традиционному `while (true)`:

```
fmt.Println("For ever...")

for {
    fmt.Println("...and ever")
}
```

Поскольку в этом цикле отсутствуют какие-либо условия завершения, он будет повторяться бесконечно.

Обход в цикле элементов массивов и срезов

В Go имеется полезное ключевое слово `range`, упрощающее обход различных коллекций в цикле.

Например, `range` можно использовать вместе с инструкцией `for` для перечисления индексов и извлечения значений элементов:

```
s := []int{2, 4, 8, 16, 32}  // Срез значений типа int

for i, v := range s {        // range возвращает каждый индекс/значение
    fmt.Println(i, "->", v)  // Выведет индекс и соответствующее ему значение
}
```

В предыдущем примере значения `i` и `v` будут обновляться в каждой итерации и содержать индекс и значение каждого следующего элемента в срезе `s`. То есть результат будет выглядеть примерно так:

```
0 -> 2
1 -> 4
2 -> 8
3 -> 16
4 -> 32
```

Но что, если вам не нужны оба этих значения? В конце концов, компилятор Go требует использовать объявленные переменные. К счастью, как и повсюду в Go, ненужные значения можно отбросить, используя «пустой идентификатор»:

```
a := []int{0, 2, 4, 6, 8}
sum := 0

for _, v := range a {
    sum += v
}

fmt.Println(sum)       // "20"
```

Как и в предыдущем примере, значение `v` будет обновляться в каждой итерации и содержать значение очередного элемента среза `a`. Однако на этот

раз значение индекса игнорируется и отбрасывается и сохраняется только значение элемента.

Обход в цикле элементов ассоциативных массивов

Ключевое слово `range` также может использоваться с оператором `for` для перебора элементов ассоциативного массива, при этом в каждой итерации будет возвращаться следующий ключ и значение:

```
m := map[int]string{
    1: "January",
    2: "February",
    3: "March",
    4: "April",
}

for k, v := range m {
    fmt.Println(k, "->", v)
}
```

Обратите внимание, что ассоциативные массивы в Go не гарантируют какой-то определенный порядок следования ключей, поэтому вывод выглядит неупорядоченным:

```
3 -> March
4 -> April
1 -> January
2 -> February
```

Инструкция if

Инструкция `if` в Go используется точно так же, как в других C-подобных языках, за исключением отсутствия круглых скобок вокруг условного выражения и обязательности фигурных скобок:

```
if 7 % 2 == 0 {
    fmt.Println("7 is even")
} else {
    fmt.Println("7 is odd")
}
```

В отличие от большинства C-подобных языков, инструкция `if` в Go не требует заключать условное выражение в круглые скобки, а фигурные скобки являются обязательными.

Интересно отметить, что Go позволяет вставлять инструкцию инициализации перед выражением проверки условия в операторе `if`. Например:

```
if _, err := os.Open("foo.ext"); err != nil {
    fmt.Println(err)
```

```
} else {
    fmt.Println("All is fine.")
}
```

Обратите внимание, как инициализируется переменная `err` перед проверкой, делая этот код примерно похожим на следующий:

```
_, err := os.Open("foo.go")
if err != nil {
    fmt.Println(err)
} else {
    fmt.Println("All is fine.")
}
```

Однако эти две конструкции не являются точно эквивалентными: в первом примере переменная `err` видима только внутри инструкции `if`; во втором примере она доступна во всей области видимости, окружающей инструкцию.

Инструкция switch

Так же как во многих других языках, в Go есть инструкция `switch`, помогающая более кратко выразить серию условных выражений `if-then-else`. Однако она имеет множество отличий от привычных реализаций, что делает ее значительно более гибкой.

Наиболее очевидное отличие для тех, кто пришел из C-подобных языков, заключается в отсутствии «проваливания» из одного варианта в другой; такое проваливание можно явно добавить с помощью ключевого слова `fallthrough`:

```
i := 0

switch i % 3 {
case 0:
    fmt.Println("Zero")
    fallthrough
case 1:
    fmt.Println("One")
case 2:
    fmt.Println("Two")
default:
    fmt.Println("Huh?")
}
```

В этом примере значение `i % 3` равно `0`, что соответствует первому варианту, и в результате выводится слово `Zero`. В языке Go варианты не проваливаются друг в друга по умолчанию, и наличие явного оператора `fallthrough` означает, что следующий вариант тоже выполнится и выведет `One`. Наконец, отсутствие `fallthrough` в этом случае приводит к завершению инструкции `switch`. Таким образом, этот пример выведет:

```
Zero
One
```

Инструкция switch в Go имеет два интересных свойства. Во-первых, выражения в case не обязательно должны быть целыми числами или даже константами: операторы case будут последовательно оцениваться сверху вниз, и выполнится первый из них, значение выражения при котором совпадет со значением выражения при switch. Во-вторых, если выражение при операторе switch оставить пустым, то оно будет интерпретировано как true и совпадет с первым вариантом case, условие в котором будет оценено как истинное. Оба этих свойства демонстрируются в следующем примере:

```
hour := time.Now().Hour()

switch {
case hour >= 5 && hour < 9:
    fmt.Println("I'm writing")
case hour >= 9 && hour < 18:
    fmt.Println("I'm working")
default:
    fmt.Println("I'm sleeping")
}
```

Здесь при инструкции switch нет выражения, поэтому она интерпретируется как switch true. В результате для выполнения будет выбран первый вариант case, условие которого также будет оценено как true. В этом случае значение hour равно 23, поэтому этот код выведет «I'm sleeping»[1].

Наконец, так же как в случае с if, условному выражению в switch может предшествовать выражение инициализации, и в этом случае любые объявленные переменные будут видны только в теле инструкции switch. Например, предыдущий пример можно переписать так:

```
switch hour := time.Now().Hour(); { // Пустое выражение эквивалентно true
case hour >= 5 && hour < 9:
    fmt.Println("I'm writing")
case hour >= 9 && hour < 18:
    fmt.Println("I'm working")
default:
    fmt.Println("I'm sleeping")
}
```

Обратите внимание на точку с запятой в конце: это пустое выражение, подразумевающее true, то есть это выражение эквивалентно switch hour: = time.Now().Hour(); true и соответствует первому варианту case с истинным условием.

Обработка ошибок

Ошибки в Go интерпретируются как еще одно значение, представленное встроенным типом error. Это упрощает обработку ошибок: идиоматические

[1] Понятно, что этот код нужно откалибровать.

функции Go могут включать в свой список возвращаемых значений значение с типом `error`, которое, если не равно `nil`, указывает на состояние ошибки, которое можно обработать в основном пути выполнения. Например, функция `os.Open` возвращает значение ошибки, отличное от `nil`, когда не может открыть файл:

```
file, err := os.Open("somefile.ext")
if err != nil {
    log.Fatal(err)
    return err
}
```

Фактическая реализация типа `error` невероятно проста: это просто универсальный интерфейс, объявляющий единственный метод:

```
type error interface {
    Error() string
}
```

Такой подход сильно отличается от исключений, имеющихся во многих языках, которые требуют специальной системы для их перехвата и обработки, что нередко ведет к созданию запутанного и малопонятного управления потоком выполнения.

Создание ошибки

Есть два простых способа создания значений ошибок и один сложный. К простым относятся вызовы функций `errors.New` и `fmt.Errorf`; последняя из которых удобнее, потому что дополнительно поддерживает форматирование строк:

```
e1 := errors.New("error 42")
e2 := fmt.Errorf("error %d", 42)
```

Тот факт, что `error` – это интерфейс, позволяет при необходимости реализовать собственные типы ошибок. Например, вот типичный шаблон создания вложенных ошибок:

```
type NestedError struct {
    Message string
    Err     error
}

func (e *NestedError) Error() string {
    return fmt.Sprintf("%s\n contains: %s", e.Message, e.Err.Error())
}
```

Дополнительную информацию об ошибках и полезные советы по их обработке в Go вы найдете в статье Эндрю Герранда (Andrew Gerrand) «Error Handling and Go» в блоге *The Go Blog* (https://oreil.ly/YQ6if).

Необычные особенности функций: переменное число параметров и замыкания

Функции в Go действуют точно так же, как в других языках: они принимают параметры, выполняют некоторые вычисления и (необязательно) что-то возвращают.

Но функции Go обладают особой гибкостью, которой нет во многих основных языках, и могут делать многое такое, что недоступно во многих других языках, например возвращать или принимать несколько значений, или использоваться в качестве обычных типов либо анонимных функций.

Функции

Объявление функции в Go аналогично объявлению функции в большинстве других языков: у них есть имя, список типизированных параметров, необязательный список типов возвращаемых значений и тело. Однако объявление функции в Go несколько отличается от объявлений в других C-подобных языках: объявление начинается со специального ключевого слова `func`; тип каждого параметра следует за его именем; а типы возвращаемых значений помещаются в конец заголовка определения функции и могут быть полностью опущены (в Go нет типа `void`).

Функция со списком типов возвращаемых значений должна заканчиваться оператором `return`, кроме случаев, когда выполнение не может достичь конца функции из-за наличия бесконечного цикла или когда функция завершается оператором `panic`:

```
func add(x int, y int) int {
    return x + y
}

func main() {
    sum := add(10, 5)
    fmt.Println(sum)   // "15"
}
```

Кроме того, дополнительный синтаксический сахар позволяет записывать тип для последовательности однотипных параметров или возвращаемых значений только один раз. Например, следующие определения `func foo` эквивалентны:

```
func foo(i int, j int, a string, b string) { /* ... */ }
func foo(i, j int, a, b string)            { /* ... */ }
```

Несколько возвращаемых значений

Функции могут возвращать любое количество значений. Например, следующая функция `swap` принимает и возвращает две строки. Список типов при возврате нескольких значений должен заключаться в круглые скобки:

```
func swap(x, y string) (string, string) {
    return y, x
}
```

Чтобы принять несколько значений, возвращаемых функцией, можно использовать множественное присваивание:

```
a, b := swap("foo", "bar")
```

В данном примере переменная `a` получит значение `"bar"`, а переменная `b` – значение `"foo"`.

Рекурсия

Go поддерживает *рекурсивные* вызовы функций, когда функции вызывают сами себя. При правильном использовании рекурсия может быть очень мощным инструментом, позволяющим решать самые разные задачи. Канонический пример – вычисление факториала положительного целого числа, то есть произведения всех положительных целых чисел, меньших или равных `n`:

```
func factorial(n int) int {
    if n < 1 {
        return 1
    }
    return n * factorial(n-1)
}

func main() {
    fmt.Println(factorial(11)) // "39916800"
}
```

Для любого целого `n` больше единицы `factorial` будет вызывать саму себя с параметром `n - 1`. Такие рекурсивные вызовы могут накапливаться очень быстро!

Отложенные вычисления

В Go имеется ключевое слово `defer`, которое можно использовать для планирования выполнения некоторых действий непосредственно перед возвратом из функции. Обычно эта возможность используется для высвобождения ресурсов некоторым способом.

Например, чтобы отложить вывод текста `"cruel world"` до конца вызова функции, можно вставить ключевое слово `defer` непосредственно перед выводом:

```
func main() {
    defer fmt.Println("cruel world")

    fmt.Println("goodbye")
}
```

Если вызвать такую функцию, она выведет:

```
goodbye
cruel world
```

Чтобы продемонстрировать более сложный пример, создадим пустой файл и попытаемся записать в него. Функция `closeFile` предназначена для закрытия файла по окончании работы с ним. Однако если просто вызвать ее в конце функции `main`, то в случае ошибки `closeFile` никогда не будет вызвана, и файл останется открытым. Поэтому мы используем `defer`, чтобы гарантировать вызов функции `closeFile` перед возвратом из функции `main`:

```
func main() {
    file, err := os.Create("/tmp/foo.txt") // Создать пустой файл
    defer closeFile(file)                  // Гарантировать вызов closeFile(file)
    if err != nil {
        return
    }

    _, err = fmt.Fprintln(file, "Your mother was a hamster")
    if err != nil {
        return
    }

    fmt.Println("File written to successfully")
}

func closeFile(f *os.File) {
    if err := f.Close(); err != nil {
        fmt.Println("Error closing file:", err.Error())
    } else {
        fmt.Println("File closed successfully")
    }
}
```

Эта функция `main` выведет:

```
File written to successfully
File closed successfully
```

Если в функции используется несколько ключевых слов `defer`, они помещаются в стек и по завершении вмещающей функции выполняются в обратном порядке, по принципу «последним пришел – первым ушел». Например:

```
func main() {
    defer fmt.Println("world")
    defer fmt.Println("cruel")
    defer fmt.Println("goodbye")
}
```

Эта функция выведет:

```
goodbye
cruel
world
```

Отложенные вычисления широко используются для высвобождения ресурсов. При работе с внешними ресурсами вы обязательно будете пользоваться этой возможностью благодаря ее удобству.

Указатели как параметры

Широта возможностей указателей становится особенно очевидной в сочетании с функциями. Как правило, параметры передаются в функции по значению: в таких случаях функция получает копию каждого параметра и изменения, внесенные функцией в копии, не влияют на вызывающую сторону. Однако указатели содержат ссылку на значение, а не само значение, и могут использоваться вызываемой функцией для косвенного изменения значения, переданного в функцию, чтобы повлиять на вызывающую функцию.

Следующая функция демонстрирует оба сценария:

```
func main() {
    x := 5

    zeroByValue(x)
    fmt.Println(x)             // "5"

    zeroByReference(&x)
    fmt.Println(x)             // "0"
}

func zeroByValue(x int) {
    x = 0
}

func zeroByReference(x *int) {
    *x = 0                     // Разыменование x и запись 0 по указателю
}
```

Такое поведение не является уникальным для указателей. В действительности некоторые типы данных являются ссылками на ячейки памяти, включая срезы, ассоциативные массивы, функции и каналы. Изменения, внесенные функциями в значения таких *ссылочных типов*, могут повлиять на вызывающую программу, при этом вызываемым функциям не требуется явно разыменовывать их:

```
func update(m map[string]int) {
    m["c"] = 2
}

func main() {
    m := map[string]int{ "a" : 0, "b" : 1}

    fmt.Println(m)                 // "map[a:0 b:1]"
```

```
    update(m)

    fmt.Println(m)                  // "map[a:0 b:1 c:2]"
}
```

В этом примере в функцию `update` передается ассоциативный массив `m` с длиной 2. Затем `update` добавляет в него пару `{"c": 2}`. Поскольку `m` является ссылочным типом, он передается в `update` как ссылка на структуру данных, а не как копия, поэтому `main` увидит в `m` новую пару после возврата из функции `update`.

Функции с переменным числом аргументов

Функции с переменным числом аргументов могут принимать ноль и более аргументов. Типичным примером могут служить функции из семейства `fmt.Printf`, которые принимают строку формата и произвольное количество дополнительных аргументов.

Вот сигнатура стандартной функции `fmt.Printf`:

```
func Printf(format string, a ...interface{}) (n int, err error)
```

Обратите внимание, что она принимает строку и ноль или более значений типа `interface{}`. Если вы незнакомы с синтаксисом `interface{}`, то не волнуйтесь, мы рассмотрим его в разделе «Интерфейсы» ниже, а пока можете рассматривать `interface{}` как «нечто произвольное». Однако самое интересное здесь то, что последний аргумент содержит многоточие (...). Это *оператор переменного числа аргументов – variadic*, который сообщает, что функцию можно вызвать с любым количеством аргументов этого типа. Например, fmt.Printf можно вызвать со строкой формата и двумя аргументами разных типов:

```
const name, age = "Kim", 22
fmt.Printf("%s is %d years old.\n", name, age)
```

В вызов такой функции список дополнительных аргументов передается как срез. В следующем примере параметр `factor` метода `product` имеет тип `[]int` и может использоваться в соответствии с типом:

```
func product(factors ...int) int {
    p := 1

    for _, n := range factors {
        p *= n
    }

    return p
}

func main() {
    fmt.Println(product(2, 2, 2)) // "8"
}
```

В этом примере функция `main` вызывает `product` с тремя аргументами (хотя в принципе она может передать любое количество аргументов, сколько потребуется). В функции `product` они доступны как срез типа `[]int` с элементами `{2, 2, 2}`, которые последовательно перемножаются для получения возвращаемого значения `8`.

Передача срезов в параметре с переменным числом значений

Что, если необходимые аргументы уже находятся в некотором срезе и было бы желательно передать его в функцию с переменным числом аргументов? Неужели придется разобрать такой срез на отдельные значения? Конечно, нет!

В этом случае можно добавить оператор переменного числа аргументов после имени переменной в вызове функции:

```
m := []int{3, 3, 3}
fmt.Println(product(m...)) // "27"
```

Здесь у нас есть переменная `m` с типом `[]int`, которую нужно передать в вызов функции `product`. Использование оператора `...` при вызове `product(m...)` делает это возможным.

Анонимные функции и замыкания

В языке Go функции считаются самыми обычными (или, как говорят, первоклассными) значениями, с которыми можно работать так же, как с любыми другими объектами: они имеют типы, их можно присваивать переменным и даже передавать и возвращать другим функциям.

Нулевое значение типа функции – `nil`; вызов функции `nil` вызовет крах (панику):

```
func sum(x, y int) int     { return x + y }
func product(x, y int) int { return x * y }

func main() {
    var f func(int, int) int // Переменная-функция имеет тип

    f = sum
    fmt.Println(f(3, 5))     // "8"

    f = product              // Допустимо: product имеет тот же тип, что и sum
    fmt.Println(f(3, 5))     // "15"
}
```

Функции могут создаваться внутри других функций как *анонимные функции*, которые можно вызывать, передавать или использовать иным способом как любые другие функции. Необыкновенно мощной особенностью Go является доступность состояния родительской функции из анонимных функций,

даже после завершения родительской функции. Фактически это позволяет определять *замыкания*.

Замыкание – это вложенная функция, сохраняющая доступ к переменным родительской функции даже после завершения родительской функции.

Возьмем, например, следующую функцию `incrementor`. Она имеет состояние в виде переменной `i` и возвращает анонимную функцию, которая увеличивает это значение перед его возвратом. Можно сказать, что возвращаемая функция *замкнута на* переменной `i`, что делает ее истинным (в простейшем случае) замыканием:

```
func incrementer() func() int {
    i := 0

    return func() int { // Возвращает анонимную функцию,
        i++             // "замкнутую на" переменной i в родительской функции
        return i
    }
}
```

Вызов `incrementor` создаст свою локальную копию `i` и вернет новую анонимную функцию, увеличивающую значение этой копии. Последующие вызовы `incrementor` будут создавать новые копии `i`. Вот как это работает:

```
func main() {
    increment := incrementer()
    fmt.Println(increment())       // "1"
    fmt.Println(increment())       // "2"
    fmt.Println(increment())       // "3"

    newIncrement := incrementer()
    fmt.Println(newIncrement())    // "1"
}
```

Как видите, `incrementer` возвращает новую функцию, которая сохраняется в переменной `increment`; каждый вызов `increment` увеличивает внутренний счетчик на единицу. Однако когда `incrementer` вызывается снова, он создает и возвращает совершенно новую функцию со своим собственным счетчиком. Ни одна из этих функций не может повлиять на другую.

Структуры, методы и интерфейсы

Одно из самых важных ментальных изменений, которые иногда приходится совершать людям, переходя на язык Go, заключается в том, что Go не является традиционным объектно-ориентированным языком. Конечно, в Go есть типы с методами, которые выглядят как традиционные объекты, но они не имеют предопределенной иерархии наследования. Вместо этого объекты в Go конструируются с использованием приема *композиции*.

Например, в строгом объектно-ориентированном языке может быть объявлен класс `Car`, расширяющий (наследующий) абстрактный класс `Vehicle`; возможно, класс `Car` добавляет свойства `Wheels` и `Engine`. В теории выглядит неплохо, но подобные отношения между классами быстро могут стать очень запутанными и сложными.

Композиционный подход в Go, напротив, позволяет «соединять» компоненты без необходимости определять их отношения. Продолжая предыдущий пример, в Go можно объявить структуру `Car` и добавить в нее различные части, такие как `Wheels` и `Engine`. Более того, методы в Go можно определить для любого типа данных – не только для структур.

Структуры

Структура в языке Go – это просто некоторый агрегат с несколькими полями, представляющий единую сущность, каждое поле которой является именованным значением произвольного типа. Структуры определяются с помощью следующего синтаксиса: `тип Имя struct`. Структура никогда не может иметь значение `nil`: нулевое значение структуры – это комплекс нулевых значений всех ее полей:

```
type Vertex struct {
    X, Y float64
}

func main() {
    var v Vertex           // Структура не может иметь значение nil
    fmt.Println(v)         // "{0 0}"

    v = Vertex{}           // Явное определение пустой структуры
    fmt.Println(v)         // "{0 0}"

    v = Vertex{1.0, 2.0}   // Определение значений полей по порядку
    fmt.Println(v)         // "{1 2}"

    v = Vertex{Y:2.5}      // Определение значений отдельных полей по их именам
    fmt.Println(v)         // "{0 2.5}"
}
```

Доступ к полям структур производится с использованием стандартной точечной нотации:

```
func main() {
    v := Vertex{X: 1.0, Y: 3.0}
    fmt.Println(v)         // "{1 3}"

    v.X *= 1.5
    v.Y *= 2.5

    fmt.Println(v)         // "{1.5 7.5}"
}
```

Операции со структурами выполняются с помощью ссылок, поэтому Go предоставляет немного синтаксического сахара: к членам структур можно обратиться по указателю на структуру с использованием точечной нотации; в таких случаях указатели разыменовываются автоматически:

```
func main() {
    var v *Vertex = &Vertex{1, 3}
    fmt.Println(v)                // &{1 3}

    v.X, v.Y = v.Y, v.X
    fmt.Println(v)                // &{3 1}
}
```

Здесь `v` – это указатель на структуру `Vertex`, и нам требуется поменять местами значения ее членов `X` и `Y`. Если бы требовалось разыменовывать указатель, то вам пришлось бы написать примерно такой код: `(*v).X, (*v).Y = (*v).Y, (*v).X`, который выглядит довольно жутко. Автоматическое разыменование ссылок на структуры позволяет записать то же самое так: `v.X, v.Y = v.Y, v.X`, что выглядит намного проще.

Методы

Методы в Go – это функции, прикрепленные к типам, включая и структуры. Синтаксис объявления метода очень похож на синтаксис объявления функции, за исключением дополнительного *аргумента получателя* перед именем функции, указывающего на тип, к которому прикрепляется метод. В вызове метода экземпляр типа будет доступен по имени аргумента получателя.

Возьмем для примера наш тип `Vertex` и добавим к нему метод `Square`, указав получателя с именем `v` и типом `*Vertex`:

```
func (v *Vertex) Square() { // Присоединить метод к типу *Vertex
    v.X *= v.X
    v.Y *= v.Y
}

func main() {
    vert := &Vertex{3, 4}
    fmt.Println(vert)      // "&{3 4}"

    vert.Square()
    fmt.Println(vert)      // "&{9 16}"
}
```


Тип получателя играет важную роль: методы, прикрепленные к типу указателя, могут вызываться только относительно указателя на экземпляр этого типа.

Помимо структур можно также создавать свои версии стандартных составных типов – структур, срезов или ассоциативных массивов – и присоединять к ним свои методы. Для примера объявим новый тип `MyMap`, версию

стандартного ассоциативного массива `map[string]int`, и присоединим к нему метод `Length`:

```
type MyMap map[string]int

func (m MyMap) Length() int {
    return len(m)
}

func main() {
    mm := MyMap{"A":1, "B": 2}

    fmt.Println(mm)            // "map[A:1 B:2]"
    fmt.Println(mm["A"])       // "1"
    fmt.Println(mm.Length())   // "2"
}
```

Результатом является новый тип `MyMap`, отображающий строки в целые числа подобно `map[string]int`, но имеющий дополнительный метод `Length`, который возвращает количество элементов в массиве.

Интерфейсы

Интерфейс в языке Go – это просто набор сигнатур методов. Как и в других языках, поддерживающих идею интерфейсов, они используются для обобщенного описания поведения других типов без привязки к деталям реализации. То есть интерфейс можно рассматривать как *контракт*, которого будет придерживаться тип, открывающий двери для мощных методов абстракции.

Например, можно определить интерфейс `Shape`, включающий сигнатуру метода `Area`. Любой тип, который должен удовлетворять требованиям интерфейса `Shape`, обязан иметь метод `Area`, возвращающий значение типа `float64`:

```
type Shape interface {
    Area() float64
}
```

Теперь определим две фигуры, `Circle` и `Rectangle`, удовлетворяющие интерфейсу `Shape`, добавив в каждую из них метод `Area`. Обратите внимание: нам не нужно явно объявлять, что они реализуют интерфейс: если тип обладает всеми методами, определяемыми интерфейсом, он будет *неявно соответствовать* этому интерфейсу. Это позволяет определять типы, соответствующие интерфейсам, которыми вы не владеете или не управляете:

```
type Circle struct {
    Radius float64
}

func (c Circle) Area() float64 {
    return math.Pi * c.Radius * c.Radius
}
```

```
type Rectangle struct {
    Width, Height float64
}

func (r Rectangle) Area() float64 {
    return r.Width * r.Height
}
```

Поскольку обе структуры, `Circle` и `Rectangle`, неявно удовлетворяют интерфейсу `Shape`, мы можем передавать их в любые функции, принимающие параметр типа `Shape`:

```
func PrintArea(s Shape) {
   fmt.Printf("%T's area is %0.2f\n", s, s.Area())
}

func main() {
   r := Rectangle{Width:5, Height:10}
   PrintArea(r)                        // "main.Rectangle's area is 50.00"

   c := Circle{Radius:5}
   PrintArea(c)                        // "main.Circle's area is 78.54"
}
```

Проверка типа

Проверку типа можно применить к экземпляру интерфейса, чтобы «подтвердить» его принадлежность к конкретному типу. Синтаксис имеет вид: `x.(T)`, где `x` – это экземпляр интерфейса, а `T` – проверяемый тип.

Вот пример для интерфейса `Shape` и структуры `Circle`, которые мы использовали выше:

```
var s Shape
s = Circle{}           // s -- это экземпляр типа Shape
c := s.(Circle)        // Проверить, является ли s экземпляром Circle
fmt.Printf("%T\n", c)  // "main.Circle"
```

Пустой интерфейс

Одна из любопытных конструкций – пустой интерфейс: `interface{}`. Пустой интерфейс не определяет никаких методов. Он не несет никакой информации и ничего не говорит о типе[1].

Переменная типа `interface{}` может содержать значение любого типа, что очень полезно, когда код должен обрабатывать значения любого типа. Отличным примером функции, использующей такую стратегию, может служить метод `fmt.Println`.

Однако у этого подхода есть свои недостатки. Работа с пустым интерфейсом требует проверки определенных предположений во время выполнения, в результате чего код становится более хрупким и менее эффективным.

---

[1] Pike, Rob. «Go Proverbs». YouTube. 1 Dec. 2015. https://oreil.ly/g8Rid.

Композиция путем встраивания типов

Go не поддерживает создания подклассов и наследования, как традиционные объектно-ориентированные языки. Вместо этого он позволяет *встраивать* типы друг в друга, расширяя функциональные возможности комбинированных типов возможностями встраиваемых типов.

Эта особенно полезная возможность Go позволяет повторно использовать функциональные возможности через *композицию* – комбинирование возможностей существующих типов для создания новых типов – вместо наследования, избавляя от необходимости создавать сложные иерархии типов, нередко обременяющие традиционные объектно-ориентированные проекты.

Встраивание интерфейсов

Популярный пример встраивания интерфейсов можно найти в пакете `io`. В частности, в виде широко используемых интерфейсов `io.Reader` и `io.Writer`, которые определены следующим образом:

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

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

Но что, если вам понадобится интерфейс с обоими методами, `io.Reader` и `io.Writer`? В таком случае *можно* реализовать третий интерфейс, содержащий методы из обоих интерфейсов, но тогда вам придется согласовать их все. Это не только добавляет ненужные накладные расходы на обслуживание, но также является отличным способом случайно добавить ошибки.

Вместо простого копирования сигнатур методов Go позволяет встроить два существующих интерфейса в третий, который возьмет на себя функции обоих. Синтаксически это делается путем добавления встроенных интерфейсов в качестве анонимных полей, как демонстрирует стандартный интерфейс `io.ReadWriter`:

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

Результатом этой композиции является новый интерфейс, определяющий все методы встроенных в него интерфейсов.

Внутрь интерфейсов можно встраивать только интерфейсы.

Встраивание структур

Поддержка встраивания не ограничивается интерфейсами: структуры тоже можно встраивать друг в друга.

Примером может служить структура из пакета `bufio`, эквивалентная примерам `io.Reader` и `io.Writer` в предыдущем разделе. В частности, она имеет два поля – `bufio.Reader` (которое реализует `io.Reader`) и `bufio.Writer` (которое реализует `io.Writer`). Кроме того, в `bufio` определена реализация `io.ReadWriter`, которая является простой композицией существующих типов `bufio.Reader` и `bufio.Writer`:

```
type ReadWriter struct {
    *Reader
    *Writer
}
```

Как видите, синтаксис встраивания структур идентичен синтаксису встраивания интерфейсов: существующие типы встраиваются в виде безымянных полей. В предыдущем случае `bufio.ReadWriter` встраивает `bufio.Reader` и `bufio.Writer` как типы указателей.

Так же как любые указатели, встроенные указатели на структуры имеют нулевое значение `nil` и должны инициализироваться ссылками на допустимые структуры перед использованием.

Продвижение

Итак, почему композиция выгоднее простого добавления полей структуры? Ответ прост: при встраивании типа его свойства и методы *продвигаются* в комбинированный тип, что позволяет вызывать их непосредственно. Например, метод `Read` интерфейса `bufio.Reader` доступен непосредственно из экземпляра `bufio.ReadWriter`:

```
var rw *bufio.ReadWriter = GetReadWriter()
var bytes []byte = make([]byte, 1024)

n, err := rw.Read(bytes) {
    // некоторые операции
}
```

Вам не нужно знать или беспокоиться о том, что метод `Read` в действительности принадлежит встроенному типу `*bufio.Reader`. Но при этом важно помнить, что получателем при вызове продвинутого метода все еще является встроенный тип, поэтому получателем `rw.Read` будет поле `Reader`, а не сам экземпляр `ReadWriter`.

Прямой доступ к встроенным полям

Иногда требуется напрямую обратиться к встроенному полю. В таких случаях в качестве имени поля следует использовать имя типа поля. В следующем

(несколько надуманном) примере функции UseReader нужно передать *bufio.Reader, но у вас есть только экземпляр *bufio.ReadWriter:

```
func UseReader(r *bufio.Reader) {
    fmt.Printf("We got a %T\n", r)     // "We got a *bufio.Reader"
}

func main() {
    var rw *bufio.ReadWriter = GetReadWriter()
    UseReader(rw.Reader)
}
```

Как видите, здесь используется имя типа поля (Reader), чтобы получить доступ к полю rw.Reader в экземпляре rw типа *bufio.Reader. Этот прием можно использовать для инициализации:

```
rw := &bufio.ReadWriter{Reader: &bufio.Reader{}, Writer: &bufio.Writer{}}
```

Если бы мы создали rw как &bufio.ReadWriter{}, его встроенные поля получили бы значение nil, а этот фрагмент создает *bufio.ReadWriter с полностью инициализированными полями *bufio.Reader и *bufio.Writer. Этот прием редко используется на практике, но он может пригодиться для создания фиктивных объектов в тестах.

Самое интересное: конкуренция

Сложности конкурентного программирования многочисленны и выходят далеко за рамки данной книги. Однако отмечу, что рассуждать о конкуренции сложно и что способ, которым обычно реализуется конкуренция, делает эти рассуждения еще сложнее. В большинстве языков типичный подход к организации взаимодействий между процессами заключается в выделении некоторого общего фрагмента памяти, доступ к которому затем ограничивается посредством блокировок, что приводит к безумно трудным для отладки ошибкам, таким как состояние гонки или взаимоблокировки.

В Go была избрана иная стратегия: в нем предлагаются два примитива поддержки конкуренции – сопрограммы (gorutines) и каналы, – которые можно использовать для структурирования программного обеспечения, выполняющегося конкурентно, и ослабления зависимости от блокировок. Она побуждает разработчиков ограничивать совместное использование памяти и организовывать взаимодействия между процессами путем передачи сообщений.

Сопрограммы

Одна из самых мощных особенностей Go – ключевое слово go. Любой вызов функции, к которому добавлено ключевое слово go, будет выполнен как обычно, но при этом вызывающий сможет продолжить работу, не ожидая,

пока функция завершит работу. За кулисами такая функция выполняется как легковесный параллельный процесс, который называют *сопрограммой*.

Синтаксис поразительно прост: функция `foo`, выполняющаяся последовательно, если ее вызвать как `foo()`, может выполняться конкурентно, как сопрограмма, достаточно лишь добавить в инструкцию вызова ключевое слово `go`:

```
foo()    // Вызвать foo() и ждать ее завершения
go foo() // Запустить новую сопрограмму, которая вызовет foo() конкурентно
```

В сопрограммах можно запускать литералы функций:

```
func Log(w io.Writer, message string) {
    go func() {
        fmt.Fprintln(w, message)
    }() // Не забудьте добавить круглые скобки в конце!
}
```

Каналы

Каналы в языке Go – это типизированные примитивы, поддерживающие возможность взаимодействий двух сопрограмм. Они действуют как трубопроводы, через которые сопрограммы могут пересылать значения друг другу.

Каналы можно создавать с помощью функции `make`. Каждый канал может передавать значения только одного определенного типа, который называют *типом элементов*. Типы каналов записываются с использованием ключевого слова `chan`, за которым следует тип элемента. Вот пример объявления и создания канала типа `int`:

```
var ch chan int = make(chan int)
```

Каналы поддерживают две основные операции: *отправка* и *прием*. Обе записываются с использованием оператора `<-`, где стрелка указывает направление потока данных, как показано ниже:

```
ch <- val  // Отправка в канал
val = <-ch // Прием из канала и запись в переменную val
<-ch       // Прием из канала и отбрасывание результата
```

Блокировка канала

По умолчанию каналы *не буферизуются*. Небуферизованные каналы имеют очень полезное свойство: отправляющая сторона блокируется до тех пор, пока принимающая сопрограмма не прочитает данные из канала, и наоборот, принимающая сторона блокируется, пока отправляющая сопрограмма не запишет значение в канал. Эту особенность можно использовать для синхронизации двух сопрограмм, как показано ниже:

```
func main() {
    ch := make(chan string)    // Создать канал для передачи строк
```

```
    go func() {
        message := <-ch        // Заблокируется при приеме; запишет строку в message
        fmt.Println(message)   // "ping"
        ch <- "pong"           // Заблокируется при передаче
    }()

    ch <- "ping"               // Послать "ping"
    fmt.Println(<-ch)          // "pong"
}
```

Хотя основная (`main`) и анонимная сопрограммы запускаются конкурентно и теоретически могут выполняться в любом порядке, блокирующее поведение небуферизованных каналов гарантирует, что этот код всегда сначала выведет «ping», а потом «pong».

Буферизация каналов

Каналы в Go можно *буферизовать*, и в этом случае они будут содержать внутреннюю очередь значений фиксированной *емкости*, которая указывается при инициализации буфера. Отправка в буферизованный канал блокируется, только когда буфер заполнен; прием из канала блокируется, только когда буфер пуст. В любом другом случае операции отправки и приема записывают или читают значения из буфера немедленно.

Чтобы создать буферизованный канал, достаточно передать функции `make` второй аргумент, определяющий емкость канала:

```
ch := make(chan string, 2) // Буферизованный канал, вмещающий 2 элемента
ch <- "foo"                // Две неблокирующие операции отправки
ch <- "bar"
fmt.Println(<-ch)          // Две неблокирующие операции приема
fmt.Println(<-ch)
fmt.Println(<-ch)          // Третья операция приема заблокируется
```

Закрытие каналов

Третья операция, доступная при работе с каналами, – *закрытие*. Она устанавливает флаг, сообщающий, что канал закрыт и через него больше не будут передаваться значения. Для закрытия канала можно использовать встроенную функцию `close(ch)`.

Операция закрытия канала просто устанавливает флаг, сообщающий получателю, что тот не должен больше ждать новых значений. Вам *не нужно* явно закрывать каналы.

Попытка отправить данные в закрытый канал вызовет панику (аварийное завершение, или крах). При приеме из закрытого канала из него будут извлечены все значения, отправленные до его закрытия; любые последующие операции приема будут возвращать нулевое значение для типа элемента канала. Получатели могут проверить, был ли канал закрыт (и наличие значений в буфере), записав второе значение, возвращаемое операцией приема, в переменную типа `bool`:

```
ch := make(chan string, 10)

ch <- "foo"

close(ch)                          // Одно значение осталось в буфере

msg, ok := <-ch
fmt.Printf("%q, %v\n", msg, ok)  // "foo", true

msg, ok = <-ch
fmt.Printf("%q, %v\n", msg, ok)  // "", false
```


Вообще говоря, закрыть канал может любая из сторон, но на практике это должен делать только отправитель. Непреднамеренная отправка в закрытый канал вызовет панику.

Прием значений из канала в цикле

Для последовательного извлечения значений из открытого канала или из канала, содержащего буферизованные значения, можно использовать ключевое слово `range`. Цикл будет блокироваться до тех пор, пока значение не станет доступно для приема или пока канал не будет закрыт. Вот как это работает:

```
ch := make(chan string, 3)

ch <- "foo"            // Послать три значения в буферизованный канал
ch <- "bar"
ch <- "baz"

close(ch)              // Закрыть канал

for s := range ch {    // range обеспечит продолжение цикла до закрытия канала
    fmt.Println(s)
}
```

В этом примере создается буферизованный канал `ch`, в который отправляются три значения, после чего канал закрывается. Поскольку три значения были отправлены в канал до его закрытия, цикл по значениям в этом канале последовательно извлечет все три строки, после чего завершится.

Если бы канал не был закрыт, цикл остановился бы и ждал, пока в канал не будет отправлено следующее значение, возможно до бесконечности.

select

Инструкция `select` в Go похожа на инструкцию `switch` и обеспечивает удобный механизм для мультиплексирования нескольких каналов. Синтаксис `select` очень похож на синтаксис `switch`: она также включает несколько операторов `case`, которые будут выполняться после успешной операции отправки или получения:

```
select {
case <-ch1:                         // Отбросить принятое значение
    fmt.Println("Got something")
case x := <-ch2:                    // Записать принятое значение в x
    fmt.Println(x)
case ch3 <- y:                      // Отправить y в канал
    fmt.Println(y)
default:
    fmt.Println("None of the above")
}
```

В этом примере перечислены три основных случая с тремя различными условиями. Если канал `ch1` готов к чтению, то его значение будет прочитано (и отброшено), а функция выведет текст «Got something». Если `ch2` готов к чтению, то его значение будет прочитано и присвоено переменной `x`, после чего выведено вызовом функции `fmt.Println(x)`.

Наконец, если `ch3` готов к отправке, то значение `y` будет отправлено в канал и выведено вызовом функции `fmt.Println(y)`.

Наконец, если нет ни одного канала, готового к выполнению соответствующей операции, то выполнится оператор `default`. Инструкция `select` без оператора `default` заблокируется до тех пор, пока какой-нибудь из перечисленных в ней каналов не будет готов к работе, после чего выполнится соответствующий оператор `case`. Если готовыми окажутся сразу несколько каналов, то `select` выберет один из них случайным образом.

Проблема! Используя `select`, имейте в виду, что закрытый канал никогда не блокируется и всегда доступен для приема.

Реализация тайм-аутов для каналов

Применение `select` для мультиплексирования каналов открывает широчайшие возможности и помогает сделать очень сложные или утомительные задачи тривиально простыми. Возьмем, к примеру, реализацию тайм-аута для произвольного канала. В некоторых языках для этого может потребоваться реализовать управление потоками, но `select` с вызовом функции `time.After`, возвращающей канал, через который будет отправлено сообщение после истечения указанного времени, делает эту задачу проще некуда:

```
var ch chan int

select {
case m := <-ch:                      // Прочитать из ch; заблокируется навсегда
    fmt.Println(m)
case <-time.After(10 * time.Second): // time.After вернет канал
    fmt.Println("Timed out")
}
```

Здесь нет оператора `default`, поэтому инструкция `select` заблокируется до выполнения одного из условий. Если канал `ch` не станет доступным для чтения до того, как в канал, возвращаемый функцией `time.After`, будет за-

писано сообщение, то сработает второй оператор `case` и инструкция `select` завершится по тайм-ауту.

Итоги

Подробное обсуждение всего, о чем рассказывалось в этой главе, потребовало бы отдельной книги. Но пространство и время ограничены (к тому же такая книга уже написана[1]), поэтому я был вынужден постараться уместить все в одну главу и дать вам только поверхностный обзор языка Go (по крайней мере, до выхода второго издания).

Но знания одного только синтаксиса и грамматики Go недостаточно. В главе 4 я представлю различные шаблоны программирования на Go, которые, как я успел убедиться, довольно часто встречаются в контексте разработки облачных приложений. Итак, если вы посчитали эту главу интересной, то следующая понравится вам еще больше.

[1] И еще раз, если вы не читали книгу «The Go Programming Language» Донавана и Кернигана, то обязательно прочитайте ее. (*Донован Алан А. А., Керниган Брайан У.* Язык программирования Go. М.: Вильямс, 2018. ISBN: 978-5-907114-21-0. – *Прим. перев.*)

Глава 4

Шаблоны программирования облачных приложений

> Прогресс возможен только в том случае, если мы научимся думать о программах, не думая о них как о фрагментах выполняемого кода[1].
>
> – Эдсгер Вибе Дейкстра (Edsger W. Dijkstra), август 1979

В 1991 году, еще работая в Sun Microsystems, Л Питер Дойч (L Peter Deutsch)[2] сформулировал список *заблуждений распределенных вычислений*, в котором перечислил некоторые из ложных предположений, которые часто делают программисты, незнакомые (и малознакомые) с распределенными приложениями:

- *сеть надежна*: коммутаторы иногда выходят из строя, маршрутизаторы могут быть неправильно настроены;
- *задержка равна нулю*: для передачи данных по сети требуется время;
- *пропускная способность бесконечна*: сеть может обрабатывать только определенное количество данных за раз;
- *сеть безопасна*: не передавайте конфиденциальную информацию открытым текстом; шифруйте все;
- *топология не меняется*: серверы и службы появляются и исчезают;
- *есть только один администратор*: несколько администраторов могут принимать противоречивые решения;
- *транспортные расходы равны нулю*: передача данных требует времени и денег;
- *сеть однородна*: каждая сеть отличается (иногда очень) от других.

---

[1] Произнесено в августе 1979 года. Авторство цитаты подтверждено Вики Альмструмом (Vicki Almstrum), Тони Хоаром (Tony Hoare), Никлаусом Виртом (Niklaus Wirth), Вимом Фейеном (Wim Feijen) и Радживом Джоши (Rajeev Joshi). В поисках простоты: симпозиум в честь профессора Эдсгера Вибе Дейкстры, 12–13 мая 2000 г.

[2] Л (да, его действительно зовут Л (L)) – блестящий и очаровательный человек. Будет возможность, обязательно встретьтесь с ним.

Если позволите, я бы хотел добавить еще одно заблуждение:

- *службы надежны*: службы, от которых вы зависите, могут прекратить работу в любой момент.

В этой главе я представлю набор идиоматических шаблонов программирования, проверенных на практике и предназначенных для решения заблуждений, описанных Дойчем, и продемонстрирую их реализацию на Go. Ни один из шаблонов, обсуждаемых в этой книге, не является уникальным для этой книги – некоторые появились вместе с появлением распределенных приложений, – но все вместе они никогда не публиковались в одной книге. Многие из них уникальны для Go или имеют уникальные реализации на Go, отличные от реализаций на других языках.

К сожалению, в этой книге не рассматриваются инфраструктурные шаблоны, такие как Bulkhead (Герметичные отсеки; https://oreil.ly/0hxmU) или Gatekeeper (Привратник; https://oreil.ly/0v5Jc). Во многом это связано с тем, что при разработке на Go основное внимание мы уделяем прикладному уровню, тогда как эти шаблоны относятся к совершенно другому уровню абстракции. Если вам интересно узнать больше, я рекомендую прочитать книгу Джастина Гаррисона (Justin Garrison) и Криса Новы (Kris Nova) *Cloud Native Infrastructure* (O'Reilly) и книгу Брендана Бернса (Brendan Burns) *Designing Distributed Systems*[1] (O'Reilly).

Пакет context

В большинстве примеров кода в этой главе используется пакет `context`, появившийся в Go 1.7 и содержащий идиоматические средства передачи между процессами крайних сроков, сигналов отмены и значений в контексте запроса. Он содержит единственный интерфейс `context.Context`, методы которого перечислены ниже:

```
type Context interface {
    // Done возвращает канал, который закрывается
    // с прекращением действия контекста.
    Done() <-chan struct{}

    // Err сообщает причину прекращения действия контекста после закрытия
    // канала Done. Если Done еще не закрыт, то Err возвращает nil.
    Err() error

    // Deadline возвращает время, когда данный контекст должен прекратить
    // действовать; он возвращает ok==false, если крайний срок не установлен.
    Deadline() (deadline time.Time, ok bool)

    // Value возвращает значение, связанное с ключом key в данном контексте,
    // или nil, если с ключом не связано никакое значение. Используйте этот
```

[1] *Бернс Брендан*. Распределенные системы. Паттерны проектирования. СПб.: Питер, 2019. ISBN: 978-5-4461-0950-0. – *Прим. перев.*

```
    // метод с большой осторожностью.
    Value(key interface{}) interface{}
}
```

С помощью первых трех методов можно получить информацию о состоянии или поведении контекста. Четвертый метод, `Value`, позволяет получить значение, связанное с произвольным ключом. Метод `Value` является предметом споров в мире Go, и мы подробно обсудим его в разделе «Определение значений в контексте запроса» ниже.

Что может дать контекст

Экземпляр `context.Context` передается службе в запросе, которая, в свою очередь, может передать его другим службам в подзапросах. Удобство контекстов состоит в том, что когда контекст прекращает действовать, все функции, удерживающие его (или производный контекст; подробнее об этом на рис. 4.1, 4.2 и 4.3), получат сигнал отмены и смогут скоординировать прекращение операций и предотвратить напрасное расходование вычислительных и других ресурсов.

Возьмем, например, запрос, посылаемый пользователем службе, которая, в свою очередь, посылает запрос к базе данных. В идеальном сценарии запросы пользователя, приложения и базы данных можно представить в виде диаграммы на рис. 4.1.

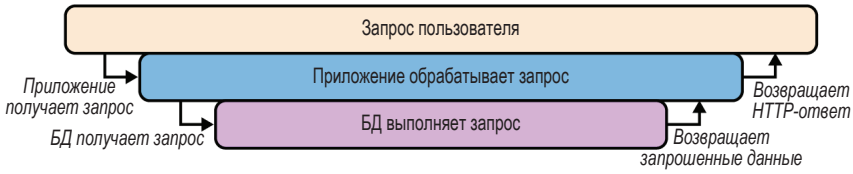


Рис. 4.1 ❖ Успешное выполнение запроса от пользователя к службе и от службы к базе данных

Но что, если пользователь отменил запрос до того, как он был полностью обработан? В большинстве случаев его обработка продолжится, несмотря ни на что (рис. 4.2), и приложение потратит ресурсы на получение результатов, которые уже стали ненужными.

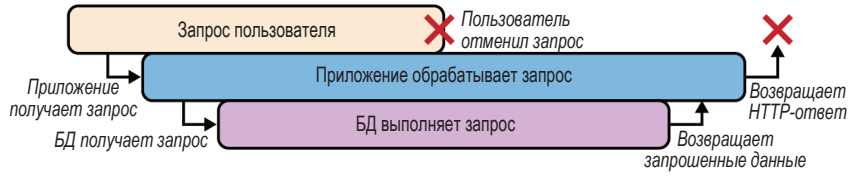


Рис. 4.2 ❖ Подпроцессы продолжат обработку запроса, не зная о его отмене

Однако, имея общий контекст Context для каждого запроса, всем процессам, участвующим в обработке запроса, можно передать сигнал «стоп», получив который, они смогут скоординированно прекратить обработку (рис. 4.3).

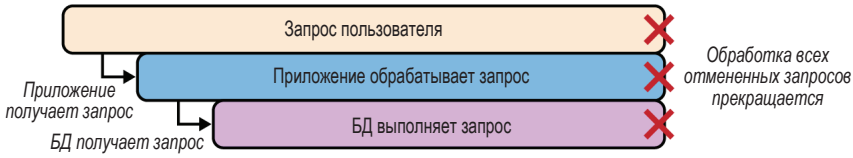


Рис. 4.3 ❖ При наличии общего контекста можно всем процессам послать сигнал отмены

Важно отметить, что экземпляры Context являются также потокобезопасными, то есть их можно использовать в нескольких одновременно выполняющихся сопрограммах, не опасаясь конфликтов.

Создание контекста

Новый экземпляр context.Context можно получить с помощью одной из двух функций:

`func Background() Context`
: Возвращает пустой контекст, который никогда не отменялся, не имеет значений и не имеет крайнего срока действия. Обычно вызывается в функции main, в функциях инициализации и в тестах и возвращает контекст верхнего уровня для входящих запросов.

`func TODO() Context`
: Тоже возвращает пустой контекст, но предназначенный для использования в качестве заполнителя, когда не ясно, какой контекст использовать, или когда родительский контекст еще недоступен.

Определение крайних сроков и тайм-аутов контекста

Пакет context также включает ряд методов для создания *производных* контекстов для управления поведением отмены или для определения тайм-аута либо обработчика, который может явно инициировать отмену.

`func WithDeadline(Context, time.Time) (Context, CancelFunc)`
: Принимает конкретное время, когда действие контекста должно быть прекращено, а канал Done – закрыт.

`func WithTimeout(Context, time.Duration) (Context, CancelFunc)`
Принимает интервал времени, по истечении которого действие контекста будет прекращено, а канал `Done` – закрыт.

`func WithCancel(Context) (Context, CancelFunc)`
В отличие от предыдущих функций, `WithCancel` ничего не принимает и возвращает только функцию, которую можно вызвать, чтобы явно отменить действие контекста.

Все три функции возвращают производный контекст `Context`, включающий любые запрошенные атрибуты, и функцию `context.CancelFunc` без параметров, которую можно вызвать для явной отмены данного контекста и всех его производных.


Когда контекст отменяется, все контексты, производные от него, тоже отменяются, а контексты, из которых был получен этот контекст, – нет.

Определение значений в контексте запроса

Наконец, пакет `context` включает функцию, которую можно использовать для определения произвольной пары ключ/значение *в контексте запроса*, значение которой можно извлечь с помощью метода `Value` из данного экземпляра контекста `Context` и из всех производных контекстов, полученных из него.

`func WithValue(parent Context, key, val interface{}) Context`
`WithValue` возвращает производный контекст, в котором ключ `key` связан со значением `val`.

О значениях в контекстах

Функции `context.WithValue` и `context.Value` позволяют определять и получать произвольные пары ключ/значение, которые могут использоваться процессами, обрабатывающими запросы. Однако, по утверждениям разработчиков, эта функциональность ортогональна функциональности отмены долгоживущих запросов, усложняет анализ потока выполнения вашей программы и может легко нарушить связи во время компиляции. Более подробно данный вопрос обсуждается в статье «Context Is for Cancelation» (https://oreil.ly/DaGN1) в блоге Дэйва Чейни (Dave Cheney).

Эта возможность не используется ни в одном из примеров в этой главе (и в этой книге). Если вы решите использовать ее, убедитесь, что все ваши значения относятся только к запросу и не влияют на работу каких-либо процессов.

Использование контекста

Когда запрос к службе инициируется входящим запросом или запускается функцией `main`, процесс верхнего уровня вызовет функцию `Background` и создаст новый экземпляр контекста `Context`, возможно, добавив в него дополни-

тельные атрибуты вызовом функций context.With*, прежде чем передать его в подзапросы. Таким подзапросам достаточно просто наблюдать за каналом Done, чтобы вовремя среагировать на сигнал отмены.

Например, взгляните на следующую функцию Stream:

```
func Stream(ctx context.Context, out chan<- Value) error {
    // Создать производный контекст с 10-секундным тайм-аутом;
    // dctx прекратит действовать спустя это время, но ctx -- нет.
    // cancel -- это функция, которую можно вызвать, чтобы явно
    // прекратить действие dctx.
    dctx, cancel := context.WithTimeout(ctx, time.Second * 10)

    // Освободить ресурсы, если SlowOperation завершилась
    // до истечения тайм-аута
    defer cancel()

    res, err := SlowOperation(dctx)
    if err != nil { // Если тайм-аут dctx истек
        return err
    }

    for {
        select {
        case out <- res:   // Прочитать из res; передать в out

        case <-ctx.Done(): // Выполнится при отмене ctx
            return ctx.Err()
        }
    }
}
```

Stream получает аргумент контекста ctx и передает его в WithTimeout, чтобы создать dctx – производный контекст с 10-секундным тайм-аутом. При таком подходе вызов SlowOperation(dctx) может завершиться по тайм-ауту через десять секунд и вернуть ошибку. Однако функции, использующие исходный контекст ctx без тайм-аута, продолжат выполнение как обычно.

Далее исходный экземпляр ctx используется в цикле for, окружающем инструкцию select, которая извлекает значения из канала res, полученного от функции SlowOperation. Обратите внимание на оператор case <-ctx.Done(), он выполняется при закрытии канала ctx.Done и возвращает соответствующее значение ошибки.

Структура этой главы

Структура описания каждого шаблона программирования в этой главе заимствована из знаменитой книги *Design Patterns*, написанной «бандой четырех»[1], и несколько упрощена. Описание каждого шаблона начинается

[1] Erich Gamma et al. «Design Patterns: Elements of Reusable Object-Oriented Software»,

с очень краткого представления его назначения и причин использования, за которым следуют разделы, перечисленные ниже:

Применимость
Контекст и порядок применения этого шаблона.

Компоненты
Список компонентов шаблона и их назначение.

Реализация
Обсуждение решения и его реализация.

Пример кода
Демонстрация одного из вариантов реализации на языке Go.

Шаблоны стабильности

Представленные здесь шаблоны стабильности предназначены для разрешения заблуждений распределенных вычислений. Обычно они применяются в распределенных приложениях для повышения их стабильности и стабильности всей системы, частью которой они являются.

Circuit Breaker (Размыкатель цепи)

Шаблон Circuit Breaker (Размыкатель цепи) автоматически отключает сервисные функции в ответ на вероятную неисправность, чтобы предотвратить более крупные или каскадные отказы, устранить повторяющиеся ошибки и обеспечить разумную реакцию на ошибки.

Применимость

Если заблуждения распределенных вычислений свести к одному пункту, то его можно сформулировать так: ошибки и сбои являются неоспоримым фактом в жизни распределенных облачных систем. Службы могут быть неправильно настроены, базы данных могут выходить из строя, сети – прекратить работать. Мы не можем этого предотвратить; мы можем только принять и учесть это.

Игнорирование вероятности ошибок и сбоев может иметь довольно неприятные последствия. Мы все сталкивались с чем-то подобным и видели, насколько это плохо. Одни службы могут продолжать безуспешно пытаться выполнять свою работу и возвращать клиентам бессмыслицу; другие впадают в смертельную спираль сбоя/перезапуска. Но все это не имеет значения,

1st edition. Addison-Wesley Professional, 1994. (*Гамма Эрих, Хелм Ричард, Джонсон Ральф, Влиссидес Джон*. Паттерны объектно-ориентированного проектирования. СПб.: Питер, 2020. ISBN: 978-5-4461-1595-2. – *Прим. перев.*)

потому что в любом случае службы тратят ресурсы, скрывая причину отказа и делая каскадные отказы еще более вероятными.

Служба, спроектированная с учетом возможного отказа ее зависимостей в любой момент, может разумно реагировать на такие ситуации. Шаблон Circuit Breaker (Размыкатель цепи) позволяет обнаруживать такие сбои и «размыкать цепь», временно прекращая обработку запросов и возвращая клиентам сообщение об ошибке в соответствии с контрактом службы.

Например, представьте службу, которая (в идеале) получает запрос от клиента, выполняет запрос к базе данных и возвращает ответ. Но как быть, если база данных выйдет из строя? Служба может продолжать безуспешно пытаться передать запрос базе данных, переполняя журналы сообщениями об ошибках, и в конечном итоге прекратить работу или вернуть бесполезное сообщение об ошибке. Такая служба может использовать размыкатель цепи при сбое базы данных, чтобы избежать повторных бессмысленных попыток выполнить запрос (по крайней мере, на некоторое время) и немедленно отправить клиенту осмысленное уведомление.

Компоненты

Этот шаблон включает следующие компоненты:

Circuit
: Функция, взаимодействующая со службой.

Breaker
: Функция-замыкание с той же сигнатурой, что и функция `Circuit`.

Реализация

По сути, Circuit Breaker (Размыкатель цепи) – это просто специализированный шаблон Adapter (Адаптер; https://oreil.ly/bEeru), в котором функция размыкателя `Breaker` обертывает функцию `Circuit` и добавляет дополнительную логику обработки ошибок.

Подобно электрическому выключателю, от которого этот шаблон получил свое название, `Breaker` имеет два возможных состояния: *замкнуто* и *разомкнуто*. В замкнутом состоянии все работает как обычно. Все запросы, полученные от клиента с помощью `Breaker`, передаются в `Circuit` без изменений, а все ответы от `Circuit` возвращаются обратно клиенту. В разомкнутом состоянии `Breaker` не передает запросы в `Circuit`, а просто «быстро терпит неудачу», возвращая информативное сообщение об ошибке.

`Breaker` следит за ошибками, которые возвращает `Circuit`; если количество ошибок, полученных подряд, превысит определенный порог, то `Breaker` срабатывает и переходит в состояние *разомкнуто*.

Большинство реализаций шаблона Circuit Breaker (Размыкатель цепи) включает некоторую логику для автоматического замыкания цепи через некоторый период времени. Однако имейте в виду, что большое количество повторных попыток обращения к уже неисправной службе может вызвать дополнительные проблемы, поэтому общепринято использовать некоторую логику *постепенного увеличения задержки*, снижающую частоту повторных

попыток с течением времени. Тема увеличения задержки на самом деле довольно скользкая, и мы подробно рассмотрим ее в разделе «Повтори еще раз: повторные запросы» в главе 9.

В распределенной службе эту реализацию можно расширить за счет включения некоторого механизма общего хранилища, такого как сетевой кеш Memcached или Redis, для хранения состояния цепи.

Пример кода

Начнем с определения типа `Circuit`, определяющего сигнатуру функции, которая взаимодействует с базой данных или другой вышестоящей службой. На практике эта сигнатура может быть любой, соответствующей вашим требованиям. Однако она должна включать ошибку в список возвращаемых значений:

```
type Circuit func(context.Context) (string, error)
```

В этом примере функция `Circuit` принимает экземпляр интерфейса `Context`, который был подробно описан в разделе «Пакет context» выше. Ваша реализация может отличаться.

Функция `Breaker` принимает любую функцию, соответствующую определению типа `Circuit`, и целое число без знака, представляющее количество отказов, следующих друг за другом, после которых должно произойти автоматическое размыкание цепи. В ответ она возвращает другую функцию, которая также соответствует определению типа `Circuit`:

```
func Breaker(circuit Circuit, failureThreshold uint) Circuit {
    var consecutiveFailures int = 0
    var lastAttempt = time.Now()
    var m sync.RWMutex

    return func(ctx context.Context) (string, error) {
        m.RLock()                       // Установить "блокировку чтения"

        d := consecutiveFailures - int(failureThreshold)

        if d >= 0 {
            shouldRetryAt := lastAttempt.Add(time.Second * 2 << d)
            if !time.Now().After(shouldRetryAt) {
                m.RUnlock()
                return "", errors.New("service unreachable")
            }
        }

        m.RUnlock()                     // Освободить блокировку чтения

        response, err := circuit(ctx)   // Послать запрос, как обычно

        m.Lock()                        // Заблокировать общие ресурсы
        defer m.Unlock()

        lastAttempt = time.Now()        // Зафиксировать время попытки

        if err != nil {                 // Если Circuit вернула ошибку,
```

```
            consecutiveFailures++        // увеличить счетчик ошибок
            return response, err         // и вернуть ошибку
        }

        consecutiveFailures = 0          // Сбросить счетчик ошибок

        return response, nil
    }
}
```

Функция `Breaker` создает другую функцию, тоже типа `Circuit`, которая обертывает `circuit`, чтобы придать ей желаемую функциональность. После прочтения раздела «Анонимные функции и замыкания» в главе 3 вы без труда опознаете в ней замыкание: вложенную функцию, обращающуюся к переменным в родительской функции. Как вы увидите далее, все шаблоны «стабильности», представленные в этой главе, реализованы именно так.

Замыкание подсчитывает количество последовательных ошибок, возвращаемых функцией `circuit`. Когда это число достигает порога, то замыкание начинает возвращать ошибку `"service unreachable"` без фактического вызова `circuit`. После любого успешного вызова `circuit` счетчик `sequenceFailures` сбрасывается в 0, и цикл начинается снова.

Замыкание даже включает механизм автоматического сброса, который позволяет производить повторные серии попыток через несколько секунд с экспоненциальным увеличением задержки между сериями – после каждой неудачной серии длительность задержки увеличивается примерно вдвое. Это простой и довольно распространенный алгоритм увеличения задержки, но далеко не идеальный, а почему не идеальный, вы узнаете в разделе «Алгоритмы увеличения задержки» главы 9.

Debounce (Антидребезг)

Шаблон Debounce (Антидребезг) ограничивает частоту вызова функции, вследствие чего выполняется только первый или последний вызов в группе.

Применимость

Debounce (Антидребезг) – это второй из наших шаблонов, получивший свое название из мира электрических цепей. Он назван в честь явления, когда контакты переключателя «дребезжат» при замыкании или размыкании цепи, в результате чего в электрической схеме возникают колебания, прежде чем она перейдет в стабильное состояние. Обычно в этом нет ничего страшного, но иногда «дребезг контактов» может стать настоящей проблемой в логических схемах, где последовательность импульсов включения/выключения может интерпретироваться как поток данных. Практика устранения дребезга, когда при замыкании или размыкании контакта передается только один сигнал, называется «антидребезгом».

В мире служб иногда можно обнаружить, что они выполняют целую последовательность потенциально медленных или дорогостоящих операций,

из которых достаточно выполнить только одну. Используя шаблон Debounce (Антидребезг), можно ограничить ряд похожих вызовов, плотно сгруппированных во времени, только одним вызовом, обычно первым или последним в пакете.

Этот метод использовался в мире JavaScript в течение многих лет для ограничения количества операций, которые могут замедлить браузер: он принимает только первое в серии пользовательских событий или задерживает вызов до тех пор, пока пользователь не будет готов. Вероятно, вы уже видели применение данного приема на практике. Все мы знакомы со строкой поиска, всплывающее окно с вариантами автозаполнения которой не отображается до тех пор, пока вы не приостановите ввод текста, еще один пример – игнорирование частых щелчков мышью на кнопке и обработка только последнего из них после истечения некоторого времени.

Те, кто специализируется на серверных службах, могут многому научиться у наших собратьев, занимающихся разработкой пользовательских интерфейсов, годами решающих проблемы надежности, задержек и пропускной способности, присущие распределенным системам. Этот подход, например, можно использовать для извлечения некоторых редко обновляющихся удаленных ресурсов без зависания и пустой траты времени клиента и сервера на бесполезные запросы.

Данный шаблон похож на шаблон Throttle (Дроссельная заслонка), представленный ниже в этой главе, который тоже ограничивает частоту вызова функций. Но, в отличие от шаблона Debounce (Антидребезг), который ограничивает пакеты вызовов, шаблон Throttle (Дроссельная заслонка) просто ограничивает частоту вызовов. Подробнее о разнице между шаблонами Debounce (Антидребезг) и Throttle (Дроссельная заслонка) рассказывается во врезке «Отличия между шаблонами Throttle и Debounce» ниже.

Компоненты

Этот шаблон включает следующие компоненты:

Circuit
: Функция, частота вызовов которой должна ограничиваться.

Debounce
: Функция-замыкание с той же сигнатурой, что и функция `Circuit`.

Реализация

Реализация шаблона Debounce (Антидребезг) на самом деле очень похожа на реализацию шаблона Circuit Breaker (Размыкатель цепи) – в нем точно так же производится обертывание функции `Circuit`, чтобы добавить в нее логику ограничения частоты вызовов. Эта логика на самом деле довольно проста: при каждом вызове внешней функции – независимо от результата – устанавливается временной интервал. Любой последующий вызов, выполненный до истечения этого интервала, игнорируется; а вызов, выполненный после этого интервала, передается внутренней функции. Эта реализация, в которой

внутренняя функция вызывается один раз при первом обращении, а последующие вызовы игнорируются, называется «по первому вызову» и полезна тем, что позволяет кешировать и возвращать первый ответ, полученный от внутренней функции.

Реализация «по последнему вызову» будет ждать завершения серии вызовов, прежде чем вызовет внутреннюю функцию. Этот вариант распространен в мире JavaScript, когда программисту требуется получить определенный объем ввода, прежде чем вызвать функцию, например когда строка поиска ожидает паузы при вводе текста перед выводом вариантов для автозаполнения. Стратегия «по последнему вызову», как правило, менее распространена в серверных службах, потому что не дает немедленного ответа, но может пригодиться, если вашей функции не требуется немедленно получить результат.

Пример кода

Так же как при обсуждении шаблона Circuit Breaker (Размыкатель цепи), начнем с определения сигнатуры функции, которую мы хотим ограничить. Снова назовем этот тип `Circuit`; он идентичен типу в предыдущем примере. Отмечу еще раз, что сигнатура может быть любой, соответствующей вашим требованиям, и должна включать ошибку в список возвращаемых значений

```
type Circuit func(context.Context) (string, error)
```

Сходство с реализацией шаблона Circuit Breaker (Размыкатель цепи) не случайно: такая совместимость позволяет использовать их в паре, как показано ниже:

```
func myFunction func(ctx context.Context) (string, error) { /* ... */ }

wrapped := Breaker(Debounce(myFunction))
response, err := wrapped(ctx)
```

Вариант «по первому вызову» шаблона Debounce (Антидребезг) – `DebounceFirst` – реализуется намного проще, чем вариант «по последнему вызову», потому что достаточно хранить лишь время последнего вызова и просто возвращать кешированный результат, если следующая попытка вызова была предпринята в пределах заданного интервала времени:

```
func DebounceFirst(circuit Circuit, d time.Duration) Circuit {
    var threshold time.Time
    var result string
    var err error
    var m sync.Mutex

    return func(ctx context.Context) (string, error) {
        m.Lock()

        defer func() {
            threshold = time.Now().Add(d)
            m.Unlock()
        }()
```

```
        if time.Now().Before(threshold) {
            return result, err
        }

        result, err = circuit(ctx)

        return result, err
    }
}
```

Данная реализация `DebounceFirst` делает все возможное, чтобы добиться максимальной безопасности в многопоточном окружении, заключая всю функцию в мьютекс. Это заставит перекрывающиеся вызовы в начале пакета ждать окончания кеширования результата, а также гарантирует, что `circuit` будет вызвана ровно один раз, в самом начале пакета. Ключевое слово `defer` гарантирует сброс значения `treshhold`, представляющего время завершения пакета вызовов, при каждом следующем фактическом вызове `circuit`.

Реализация «по последнему вызову» более запутанная, потому что включает использование `time.Ticker`, чтобы определить, достаточно ли прошло времени с момента последней попытки вызвать функцию, и действительно вызвать `circuit`. Как вариант можно создавать новый экземпляр `time.Ticker` при каждом вызове, но это будет стоить очень дорого, если вызовы будут следовать слишком часто:

```
type Circuit func(context.Context) (string, error)

func DebounceLast(circuit Circuit, d time.Duration) Circuit {
    var threshold time.Time = time.Now()
    var ticker *time.Ticker
    var result string
    var err error
    var once sync.Once
    var m sync.Mutex

    return func(ctx context.Context) (string, error) {
        m.Lock()
        defer m.Unlock()

        threshold = time.Now().Add(d)

        once.Do(func() {
            ticker = time.NewTicker(time.Millisecond * 100)

            go func() {
                defer func() {
                    m.Lock()
                    ticker.Stop()
                    once = sync.Once{}
                    m.Unlock()
                }()

                for {
                    select {
```

```
                case <-ticker.C:
                    m.Lock()
                    if time.Now().After(threshold) {
                        result, err = circuit(ctx)
                        m.Unlock()
                        return
                    }
                    m.Unlock()
                case <-ctx.Done():
                    m.Lock()
                    result, err = "", ctx.Err()
                    m.Unlock()
                    return
                }
            }
        }()
    })

    return result, err
  }
}
```

Как и `DebounceFirst`, реализация `DebounceLast` использует порог `treshold` для обозначения конца пакета вызовов (при условии отсутствия дополнительных вызовов). Однако на этом сходство заканчивается.

Обратите внимание, что почти вся функция выполняется внутри метода `Do` экземпляра `sync.Once`, это гарантирует, что функция будет вызвана *ровно один раз*. Внутри этого блока `time.Ticker` используется для проверки преодоления порога и вызова `circuit`. Наконец, `time.Ticker` останавливается, `sync.Once` сбрасывается, и цикл подготавливается к повторению.

Retry (Повтор)

Шаблон Retry (Повтор) учитывает возможный временный характер ошибки в распределенной системе и осуществляет повторные попытки выполнить неудачную операцию.

Применимость

Временный характер ошибки – это реальность при работе со сложными распределенными системами. Неудача может быть вызвана любым количеством (будем надеяться) временных факторов, особенно если нижестоящая служба или сетевой ресурс реализует стратегию защиты, такую как регулирование, которая временно отклоняет запросы при высокой рабочей нагрузке, или адаптивную стратегию, такую как автоматическое масштабирование, увеличивающее емкость при необходимости.

Подобные сбои обычно исчезают сами собой через некоторое время, поэтому повторная попытка после разумной задержки вероятно (но не гарантированно) будет успешной. Отказ от учета возможности кратковременных

неисправностей может привести к излишней хрупкости системы. И напротив, реализация стратегии автоматического повтора может значительно повысить стабильность службы, что принесет пользу как ей, так и ее вышестоящим потребителям.

Компоненты

Этот шаблон включает следующие компоненты:

Effector
: Функция, взаимодействующая со службой.

Retry
: Функция, принимающая *Effector* и возвращающая замыкание с той же сигнатурой, что и *Effector*.

Реализация

Этот шаблон действует подобно шаблону Circuit Breaker (Размыкатель цепи) или Debounce (Антидребезг) в том смысле, что так же объявляет тип *Effector*, определяющий сигнатуру функции. Эта сигнатура может быть любой, соответствующей вашим требованиям, то есть функция, выполняющая потенциально неудачную операцию, должна соответствовать сигнатуре, определяемой типом *Effector*.

Функция *Retry* принимает пользовательскую функцию с сигнатурой *Effector* и возвращает функцию-замыкание *Effector*, которая обертывает пользовательскую функцию и добавляет дополнительную логику, выполняющую повторные попытки. Кроме пользовательской функции, *Retry* также принимает целое число `retries`, определяющее максимальное количество повторных попыток, и значение `time.Duration`, определяющее задержку между попытками. Если параметр `retries` равен 0, то повторные попытки выполняться не будут.

Обычно логика повторных попыток включает некоторый алгоритм увеличения задержки, но в этом примере она отсутствует.

Пример кода

Сигнатура функции `Effector`, которая передается в `Retry`, выглядит точно так же, как сигнатуры в предыдущих шаблонах:

```
type Effector func(context.Context) (string, error)
```

Сама функция `Retry` выглядит просто, по крайней мере в сравнении с предыдущими функциями:

```
func Retry(effector Effector, retries int, delay time.Duration) Effector {
    return func(ctx context.Context) (string, error) {
```

```
        for r := 0; ; r++ {
            response, err := effector(ctx)
            if err == nil || r >= retries {
                return response, err
            }

            log.Printf("Attempt %d failed; retrying in %v", r + 1, delay)

            select {
            case <-time.After(delay):
            case <-ctx.Done():
                return "", ctx.Err()
            }
        }
    }
}
```

Возможно, вы сами заметили, почему функция `Retry` имеет такую стройную реализацию: она точно так же возвращает функцию, но у этой функции нет внешнего состояния, а это означает, что нет необходимости использовать сложные механизмы поддержки конкуренции.

Чтобы опробовать `Retry`, нужно реализовать функцию, которая выполняет потенциально неудачную операцию, чья сигнатура соответствует типу `Effector`; выберем на эту роль функцию `EmulateTransientError`:

```
var count int

func EmulateTransientError(ctx context.Context) (string, error) {
    count++

    if count <= 3 {
        return "intentional fail", errors.New("error")
    } else {
        return "success", nil
    }
}

func main() {
    r := Retry(EmulateTransientError, 5, 2*time.Second)

    res, err := r(context.Background())

    fmt.Println(res, err)
}
```

Функция `main` передает функцию `EmulateTransientError` в `Retry` и сохраняет полученную функцию в переменной `r`. Когда будет вызвана функция `r`, она вызовет `EmulateTransientError` и повторит попытку после задержки, если `EmulateTransientError` вернет ошибку, в соответствии с логикой повтора, показанной выше. Наконец, после четвертой попытки `EmulateTransientError` возвращает `nil` в качестве ошибки и завершает работу.

Throttle (Дроссельная заслонка)

Шаблон Throttle (Дроссельная заслонка) ограничивает частоту вызовов функции некоторым предельным числом вызовов в единицу времени.

Применимость

Шаблон Throttle (Дроссельная заслонка) назван в честь устройства, регулирующего поток жидкости, например топлива, поступающего в двигатель автомобиля. Как и его одноименный механизм, шаблон Throttle (Дроссельная заслонка) ограничивает количество вызовов функции в течение определенного периода времени. Например:

- пользователю может быть разрешено обращаться к службе не чаще, чем 10 раз в секунду;
- клиенту может быть позволено вызывать определенную функцию только один раз в каждые 500 миллисекунд;
- учетной записи может быть разрешено только три неудачные попытки входа в систему в течение 24 часов.

Наиболее распространенной причиной применения шаблона Throttle (Дроссельная заслонка) является устранение резких всплесков активности, способных привести к насыщению системы необоснованным количеством дорогостоящих запросов, которые могут привести к ухудшению качества обслуживания и в конечном итоге к отказу. Система может поддерживать автоматическое масштабирование для удовлетворения потребностей пользователей, но на это требуется время, и система может быть не в состоянии реагировать достаточно быстро.

Отличия между шаблонами Throttle и Debounce

Концептуально шаблоны Debounce (Антидребезг) и Throttle (Дроссельная заслонка) выглядят очень похожими. В конце концов, оба преследуют цель сократить количество вызовов в единицу времени. Однако, как показано на рис. 4.4, точная синхронизация каждого из них немного отличается:

- *Throttle* (*Дроссельная заслонка*) действует подобно дроссельной заслонке в автомобиле, ограничивая количество топлива, поступающего в двигатель, некоторым максимальным значением. Это показано на рис. 4.4: независимо от того, сколько попыток вызова функции Input будет выполнено, логика Throttle позволит вызвать ее не более установленного количества раз в единицу времени;
- *Debounce* (*Антидребезг*) анализирует пакетные вызовы, следя за тем, чтобы функция вызывалась только один раз в ответ на пакет вызовов – в начале или в конце пакета. На рис. 4.4 проиллюстрирована реализация «по первому вызову»: для каждого из двух пакетов вызовов функции Input шаблон Debounce позволит выполнить только один вызов в начале каждого пакета.

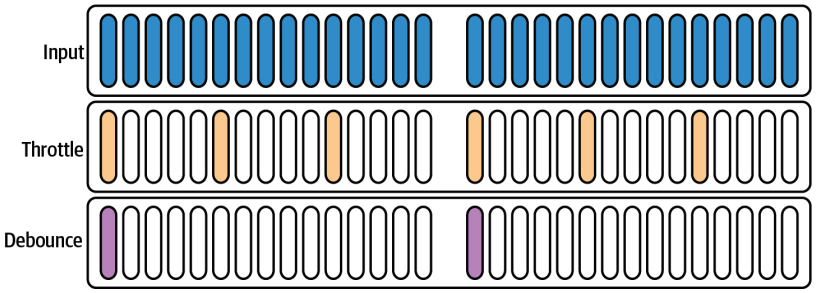


Рис. 4.4 ❖ Throttle ограничивает частоту события; Debounce пропускает только одно событие из пакета

Компоненты

Этот шаблон включает следующие компоненты:

Effector
: Функция, частоту вызовов которой нужно ограничить.

Throttle
: Функция, принимающая *Effector* и возвращающая замыкание с той же сигнатурой, что и *Effector*.

Реализация

Шаблон Throttle (Дроссельная заслонка) похож на многие другие шаблоны, описанные в этой главе: он реализован как функция, которая принимает функцию *Effector* и возвращает замыкание с той же сигнатурой, которое реализует логику ограничения частоты вызовов.

Чаще всего для реализации ограничения скорости используется алгоритм *корзины жетонов*[1] (*token bucket*; https://oreil.ly/5A5aP), по аналогии с корзиной, в которой можно хранить не больше некоторого максимального числа жетонов. Когда вызывается функция, из корзины извлекается один жетон, при этом корзина пополняется новыми жетонами с некоторой фиксированной скоростью.

Как поступать с запросами, когда в корзине недостаточно жетонов, может зависеть от потребностей разработчика. Вот некоторые общие стратегии:

Вернуть ошибку
: Это самая простая стратегия, которая часто используется для ограничения необоснованного количества клиентских запросов. Служба RESTful, использующая эту стратегию, может ответить кодом 429 (Слишком много запросов).

[1] Его еще называют алгоритмом *протекающего ведра*. – *Прим. перев.*

Вернуть ответ последнего успешного вызова функции
Эта стратегия может пригодиться, когда обращение к службе или дорогостоящей функции может дать идентичный результат, если вызвать ее слишком рано. Этот подход широко используется в мире JavaScript.

Поставить запрос в очередь и выполнить его, когда в корзину поступит достаточное количество жетонов
Этот подход может пригодиться, когда желательно обработать все запросы, пусть и с задержкой, но он также самый сложный и требует осторожности, чтобы не исчерпать память.

Пример кода

В следующем примере показана реализация алгоритма «корзина жетонов» с самой простой стратегией «возврат ошибки»:

```
type Effector func(context.Context) (string, error)

func Throttle(e Effector, max uint, refill uint, d time.Duration) Effector {
    var tokens = max
    var once sync.Once

    return func(ctx context.Context) (string, error) {
        if ctx.Err() != nil {
            return "", ctx.Err()
        }

        once.Do(func() {
            ticker := time.NewTicker(d)

            go func() {
                defer ticker.Stop()

                for {
                    select {
                    case <-ctx.Done():
                        return

                    case <-ticker.C:
                        t := tokens + refill
                        if t > max {
                            t = max
                        }
                        tokens = t
                    }
                }
            }()
        })

        if tokens <= 0 {
            return "", fmt.Errorf("too many calls")
        }
```

```
        tokens--

        return e(ctx)
    }
}
```

Эта реализация шаблона Throttle (Дроссельная заслонка) похожа на другие наши примеры: она точно так же обертывает функцию *Effector* замыканием с логикой ограничения частоты вызовов. В корзине изначально хранится максимальное `max` количество жетонов; каждый раз, когда вызывается функция-замыкание, она проверяет, имеются ли в корзине неиспользованные жетоны. Если есть, то она уменьшает счетчик жетонов на единицу и запускает пользовательскую функцию, иначе возвращает ошибку. Жетоны добавляются со скоростью `refill` жетонов через каждый интервал времени `d`.

Timeout (Тайм-аут)

Шаблон Timeout (Тайм-аут) позволяет процессу прекратить ожидание ответа, когда станет очевидно, что его можно вообще не получить.

Применимость

Первое из заблуждений распределенных вычислений гласит: «сеть надежна», и небезосновательно. Коммутаторы могут выйти из строя, маршрутизаторы и межсетевые экраны могут быть неправильно настроены; пакеты теряются. Даже если ваша сеть работает безупречно, не всякая служба спроектирована настолько идеально, чтобы гарантировать значимый и своевременный ответ – или вообще какой-либо ответ – в случае сбоя.

Шаблон Timeout (Тайм-аут) предлагает распространенное решение этой дилеммы, и он настолько прост, что его вообще едва ли можно квалифицировать как шаблон: если вызов службы или функции выполняется дольше ожидаемого, вызывающий код просто… перестает ждать.

Но не путайте «простой» или «обычный» с «бесполезный». Напротив, повсеместное распространение стратегии тайм-аута свидетельствует о ее полезности. Разумное использование тайм-аутов может обеспечить хорошую изоляцию сбоев, предотвратить каскадные сбои и снизить вероятность того, что проблема в нижестоящем ресурсе станет *вашей* проблемой.

Компоненты

Этот шаблон включает следующие компоненты:

Client
: Клиент, обращающийся к *SlowFunction*.

SlowFunction
: Функция, возвращающая ответ, необходимый клиенту, и выполняющаяся очень долго.

Timeout

Функция-обертка вокруг *SlowFunction*, которая реализует логику тайм-аута.

Реализация

Реализовать тайм-аут на языке Go можно несколькими способами, но наиболее идиоматический – использовать возможности пакета `context`. См. раздел «Пакет context» выше.

В идеальном случае долго выполняющаяся функция принимает параметр `context.Context`. В такой ситуации вам останется только передать ей экземпляр `Context`, инициированный функцией `context.WithTimeout`:

```
ctx := context.Background()
ctxt, cancel := context.WithTimeout(ctx, 10 * time.Second)
defer cancel()

result, err := SomeFunction(ctxt)
```

Однако в реальности, особенно при использовании сторонних библиотек, не всегда есть возможность рефакторинга, чтобы организовать передачу экземпляра `Context`. В таких случаях лучшим вариантом может стать оберты-вание вызова функции так, чтобы он учитывал контекст `Context`.

Например, представьте, что у вас есть долго выполняющаяся функция, которая не только не принимает экземпляра `Context`, но и находится в стороннем пакете. Если клиент *Client* будет вызывать *SlowFunction* непосредственно, ему придется ждать завершения функции, если это вообще произойдет. И как быть в этом случае?

Функцию *SlowFunction* можно вызвать в сопрограмме. Это позволит получить возвращаемый результат, если функция завершит работу в течение приемлемого периода времени, и продолжить двигаться дальше, если этого не случится.

Для этого можно использовать несколько инструментов, которые мы видели раньше: `context.Context` – для организации тайм-аута, каналы – для передачи результатов и `select` – чтобы обработать событие, которое произойдет первым.

Пример кода

В следующем примере предполагается наличие вымышленной функции `SlowFunction`, выполнение которой может завершиться или не завершиться за некоторый разумный промежуток времени и чья сигнатура соответствует следующему определению типа:

```
type SlowFunction func(string) (string, error)
```

Мы не будем вызывать `SlowFunction` напрямую, а определим функцию `Timeout`, которая принимает `SlowFunction` и возвращает функцию-замыкание `WithContext`, добавляющую `context.Context` в список параметров `SlowFunction`:

```
type WithContext func(context.Context, string) (string, error)

func Timeout(f SlowFunction) WithContext {
    return func(ctx context.Context, arg string) (string, error) {
        chres := make(chan string)
        cherr := make(chan error)

        go func() {
            res, err := f(arg)
            chres <- res
            cherr <- err
        }()

        select {
        case res := <-chres:
            return res, <-cherr
        case <-ctx.Done():
            return "", ctx.Err()
        }
    }
}
```

Внутри функции, которую создает `Timeout`, `SlowFunction` запускается в сопрограмме, которая отправляет возвращаемые значения в каналы, созданные для этой цели, если и когда `SlowFunction` завершится.

За запуском сопрограммы следует инструкция `select`, реализующая выбор из двух каналов: через первый канал передается возвращаемое значение функции `SlowFunction`, а через канал `Done` – экземпляр `Context`. Если функция выполнится раньше, чем истечет тайм-аут, то замыкание вернет значения, возвращаемые функцией `SlowFunction`; в противном случае она вернет ошибку, предоставленную контекстом `Context`.

Пользоваться функцией `Timeout` ненамного сложнее, чем вызывать `SlowFunction` напрямую, за исключением того, что вместо одной мы должны вызвать две функции: `Timeout`, чтобы получить замыкание, и само замыкание:

```
func main() {
    ctx := context.Background()
    ctxt, cancel := context.WithTimeout(ctx, 1*time.Second)
    defer cancel()

    timeout := Timeout(Slow)
    res, err := timeout(ctxt, "some input")

    fmt.Println(res, err)
}
```

Наконец, несмотря на то что для реализации тайм-аутов обычно предпочтительнее использовать `context.Context`, для этой цели также можно использовать канал, возвращаемый функцией `time.After`. См. раздел «Реализация тайм-аутов для каналов» в главе 3, где показано, как это делается.

Шаблоны конкуренции

Облачные службы часто используются для эффективного управления несколькими процессами и обслуживания высоких (и очень изменчивых) нагрузок, в идеале без затрат на масштабирование. По этой причине подобные службы должны поддерживать возможность параллельного выполнения и одновременной обработки нескольких запросов от нескольких клиентов. Язык Go славится своей поддержкой конкуренции, но и в нем есть свои узкие места. Ниже представлены некоторые шаблоны, разработанные для предотвращения их появления.

Fan-In (Мультиплексор)

Шаблон Fan-In (Мультиплексор) мультиплексирует несколько входных каналов в один выходной канал.

Применимость

В службах, имеющих несколько рабочих процессов, генерирующих результаты, бывает полезно объединить эти результаты в один общий поток. В таких случаях используется шаблон Fan-In (Мультиплексор), позволяющий читать данные из нескольких входных каналов и передавать их в один целевой канал.

Компоненты

Этот шаблон включает следующие компоненты:

Sources
: Набор из одного или нескольких входных каналов одного типа, которые может принять *Funnel*.

Destination
: Выходной канал того же типа, что и каналы в наборе *Sources*. Создается функцией *Funnel*.

Funnel
: Принимает набор *Sources* и сразу же возвращает *Destination*. Любые данные, поступающие из каналов в *Sources*, передаются в *Destination*.

Реализация

Funnel – это функция, которая принимает до N входных каналов (*Sources*). Для каждого входного канала в *Sources* функция *Funnel* запускает отдельную сопрограмму, читающую значения из назначенного ей канала и передающую их в выходной канал, общий для всех сопрограмм (*Destination*).

Пример кода

Функция Funnel принимает переменное число аргументов – ноль или более каналов некоторого типа (int в нашем примере):

```
func Funnel(sources ...<-chan int) <-chan int {
    dest := make(chan int)       // Общий выходной канал

    var wg sync.WaitGroup        // Для автоматического закрытия dest,
                                 // когда закроются все входящие каналы sources

    wg.Add(len(sources))         // Установить размер WaitGroup

    for _, ch := range sources { // Запуск сопрограммы для каждого входного канала
        go func(c <-chan int) {
            defer wg.Done()      // Уведомить WaitGroup, когда с закроется

            for n := range c {
                dest <- n
            }
        }(ch)
    }

    go func() {                  // Запустить сопрограмму, которая закроет dest
        wg.Wait()                // после закрытия всех входных каналов
        close(dest)
    }()

    return dest
}
```

Для каждого входного канала Funnel запускает специальную сопрограмму, которая читает значения из назначенного ей канала и пересылает их в dest, общий выходной канал для всех сопрограмм.

Обратите внимание, как используется sync.WaitGroup, чтобы гарантировать закрытие выходного канала. Сначала создается группа ожидания WaitGroup, в которой настраивается общее количество входных каналов. Когда какой-то канал закрывается, связанная с ним сопрограмма завершается и вызывает wg.Done. Когда закроется последний входной канал, счетчик в WaitGroup обнулится, блокировка, установленная вызовом wg.Wait, освободится, и выходной канал закроется.

Пользоваться функцией Funnel достаточно просто, нужно лишь передать ей N входных каналов (или срез с N каналами). Полученный выходной канал можно использовать как обычно, и он будет закрыт после закрытия всех входных каналов:

```
func main() {
    sources := make([]<-chan int, 0)  // Создать пустой срез с каналами

    for i := 0; i < 3; i++ {
        ch := make(chan int)
        sources = append(sources, ch) // Создать канал; добавить в срез sources
```

```
        go func() {                    // Запустить сопрограмму для каждого
            defer close(ch)            // Закрыть канал по завершении сопрограммы

            for i := 1; i <= 5; i++ {
                ch <- i
                time.Sleep(time.Second)
            }
        }()
    }

    dest := Funnel(sources...)
    for d := range dest {
        fmt.Println(d)
    }
}
```

Код в этом примере создает срез с тремя каналами `int`, в каждый из которых перед закрытием отправляются значения от 1 до 5. В отдельной сопрограмме извлекаются выходные данные из единственного канала `dest`. Если запустить этот пример, он выведет 15, после чего закроет `dest` и завершится.

Fan-Out (Демультиплексор)

Шаблон Fan-Out (Демультиплексор) равномерно распределяет сообщения из одного входного канала между несколькими выходными каналами.

Применимость

Шаблон Fan-out (Демультиплексор) принимает сообщения из входного канала и равномерно распределяет их между выходными каналами. Это очень полезный шаблон для распараллеливания вычислительных ресурсов и ресурсов ввода/вывода.

Например, представьте, что у вас есть источник данных, входной поток или брокер сообщений, который предоставляет входные данные для некоторой ресурсоемкой работы. Вместо того чтобы связывать процессы ввода и вычислений в один последовательный процесс, можно просто распараллелить рабочую нагрузку, распределив ее между несколькими вычислительными процессами.

Компоненты

Этот шаблон включает следующие компоненты:

Source
: Входной канал. Принимается компонентом *Split*.

Destinations
: Набор выходных каналов того же типа, что и *Source*. Создается и возвращается компонентом *Split*.

Split

Функция, которая принимает *Source* и сразу же возвращает *Destinations*. Любые данные, поступившие из *Source*, выводятся в *Destinations*.

Реализация

Шаблон Fan-out (Демультиплексор) относительно прост в концептуальном плане, но, как известно, дьявол кроется в деталях.

Обычно демультиплексирование реализуется как функция *Split*, которая принимает один входной канал и целое число, представляющее желаемое количество выходных каналов *Destinations*. Функция *Split* создает выходные каналы и запускает некоторый фоновый процесс, который извлекает значения из входного канала и пересылает их в один из выходных каналов.

Реализовать логику пересылки можно одним из двух способов:

- использовать общую сопрограмму, которая читает значения из входного канала и пересылает их в выходные каналы в циклическом режиме. Достоинство этого способа в том, что он требует только одной сопрограммы, но если следующий выбранный ею канал не будет готов к чтению, то это замедлит весь процесс;
- использовать отдельные сопрограммы для каждого выходного канала, которые будут состязаться за чтение следующего значения из *Source* и пересылать его в соответствующий выходной канал. Это потребует немного больше ресурсов, но вероятность зависнуть из-за одного медленно работающего рабочего процесса будет ниже.

Следующий пример реализует второй подход.

Пример кода

В этом примере функция `Split` принимает канал `source`, доступный только для приема, и целое число `n`, определяющее количество выходных каналов. Она возвращает срез с `n` каналами, доступными только для отправки, того же типа, что и `source`.

Внутри `Split` создает выходные каналы. Для каждого созданного канала она запускает сопрограмму, которая извлекает значения из `source` в цикле `for` и пересылает их в назначенный ей выходной канал. Фактически все эти сопрограммы состязаются между собой за чтение из `source`; победитель в таком состязании выбирается случайным образом. Если `source` закроется, то все сопрограммы завершатся и все выходные каналы закроются:

```
func Split(source <-chan int, n int) []<-chan int {
    dests := make([]<-chan int, 0) // Создать срез dests

    for i := 0; i < n; i++ {       // Создать n выходных каналов
        ch := make(chan int)
        dests = append(dests, ch)

        go func() {                // Каждый выходной канал передается
            defer close(ch)        // своей сопрограмме, которая состязается
                                   // с другими за доступ к source
```

```
            for val := range source {
                ch <- val
            }
        }()
    }

    return dests
}
```

Получив канал определенного типа, функция `Split` вернет заданное количество выходных каналов. Далее каждый из них можно передать отдельной сопрограмме, как показано в следующем примере:

```
func main() {
    source := make(chan int)       // Входной канал
    dests := Split(source, 5)      // Получить 5 выходных каналов

    go func() {                    // Передать числа 1..10 в source
        for i := 1; i <= 10; i++ { // и закрыть его по завершении
            source <- i
        }

        close(source)
    }()

    var wg sync.WaitGroup          // Использовать WaitGroup для ожидания, пока
    wg.Add(len(dests))             // не закроются выходные каналы

    for i, ch := range dests {
        go func(i int, d <-chan int) {
            defer wg.Done()

            for val := range d {
                fmt.Printf("#%d got %d\n", i, val)
            }
        }(i, ch)
    }

    wg.Wait()
}
```

В этом примере создается входной канал `source`, который передается в `Split` для получения выходных каналов. Затем в отдельной сопрограмме в него записываются числа от 1 до 10, которые извлекаются из `dests` в пяти других сопрограммах. По завершении ввода канал `source` закрывается, что вызывает закрытие выходных каналов и завершение циклов чтения, после чего каждая сопрограмма вызывает `wg.Done`, снимается блокировка `wg.Wait`, и функция `main` завершается.

Future (В будущем)

Шаблон Future (В будущем) возвращает заполнитель для значения, которое пока неизвестно.

Применимость

Шаблон Future (В будущем) (также известен под названиями Promises (Обещанное значение) или Delays (Задерживаемое значение)[1]) служит для синхронизации и предоставляет заполнитель для значения, которое будет сгенерировано асинхронным процессом когда-то в будущем.

В Go этот шаблон используется реже, чем в некоторых других языках, потому что каналы предлагают аналогичные возможности. Например, блокирующуюся функцию `BlockingInverse` (здесь не показана) можно вызвать в сопрограмме, которая вернет результат (когда он поступит) по каналу. Функция `ConcurrentInverse` именно так и работает, возвращая канал, который можно прочитать, когда результат станет доступен:

```
func ConcurrentInverse(m Matrix) <-chan Matrix {
    out := make(chan Matrix)

    go func() {
        out <- BlockingInverse(m)
        close(out)
    }()

    return out
}
```

Используя `ConcurrentInverse`, можно построить функцию для вычисления произведения двух матриц, обратных заданным:

```
func InverseProduct(a, b Matrix) Matrix {
    inva := ConcurrentInverse(a)
    invb := ConcurrentInverse(b)

    return Product(<-inva, <-invb)
}
```

На первый взгляд выглядит неплохо, но есть один недостаток, который делает такой подход нежелательным для реализации, например, общедоступного API. Во-первых, вызывающий код должен проявить осторожность и точно синхронизировать вызовы `ConcurrentInverse`. Чтобы понять, о чем я говорю, взгляните на следующий код:

```
return Product(<-ConcurrentInverse(a), <-ConcurrentInverse(b))
```

Видите проблему? Поскольку вычисления не начнутся до фактического вызова `ConcurrentInverse`, эта конструкция в действительности будет выполняться последовательно, что потребует вдвое большего времени.

Более того, при таком использовании каналов функции с несколькими возвращаемыми значениями обычно возвращают несколько каналов – по одному для каждого члена списка возвращаемых значений, что может пре-

---

[1] Эти термины часто используются как синонимы, но иногда они могут иметь свои нюансы, в зависимости от контекста. Я знаю об этом, и, пожалуйста, не пишите мне гневных писем по данному поводу.

вратиться в неудобство с ростом такого списка или когда значения должны читаться несколькими сопрограммами.

Шаблон Future (В будущем) изолирует эту сложность за фасадом API, который предоставляет пользователю простой интерфейс с методами, вызываемыми обычным образом и блокирующимися до тех пор, пока не будут получены все результаты. Для этого даже не нужно создавать специальный интерфейс, которому удовлетворяет значение; можно использовать любой удобный интерфейс.

Компоненты

Этот шаблон включает следующие компоненты:

Future
: Интерфейс, который получает пользователь для извлечения окончательного результата.

SlowFunction
: Функция-обертка вокруг некоторой функции, которую требуется выполнить асинхронно; возвращает *Future*.

InnerFuture
: Удовлетворяет интерфейсу *Future*; включает присоединенный метод с логикой доступа к результату.

Реализация

Прикладной интерфейс (API), предлагаемый пользователю, довольно прост: программист вызывает *SlowFunction*, которая возвращает значение, удовлетворяющее интерфейсу *Future*. *Future* может быть специализированным интерфейсом, как в следующем примере, или быть чем-то более похожим на `io.Reader`, экземпляры которого можно передавать его собственным функциям.

Когда вызывается *SlowFunction*, она выполняет заданную функцию как сопрограмму и определяет каналы для передачи результата функции, которые заключает в экземпляр *InnerFuture*.

InnerFuture имеет один или несколько методов, которые удовлетворяют интерфейсу *Future*, извлекают из каналов значения, возвращаемые функцией, кешируют их и возвращают. Если значения недоступны в канале, то вызов блокируется. Если они уже были получены раньше, то возвращаются кешированные значения.

Пример кода

В этом примере мы используем интерфейс *Future*, которому удовлетворяет *InnerFuture*:

```
type Future interface {
    Result() (string, error)
}
```

Структура `InnerFuture` используется внутри реализации для поддержки конкуренции. В этом примере она удовлетворяет интерфейсу *Future*, но с таким же успехом можно было бы выбрать, например, `io.Reader`, присоединив к нему метод `Read`:

```
type InnerFuture struct {
    once sync.Once
    wg sync.WaitGroup

    res string
    err error
    resCh <-chan string
    errCh <-chan error
}

func (f *InnerFuture) Result() (string, error) {
    f.once.Do(func() {
        f.wg.Add(1)
        defer f.wg.Done()
        f.res = <-f.resCh
        f.err = <-f.errCh
    })

    f.wg.Wait()

    return f.res, f.err
}
```

В этой реализации сама структура содержит канал и переменную для каждого значения, возвращаемого методом `Result`. При первом вызове `Result` пытается прочитать результаты из каналов и сохранить их в структуре `InnerFuture`, чтобы последующие вызовы могли сразу вернуть кешированные значения.

Обратите внимание на использование экземпляров `sync.Once` и `sync.WaitGroup`. Первый делает то, о чем говорит его имя: гарантирует ровно один вызов указанной функции. `WaitGroup` обеспечивает безопасность вызова функции в многопоточном окружении: любые вызовы после первого будут блокироваться на `wg.Wait`, пока не завершится чтение канала.

`SlowFunction` – это обертка для указанной функции, которую требуется выполнить асинхронно. Ее задача – создать каналы результатов, запустить заданную функцию в сопрограмме, а также создать и вернуть реализацию `Future` (`InnerFuture` в этом примере):

```
func SlowFunction(ctx context.Context) Future {
    resCh := make(chan string)
    errCh := make(chan error)

    go func() {
        select {
        case <-time.After(time.Second * 2):
            resCh <- "I slept for 2 seconds"
            errCh <- nil
```

```
        case <-ctx.Done():
            resCh <- ""
            errCh <- ctx.Err()
        }
    }()

    return &InnerFuture{resCh: resCh, errCh: errCh}
}
```

Чтобы использовать этот шаблон, достаточно просто вызвать `SlowFunction` и использовать полученный экземпляр `Future`, как любое другое значение:

```
func main() {
    ctx := context.Background()
    future := SlowFunction(ctx)

    res, err := future.Result()
    if err != nil {
        fmt.Println("error:", err)
        return
    }

    fmt.Println(res)
}
```

Такой подход обеспечивает простой и понятный интерфейс. Программист может создать экземпляр `Future`, обращаться к нему и даже применять тайм-ауты или крайние сроки с помощью контекста `Context`.

Sharding (Сегментирование)

Шаблон *Sharding* (*Сегментирование*) разбивает большую структуру данных на несколько сегментов, чтобы локализовать влияние блокировок чтения/записи.

Применимость

Термин *сегментирование* (*sharding*) обычно используется в контексте поддержки состояния распределенных приложений и подразумевает разделение данных между экземплярами сервера. Такое *горизонтальное сегментирование* обычно используется в базах данных и других хранилищах для распределения нагрузки и обеспечения избыточности.

В службах, выполняющихся конкурентно и использующих общую структуру данных с механизмом блокировки для защиты от конфликтов при записи, может возникнуть несколько иная проблема. В этом сценарии блокировки, обеспечивающие целостность данных, могут превратиться в узкое место, когда процессы начинают тратить больше времени на ожидание освобождения блокировок, чем на выполнение своей работы. Это досадное явление называется *конфликтом блокировок*.

В некоторых случаях проблему можно решить путем масштабирования числа экземпляров, но такое решение также увеличивает сложность и длительность задержек, потому что требует использовать распределенные блокировки, обеспечивающие согласованность операций записи. Альтернативной стратегией уменьшения конфликтов блокировок при работе с общими структурами данных является *вертикальное сегментирование*, когда большая структура данных делится на две или более структур, каждая из которых является частью целого. При использовании этой стратегии блокировать требуется только часть общей структуры, что снижает вероятность конфликтов на блокировках.

Горизонтальное и вертикальное сегментирование

Большие структуры данных можно сегментировать двумя способами:

- *по горизонтали* – разделив данные между экземплярами службы. Этот прием может обеспечить избыточность данных и позволяет сбалансировать нагрузку между экземплярами, но также увеличивает задержку и сложность доступа к распределенным данным;
- *по вертикали* – разделив данные в пределах одного экземпляра. Этот прием может уменьшить конфликты между операциями чтения/записи, которые выполняют конкурирующие процессы, но он не обеспечивает избыточности и возможности масштабирования.

Компоненты

Этот шаблон включает следующие компоненты:

ShardedMap
: Абстракция вокруг одного или нескольких сегментов *Shard*, обеспечивающая доступ для чтения и записи, как если бы сегменты были единой структурой.

Shard
: Отдельная коллекция со своей блокировкой, представляющая один сегмент данных.

Реализация

Идиоматический подход в языке Go заключается в использовании каналов (https://oreil.ly/BipeP) вместо общих ресурсов, защищенных блокировками[1], но это не всегда возможно. Ассоциативные массивы особенно уязвимы при конкурентном использовании, что делает применение блокировок для синхронизации доступа к ним неизбежным злом. К счастью, как раз для этой цели в Go имеется `sync.RWMutex`.

[1] См. статью «Share Memory By Communicating» в блоге *The Go Blog*.

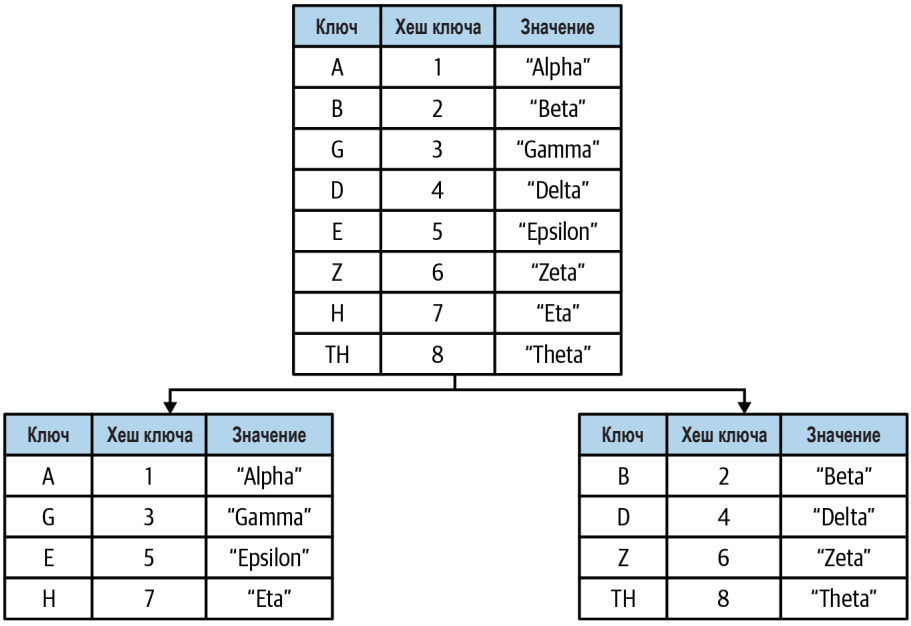
RWMutex предоставляет методы установки блокировок для чтения и для записи, как показано ниже. Используя этот метод, конкурирующие процессы могут одновременно устанавливать блокировки для чтения, если не установлена блокировка для записи; процесс может установить блокировку для записи, только когда не установлено ни одной блокировки для чтения или записи. Попытки установить дополнительные блокировки для записи будут вызывать приостановку процесса до тех пор, пока не будут сняты все блокировки, установленные ранее:

```
var items = struct{        // Структура с ассоциативным массивом
    sync.RWMutex           // и мьютексом sync.RWMutex
    m map[string]int
}{m: make(map[string]int)}

func ThreadSafeRead(key string) int {
    items.RLock()          // Установить блокировку для чтения
    value := items.m[key]
    items.RUnlock()        // Снять блокировку для чтения
    return value
}

func ThreadSafeWrite(key string, value int) {
    items.Lock()           // Установить блокировку для записи
    items.m[key] = value
    items.Unlock()         // Снять блокировку для записи
}
```

В общем случае эта стратегия прекрасно работает. Однако, поскольку блокировки разрешают доступ только одному процессу за раз, среднее время ожидания снятия блокировок в приложении, часто выполняющем операции чтения/записи, может резко возрасти с увеличением числа процессов, одновременно использующих ресурс. Возникающая в результате конкуренция за блокировки может ограничить общую производительность.

Вертикальное сегментирование снижает конкуренцию за блокировки путем деления общей структуры данных – обычно ассоциативного массива – на несколько подмассивов, блокируемых по отдельности. Слой абстракции обеспечивает доступ к сегментам, как если бы они были единой структурой (см. рис. 4.5).

Это достигается путем создания слоя абстракции вокруг общей структуры, которая по сути является ассоциативным массивом, состоящим из ассоциативных массивов. Перед каждой операцией чтения или записи слой абстракции вычисляет хеш ключа и с учетом количества сегментов определяет соответствующий индекс сегмента. Это позволяет изолировать необходимую блокировку, защищающую только сегмент с этим индексом.



Ключ	Хеш ключа	Значение
A	1	“Alpha”
B	2	“Beta”
G	3	“Gamma”
D	4	“Delta”
E	5	“Epsilon”
Z	6	“Zeta”
H	7	“Eta”
TH	8	“Theta”

Ключ	Хеш ключа	Значение
A	1	“Alpha”
G	3	“Gamma”
E	5	“Epsilon”
H	7	“Eta”

Ключ	Хеш ключа	Значение
B	2	“Beta”
D	4	“Delta”
Z	6	“Zeta”
TH	8	“Theta”

Рис. 4.5 ❖ Вертикальное сегментирование ассоциативного массива по хешу ключа

Пример кода

В следующем примере представлена реализация простого сегментированного ассоциативного массива `ShardedMap` с использованием стандартных пакетов `sync` и `crypto/sha1`.

Внутренне `ShardedMap` – это самый обычный срез с указателями на экземпляры `Shard`, но мы определяем его как отдельный тип, чтобы получить возможность присоединять к нему методы. Каждый экземпляр `Shard` включает поле `map[string]interface{}`, содержащее информацию о сегменте, и скомпонован с `sync.RWMutex`, чтобы дать возможность заблокировать этот сегмент отдельно:

```
type Shard struct {
    sync.RWMutex             // Встраивание sync.RWMutex
    m map[string]interface{} // m содержит информацию о сегменте
}

type ShardedMap []*Shard     // ShardedMap является срезом с экземплярами *Shard
```

В языке Go нет понятия «конструктор», поэтому добавим функцию `NewShardedMap` для создания нового экземпляра `ShardedMap`:

```
func NewShardedMap(nshards int) ShardedMap {
    shards := make([]*Shard, nshards)  // Инициализировать срез с экземплярами *Shard
```

```
    for i := 0; i < nshards; i++ {
        shard := make(map[string]interface{})
        shards[i] = &Shard{m: shard}
    }

    return shards  // ShardedMap ЯВЛЯЕТСЯ срезом с экземплярами *Shard!
}
```

`ShardedMap` имеет два внутренних метода, `getShardIndex` и `getShard`, которые используются для вычисления индекса сегмента по ключу и получения соответствующего сегмента. Их можно объединить в один метод, но такое разделение, как здесь, упрощает их тестирование:

```
func (m ShardedMap) getShardIndex(key string) int {
    checksum := sha1.Sum([]byte(key)) // Использовать Sum из "crypto/sha1"
    hash := int(checksum[17])         // Выбрать произвольный байт на роль хеша
    return hash % len(m)              // Взять остаток от деления на len(m),
                                      // чтобы получить индекс
}

func (m ShardedMap) getShard(key string) *Shard {
    index := m.getShardIndex(key)
    return m[index]
}
```

Обратите внимание на очевидный недостаток в предыдущем примере: в нем в качестве хеша используется значение типа `byte`, поэтому он сможет обрабатывать не более 255 сегментов. Если по какой-то причине вам потребуется больше сегментов, то добавьте немного двоичной арифметики: `hash := int(sum[13]) << 8 | int(sum[17])`.

Наконец, добавим в `ShardedMap` методы чтения и записи значений. Очевидно, что это далеко не все функции, которые могут понадобиться при работе с таким ассоциативным массивом. Однако исходный код этого примера имеется в репозитории GitHub для данной книги, и вы сможете в качестве самостоятельного упражнения добавить в него любые функции, которые могут вам понадобиться. Было бы неплохо, например, добавить методы `Delete` и `Contains`:

```
func (m ShardedMap) Get(key string) interface{} {
    shard := m.getShard(key)
    shard.RLock()
    defer shard.RUnlock()

    return shard.m[key]
}

func (m ShardedMap) Set(key string, value interface{}) {
    shard := m.getShard(key)
    shard.Lock()
    defer shard.Unlock()

    shard.m[key] = value
}
```

Если понадобится установить блокировки для всех сегментов, то лучше делать это одновременно. Ниже показана реализация функции Keys с использованием сопрограмм и нашего старого знакомого sync.WaitGroup:

```
func (m ShardedMap) Keys() []string {
    keys := make([]string, 0)      // Создать пустой срез ключей

    mutex := sync.Mutex{}          // Мьютекс для безопасной записи в keys

    wg := sync.WaitGroup{}         // Создать группу ожидания и установить
    wg.Add(len(m))                 // счетчик равным количеству сегментов

    for _, shard := range m {      // Запустить сопрограмму для каждого сегмента
        go func(s *Shard) {
            s.RLock()              // Установить блокировку для чтения в s

            for key := range s.m { // Получить ключи из сегмента
                mutex.Lock()
                keys = append(keys, key)
                mutex.Unlock()
            }

            s.RUnlock()            // Снять блокировку для чтения
            wg.Done()              // Сообщить WaitGroup, что обработка завершена
        }(shard)
    }

    wg.Wait()                      // Приостановить выполнение до выполнения
                                   // всех операций чтения

    return keys                    // Вернуть срез с ключами
}
```

К сожалению, использование ShardedMap отличается от использования стандартного ассоциативного массива, но оно ничуть не сложнее:

```
func main() {
    shardedMap := NewShardedMap(5)

    shardedMap.Set("alpha", 1)
    shardedMap.Set("beta", 2)
    shardedMap.Set("gamma", 3)

    fmt.Println(shardedMap.Get("alpha"))
    fmt.Println(shardedMap.Get("beta"))
    fmt.Println(shardedMap.Get("gamma"))

    keys := shardedMap.Keys()
    for _, k := range keys {
        fmt.Println(k)
    }
}
```

Возможно, самым большим недостатком ShardedMap (помимо его сложности, конечно) является потеря безопасности типов из-за использования

`interface{}` и необходимость их проверки. Надеюсь, с появлением дженериков в Go (это произойдет очень скоро, если уже не произошло) эта проблема останется в прошлом!

Итоги

В этой главе мы рассмотрели несколько очень интересных и полезных идиом. В реальности их гораздо больше[1], но я перечислил здесь лишь те, которые считаю наиболее важными, потому что они имеют прямое практическое применение или демонстрируют некоторые интересные особенности языка Go, но в большинстве и то, и другое.

В главе 5 мы перейдем на следующий уровень, взяв на вооружение некоторые приемы, обсуждавшиеся в главах 3 и 4, и применим их на практике, создав с нуля простое хранилище пар ключ/значение.

[1] Я пропустил вашу любимую идиому? Дайте мне знать, и я постараюсь включить ее в следующее издание!

Глава 5

Конструирование облачной службы

> До Второй мировой войны жизнь была простой. После нее у нас появились системы[1].
>
> – Грейс Хоппер (Grace Hopper), информационный бюллетень OCLC (1987)

В этой главе, наконец, начнется настоящая работа.

Мы объединим многое из того, что обсуждалось в части II, и создадим службу, которая станет отправной точкой для оставшейся части книги. По мере продвижения вперед мы будем снова и снова возвращаться к начатому здесь, и в каждой главе будем добавлять новые функциональные возможности, пока, наконец, не получим настоящее облачное приложение.

Естественно, это приложение не готово к промышленной эксплуатации, потому что в нем, например, отсутствуют важные функции безопасности, но оно послужит нам основой для нашей работы.

Так что мы создадим?

Давайте создадим службу!

Хорошо. Давайте. Но сначала определимся, какая это будет служба.

Она должна быть концептуально простой, достаточно понятной для реализации в самой простой форме, но нетривиальной и поддающейся масштабированию. Это должно быть что-то, что мы сможем постепенно совершенствовать в оставшейся части книги. Я много думал над этим, рассматривая разные идеи, и в конце концов пришел к простому решению.

Мы создадим распределенное хранилище пар ключ/значение.

---

[1] Schieber, Philip. «The Wit and Wisdom of Grace Hopper». OCLC Newsletter, March/April, 1987, No. 167.

Что такое хранилище пар ключ/значение?

Хранилище пар ключ/значение – это разновидность нереляционной базы данных, в которой данные хранятся в виде набора пар ключ/значение. Такие хранилища сильно отличаются от более известных реляционных баз данных, таких как Microsoft SQL Server или PostgreSQL, которые мы знаем и любим[1]. В отличие от реляционных баз данных, которые структурируют хранимые данные по фиксированным таблицам с четко определенными типами данных, хранилища пар ключ/значение устроены намного проще и позволяют связывать уникальные идентификаторы (ключи) с произвольными значениями.

Иначе говоря, хранилище пар ключ/значение – это просто ассоциативный массив с конечной точкой в виде службы, как показано на рис. 5.1. Это самая простая из возможных баз данных.

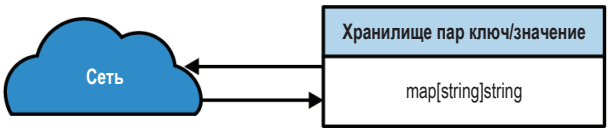


Рис. 5.1 ❖ Хранилище пар ключ/значение – это просто ассоциативный массив с конечной точкой в виде службы

Требования

К концу этой главы мы создадим простое нераспределенное хранилище пар ключ/значение, которое сможет делать все то же, что и (монолитное) хранилище.

- Оно должно поддерживать хранение произвольных пар ключ/значение.
- Оно должно предоставлять конечные точки (службы), позволяющие пользователю добавлять, получать и удалять пары ключ/значение.
- Оно должно обеспечивать долговременное хранение данных.

Наконец, служба должна быть идемпотентной. Но почему?

Что такое идемпотентность, и почему это важно?

Идея *идемпотентности* (тождественности влияния) зародилась в алгебре, где описывает определенные свойства некоторых математических операций. К счастью, это не книга по математике, и мы не будем углубляться в нее (разве что во врезке в конце этого раздела).

В мире программирования операция (например, вызов метода или службы) считается идемпотентной, если ее однократное выполнение имеет тот

[1] «Любим» в некотором определенном смысле.

же эффект, что и многократное. Например, операция присваивания `x = 1` идемпотентна, потому что после ее выполнения `x` всегда будет хранить `1`, независимо от того, сколько раз ее выполнить. Точно так же HTTP-метод `PUT` является идемпотентным, поскольку многократная отправка ресурса в какое-либо место ничего не изменит: повторная отправка не даст ничего нового[1]. Операция `x += 1`, однако, не является идемпотентной, потому что после каждого ее выполнения создается новое состояние.

Менее обсуждаемым, но не менее важным является родственное свойство *отсутствия влияния* (nullpotence), согласно которому функция или операция вообще не имеет побочных эффектов. Например, присваивание `x = 1` и HTTP-метод `PUT` идемпотентны, но не лишены влияния, потому что запускают изменение состояния. Присваивание переменной самой себе, такое как `x = x`, не имеет влияния, поскольку в результате его выполнения состояние не меняется. Точно так же простое чтение данных, как, например, HTTP-метод `GET`, обычно не имеет побочных эффектов, поэтому оно также лишено влияния на окружение.

Конечно, все это прекрасно в теории, но какое нам дело до этого в реальном мире? Как оказывается, реализация методов службы в идемпотентной манере дает ряд преимуществ:

Идемпотентные операции безопаснее
: Представьте, что вы отправили запрос службе, но не получили ответа. Как вы поступите? Наверняка попробуете еще раз. Но что, если служба получила и обработала ваш первый запрос[2]? Если метод идемпотентный, то никакого вреда от этого не будет. Но если это не так, то могут возникнуть проблемы. Этот сценарий встречается чаще, чем вы думаете. Сети ненадежны. Ответы могут задерживаться; пакеты могут теряться.

Идемпотентные операции часто проще
: Идемпотентные операции более автономны и проще в реализации. Сравните, например, идемпотентный метод `PUT`, который просто добавляет пару ключ/значение в хранилище данных, и аналогичный, но неидемпотентный метод `CREATE`, который возвращает ошибку, если хранилище данных уже содержит такой ключ. Логика `PUT` проста: проанализировать запрос и изменить значение. Логика `CREATE` сложнее и требует дополнительных проверок и обработки ошибок и, возможно, даже использования распределенной блокировки для координации действий с любыми другими экземплярами службы, что затрудняет масштабирование.

Идемпотентные операции более декларативны
: Конструирование идемпотентного API побуждает проектировщика сосредоточиться на конечных состояниях и создавать методы, которые являются более *декларативными*: они позволяют пользователям сообщать службе, *что нужно сделать*, вместо того чтобы указывать ей, *как это делать*. На первый взгляд разница может показаться несущественной, тем не менее

---

[1] А если даст, то, значит, что-то пошло не так.

[2] Как мой сын, который только притворяется, что не слышит меня.

декларативные методы – в отличие от *императивных* – освобождают пользователей от необходимости иметь дело с низкоуровневыми конструкциями, позволяя им сосредоточиться на конкретных целях и сводя к минимуму возможные побочные эффекты.

В действительности идемпотентность дает такие преимущества, особенно в контексте облачных вычислений, что некоторые очень умные люди даже утверждают, что идемпотентность – это *синоним* понятия «облачное окружение»[1]. Я не сторонник таких радикальных суждений, но *могу* сказать, что если вы собираетесь разместить свою службу в облаке, то отступление от идемпотентности обязательно приведет к проблемам.

Математическое определение идемпотентности

Понятие идемпотентности родилось в математике, где оно описывает операции, которые при многократном применении не меняют результата, кроме самого первого раза.

Выражаясь чисто математическим языком: функция считается идемпотентной, если $f(f(x)) = f(x)$ для всех x.

Например, взятие абсолютного значения $abs(x)$ целого числа x является идемпотентной функцией, потому что условие $abs(x) = abs(abs(x))$ выполняется для любого действительного числа x.

Конечная цель

Эти требования довольно сложны, но они представляют абсолютный минимум для нашего хранилища пар ключ/значение. Позже мы добавим некоторые важные функции, такие как поддержка нескольких пользователей и шифрование данных при передаче. Но, что еще более важно, мы представим методы и технологии, которые сделают службу более масштабируемой, устойчивой и способной выживать и работать в жестокой и неопределенной вселенной.

Итерация 0: базовая функциональность

Итак, приступим. Для начала забудем о пользовательских запросах и долговременном хранении и реализуем самые основные функции, которые потом можно будет вызывать из любого веб-фреймворка, который мы решим использовать.

---

[1] «Облачное окружение не является синонимом для микросервисов… если термин “облачное окружение” и может быть синонимом, то только синонимом понятию идемпотентности, которому определенно нужен синоним». – Холли Камминс (Holly Cummins; Cloud Native, Лондон, 2018).

Хранение произвольных пар ключ/значение
: На данный момент мы можем взять за основу простой ассоциативный массив, но какой? Для простоты ограничимся строковыми ключами и значениями, но позже мы разрешим использовать произвольные типы. Для этого используем простое определение `map[string]string` для нашей базовой структуры данных.

Добавление, получение и удаление пар ключ/значение
: В этой первой итерации мы создадим простой API на языке Go, который можно будет вызывать для выполнения основных операций. Такое разделение функциональности упростит тестирование и совершенствование в будущих итерациях.

Наш суперпростой API

Первое, что нужно сделать, – создать ассоциативный массив; сердце нашего хранилища пар ключ/значение:

```
var store = make(map[string]string)
```

Ну разве это не прекрасно своей простотой? Не волнуйтесь, позже мы все усложним.

Первая функция, которую мы создадим, – это, соответственно, `PUT`, которая будет использоваться для добавления записей в хранилище. Она делает именно то, что предлагает ее имя: принимает строки `key` и `value` и помещает их в хранилище. Сигнатура функции `PUT` включает возврат ошибки `error`, которая нам понадобится позже:

```
func Put(key string, value string) error {
    store[key] = value

    return nil
}
```

Поскольку мы сделали сознательный выбор в пользу идемпотентности, `Put` не проверяет существование пары ключ/значение, поэтому с радостью добавит новую, если об этом попросят. Многократное выполнение `Put` с одинаковыми параметрами всегда будет давать один и тот же результат, независимо от текущего состояния.

Теперь, когда мы определили базовый шаблон, напишем операции `Get` и `Delete`:

```
var ErrorNoSuchKey = errors.New("no such key")

func Get(key string) (string, error) {
    value, ok := store[key]

    if !ok {
        return "", ErrorNoSuchKey
    }
```

```
    return value, nil
}

func Delete(key string) error {
    delete(store, key)

    return nil
}
```

Но посмотрите внимательно, как `Get` возвращает ошибку, она не использует `errors.New`, а просто возвращает предварительно созданный экземпляр ошибки `ErrorNoSuchKey`. Почему? Это пример *сигнальной ошибки*, которая позволяет службе-потребителю точно определить тип ошибки, которую она получает, и отреагировать соответствующим образом. Например, она может сделать что-то вроде этого:

```
if errors.Is(err, ErrorNoSuchKey) {
    http.Error(w, err.Error(), http.StatusNotFound)
    return
}
```

Теперь, получив минимальный набор функций (просто меньше некуда), не забудьте написать тесты. Мы не будем делать этого здесь, на страницах книги, и я предлагаю вам сделать это самостоятельно, но если вам не терпится или лень (лень – в какой-то степени тоже двигатель прогресса), то можете взять код из репозитория GitHub, созданного для этой книги (https://oreil.ly/ois1B).

Итерация 1: монолит

Теперь, имея минимальный API хранилища пар ключ/значение, можно приступить к созданию службы на его основе. У нас есть несколько разных вариантов, как это сделать. Мы можем использовать что-то вроде GraphQL. Есть множество достойных сторонних пакетов, которые мы могли бы использовать, но наш ландшафт данных не настолько сложен, чтобы мы не могли обойтись без этого. Мы также могли бы использовать механизм вызова удаленных процедур (Remote Procedure Call, RPC), который поддерживается стандартным пакетом `net/rpc`, или даже gRPC, но это потребует дополнительных накладных расходов для клиента, и, опять же, наши данные не настолько сложны, чтобы в этом была необходимость.

У нас остается только технология репрезентативной передачи состояния (Representational State Transfer, REST). Технология REST не очень нравится людям, но она проста и полностью соответствует нашим потребностям.

Создание HTTP-сервера с использованием net/http

В Go нет веб-фреймворков, которые были бы столь же развитыми или имели столь же долгую историю развития, как Django или Flask. Однако есть надежный набор стандартных библиотек, которых вполне достаточно в 80 % случаев. Более того, они поддерживают возможность расширения, поэтому *существует* несколько веб-фреймворков на Go, которые их дополняют.

А сейчас давайте рассмотрим стандартную идиому обработчика HTTP-запросов на Go, реализованную с использованием `net/http`:

```
package main

import (
    "log"
    "net/http"
)

func helloGoHandler(w http.ResponseWriter, r *http.Request) {
    w.Write([]byte("Hello net/http!\n"))
}

func main() {
    http.HandleFunc("/", helloGoHandler)

    log.Fatal(http.ListenAndServe(":8080", nil))
}
```

В этом примере мы определили метод `helloGoHandler`, который удовлетворяет определению `http.HandlerFunc`:

```
type HandlerFunc func(http.ResponseWriter, *http.Request)
```

Параметры `http.ResponseWriter` и `*http.Request` можно использовать для создания HTTP-ответа и приема запроса соответственно. С помощью функции `http.HandleFunc` можно зарегистрировать `helloGoHandler` как функцию-обработчик для обработки любых запросов, соответствующих заданному шаблону (в этом примере – корневой путь).

После регистрации обработчика можно вызвать `ListenAndServe` для приема запросов, поступающих на адрес `addr`. Эта функция принимает также второй параметр, но в нашем примере в нем передается `nil`.

Обратите внимание, что вызов `ListenAndServe` завернут в вызов `log.Fatal`. Это связано с тем, что `ListenAndServe` всегда останавливает поток выполнения, возвращая управление только в случае ошибки. То есть она всегда возвращает ошибку, отличную от нуля, которую желательно зафиксировать.

Предыдущий пример представляет собой законченную программу, которую можно скомпилировать и запустить командой `go run`:

```
$ go run .
```

Поздравляю! Вы только что запустили пусть и маленькую, но самую настоящую службу. Теперь сделайте еще шаг вперед и протестируйте ее с помощью curl или веб-браузера:

```
$ curl http://localhost:8080
Hello net/http!
```

ListenAndServe, обработчики и мультиплексоры HTTP-запросов

Функция http.ListenAndServe запускает HTTP-сервер с заданным адресом и обработчиком. Если вместо обработчика передать nil, как это часто бывает при использовании только стандартной библиотеки net/http, то будет использован экземпляр DefaultServeMux. Но что такое обработчик? Что такое DefaultServeMux? И что такое «мультиплексор»?

Обработчик – это любой тип, определяющий метод ServeHTTP, в соответствии с требованием интерфейса Handler, со следующей сигнатурой:

```
type Handler interface {
    ServeHTTP(ResponseWriter, *Request)
}
```

Большинство реализаций обработчиков, включая обработчик по умолчанию, действуют как «мультиплексор», который может передавать входящие сигналы на один из нескольких выходов. Когда служба, запущенная функцией ListenAndServe, получает запрос, то мультиплексор сравнивает адрес URL в запросе с зарегистрированными шаблонами и вызывает функцию обработчика, соответствующую наиболее подходящему шаблону.

DefaultServeMux – это глобальный экземпляр типа ServeMux, который реализует логику по умолчанию HTTP-мультиплексора.

Создание HTTP-сервера с использованием gorilla/mux

Для многих веб-служб вполне достаточно возможностей net/http и DefaultServeMux. Однако иногда может понадобиться что-то особенное, предлагаемое только сторонними веб-инструментами. Большой популярностью пользуется библиотека Gorilla (https://oreil.ly/15sGK), которая хоть и является относительно новой, менее развитой и богатой возможностями, чем, например, Django или Flask, но все же основывается на стандартном пакете net/http и добавляет некоторые замечательные усовершенствования.

Пакет gorilla/mux – один из нескольких пакетов, входящих в состав набора веб-инструментов Gorilla, – включает в себя маршрутизатор и диспетчер HTTP-запросов, которые могут полностью заменить DefaultServeMux, обработчик по умолчанию для служб на Go, и предлагают несколько очень полезных улучшений, касающихся маршрутизации и обработки запросов. Мы пока не будем использовать эти функции, но они пригодятся в будущем. Однако если вам интересно, то можете заглянуть в документацию gorilla/mux (https://oreil.ly/qflph), чтобы получить дополнительную информацию.

Создание минимальной службы

Для этого достаточно задействовать минимальный маршрутизатор `gorilla/mux`, добавив инструкцию импорта и одну строку кода, инициализирующую новый маршрутизатор перед передачей функции `ListenAndServe` в параметре `handler`:

```
package main

import (
    "log"
    "net/http"

    "github.com/gorilla/mux"
)

func helloMuxHandler(w http.ResponseWriter, r *http.Request) {
    w.Write([]byte("Hello gorilla/mux!\n"))
}

func main() {
    r := mux.NewRouter()

    r.HandleFunc("/", helloMuxHandler)

    log.Fatal(http.ListenAndServe(":8080", r))
}
```

После этого можно запустить службу с помощью `go run`, верно? Попробуйте:

```
$ go run .
main.go:7:5: cannot find package "github.com/gorilla/mux" in any of:
        /go/1.15.8/libexec/src/github.com/gorilla/mux (from $GOROOT)
        /go/src/github.com/gorilla/mux (from $GOPATH)
```

Как оказывается – нет (пока). Поскольку теперь используется сторонний пакет – пакет не из стандартной библиотеки, – вам придется использовать модули Go.

Инициализация проекта с помощью модулей Go

Использование пакета не из стандартной библиотеки требует использования модулей Go (https://oreil.ly/QJzOi), которые появились в Go 1.12 и заменили рудиментарную систему управления зависимостями более явной и простой в использовании. Все операции, которые вы будете использовать для управления зависимостями, сводятся к одной из небольшой группы команд `go mod`.

Первое, что вам нужно сделать, – инициализировать свой проект. Начните с создания нового пустого каталога, перейдите в него и создайте там (или переместите) файл службы на Go. После этого каталог должен содержать только один файл Go.

Затем выполните команду `go mod init`, чтобы инициализировать проект. Обычно, если проект предполагается импортировать в другие проекты, он

должен инициализироваться с его путем импорта. Однако для автономной службы, такой как наша, это не важно, поэтому можете не особенно заботиться о выборе имени. Я просто использую имя `example.com/gorilla`; вы можете использовать любое другое, какое захотите:

```
$ go mod init example.com/gorilla
go: creating new go.mod: module example.com/gorilla
```

После этого в вашем каталоге появится пустой (почти) файл модуля `go.mod`[1]:

```
$ cat go.mod
module example.com/gorilla

go 1.15
```

Затем нужно добавить зависимости, что можно сделать автоматически с помощью `go mod tidy`:

```
$ go mod tidy
go: finding module for package github.com/gorilla/mux
go: found github.com/gorilla/mux in github.com/gorilla/mux v1.8.0
```

Если теперь заглянуть в файл `go.mod`, то можно увидеть, что в него добавилась зависимость (с номером версии):

```
$ cat go.mod
module example.com/gorilla

go 1.15

require github.com/gorilla/mux v1.8.0
```

Вы не поверите, но это все, что нужно. Если ваши зависимости изменятся в будущем, то просто снова запустите `go mod tidy`, чтобы обновить файл. Теперь опять попробуйте запустить службу:

```
$ go run .
```

Поскольку служба выполняется на переднем плане, терминал должен приостановиться. Вызов конечной точки с помощью `curl` из другого терминала или из веб-браузера должен дать ожидаемый ответ:

```
$ curl http://localhost:8080
Hello gorilla/mux!
```

Отлично! Но вы же наверняка хотите, чтобы ваша служба делала что-то большее, чем просто выводила приветствие, верно? Конечно! Читайте дальше!

Переменные в путях URI

Библиотека веб-инструментов Gorilla предлагает множество дополнительных возможностей, по сравнению со стандартным пакетом `net/http`, и одна

[1] Ну разве это не восхитительно?

из них представляет для нас особый интерес: возможность создавать пути с переменными сегментами, которые могут даже содержать шаблон регулярного выражения. Используя пакет gorilla/mux, программист может определять переменные в формате {name} или {name:pattern} следующим образом:

```
r := mux.NewRouter()
r.HandleFunc("/products/{key}", ProductHandler)
r.HandleFunc("/articles/{category}/", ArticlesCategoryHandler)
r.HandleFunc("/articles/{category}/{id:[0-9]+}", ArticleHandler)
```

Получить значения переменных в виде map[string]string можно с помощью функции mux.Vars:

```
vars := mux.Vars(request)
category := vars["category"]
```

В следующем разделе мы используем эту возможность, чтобы дать клиентам возможность выполнять операции с произвольными ключами.

Множество сопоставлений

Еще одна особенность пакета gorilla/mux – поддержка *сопоставления* маршрутов, что позволяет добавлять дополнительные критерии выбора. К ним относятся сопоставление с определенными доменами или поддоменами, с префиксами путей, схемами, заголовками и даже нестандартные сопоставления с использованием функций, определяемых вами.

Сопоставление осуществляется путем вызова соответствующих методов экземпляра *Route, возвращаемого реализацией HandleFunc в Gorilla. Каждая функция, выполняющая сопоставление, возвращает обратно экземпляр *Route, что позволяет объединять их в цепочки. Например:

```
r := mux.NewRouter()

r.HandleFunc("/products", ProductsHandler).
    Host("www.example.com").      // Соответствует только конкретному домену
    Methods("GET", "PUT").        // Соответствует только методам GET+PUT
    Schemes("http")               // Соответствует только схеме http
```

Исчерпывающий список функций сопоставления вы найдете в документации к пакету gorilla/mux (https://oreil.ly/6ztZe).

Создание службы RESTful

Теперь, познакомившись со стандартной библиотекой HTTP в Go, вы сможете использовать ее для создания службы RESTful, с которой клиент может взаимодействовать для вызова функций API, созданных в разделе «Наш суперпростой API» выше. Сделав это, вы получите минимально возможное хранилище пар ключ/значение.

Методы RESTful

Мы постараемся максимально точно следовать соглашениям RESTful, поэтому наш API будет интерпретировать каждую пару ключ/значение как отдельный ресурс с отдельным URI, которым можно манипулировать с использованием разных HTTP-методов. Все три наши основные операции – `Put`, `Get` и `Delete` – будут вызываться с использованием разных HTTP-методов, перечисленных в табл. 5.1.

URI, представляющий ресурсы (пары ключ/значение), будет иметь форму `/v1/key/{key}`, где `{key}` – уникальная строка ключа. Сегмент `v1` определяет версию API. Это соглашение часто используется для управления изменениями в API, и хотя данная практика никоим образом не является ни обязательной, ни универсальной, она может пригодиться для управления совместимостью будущих изменений, которые могут нарушить интеграцию существующих клиентов.

Таблица 5.1. Методы RESTful

Операция	Метод	Возможные коды возврата
Добавить пару ключ/значение в хранилище	`PUT`	`201 (Created)`
Прочитать пару ключ/значение из хранилища	`GET`	`200 (OK)`, `404 (Not Found)`
Удалить пару ключ/значение	`DELETE`	`200 (OK)`

В разделе «Переменные в путях URI» выше мы узнали, как с помощью пакета `gorilla/mux` создавать пути, содержащие переменные сегменты. Эта возможность позволит нам определить единственный путь с переменной частью, соответствующий *всем* ключам, что избавит нас от необходимости регистрировать каждый ключ отдельно. Затем, в разделе «Множество сопоставлений», мы познакомились с поддержкой сопоставления маршрутов для выбора определенной функции-обработчика на основе различных критериев, благодаря которой можно создать отдельную функцию-обработчик для каждого из пяти HTTP-методов.

Реализация функции создания

Теперь у нас есть все необходимое, и мы можем приступить к созданию службы RESTful! Давайте сначала реализуем функцию-обработчик для создания пар ключ/значение. Эта функция должна удовлетворять нескольким требованиям:

- она должна получать только запросы `PUT` с URI `/v1/key/{key}`;
- она должна вызывать метод `Put` из раздела «Наш суперпростой API»;
- она должна возвращать ответ `201 (Created)` в случае успешного создания пары ключ/значение;
- она должна возвращать ответ `500 (Internal Server Error)`, столкнувшись с непредвиденной ошибкой.

Все эти требования реализованы в функции `keyValuePutHandler`. Обратите внимание, что значение извлекается из тела запроса:

```
// keyValuePutHandler ожидает получить PUT-запрос с
// ресурсом "/v1/key/{key}".
func keyValuePutHandler(w http.ResponseWriter, r *http.Request) {
    vars := mux.Vars(r)               // Получить ключ из запроса
    key := vars["key"]

    value, err := io.ReadAll(r.Body)  // Тело запроса хранит значение
    defer r.Body.Close()

    if err != nil {                   // Если возникла ошибка, сообщить о ней
        http.Error(w,
            err.Error(),
            http.StatusInternalServerError)
        return
    }

    err = Put(key, string(value))     // Сохранить значение как строку
    if err != nil {                   // Если возникла ошибка, сообщить о ней
        http.Error(w,
            err.Error(),
            http.StatusInternalServerError)
        return
    }

    w.WriteHeader(http.StatusCreated) // Все хорошо! Вернуть StatusCreated
}
```

Имея функцию-обработчик «создания пары ключ/значение», ее можно зарегистрировать с помощью маршрутизатора Gorilla для обработки запросов с конкретными характеристиками:

```
func main() {
    r := mux.NewRouter()

    // Зарегистрировать keyValuePutHandler как обработчик HTTP-запросов PUT,
    // в которых указан путь "/v1/{key}"
    r.HandleFunc("/v1/{key}", keyValuePutHandler).Methods("PUT")

    log.Fatal(http.ListenAndServe(":8080", r))
}
```

Теперь можно запустить службу командой `go run .` из корневого каталога проекта. Сделайте это и отправьте службе несколько запросов, чтобы узнать, как она отреагирует.

Сначала с помощью нашего старого знакомого `curl` отправьте запрос `PUT` с коротким фрагментом текста в конечную точку `/v1/key-a`, чтобы создать ключ `key-a` со значением `Hello, key-value store!`:

```
$ curl -X PUT -d 'Hello, key-value store!' -v http://localhost:8080/v1/key-a
```

Эта команда выведет строки, как показано ниже. Полный вывод получился довольно объемным, поэтому я выбрал лишь интересующие нас строки:

```
> PUT /v1/key-a HTTP/1.1
< HTTP/1.1 201 Created
```

Первая строка, начинающаяся с символа «больше» (>), показывает некоторые сведения о запросе. Вторая строка, начинающаяся с символа «меньше» (<), сообщает подробную информацию об ответе сервера.

Как показывает этот вывод, мы действительно послали запрос `PUT` в конечную точку `/v1/key-a`, и сервер ответил кодом `201 Created`, как и ожидалось.

А что произойдет, если в конечную точку `/v1/key-a` послать пока не поддерживаемый запрос `GET`? Если функция сопоставления работает правильно, то мы должны получить сообщение об ошибке:

```
$ curl -X GET -v http://localhost:8080/v1/key-a
> GET /v1/key-a HTTP/1.1
< HTTP/1.1 405 Method Not Allowed
```

Как видите, сервер действительно вернул ошибку `405 Method Not Allowed`. Похоже, что все работает правильно.

Реализация функции чтения

Теперь, получив полностью работающий метод `Put`, было бы неплохо реализовать возможность прочитать свои данные обратно! Далее мы реализуем функцию `Get`, к которой предъявляются следующие требования:

- она должна получать только запросы `GET` с URI `/v1/key/{key}`;
- она должна вызывать метод `Get` из раздела «Наш суперпростой API»;
- она должна возвращать ответ `404 (Not Found)`, если запрошенный ключ не существует;
- она должна возвращать запрошенное значение и код `200`, если ключ существует;
- она должна возвращать ответ `500 (Internal Server Error)`, столкнувшись с непредвиденной ошибкой.

Все эти требования реализованы в функции `keyValueGetHandler`. Обратите внимание, что значение записывается в `w` – параметр типа `http.Response-Writer` функции-обработчика – после получения из API хранилища пар ключ/значение:

```
func keyValueGetHandler(w http.ResponseWriter, r *http.Request) {
    vars := mux.Vars(r)        // Извлечь ключ из запроса
    key := vars["key"]

    value, err := Get(key)     // Получить значение для данного ключа
    if errors.Is(err, ErrorNoSuchKey) {
        http.Error(w,err.Error(), http.StatusNotFound)
        return
    }

    if err != nil {
        http.Error(w, err.Error(), http.StatusInternalServerError)
        return
    }
```

```
    w.Write([]byte(value))     // Записать значение в ответ
}
```

А теперь зарегистрируем функцию-обработчик «get», действуя по аналогии с функцией-обработчиком «put»:

```
func main() {
    r := mux.NewRouter()

    r.HandleFunc("/v1/{key}", keyValuePutHandler).Methods("PUT")
    r.HandleFunc("/v1/{key}", keyValueGetHandler).Methods("GET")

    log.Fatal(http.ListenAndServe(":8080", r))
}
```

И проверим, как работает наша обновленная служба:

```
$ curl -X PUT -d 'Hello, key-value store!' -v http://localhost:8080/v1/key-a
> PUT /v1/key-a HTTP/1.1
< HTTP/1.1 201 Created

$ curl -v http://localhost:8080/v1/key-a
> GET /v1/key-a HTTP/1.1
< HTTP/1.1 200 OK
Hello, key-value store!
```

Она действительно работает! Теперь, имея возможность получить сохраненное значение, можно также проверить идемпотентность. Давайте повторно выполним те же запросы и посмотрим, изменятся ли результаты:

```
$ curl -X PUT -d 'Hello, key-value store!' -v http://localhost:8080/v1/key-a
> PUT /v1/key-a HTTP/1.1
< HTTP/1.1 201 Created

$ curl -v http://localhost:8080/v1/key-a
> GET /v1/key-a HTTP/1.1
< HTTP/1.1 200 OK
Hello, key-value store!
```

Все в порядке! Но что, если потребуется сохранить новое значение для существующего ключа? Вернет ли последующий запрос `GET` новое значение? Для проверки изменим значение, отправляемое командой `curl`, на `Hello, again, key-value store!`:

```
$ curl -X PUT -d 'Hello, again, key-value store!' \
    -v http://localhost:8080/v1/key-a
> PUT /v1/key-a HTTP/1.1
< HTTP/1.1 201 Created

$ curl -v http://localhost:8080/v1/key-a
> GET /v1/key-a HTTP/1.1
< HTTP/1.1 200 OK
Hello, again, key-value store!
```

Как и ожидалось, на запрос GET служба ответила кодом 200 и вернула обновленное значение.

Наконец, чтобы завершить создание службы, нужно добавить обработчик запросов DELETE. Я оставлю это вам в качестве самостоятельного упражнения. Испытайте себя!

Добавление в структуру данных поддержки использования в конкурентном окружении

Операции с ассоциативными массивами в Go не атомарны и небезопасны в конкурентном окружении. К сожалению, наша служба, предназначенная для одновременной обработки нескольких запросов, использует именно такой ассоциативный массив.

И как нам быть? Обычно, когда имеется структура данных, доступ к которой может осуществляться одновременно из нескольких сопрограмм, программисты используют что-то вроде мьютекса, также известного как блокировка, чтобы синхронизировать доступ. С помощью мьютекса можно гарантировать, что только один процесс будет иметь монопольный доступ к определенному ресурсу.

К счастью, нам не нужно разрабатывать свой механизм синхронизации[1]: пакет sync в языке Go предоставляет именно то, что нужно, – sync.RWMutex. Следующая инструкция использует волшебство композиции и создает *анонимную структуру*, содержащую ассоциативный массив и встроенный мьютекс sync.RWMutex:

```
var myMap = struct{
    sync.RWMutex
    m map[string]string
}{m: make(map[string]string)}
```

Структура myMap получает все методы встроенного типа sync.RWMutex, что дает возможность вызывать метод Lock, чтобы установить блокировку для записи, когда потребуется изменить содержимое ассоциативного массива myMap:

```
myMap.Lock()                      // Установить блокировку для записи
myMap.m["some_key"] = "some_value"
myMap.Unlock()                    // Снять блокировку для записи
```

Если другой процесс уже установил блокировку для чтения или записи, то вызов метода Lock заблокирует вызывающий код до момента, когда эта блокировка будет снята.

Аналогично для чтения из ассоциативного массива используется метод RLock, устанавливающий блокировку для чтения:

---

[1] И это хорошо, потому что правильная реализация мьютексов – довольно утомительная задача!

```
myMap.RLock()                  // Установить блокировку для чтения
value := myMap.m["some_key"]
myMap.RUnlock()                // Снять блокировку для чтения

fmt.Println("some_key:", value)
```

Блокировки для чтения менее строгие, чем блокировки для записи, и одновременно может быть установлено несколько блокировок для чтения. Однако `RLock` будет блокироваться до момента, когда будет снята блокировка для записи, если она установлена.

Интеграция мьютекса чтения/записи в приложение

Теперь, узнав, как использовать `sync.RWMutex` для синхронизации операций чтения/записи, вернемся к нашей службе и добавим мьютекс в код, созданный в разделе «Наш суперпростой API».

Прежде всего нужно реорганизовать ассоциативный массив `store`[1]. Его можно определить так же, как `myMap`, то есть как анонимную структуру, содержащую ассоциативный массив и встроенный тип `sync.RWMutex`:

```
var store = struct{
    sync.RWMutex
    m map[string]string
}{m: make(map[string]string)}
```

После доработки структуры `store` можно добавить в функции `Get` и `Put` установку и снятие соответствующих блокировок. Поскольку `Get` только *читает* ассоциативный массив `store`, в ней будет вызываться метод `RLock`, устанавливающий блокировку для чтения. Функция `Put`, напротив, *изменяет* ассоциативный массив, поэтому должна вызывать метод `Lock`, чтобы получить блокировку для записи:

```
func Get(key string) (string, error) {
    store.RLock()
    value, ok := store.m[key]
    store.RUnlock()

    if !ok {
        return "", ErrorNoSuchKey
    }

    return value, nil
}

func Put(key string, value string) error {
    store.Lock()
    store.m[key] = value
    store.Unlock()

    return nil
}
```

[1] Я же говорил вам, что мы все усложним!

Шаблон прост и понятен: если функция изменяет ассоциативный массив (`Put`, `Delete`), то она должна использовать `Lock`, чтобы установить блокировку для записи. Если функции требуется только прочитать существующие данные (`Get`), то она должна использовать `RLock`, чтобы установить блокировку для чтения. Я оставляю создание функции `Delete` вам как самостоятельное упражнение.

Не забудьте снять блокировку и убедитесь, что используете блокировку правильного типа!

Итерация 2: долговременное хранение ресурса

Одна из самых серьезных проблем, связанных с распределенными облачными приложениями, – это обработка состояния.

Есть разные методы распределения состояния ресурсов приложения между несколькими экземплярами службы, но пока мы просто сосредоточимся на минимально жизнеспособном продукте и рассмотрим два способа поддержки состояния нашего приложения:

- в разделе «Сохранение состояния в журнале транзакций» ниже мы используем *журнал транзакций* для фиксации каждого изменения ресурса. Если служба потерпит сбой, перезапустится или иным образом окажется в несогласованном состоянии, то журнал транзакций позволит восстановить исходное состояние службы путем простого воспроизведения транзакций;
- в разделе «Сохранение состояния во внешней базе данных» ниже мы используем внешнюю базу данных вместо журнала транзакций. С учетом характера создаваемого приложения использование базы данных может показаться избыточным, тем не менее перенос данных в другую службу, созданную специально для этой цели, является распространенным средством обмена состоянием между экземплярами службы и обеспечения устойчивости.

Возможно, вам интересно узнать, зачем использовать стратегию на основе журнала транзакций, если можно просто использовать базу данных для хранения текущих значений. Эта стратегия имеет смысл, когда предполагается хранить данные в памяти большую часть времени и обращаться к механизму долговременного хранения только в фоновом режиме и во время запуска.

Кроме того, исследование двух стратегий дает нам еще одну возможность: поскольку предполагается создание двух разных реализаций схожей функциональности – журнала транзакций в файле и базы данных, – мы можем описать необходимую функциональность в виде интерфейса, которому будут удовлетворять обе реализации. Это может пригодиться, когда потом вы решите добавить возможность выбора реализации в соответствии с потребностями.

Состояние приложения и ресурса

В контексте облачной архитектуры часто используется термин «без состояния» (stateless), и состояние нередко рассматривается как нечто очень плохое. Но что такое состояние, и почему наличие состояния – это плохо? Должно ли приложение быть полностью лишено какого-либо состояния, чтобы быть «облачным»? На этот вопрос нет однозначного ответа.

Во-первых, важно различать *состояние приложения* и *состояние ресурса*. Это разные вещи, но их легко спутать.

Состояние приложения
: Это данные на стороне сервера о приложении или о том, как оно используется клиентом. Типичным примером является информация об открытых клиентами сеансах, например для доступа к их учетным данным или какой-либо другой контекстной информации.

Состояние ресурса
: Текущее состояние ресурса в службе в любой момент времени. Это состояние одинаково для каждого клиента и никак не связано со взаимодействиями между клиентом и сервером.

Поддержка любого состояния порождает технические проблемы, но поддержка состояния приложения особенно проблематична, потому что заставляет службы зависеть от *привязки к серверу* – отправлять каждый запрос пользователя на тот же сервер, на котором был инициирован сеанс, – что приводит к усложнению приложения, затрудняет уничтожение или замену одного экземпляра службы другим.

Более подробно проблемы служб с состоянием и без состояния будут обсуждаться в разделе «С состоянием и без состояния» главы 7.

Что такое журнал транзакций?

В простейшем случае *журнал транзакций* – это просто файл, в котором хранится история изменений в хранилище данных. Если служба потерпит сбой, перезапустится или иным образом окажется в несогласованном состоянии, то журнал транзакций позволит восстановить исходное состояние службы путем простого воспроизведения транзакций.

Журналы транзакций широко используются в системах управления базами данных для обеспечения устойчивости к сбоям или отказам оборудования. И хотя этот метод может быть довольно сложным, мы постараемся реализовать его как можно проще.

Формат журнала транзакций

Прежде чем перейти к коду, давайте решим, что должен содержать журнал транзакций.

Предполагается, что журнал транзакций будет читаться только при перезапуске службы или когда ей потребуется восстановить свое состояние, и чтение будет производиться сверху вниз с последовательным воспроизведением каждого события. То есть журнал транзакций должен содержать

упорядоченный список событий изменения содержимого хранилища. Кроме того, запись в журнал транзакций обычно осуществляется только в конец, поэтому, например, когда пара ключ/значение удаляется из хранилища, в журнал записывается соответствующее событие удаления.

Учитывая все, что мы обсуждали до сих пор, каждое событие (или транзакция), записываемое в журнал, должно включать следующие атрибуты:

Порядковый номер
Уникальный идентификатор записи – монотонно увеличивающееся число.

Тип события
Дескриптор типа выполненного действия; это может быть `PUT` или `DELETE`.

Ключ
Строка с ключом, затронутым этой транзакцией.

Значение
Для события `PUT` определяет значение для заданного ключа.

Просто и красиво. Надеюсь, мы сможем сохранить эти простоту и красоту.

Интерфейс регистратора транзакций

Первое, что нужно сделать, – определить интерфейс `TransactionLogger`. На данный момент предполагается определить только два метода: `WritePut` и `WriteDelete`, которые будут записывать в журнал события `PUT` и `DELETE` соответственно:

```
type TransactionLogger interface {
    WriteDelete(key string)
    WritePut(key, value string)
}
```

Позднее нам несомненно потребуется добавить другие методы, но мы рассмотрим их, когда придет время. А пока сосредоточимся на первой реализации и будем добавлять новые методы в интерфейс по мере их появления.

Сохранение состояния в журнале транзакций

Для начала используем самую простую (и наиболее распространенную) форму журнала транзакций – обычный файл, в который новые записи добавляются в конец и хранящий историю изменений в хранилище. Реализация на основе файла имеет несколько заманчивых плюсов, но есть и довольно существенные минусы.

Плюсы

Отсутствие нижестоящих зависимостей
Реализация журнала не зависит от внешних служб, которые могут выйти из строя или к которым можно потерять доступ.

Техническая простота
Логика не особо сложная. Мы можем быстро реализовать и запустить ее.

Минусы

Сложность масштабирования
Понадобится дополнительный механизм распределения состояния между узлами, когда потребуется реализовать поддержку масштабирования.

Неконтролируемый рост
Журналы должны храниться на диске, поэтому мы не можем позволить им расти до бесконечности. Соответственно, нам понадобится некоторый способ их уплотнения.

Создание прототипа регистратора транзакций

Прежде чем мы перейти к коду, давайте примем несколько проектных решений. Во-первых, для простоты журнал будет храниться в виде обычного текстового файла; двоичный сжатый формат может оказаться более эффективным, но мы сможем оптимизировать его позже. Во-вторых, каждая запись будет занимать отдельную строку; это значительно упростит чтение данных позже.

Наконец, каждая запись будет включать четыре поля, перечисленных в разделе «Формат журнала транзакций» выше, которые будут разделяться символами табуляции. Вспомним эти поля:

Порядковый номер
Уникальный идентификатор записи – монотонно увеличивающееся число.

Тип события
Дескриптор типа выполненного действия; это может быть `PUT` или `DELETE`.

Ключ
Строка с ключом, затронутым этой транзакцией.

Значение
Для события `PUT` определяет значение для заданного ключа.

Теперь, заложив основы, продолжим и определим тип `FileTransactionLogger`, неявно реализующий интерфейс `TransactionLogger`, описанный в разделе «Интерфейс регистратора транзакций» выше, объявляя методы `WritePut` и `WriteDelete` для записи событий `PUT` и `DELETE` в журнал:

```
type FileTransactionLogger struct {
    // Некоторые поля
}

func (l *FileTransactionLogger) WritePut(key, value string) {
    // Некоторая логика
}

func (l *FileTransactionLogger) WriteDelete(key string) {
```

```
    // Некоторая логика
}
```

Сигнатуры методов не особенно проясняют детали, но мы скоро их уточним!

Определение типа события

Забегая вперед, отмечу, что методы `WritePut` и `WriteDelete` должны работать асинхронно. Мы можем обеспечить такую асинхронность, используя некоторый канал событий `events`, откуда некоторая сопрограмма, выполняющаяся конкурентно, будет читать и воспроизводить записи. Идея хорошая, но в этом случае нам понадобится какое-то внутреннее представление «события».

Создание такого представления не должно доставить нам особых хлопот. Включив все поля, перечисленные в разделе «Формат журнала транзакций», мы получаем примерно такую структуру `Event`:

```
type Event struct {
    Sequence uint64     // Уникальный порядковый номер записи
    EventType EventType // Выполненное действие
    Key string          // Ключ, затронутый этой транзакцией
    Value string        // Значение для транзакции PUT
}
```

Выглядит довольно просто, верно? `Sequence` – это порядковый номер, а `Key` и `Value` не требуют пояснений. Но… что такое `EventType`? Именно об этом мы сейчас и говорим – это константа, которую можно использовать для обозначения различных типов событий и которая, как мы уже определили, будет включать по одному значению для событий `PUT` и `DELETE`.

Один из способов определить типы событий – просто присвоить им некоторые постоянные значения, например:

```
const (
    EventDelete byte = 1
    EventPut    byte = 2
)
```

Это вполне жизнеспособное решение, но в Go есть способ более удобный (и идиоматичный) способ: `iota`. `iota` – это предопределенное значение, которое можно использовать в объявлении `const` для создания последовательности связанных значений.

Использование приема с `iota` избавляет от необходимости вручную присваивать значения константам. Вместо этого можно записать такое определение:

```
type EventType byte

const (
    _                   = iota // iota == 0; игнорировать нулевое значение
    EventDelete EventType = iota // iota == 1
    EventPut                     // iota == 2; неявное присваивание
)
```

Объявление констант с использованием iota

В объявлении константы `iota` представляет последовательно увеличивающиеся нетипизированные целочисленные значения, которые можно использовать для создания набора связанных констант. В начале каждого объявления `const` идентификатор `iota` получает нулевое значение и затем увеличивается с каждой операцией присваивания значения новой константе (независимо от использования ссылки на идентификатор `iota`).

`iota` может также использоваться в выражениях, как показано в примере ниже, где применяются операции умножения, поразрядного сдвига влево и деления:

```
const (
    a = 42 * iota // iota == 0; a == 0
    b = 1 << iota // iota == 1; b == 2
    c = 3         // iota == 2; c == 3 (iota постоянно увеличивается!)
    d = iota / 2  // iota == 3; d == 1
)
```

Поскольку `iota` является нетипизированным числом, его можно использовать для присваивания типизированных значений без явного приведения типов, даже константе с типом `float64`:

```
const (
    u = iota * 42         // iota == 0; u == 0 (нетипизированная целочисленная
                          // константа)
    v float64 = iota * 42 // iota == 1; v == 42.0 (константа типа float64)
)
```

Ключевое слово `iota` допускает неявное перечисление, что делает тривиальным создание произвольно длинных наборов связанных констант, как это показано в следующем примере, где перечисляются различные единицы измерения размеров в байтах:

```
type ByteSize uint64

const (
    _           = iota             // iota == 0; игнорировать нулевое значение
    KB ByteSize = 1 << (10 * iota) // iota == 1; KB == 2^10
    MB                             // iota == 2; MB == 2^20
    GB                             // iota == 3; GB == 2^30
    TB                             // iota == 4; TB == 2^40
    PB                             // iota == 5; PB == 2^50
)
```

Для двух констант выбор того или иного способа их определения не имеет большого значения, но второй способ намного удобнее, когда требуется определить большую группу связанных констант, потому что избавляет от необходимости просматривать код, чтобы выяснить, какое значение уже было присвоено.

Если вы будете использовать `iota` для определения последовательных перечислений (как мы в этом примере), то добавляйте новые члены перечисления *только в конец списка*, но не переупорядочивайте и не вставляйте значения в середину, иначе потом вы рискуете столкнуться с трудностями.

Теперь мы знаем, как выглядит тип `TransactionLogger` и что он имеет два основных метода записи. Мы также определили структуру, описывающую отдельные события, создали новый тип `EventType` и использовали `iota` для определения допустимых значений. Мы, наконец, готовы приступить к работе.

Реализация FileTransactionLogger

Мы добились определенного прогресса и знаем, что нам нужна реализация `TransactionLogger` с методами для записи событий. Мы описали события в коде. Но как насчет типа `FileTransactionLogger`?

Служба должна знать, где физически находится журнал транзакций, поэтому имеет смысл добавить атрибут `os.File` для представления этой информации. Также регистратор должен запоминать порядковый номер последней записи, чтобы присвоить правильный порядковый номер следующей записи – этот номер можно хранить в виде 64-битного целого числа без знака. Все это хорошо, но как `FileTransactionLogger` будет фактически записывать события?

Одно из возможных решений – добавить атрибут `io.Writer`, который могли бы использовать методы `WritePut` и `WriteDelete`, но это немасштабируемое решение, и, выбрав его, вы можете обнаружить, что на ввод/вывод в нескольких параллельно выполняющихся сопрограммах тратится больше времени, чем хотелось бы. Альтернативное решение – создать буфер в форме среза с элементами типа `Event`, который будет обслуживаться отдельной сопрограммой. Это уже теплее, но слишком сложно.

В конце концов, зачем делать всю эту работу, если можно просто использовать стандартные буферизованные каналы? Учитывая вышесказанное, получаем следующие определения типа `FileTransactionLogger` и методов `Write*`:

```
type FileTransactionLogger struct {
    events       chan<- Event // Канал только для записи; для передачи событий
    errors       <-chan error // Канал только для чтения; для приема ошибок
    lastSequence uint64       // Последний использованный порядковый номер
    file         *os.File     // Местоположение файла журнала
}

func (l *FileTransactionLogger) WritePut(key, value string) {
    l.events <- Event{EventType: EventPut, Key: key, Value: value}
}

func (l *FileTransactionLogger) WriteDelete(key string) {
    l.events <- Event{EventType: EventDelete, Key: key}
}

func (l *FileTransactionLogger) Err() <-chan error {
    return l.errors
}
```

Теперь у нас есть структура `FileTransactionLogger` с полем `uint64` для хранения последнего использованного порядкового номера события и с каналом только для записи, через который регистратор будет получать экземпля-

ры событий, а также методы `WritePut` и `WriteDelete`, отправляющие события в этот канал.

Добавился еще метод `Err`, который возвращает канал, доступный только для приема, через который передаются экземпляры ошибок. Этот канал мы добавили не просто так. Как уже упоминалось, запись событий в журнал транзакций будет выполняться конкурентно с помощью сопрограммы, извлекающей события из канала `events`. Это не только обеспечит более высокую эффективность записи, но также означает, что `WritePut` и `WriteDelete` не смогут просто вернуть ошибку, столкнувшись с проблемой, поэтому мы предоставляем специальный канал для передачи сообщений об ошибках.

Создание экземпляра FileTransactionLogger

Если вы внимательно следили за развитием примера, то наверняка заметили, что ни один из атрибутов в `FileTransactionLogger` не был инициализирован. Этот недостаток нужно исправить, иначе могут возникнуть некоторые другие проблемы. Однако в Go нет конструкторов, поэтому для решения этой задачи нам необходимо определить функцию, и, за неимением лучшего имени[1], назовем ее NewFileTransactionLogger:

```
func NewFileTransactionLogger(filename string) (TransactionLogger, error) {
    file, err := os.OpenFile(filename, os.O_RDWR|os.O_APPEND|os.O_CREATE, 0755)
    if err != nil {
        return nil, fmt.Errorf("cannot open transaction log file: %w", err)
    }

    return &FileTransactionLogger{file: file}, nil
}
```

Обратите внимание, что функция `NewFileTransactionLogger` возвращает тип указателя, но в ее списке возвращаемых значений явно указан тип интерфейса `TransactionLogger`, не являющийся указателем. Как такое возможно?

Причина в том, что Go поддерживает типы указателей на реализации интерфейсов, но не поддерживает типы указателей на интерфейсы.

`NewFileTransactionLogger` вызывает функцию `os.OpenFile`, чтобы открыть файл с именем в параметре `filename`. Обратите внимание, что ей передается несколько флагов, объединенных оператором поразрядного ИЛИ (`|`), определяющих некоторые особенности поведения функции:

`os.O_RDWR`
: Открывает файл в режиме чтения/записи.

`os.O_APPEND`
: Любые записи в этот файл будут добавляться в конец.

---

[1] Конечно, это неправда. Наверняка можно подобрать множество гораздо более удачных имен.

`os.O_CREATE`
Если файл отсутствует, то он должен быть создан.

Таких флагов довольно много, помимо этих трех, которые мы здесь используем. Загляните в документацию с описанием пакета `os` (https://pkg.go.dev/os), чтобы увидеть полный список.

Теперь у нас есть функция, гарантирующая корректное создание файла журнала транзакций. Но как быть с каналами? Мы *могли бы* создать каналы и запустить сопрограмму в `NewFileTransactionLogger`, но тогда мы скроем слишком много функциональности за завесой таинственности. Давайте вместо этого создадим метод `Run`.

Добавление записей в конец журнала транзакций

Наш канал `events` пока бездействует, что удручает. Хуже того, каналы даже не инициализированы. Давайте исправим этот недостаток, создав метод `Run`, как показано ниже:

```
func (l *FileTransactionLogger) Run() {
    events := make(chan Event, 16)     // Создать канал событий
    l.events = events

    errors := make(chan error, 1)      // Создать канал ошибок
    l.errors = errors

    go func() {
        for e := range events {        // Извлечь следующее событие Event

            l.lastSequence++           // Увеличить порядковый номер

            _, err := fmt.Fprintf(     // Записать событие в журнал
                l.file,
                "%d\t%d\t%s\t%s\n",
                l.lastSequence, e.EventType, e.Key, e.Value)

            if err != nil {
                errors <- err
                return
            }
        }
    }()
}
```

Это весьма упрощенная реализация. Он не будет правильно обрабатывать записи с пробелами или с несколькими строками!

Функция `Run` выполняет несколько важных действий.

Во-первых, создает буферизованный канал `events`. Использование буферизованного канала в `TransactionLogger` означает, что вызовы `WritePut` и `WriteDelete` не будут блокироваться, пока буфер не заполнится. Это позволит службе-потребителю обрабатывать короткие всплески событий без замедления из-за

дискового ввода/вывода. Если буфер заполнится, то методы записи будут блокироваться до момента, когда сопрограмма записи в журнал освободит в нем место.

Во-вторых, создает канал `errors`, также буферизованный, который мы будем использовать для сигнализации о любых ошибках, возникающих в сопрограмме, которая отвечает за запись событий в журнал транзакций. Установка размера буфера равным 1 позволит нам отправлять ошибки без приостановки.

Наконец, запускает сопрограмму, которая извлекает экземпляры `Event` событий из канала `events` и использует функцию `fmt.Fprintf` для записи их в журнал транзакций. Если `fmt.Fprintf` возвращает ошибку, сопрограмма отправляет ее в канал `errors` и останавливается.

Использование bufio.Scanner для воспроизведения транзакций из журнала

Даже самый лучший журнал транзакций бесполезен, если он никогда не будет читаться[1]. Но как это сделать?

Мы должны читать журнал с самого начала и разбирать каждую строку; `io.ReadString` и `fmt.Sscanf` помогут сделать это с минимальными усилиями.

Каналы, наши верные друзья, помогут службе организовать потоковую передачу результатов потребителю. Такой подход может показаться излишне утомительным, но давайте приостановимся ненадолго и задумаемся. В большинстве других языков простейшее решение заключается в том, чтобы прочитать весь файл в массив и выполнить обход элементов этого массива, чтобы воспроизвести события. Удобные примитивы конкуренции в Go упрощают потоковую передачу данных потребителю, обеспечивающую гораздо более эффективное расходование памяти.

Метод `ReadEvents`[2] демонстрирует это:

```
func (l *FileTransactionLogger) ReadEvents() (<-chan Event, <-chan error) {
    scanner := bufio.NewScanner(l.file) // Создать Scanner для чтения l.file
    outEvent := make(chan Event)        // Небуферизованный канал событий
    outError := make(chan error, 1)     // Буферизованный канал ошибок

    go func() {
        var e Event

        defer close(outEvent)           // Закрыть каналы
        defer close(outError)           // по завершении сопрограммы

        for scanner.Scan() {
            line := scanner.Text()

            if err := fmt.Sscanf(line, "%d\t%d\t%s\t%s",
                &e.Sequence, &e.EventType, &e.Key, &e.Value); err != nil {
```

[1] Что вообще делает журнал транзакций «хорошим»?

[2] Выбор хорошего имени – сложная задача.

```
                outError <- fmt.Errorf("input parse error: %w", err)
                return
            }

            // Проверка целостности!
            // Порядковые номера последовательно увеличиваются?
            if l.lastSequence >= e.Sequence {
                outError <- fmt.Errorf("transaction numbers out of sequence")

                return
            }

            l.lastSequence = e.Sequence // Запомнить последний использованный
                                        // порядковый номер

            outEvent <- e               // Отправить событие along
        }

        if err := scanner.Err(); err != nil {
            outError <- fmt.Errorf("transaction log read failure: %w", err)
            return
        }
    }()

    return outEvent, outError
}
```

Можно сказать, что метод `ReadEvents` – это две функции в одной: внешняя функция инициализирует средство чтения файлов, а также создает и возвращает каналы событий и ошибок. Внутренняя функция выполняется конкурентно и построчно извлекает содержимое из файла и отправляет результаты в каналы.

Интересно отметить, что атрибут `file` в `TransactionLogger` имеет тип `*os.File`, у которого есть метод `Read`, который удовлетворяет интерфейсу `io.Reader`. Метод `Read` довольно низкоуровневый, но при желании его вполне можно использовать для извлечения данных. Однако пакет `bufio` предлагает более удобный способ: интерфейс `Scanner`, дающий возможность чтения текстовых строк, разделенных символом перевода строки. Получить экземпляр `Scanner` можно, передав `io.Reader` – в данном случае `os.File` – в `bufio.NewScanner`.

Каждый вызов метода `scanner.Scan` переносит внутренний указатель в начало следующей строки и возвращает `false`, если достигнут конец файла. Последующий вызов `scanner.Text` возвращает строку.

Обратите внимание на оператор `defer` во внутренней анонимной сопрограмме. Он гарантирует закрытие каналов по завершении сопрограммы. Оператор `defer` привязан к области видимости функции, в которой объявлен, поэтому будет вызван в конце сопрограммы, а не `ReadEvents`.

Как рассказывалось во врезке «Форматирование ввода/вывода в Go» в главе 3, функция `fmt.Sscanf` предлагает простые (иногда чересчур упрощенные) средства синтаксического анализа обычных строк. Как и другие методы в пакете `fmt`, ожидаемый формат задается с помощью строки формата, содержа-

щей различные встроенные спецификаторы («глаголы»): в данном случае мы ожидаем два числа (%d) и две строки (%s), разделенные символами табуляции (\t). Очень удобно, что fmt.Sscanf позволяет передавать указатели на целевые значения, соответствующие спецификаторам, куда он сможет записать прочитанные данные[1].

Строки формата в Go имеют долгую историю, восходящую к printf и scanf в языке C. За прошедшие годы они были реализованы во многих других языках, включая C++, Java, Perl, PHP, Ruby и Scala. Возможно, вы уже знакомы с ними, а если нет, то сделайте перерыв и загляните в документацию с описанием пакета fmt (https://pkg.go.dev/fmt).

В конце каждой итерации последний использованный порядковый номер обновляется только что прочитанным значением, и событие отправляется своим путем. Небольшое замечание: обратите внимание, что в каждой итерации повторно используется один и тот же экземпляр Event, а не создается новый. Такое возможно, потому что канал outEvent отправляет значения структуры, а не указатели на значения структуры, то есть он копирует любые значения, которые мы отправляем в него.

Наконец, функция проверяет наличие ошибок в Scanner. Метод Scan возвращает единственное логическое значение, что действительно удобно для организации обхода в цикле. Столкнувшись с ошибкой, Scan возвращает false, а мы получаем ее с помощью метода Err.

Интерфейс регистратора транзакций (еще раз)

Теперь, закончив реализацию FileTransactionLogger, можно оглянуться назад и посмотреть, какие новые методы можно включить в интерфейс TransactionLogger. На самом деле таких методов, которые желательно было бы иметь в каждой реализации, довольно мало. В результате мы приходим к следующему и окончательному определению интерфейса TransactionLogger:

```
type TransactionLogger interface {
    WriteDelete(key string)
    WritePut(key, value string)
    Err() <-chan error

    ReadEvents() (<-chan Event, <-chan error)

    Run()
}
```

Теперь, зафиксировав интерфейс, мы наконец можем приступить к интеграции журнала транзакций в службу хранилища пар ключ/значение.

Инициализация FileTransactionLogger в веб-службе

Реализация FileTransactionLogger завершена! Осталось лишь интегрировать ее в нашу веб-службу. Первым делом добавим функцию, которая будет

[1] Даже спустя годы я считаю это удобным.

создавать экземпляр TransactionLogger, читать и воспроизводить любые существующие события и затем вызывать метод Run.

Для начала добавим ссылку на TransactionLogger в наш файл service.go. Назовем ее logger, потому что трудно подобрать более подходящее имя:

```
var logger TransactionLogger
```

Добавив эту деталь, можно определить метод инициализации, как показано ниже:

```
func initializeTransactionLog() error {
    var err error

    logger, err = NewFileTransactionLogger("transaction.log")
    if err != nil {
        return fmt.Errorf("failed to create event logger: %w", err)
    }

    events, errors := logger.ReadEvents()
    e, ok := Event{}, true

    for ok && err == nil {
        select {
        case err, ok = <-errors: // Получает ошибки
        case e, ok = <-events:
            switch e.EventType {
            case EventDelete:    // Получено событие DELETE!
                err = Delete(e.Key)
            case EventPut:       // Получено событие PUT!
                err = Put(e.Key, e.Value)
            }
        }
    }

    logger.Run()

    return err
}
```

Эта функция начинается, как и следовало ожидать, с вызова NewFileTransactionLogger и сохранения экземпляра регистратора в logger.

Следующая часть функции гораздо интереснее: в ней она вызывает logger.ReadEvents и воспроизводит результаты в зависимости от типов событий. Для этого в цикле выполняется инструкция select с вариантами case, выполняющими прием из обоих каналов – events и errors. Обратите внимание, что в инструкции select используется формат записи вариантов case foo, ok = <-ch. В этом случае, если канал закроется, в ok будет записано логическое значение false, и цикл for завершится.

Получив экземпляр Event из канала events, мы вызываем Delete или Put, в зависимости от ситуации; если будет получена ошибка из канала errors, то err получит значение, отличное от nil, и цикл for завершится.

Интеграция FileTransactionLogger в веб-службу

Теперь, реализовав логику инициализации, нам остается только добавить ровно три вызова в веб-службу, чтобы завершить интеграцию `TransactionLogger`. Сделать это просто, и поэтому я не буду вдаваться в подробные рассуждения, а просто перечислю вызовы, которые нужно добавить:

- `initializeTransactionLog` в метод `main`;
- `logger.WriteDelete` в `keyValueDeleteHandler`;
- `logger.WritePut` в `keyValuePutHandler`.

Фактическую реализацию интеграции я оставляю вам в качестве самостоятельного упражнения[1].

Будущие улучшения

Мы завершили минимальную жизнеспособную реализацию регистратора транзакций, но у него полно недостатков, которые желательно устранить, например:

- отсутствуют тесты;
- нет метода `Close` для корректного закрытия файла;
- служба может закрыться, когда события все еще находятся в буфере записи: в этом случае события могут быть потеряны;
- ключи и значения никак не кодируются в журнале транзакций: если событие будет содержать несколько строк или пробелов, то его не удастся правильно проанализировать;
- размеры ключей и значений не ограничены: служба позволяет добавлять огромные ключи или значения, что может привести к переполнению диска;
- журнал транзакций хранится в простом текстовом виде: он будет занимать больше места на диске, чем мог бы;
- журнал хранит записи об удаленных значениях: он будет неограниченно расти.

Все эти недостатки являются препятствием для использования службы в производстве. Я призываю вас найти время, чтобы рассмотреть эти недостатки и, может быть, даже реализовать их решения.

Сохранение состояния во внешней базе данных

Базы данных и данные лежат в основе многих, если не большинства, бизнес- и веб-приложений, поэтому вполне логично, что в состав стандартных библиотек Go входит и интерфейс (https://oreil.ly/NosgK) для работы с базами данных SQL (или SQL-подобными).

Но имеет ли смысл использовать базу данных SQL для поддержки нашего хранилища пар ключ/значение? В конце концов, разве не было бы странно, если бы наше хранилище данных зависело от другого хранилища данных?

[1] Пожалуйста.

Да, верно. И все же перенос данных службы в другую службу, спроектированную специально для этой цели, – базу данных – является распространенным приемом, позволяющим разделять состояние между экземплярами службы и обеспечивать целостность данных. Кроме того, цель этого раздела состоит в том, чтобы показать, как взаимодействовать с базой данных, а не как разработать идеальное приложение.

В этом разделе мы реализуем журнал транзакций, основанный на внешней базе данных и удовлетворяющий интерфейсу `TransactionLogger`, как мы сделали это в разделе «Сохранение состояния в журнале транзакций». Это, безусловно, будет работоспособное решение и даже имеющее некоторые преимущества, но оно не лишено и недостатков.

Плюсы

Перенос состояния приложения вовне
Распределение состояния доставляет меньше беспокойства и ближе к «истинно облачному».

Упрощается масштабирование
Отсутствие необходимости обмениваться данными между экземплярами *упрощает* масштабирование (но не делает его *совсем простым*).

Минусы

Вводит узкое место
Что, если вам понадобится значительно увеличить масштаб? Что, если всем экземплярам сразу понадобится прочитать состояние из базы данных?

Вводит нижестоящую зависимость
Создает зависимость от другого ресурса, который может выйти из строя.

Требуется инициализация
Что делать, если в базе данных не окажется таблицы `Transactions`?

Увеличивает сложность
Появляется еще один ресурс, который нужно настраивать и которым нужно управлять.

Работа с базами данных в Go

Базы данных, особенно базы данных SQL, используются повсюду. Можно, конечно, попытаться обойтись без них, но если вы создаете приложения с каким-либо компонентом данных, то рано или поздно вам все равно придется взаимодействовать с базами данных.

К счастью для нас, создатели стандартной библиотеки Go добавили в нее пакет `database/sql` (https://oreil.ly/YKPZ6), предлагающий идиоматический и простой интерфейс к базам данных SQL. В этом разделе я покажу, как использовать этот пакет, и расскажу о некоторых подводных камнях.

В число наиболее часто используемых членов пакета `database/sql` входит `sql.DB`: базовая абстракция базы данных на языке Go и точка входа для создания функций и транзакций, выполняющих запросы и получающих результа-

ты. Как следует из ее имени, эта абстракция не связана с какой-то конкретной базой данных или схемой, но делает очень многое, включая согласование соединения с базой данных и управление пулом подключений.

Далее мы коснемся вопросов создания своего экземпляра `sql.DB`, но сначала поговорим о драйверах баз данных.

Импортирование драйвера базы данных

Тип `sql.DB` предлагает обобщенный интерфейс для взаимодействий с базами данных SQL, но реализация специфики конкретных типов баз данных зависит от драйверов. На момент написания этих строк в репозитории Go имелось 45 драйверов (https://oreil.ly/QDQIe).

В следующем разделе мы будем работать с базой данных Postgres, поэтому используем стороннюю реализацию драйвера Postgres `lib/pq` (https://oreil.ly/hYW8r).

Чтобы загрузить драйвер базы данных, выполните анонимное импортирование пакета драйвера с псевдонимом `_`. Эта операция запустит процедуру инициализации в пакете, а также информирует компилятор о том, что вы не собираетесь напрямую использовать его:

```
import (
    "database/sql"
    _ "github.com/lib/pq" // Анонимный импорт пакета драйвера
)
```

Теперь можно создать экземпляр `sql.DB` и получить доступ к базе данных.

Реализация PostgresTransactionLogger

Выше мы определили интерфейс `TransactionLogger`, представляющий стандартное определение журнала транзакций. Как вы наверняка помните, в нем определены методы для запуска регистратора, а также для чтения и записи событий в журнал, как подробно описано здесь:

```
type TransactionLogger interface {
    WriteDelete(key string)
    WritePut(key, value string)
    Err() <-chan error

    ReadEvents() (<-chan Event, <-chan error)

    Run()
}
```

Наша текущая цель – создать реализацию `TransactionLogger` для базы данных. К счастью, большая часть нашей работы уже сделана за нас. Оглядываясь назад на раздел «Реализация FileTransactionLogger», мы можем создать `PostgresTransactionLogger`, используя очень похожую логику.

Начав с методов `WritePut`, `WriteDelete` и `Err`, приходим к следующему определению:

```
type PostgresTransactionLogger struct {
    events  chan<- Event  // Канал только для записи; для передачи событий
    errors  <-chan error  // Канал только для чтения; для приема ошибок
    db      *sql.DB       // Интерфейс доступа к базе данных
}

func (l *PostgresTransactionLogger) WritePut(key, value string) {
    l.events <- Event{EventType: EventPut, Key: key, Value: value}
}

func (l *PostgresTransactionLogger) WriteDelete(key string) {
    l.events <- Event{EventType: EventDelete, Key: key}
}

func (l *PostgresTransactionLogger) Err() <-chan error {
    return l.errors
}
```

Сравнив этот тип с `FileTransactionLogger`, можно заметить, что они почти идентичны. В действительности мы лишь:

- переименовали тип в `PostgresTransactionLogger` (это очевидно);
- заменили `*os.File` на `*sql.DB`;
- удалили `lastSequence`; мы можем возложить заботу об упорядочении на базу данных.

Создание экземпляра PostgresTransactionLogger

Это все хорошо, но я пока ничего не сказал о том, как создать экземпляр `sql.DB`. Я знаю, что вы чувствуете. Меня тоже ожидание просто убивает.

Действуя по аналогии с функцией `NewFileTransactionLogger`, мы создадим функцию, конструирующую экземпляр `PostgresTransactionLogger`, которую назовем (как нетрудно догадаться) `NewPostgresTransactionLogger`. Однако в отличие от `NewFileTransactionLogger` она будет не открывать файл, а устанавливать соединение с базой данных и возвращать ошибку в случае сбоя.

В этой стройной схеме есть одна маленькая проблема: настройка подключения к Postgres требует множества параметров. Как минимум мы должны знать имя хоста, на котором находится база данных, имя базы данных, а также имя пользователя и пароль. Одно из решений – создать функцию, как показано ниже, которая просто принимает кучу строковых параметров:

```
func NewPostgresTransactionLogger(host, dbName, user, password string)
    (TransactionLogger, error) { ... }
```

Однако это не лучшее решение. Представьте, например, что вам понадобилось добавить дополнительный параметр. Вы поместите его в конец списка параметров и тем самым сломаете код, который уже использует эту функцию. Что еще хуже, порядок следования параметров неочевиден, и чтобы выяснить его, придется заглянуть в документацию.

Нам нужен другой, более удобный способ. И такой способ есть! Вместо этого потенциального шоу ужасов можно создать небольшую вспомогательную структуру:

```
type PostgresDBParams struct {
    dbName   string
    host     string
    user     string
    password string
}
```

В отличие от подхода «большой мешок строк», эта структура невелика, удобочитаема и легко расширяется. Для работы с ней можно создать переменную `PostgresDBParams` и передать ее своей функции. Вот как это выглядит:

```
logger, err = NewPostgresTransactionLogger(PostgresDBParams{
    host:     "localhost",
    dbName:   "kvs",
    user:     "test",
    password: "hunter2"
})
```

Вот как выглядит новая функция, конструирующая экземпляр `PostgresTransactionLogger`:

```
func NewPostgresTransactionLogger(config PostgresDBParams)
    (TransactionLogger, error) {

    connStr := fmt.Sprintf("host=%s dbname=%s user=%s password=%s",
        config.host, config.dbName, config.user, config.password)

    db, err := sql.Open("postgres", connStr)
    if err != nil {
        return nil, fmt.Errorf("failed to open db: %w", err)
    }

    err = db.Ping()     // Проверка соединения с базой данных
    if err != nil {
        return nil, fmt.Errorf("failed to open db connection: %w", err)
    }

    logger := &PostgresTransactionLogger{db: db}

    exists, err := logger.verifyTableExists()
    if err != nil {
        return nil, fmt.Errorf("failed to verify table exists: %w", err)
    }
    if !exists {
        if err = logger.createTable(); err != nil {
            return nil, fmt.Errorf("failed to create table: %w", err)
        }
    }

    return logger, nil
}
```

Она делает довольно много, но принципиально не сильно отличается от `NewFileTransactionLogger`.

Во-первых, она использует `sql.Open`, чтобы получить экземпляр `*sql.DB`. Обратите внимание, что строка подключения, которая передается в вызов `sql.Open`, содержит несколько параметров; пакет `lib/pq` поддерживает еще несколько параметров, кроме этих. Полный список вы найдете в документации к пакету (https://oreil.ly/uIgyN).

Многие драйверы, включая `lib/pq`, на самом деле создают соединение с базой данных не сразу, поэтому функция использует `db.Ping`, чтобы заставить драйвер установить и проверить соединение.

В заключение создается экземпляр `PostgresTransactionLogger` и используется для проверки существования таблицы `transactions` и ее создания при необходимости. Без этого шага `PostgresTransactionLogger` будет считать, что таблица уже существует, и завершится ошибкой, если это не так.

Возможно, вы заметили, что в примере отсутствует реализация используемых здесь методов `verifyTableExists` и `createTable`. Это сделано намеренно. Я предлагаю вам окунуться в документацию к `database/sql` (https://oreil.ly/xuFlE) и подумать, как их можно реализовать. Но если вы не хотите заняться этим, то можете найти реализацию в репозитории GitHub (https://oreil.ly/1MEIr) для данной книги.

Теперь у нас есть функция, которая устанавливает соединение с базой данных и возвращает вновь созданный экземпляр `TransactionLogger`. Но нам опять нужен механизм запуска регистратора в работу. Для этого мы реализуем метод `Run`, который создаст каналы `events` и `errors` и запустит сопрограмму, принимающую события.

Выполнение SQL-запроса INSERT с помощью db.Exec

Для `FileTransactionLogger` мы реализовали метод `Run`, который инициализировал каналы и запускал сопрограмму, осуществляющую запись событий в журнал транзакций.

`PostgresTransactionLogger` действует похожим образом, только вместо добавления строк в конец файла журнала он использует `db.Exec`, чтобы выполнить SQL-запрос `INSERT`:

```
func (l *PostgresTransactionLogger) Run() {
    events := make(chan Event, 16) // Создать канал событий
    l.events = events

    errors := make(chan error, 1)  // Создать канал ошибок
    l.errors = errors

    go func() {                    // Запрос INSERT
        query := `INSERT INTO transactions
                  (event_type, key, value)
                  VALUES ($1, $2, $3)`

        for e := range events {    // Извлечь следующее событие Event

            _, err := l.db.Exec(   // Выполнить запрос INSERT
                query,
                e.EventType, e.Key, e.Value)
```

```
            if err != nil {
                errors <- err
            }
        }
    }()
}
```

Эта реализация метода `Run` делает почти то же самое, что и его эквивалент в `FileTransactionLogger`: создает буферизованные каналы `events` и `errors` и запускает сопрограмму, которая извлекает события из канала `events` и записывает их в журнал транзакций.

В отличие от сопрограммы в `FileTransactionLogger`, которая добавляет события в файл, эта сопрограмма с помощью `db.Exec` выполняет SQL-запрос, добавляющий запись в таблицу `transactions`. Нумерованные аргументы (`$1`, `$2`, `$3`) в тексте запроса представляют параметры, которые должны передаваться функции `db.Exec`.

Использование db.Query для воспроизведения транзакций из журнала

В разделе «Использование bufio.Scanner для воспроизведения транзакций из журнала» выше мы использовали `bufio.Scanner` для чтения из журнала транзакций, записанных ранее.

Реализация на основе Postgres не получится *такой же* простой, но суть та же: вы переходите в начало источника данных и последовательно читаете записи, пока не достигнете конца:

```
func (l *PostgresTransactionLogger) ReadEvents() (<-chan Event, <-chan error) {
    outEvent := make(chan Event)     // Небуферизованный канал событий
    outError := make(chan error, 1)  // Буферизованный канал ошибок

    go func() {
        defer close(outEvent)        // Закрыть каналы
        defer close(outError)        // по завершении сопрограммы

        query := `SELECT sequence, event_type, key, value FROM transactions
                  ORDER BY sequence`

        rows, err := db.Query(query) // Выполнить запрос; получить набор результатов
        if err != nil {
            outError <- fmt.Errorf("sql query error: %w", err)
            return
        }

        defer rows.Close()           // Это важно!

        e := Event{}                 // Создать пустой экземпляр Event

        for rows.Next() {            // Цикл по записям

            err = rows.Scan(              // Прочитать значения
                &e.Sequence, &e.EventType, // из записи в Event.
```

```
                &e.Key, &e.Value)

            if err != nil {
                outError <- fmt.Errorf("error reading row: %w", err)
                return
            }

            outEvent <- e            // Послать e в канал
        }

        err = rows.Err()
        if err != nil {
            outError <- fmt.Errorf("transaction log read failure: %w", err)
        }
    }()

    return outEvent, outError
}
```

Все самое интересное (или, по крайней мере, новое) происходит в сопрограмме. Давайте разберем ее:

- `query` – строка, содержащая SQL-запрос. В запросе перечислены четыре столбца: `sequence`, `event_type`, `key` и `value`;
- `db.Query` отправляет запрос в базу данных и возвращает значения типа `*sql.Rows` и `error`;
- вызов `rows.Close` откладывается до завершения сопрограммы. Несоблюдение этого правила может привести к утечке соединений;
- `rows.Next` помогает перебирать записи; этот метод возвращает `false`, если достигнут конец набора результатов или произошла ошибка;
- `rows.Scan` копирует столбцы из текущей записи в переменные, ссылки на которые указаны в вызове функции;
- событие `e` отправляется в выходной канал;
- в `Err` возвращается ошибка, если таковая имеется, из-за которой `rows.Next` мог вернуть `false`.

Инициализация PostgresTransactionLogger в веб-службе

Мы почти закончили `PostgresTransactionLogger`. Осталось только интегрировать его в веб-службу.

К счастью, поскольку у нас уже есть `FileTransactionLogger`, мы должны изменить только одну строку:

```
logger, err = NewFileTransactionLogger("transaction.log")
```

которая теперь будет выглядеть так...

```
logger, err = NewPostgresTransactionLogger("localhost")
```

Вот и все! Действительно все.

Поскольку эта реализация в полной мере соответствует интерфейсу `TransactionLogger`, все остальное остается без изменений. Вы можете взаимодействовать с `PostgresTransactionLogger`, используя те же методы, что и раньше.

Будущие улучшения

Так же как `FileTransactionLogger`, тип `PostgresTransactionLogger` является минимальной жизнеспособной реализацией регистратора транзакций и имеет множество недостатков, которые можно исправить. Вот некоторые из них:

- предполагается, что база данных и таблица существуют, и код потерпит неудачу, если это не так;
- строка подключения жестко запрограммирована, даже пароль;
- по-прежнему отсутствует метод `Close`, который закрывал бы соединение;
- служба может закрыться, когда события все еще находятся в буфере записи: в этом случае события могут быть потеряны;
- журнал хранит записи об удаленных значениях: он будет неограниченно расти.

Все эти недостатки являются препятствием для использования службы в производстве. Я призываю вас найти время, чтобы рассмотреть эти недостатки и, может быть, даже реализовать их решения.

Итерация 3: реализация безопасности транспортного уровня

Безопасность. Нравится вам это или нет, но безопасность является критически важной частью *любого* приложения, будь оно облачным или нет. К сожалению, о безопасности часто начинают думать в последнюю очередь, что может иметь катастрофические последствия.

Для традиционных окружений существует большое разнообразие инструментов и методов обеспечения безопасности, но к облачным приложениям, которые обычно принимают форму нескольких небольших, часто эфемерных микросервисов, это относится в меньшей степени. Эта архитектура предлагает значительные преимущества гибкости и масштабируемости, но она также открывает широкие возможности потенциальным злоумышленникам: каждое взаимодействие между службами передается по сети, что позволяет подслушивать их и подделывать.

Теме безопасности можно посвятить целую книгу[1], поэтому мы сосредоточимся только на одном распространенном методе: шифровании. Шифрование данных «в процессе передачи» часто используется для защиты от подслу-

[1] В идеале написанной кем-то, кто знает о безопасности больше, чем я.

шивания и подделки, и любой язык, заслуживающий внимания, в том числе и Go, делает реализацию такого шифрования относительно простой задачей.

Transport Layer Security

Transport Layer Security (TLS) – это криптографический протокол, предназначенный для обеспечения безопасности связи в компьютерной сети. Он имеет самое широкое применение и используется для защиты подавляющего числа взаимодействий в интернете. Вы почти наверняка знакомы с ним (и, возможно, используете его прямо сейчас) в форме HTTPS, также известного как «HTTP через TLS», который использует TLS для шифрования данных, передаваемых по протоколу HTTP.

TLS шифрует сообщения с использованием *криптографии с открытым ключом*, в которой обе стороны обладают своей парой ключей, состоящей из *открытого ключа*, который можно передавать другим без ограничений, и *закрытого ключа*, известного только его владельцу, как показано на рис. 5.2. Любой желающий может использовать открытый ключ для шифрования сообщений, но расшифровать такие сообщения можно только с помощью соответствующего закрытого ключа. Используя этот протокол, две стороны, вступающие во взаимодействие и желающие сохранить пересылаемые данные в секрете, могут обменяться своими открытыми ключами и затем использовать их для защиты всех последующих взаимодействий, чтобы прочитать данные мог только конкретный адресат, владеющий соответствующим закрытым ключом[1].

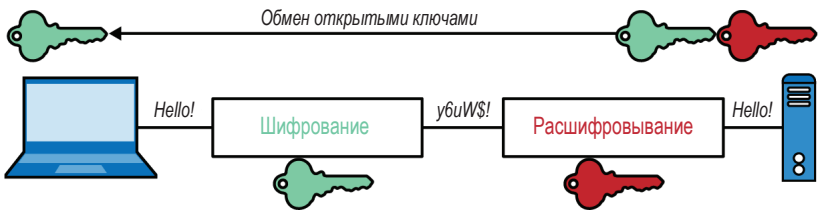


Рис. 5.2 ❖ Обмен открытыми ключами

Сертификаты, центры сертификации и доверие

Если бы у TLS был девиз, то он звучал бы так: «Доверяй, но проверяй». На самом деле не доверяйте безоглядно. Проверяйте все.

Службе недостаточно просто предоставить открытый ключ[2]. Каждый открытый ключ связан еще с *цифровым сертификатом*, электронным документом, подтверждающим права собственности на ключ. Сертификат показывает, что владелец открытого ключа фактически является тем, за кого

[1] Это сильно упрощенное описание, но его вполне достаточно для общего понимания. Я предлагаю вам узнать больше об этом протоколе и поправить меня.

[2] Потому что вам неизвестно, откуда взялся этот ключ.

себя выдает, и описывает, как этот ключ может использоваться. Благодаря этому получатель может проверить сертификат и решить, примет ли он его как действительный.

Во-первых, сертификат должен быть подписан цифровой подписью и заверен *центром сертификации* – доверенной стороной, выдающей цифровые сертификаты.

Во-вторых, тема сертификата должна совпадать с доменным именем службы, к которой клиент пытается подключиться. Помимо прочего, это помогает гарантировать, что полученные вами сертификаты действительны и не были подменены злоумышленником.

Только после этого ваш диалог продолжится.

Веб-браузеры и другие инструменты обычно позволяют продолжить диалог, даже если сертификат невозможно подтвердить. Это может иметь смысл, например, если вы используете самоподписанные сертификаты для разработки. Но в целом прислушивайтесь к предупреждениям.

Закрытый ключ и файлы сертификатов

Протокол TLS (и его предшественник, SSL) существует достаточно давно[1], поэтому вы могли бы подумать, что повсюду используется единый формат представления ключей, но вы ошибаетесь. Поиск в интернете по запросу «формат файла ключа» вернет длинный список расширений таких файлов. Вот лишь некоторые из них: *.csr*, *.key*, *.pkcs12*, *.der* и *.pem*.

Из них наиболее широко, пожалуй, используется формат *.pem*. Кроме того, поддержка этого формата включена в пакет `net/http`, поэтому мы будем использовать именно его.

Формат Privacy Enhanced Mail (PEM)

Privacy Enhanced Mail (PEM) – это универсальный формат файлов сертификатов, обычно с расширением *.pem*, но также может использоваться для файлов *.cer* или *.crt* (для сертификатов) и *.key* (для открытых или закрытых ключей). Формат PEM удобен тем, что сертификаты и ключи в этом формате дополнительно кодируются в представление base64 и, соответственно, могут просматриваться в текстовом редакторе и передаваться в теле сообщения электронной почты[2].

Часто файлы *.pem* образуют пары, представляя полную пару ключей:

cert.pem
: Сертификат сервера (включая открытый ключ, подписанный центром сертификации).

---

[1] SSL 2.0 был выпущен в 1995 году, а TLS 1.0 – в 1999 году. Интересно отметить, что SSL 1.0 имел довольно серьезные недостатки безопасности и так и не был опубликован.

[2] Только открытые ключи! Помните об этом.

key.pem
Закрытый ключ, не подлежит разглашению.

Далее будем предполагать, что ваши ключи соответствуют этой конфигурации. Если у вас еще нет ключей и вам нужно сгенерировать их для целей разработки, то необходимые инструкции вы без труда найдете в интернете. Если у вас уже есть файл ключа в каком-то другом формате, то его можно преобразовать. Обсуждение этого процесса выходит за рамки книги, тем не менее интернет – это волшебное место, и там вы найдете множество учебных пособий по преобразованию ключей из одного формата в другой.

Защита веб-службы с помощью HTTPS

Итак, теперь, когда мы узнали, что к безопасности следует относиться серьезно и что обмен данными через TLS является минимальным первым шагом на пути к обеспечению безопасности наших взаимодействий, пришла пора узнать, как это сделать.

Одно из возможных решений – установка обратного прокси-сервера перед службой, который будет обрабатывать запросы HTTPS и пересылать их в нашу службу хранилища пар ключ/значение как HTTP-запросы, но если служба и прокси окажутся на разных хостах, то получится так, что мы все равно будем отправлять незашифрованные сообщения по сети. Кроме того, дополнительная служба добавит сложности в архитектуру, чего, конечно, хотелось бы избежать. Может быть, мы сможем сделать так, чтобы наша служба хранилища пар ключ/значение сама обслуживала бы протокол HTTPS?

Да, сможем. Как рассказывалось в разделе «Создание HTTP-сервера с использованием net/http» выше, в пакете `net/http` имеется функция `ListenAndServe`, которая в самом простом случае используется примерно так:

```
func main() {
    http.HandleFunc("/", helloGoHandler) // Зарегистрировать обработчик корневого пути

    http.ListenAndServe(":8080", nil)    // Запустить HTTP-сервер
}
```

В этом примере мы вызываем `HandleFunc`, чтобы зарегистрировать функцию-обработчик корневого пути, а затем `ListenAndServe`, чтобы начать прием входящих запросов и их обработку. Для простоты мы игнорируем любые ошибки, возвращаемые `ListenAndServe`.

Здесь не так много движущихся частей, что само по себе неплохо. Следуя этой философии, разработчики `net/http` любезно предоставили знакомый нам вариант функции `ListenAndServe` с поддержкой TLS:

```
func ListenAndServeTLS(addr, certFile, keyFile string, handler Handler) error
```

Как видите, ListenAndServeTLS выглядит и действует почти так же, как ListenAndServe, за исключением двух дополнительных параметров: certFile и keyFile. Если у вас есть файлы сертификата и закрытого ключа в формате PEM, то для обслуживания HTTPS-соединений достаточно передать имена этих файлов:

```
http.ListenAndServeTLS (":8080", "cert.pem", "key.pem", nil)
```

Очень удобно, но работает ли? Давайте запустим нашу службу (используя самоподписанные сертификаты) и узнаем.

Смахнем пыль с нашего старого знакомого curl и попробуем добавить пару ключ/значение в хранилище. Обратите внимание на схему https в URL-адресе:

```
$ curl -X PUT -d 'Hello, key-value store!' -v https://localhost:8080/v1/key-a
* SSL certificate problem: self signed certificate
curl: (60) SSL certificate problem: self signed certificate
```

Что ж, фокус не удался. Как уже отмечалось в разделе «Сертификаты, центры сертификации и доверие», протокол TLS ожидает, что все сертификаты будут подписаны центром сертификации. Он не любит самоподписанные сертификаты.

К счастью, в curl можно отключить эту проверку безопасности, добавив флаг --insecure:

```
$ curl -X PUT -d 'Hello, key-value store!' --insecure -v \
    https://localhost:8080/v1/key-a
* SSL certificate verify result: self signed certificate (18), continuing anyway.
> PUT /v1/key-a HTTP/2
< HTTP/2 201
```

Да, мы получили строгое предупреждение, но этот прием сработал!

В заключение о транспортном уровне

Мы охватили довольно много информации всего на нескольких страницах. Тема безопасности обширна, и я не претендую на полноту ее освещения, но мне удалось хотя бы представить протокол TLS и как он может служить одним из относительно недорогих и очень полезных компонентов более широкой стратегии безопасности.

Также у меня получилось продемонстрировать порядок внедрения TLS в веб-службу на Go с использованием net/http и показать, как – при наличии действительных сертификатов – обеспечить безопасность взаимодействий со службой.

Контейнеризация хранилища пар ключ/значение

Контейнер – это облегченная абстракция виртуализации на уровне операционной системы[1], которая обеспечивает процессам определенную степень изоляции не только от хоста, но и от других контейнеров. Идея контейнеров существует, по крайней мере, с 2000 года, но только появление фреймворка Docker в 2013 году сделало контейнеры доступными для массового применения и обеспечило рост популярности контейнеризации.

Важно отметить, что контейнеры – это не виртуальные машины[2]: они не используют гипервизоры и все вместе используют одно и то же ядро хоста, а не запускают свою гостевую операционную систему. Их изоляция обеспечивается особым применением некоторых возможностей ядра Linux, включая `chroot`, cgroups и пространств имен ядра. Фактически можно даже утверждать, что контейнеры – это не более чем удобная абстракция и на самом деле нет такой вещи, как контейнер.

Несмотря на то что контейнеры не являются виртуальными машинами[3], они действительно дают некоторые преимущества, характерные для виртуальных машин. Наиболее очевидным является возможность упаковки приложения с его зависимостями и большей части окружения в один распространяемый артефакт – образ контейнера, – который может выполняться на любом подходящем хосте.

Но на этом преимущества не заканчиваются. Вот еще несколько.

Гибкость
: В отличие от виртуальных машин, обремененных своей операционной системой и требующих выделять им колоссальные объемы памяти, контейнеры могут похвастаться небольшими размерами образов, порядка нескольких мегабайтов, и коротким временем запуска, которое исчисляется миллисекундами. Это особенно верно для приложений на Go, двоичные файлы которых имеют мало зависимостей, если они вообще есть.

Изоляция
: Об этом уже говорилось выше, но стоит повторить. Контейнеры виртуализируют процессоры, память, устройства хранения и сетевые ресурсы на уровне операционной системы, предоставляя разработчикам образ операционной системы, логически изолированный от других приложений.

Стандартизация и производительность
: Контейнеры позволяют упаковать приложение вместе с зависимостями, такими как определенные версии среды выполнения и библиотек, в один двоичный файл, что обеспечивает воспроизводимость развертываний, их

---

[1] Контейнеры – это не виртуальные машины. Они виртуализируют операционную систему, а не оборудование.

[2] Намеренное повторение, так как это очень важный момент.

[3] Ну да. Я повторился. Снова.

предсказуемость и возможность версионирования.

Оркестровка (управление)
Сложные системы оркестровки контейнеров, такие как Kubernetes, предоставляют огромное количество преимуществ. Помещая свои приложения в контейнеры, вы делаете первый шаг к тому, чтобы воспользоваться их преимуществами.

Итак, мы имеем четыре (очень) мотивирующих аргумента в пользу контейнеров[1]. Иначе говоря, контейнеризация – это суперполезно.

Для создания образов контейнеров в этой книге мы будем использовать Docker. Конечно, существуют и другие инструменты сборки, но Docker – наиболее распространенный инструмент контейнеризации, используемый в настоящее время, а синтаксис его файла сборки, так называемого *Dockerfile*, позволяет использовать знакомые команды сценариев командной оболочки и служебные программы.

Тем не менее я должен отметить, что эта книга не о Docker и не о контейнеризации, поэтому наше обсуждение будет ограничено простыми основами использования Docker с Go. Если вам интересно узнать больше, я предлагаю прочитать книгу Шона П. Кейна (Sean P. Kane) и Карла Маттиаса (Karl Matthias) *Docker: Up & Running: Shipping Reliable Containers in Production* (O'Reilly; https://oreil.ly/rqGoI).

Основы Docker

Прежде чем продолжить, я хочу провести различие между образами контейнеров и самими контейнерами. *Образ контейнера* – это, по сути, выполняемый двоичный файл, который содержит среду выполнения для вашего приложения и его зависимости. Когда образ запускается, то получившийся процесс становится *контейнером*. Образ можно запустить много раз и создать несколько идентичных контейнеров.

На следующих нескольких страницах мы создадим простой файл Dockerfile, а также соберем и запустим образ. Если вы еще этого не сделали, найдите время и установите Docker Community Edition (CE; https://oreil.ly/yYwKL).

Dockerfile

Файл Dockerfile – это файл сборки, описывающий шаги, которые необходимо выполнить, чтобы собрать образ. Вот хоть и минимальный, но полный пример:

```
# Родительский образ. На этапе сборки этот образ будет извлечен,
# и последующие инструкции будут манипулировать им.
FROM ubuntu:20.04
```

[1] В первоначальном варианте их было больше, но эта глава и так получилась довольно длинной.

```
# Обновить кеш apt и установить nginx без запроса подтверждения.
RUN apt-get update && apt-get install --yes nginx

# Сообщить Docker, что контейнеры, создаваемые на основе этого образа,
# будут использовать порт 80.
EXPOSE 80

# Запустить Nginx на переднем плане. Это важно: в отсутствие
# процесса переднего плана контейнер автоматически остановится.
CMD ["nginx", "-g", "daemon off;"]
```

Как видите, этот файл Dockerfile включает четыре разные команды:

`FROM`
: Определяет *базовый образ*, который будет расширяться и дополняться этой сборкой. Как правило, это обычный дистрибутив Linux, такой как `ubuntu` или `alpine`. Во время сборки этот образ извлекается из репозитория и запускается, и к нему применяются последующие команды.

`RUN`
: Выполняет любые команды поверх текущего образа. Результат используется на следующем шаге в Dockerfile.

`EXPOSE`
: Сообщает фреймворку Docker, какие порты будет использовать контейнер. См. врезку «В чем разница между открытием и публикацией портов?» ниже для получения дополнительной информации об использовании портов.

`CMD`
: Команда, выполняемая при запуске контейнера. В Dockerfile может быть только одна инструкция `CMD`.

Это лишь четыре из множества доступных инструкций. Полный список см. в официальном справочнике Dockerfile (https://oreil.ly/8LGdP).

Как вы наверняка догадались, предыдущий пример начинается с существующего образа дистрибутива Linux (Ubuntu 20.04) и устанавливает Nginx, который запускается при запуске контейнера.

В соответствии с соглашениями файлу Dockerfile дается имя *Dockerfile*. Создайте новый файл с именем *Dockerfile* и вставьте в него предыдущий пример.

Сборка образа контейнера

Теперь, имея простой Dockerfile, можно собрать свой образ контейнера! Убедитесь, что находитесь в каталоге с файлом *Dockerfile*, и выполните команду

```
$ docker build --tag my-nginx .
```

Она запустит процесс сборки. Если не возникнет никаких ошибок (а почему они должны возникнуть?), то вы увидите, как Docker загрузит родительский образ и выполнит команды `apt`. При первом запуске это, вероятно, займет пару минут.

В конце вы увидите строку с примерно таким текстом: `Successfully tagged my-nginx:latest`.

После этого можно выполнить команду `docker images`, чтобы убедиться, что образ создан и сохранен. Вы должны увидеть что-то вроде следующего:

```
$ docker images
REPOSITORY      TAG       IMAGE ID         CREATED           SIZE
my-nginx        latest    64ea3e21a388     29 seconds ago    159MB
ubuntu          20.04     f63181f19b2f     3 weeks ago       72.9MB
```

Если все прошло благополучно, то вы увидите как минимум два образа в списке: родительский образ `ubuntu:20.04` и вновь созданный образ `my-nginx:latest`. Следующий шаг: запуск контейнера со службой!

Что означает latest?

Обратите внимание на название образа. Что означает слово `latest` в нем? Это простой вопрос со сложным ответом. Имена образов Docker состоят из двух частей: названия репозитория и тега.

Часть с названием репозитория может включать доменное имя хоста, где хранится образ (или где будет храниться). Например, имя репозитория для образа, размещенного на FooCorp, может иметь примерно такой вид: `docker.foo.com/ubuntu`. Если URL-адрес репозитория не очевиден, то образ либо на 100 % локальный (как образ, который мы только что создали), либо находится в репозитории Docker Hub (https://hub.docker.com/).

Тег служит для обозначения конкретной версии образа и часто имеет форму номера версии. Тег `latest` – это тег по умолчанию, который добавляется многими операциями команды `docker`, если тег не указан явно.

Однако использование тега `latest` в производственном окружении считается плохой практикой, потому что содержимое образа с этим тегом может измениться – иногда значительно – с печальными последствиями.

Запуск образа контейнера

После создания образа его можно запустить. Делается это с помощью команды `docker run`:

```
$ docker run --detach --publish 8080:80 --name nginx my-nginx
61bb4d01017236f6261ede5749b421e4f65d43cb67e8e7aa8439dc0f06afe0f3
```

Она загрузит указанный образ контейнера и запустит его. Флаг `--detach` требует запустить контейнер в фоновом режиме. Параметр `--publish 8080:80` требует от Docker опубликовать порт 8080 хоста (открыть для доступа из внешнего мира) и связать его с портом 80 контейнера, в результате этого любые подключения к `localhost:8080` будут перенаправляться на порт 80 контейнера. Наконец, флаг `--name nginx` задает имя контейнера; без этого контейнеру будет присвоено случайно сгенерированное имя.

Обратите внимание, что эта команда выводит загадочную строку, содержащую 65 очень загадочных шестнадцатеричных символов. Это *идентифи-*

катор контейнера, который можно использовать для ссылки на контейнер вместо его имени.

В чем разница между открытием и публикацией портов[1]?

Разница между «открытием» (EXPOSE) и «публикацией» (--publish) портов контейнера может показаться трудноуловимой, но на самом деле между ними есть важное различие:

- *открытие* (EXPOSE) портов – это способ ясно задокументировать для пользователей и для Docker, какие порты использует контейнер. Команда EXPOSE не отображает и не открывает какие-либо порты хоста. Порты можно открыть с помощью ключевого слова EXPOSE в файле Dockerfile или флага --expose в команде docker run;
- *публикация* (--publish) портов сообщает фреймворку Docker, какие порты контейнера должны быть открыты для доступа извне. Опубликовать порты можно с помощью флага --publish или --publish-all в команде docker run, которые создадут правила брандмауэра, отображающие порты контейнера в порты хоста.

Проверка запущенного образа контейнера

Чтобы убедиться, что контейнер запущен и работает, можете воспользоваться командой docker ps и вывести список всех запущенных контейнеров. Например:

```
$ docker ps
CONTAINER ID    IMAGE       STATUS          PORTS                   NAMES
4cce9201f484    my-nginx    Up 4 minutes    0.0.0.0:8080->80/tcp    nginx
```

Предыдущий вывод был немного сокращен (в действительности в нем присутствовали также столбцы COMMAND и CREATED). У вас вывод должен включать семь столбцов:

CONTAINER ID
: Первые 12 символов идентификатора контейнера, который сообщает команда docker run.

IMAGE
: Имя (и тег, если задан) образа контейнера. Тег latest не выводится.

---

[1] *От переводчика:* считаю необходимым дополнить объяснение автора. Всего возможны три варианта: 1) не указать ни **EXPOSE**, ни **--publish**; 2) указать только **EXPOSE**; 3) указать и **EXPOSE**, и **--publish**. В первом случае служба будет доступна только изнутри контейнера. Во втором случае служба будет доступна из других контейнеров, выполняющихся на том же хосте, но не будет доступна другим хостам. В третьем случае служба будет доступна откуда угодно. Если не указать **EXPOSE**, но указать флаг **--publish** в команде **docker run**, то порт будет доступен не только извне, как предполагает флаг **--publish**, но также и другим контейнерам на хосте, как предполагает команда **EXPOSE**, то есть флаг **--publish** неявно выполняет команду **EXPOSE**.

COMMAND (*в примере выше не показан*)
: Команда, запущенная внутри контейнера. Если она не была переопределена в команде запуска `docker run`, то это будет та же команда, что указана в инструкции `CMD` в файле Dockerfile. В нашем случае это будет команда `nginx -g 'daemon off;'`.

CREATED (*в примере выше не показан*)
: Как давно создан контейнер.

STATUS
: Текущее состояние контейнера (`up`, `exited`, `restarting` и т. д.) и как давно он находится в этом состоянии. Если состояние изменилось, то время будет отличаться от времени в столбце `CREATED`.

PORTS
: Список всех открытых и опубликованных портов (см. врезку «В чем разница между открытием и публикацией портов?» выше). В нашем случае мы опубликовали порт `8080` хоста и отобразили его в порт `80` контейнера, поэтому все запросы, присылаемые на порт хоста 8080, будут перенаправляться в порт 80 контейнера.

NAMES
: Имя контейнера. Если имя не было задано явно, то Docker выберет случайное имя. Два контейнера с одинаковыми именами, независимо от состояния, не могут существовать на одном хосте одновременно. Чтобы повторно использовать имя, сначала нужно остановить и удалить контейнер с этим именем.

Отправка запроса в опубликованный порт контейнера

Итак, мы благополучно добрались до того момента, когда команда `docker ps` показала, что наш контейнер `nginx` работает и имеет опубликованный порт 8080, связанный с портом 80 контейнера. Теперь мы готовы отправить запрос нашему контейнеру. Но в какой порт мы должны послать запрос?

Контейнер Nginx прослушивает порт 80. Можем ли мы послать запрос в него? Вообще-то, нет. Этот порт не будет доступен извне, потому что он не был опубликован ни в одном сетевом интерфейсе в команде `docker run`. Любая попытка подключиться к неопубликованному порту контейнера обречена на провал:

```
$ curl localhost:80
curl: (7) Failed to connect to localhost port 80: Connection refused
```

Мы не опубликовали порт 80, *но опубликовали* порт 8080 и перенаправили его в порт 80 контейнера. Его можно проверить с помощью нашего старого знакомого `curl` или открыв адрес `localhost:8080` в браузере. Если все работает правильно, то откроется страница приветствия Nginx, показанная на рис. 5.3.

Рис. 5.3 ❖ Страница приветствия Nginx

Запуск нескольких контейнеров

Одна из уникальных особенностей контейнеризации заключается в следующем: поскольку все контейнеры, выполняющиеся на хосте, изолированы друг от друга, их можно запустить довольно много, даже если они будут использовать совершенно разные технологии и стеки, при условии что каждый будет прослушивать свой отдельный опубликованный порт. Например, если вы решите запустить контейнер `httpd` параллельно с уже запущенным контейнером `my-nginx`, то сможете сделать это.

«Но, – можете сказать вы, – оба этих контейнера открывают (`EXPOSE`) порт 80! Разве они не будут конфликтовать друг с другом?»

К счастью, нет! На самом деле вы можете запустить сколько угодно контейнеров, открывающих один и тот же порт, – даже несколько экземпляров одного и того же образа – при условии что в командах запуска вы не будете пытаться *опубликовать* один и тот же порт на одном и том же сетевом интерфейсе.

Например, если вы решите запустить стандартный образ `httpd`, то сможете запустить его командой `docker run`, если позаботитесь о публикации другого порта (например, 8081):

```
$ docker run --detach --publish 8081:80 --name httpd httpd
```

Если не случится ничего непредвиденного, то на хосте появится новый контейнер, который будет прослушивать порт 8081 хоста. Убедиться в этом можно с помощью `docker ps` и `curl`:

```
$ curl localhost:8081
<html><body><h1>It works!</h1></body></html>
```

Остановка и удаление контейнеров

После успешного запуска контейнера вам в какой-то момент может понадобиться остановить и удалить его, особенно если вы решите повторно запустить новый контейнер с тем же именем.

Остановить работающий контейнер можно с помощью команды `docker stop`, передав ей имя контейнера или первые несколько символов его идентификатора (количество символов не имеет значения, если они однозначно идентифицируют желаемый контейнер). Вот как можно остановить наш контейнер `nginx` с использованием идентификатора:

```
$ docker stop 4cce     # также можно выполнить команду "docker stop nginx"
4cce
```

В случае успеха `docker stop` выведет имя или идентификатор, переданный команде. Убедиться, что контейнер действительно остановлен, можно с помощью команды `docker ps --all`, которая покажет все контейнеры, а не только запущенные:

```
$ docker ps
CONTAINER ID    IMAGE       STATUS                      PORTS   NAMES
4cce9201f484    my-nginx    Exited (0) 3 minutes ago            nginx
```

Если вы также запустили контейнер `httpd`, то он появится в выводе со статусом `Up`. Возможно, вы захотите остановить и его.

Как видите, статус контейнера `nginx` изменился на `Exited`, за которым следует код завершения 0, указывающий, что работа была завершена штатно, и как давно контейнер перешел в это состояние.

После остановки контейнера его можно удалить.

Нельзя удалить работающий контейнер или образ, используемый работающим контейнером.

Сделать это можно командой `docker rm` (или более новой командой `docker container rm`), передав ей либо имя контейнера, либо первые несколько символов идентификатора:

```
$ docker rm 4cce     # также можно выполнить команду "docker rm nginx"
4cce
```

Как и раньше, вывод имени или идентификатора указывает на успех. Если снова выполнить команду `docker ps --all`, то вы не увидите своего контейнера в списке.

Сборка контейнера для службы хранилища пар ключ/значение

Теперь, познакомившись с основами контейнеризации служб, можно приступить к созданию образа контейнера для нашей службы хранилища пар ключ/значение.

К счастью, способность Go компилировать код в статически связанные двоичные файлы делает его особенно подходящим для контейнеризации. В отличие от большинства других языков, требующих встраивать прило-

жения на них в родительский образ со средой выполнения языка, например 486-мегабайтный образ `openjdk:15` для Java или 885-мегабайтный образ `python:3.9` для Python[1], двоичные файлы Go вообще не нуждаются в среде выполнения. Их можно включить в «пустой» образ, то есть создать образ, не имеющий родителя.

Итерация 1: добавление двоичного файла в пустой образ

Чтобы собрать образ, нам понадобится файл Dockerfile. Вот довольно типичный пример файла Dockerfile, определяющий образ контейнера для двоичного файла Go:

```
# Мы используем образ "scratch", не содержащий распределяемых файлов.
# Получившийся образ будет содержать только двоичный файл службы.
FROM scratch

# Скопировать имеющийся двоичный файл с хоста.
COPY kvs .

# Скопировать файлы PEM.
COPY *.pem .

# Сообщить фреймворку Docker, что служба будет использовать порт 8080.
EXPOSE 8080

# Команда, которая должна быть выполнена при запуске контейнера.
CMD ["/kvs"]
```

Этот файл Dockerfile очень похож на предыдущий, только вместо `apt` для установки приложения из репозитория он использует `COPY`, чтобы скопировать скомпилированный двоичный файл. В данном примере предполагается, что двоичный файл называется `kvs`. Чтобы сборка удалась, нам нужно сначала собрать двоичный файл.

Чтобы двоичный файл можно было использовать внутри контейнера, он должен соответствовать нескольким критериям:

- быть скомпилирован для Linux;
- быть статически скомпонован;
- иметь имя `kvs` (потому что файл с этим именем ожидается в Dockerfile).

Все это можно сделать одной командой

```
$ CGO_ENABLED=0 GOOS=linux go build -a -o kvs
```

Давайте разберем, что она делает:

- `CGO_ENABLED=0` сообщает компилятору о необходимости отключить `cgo` и статически скомпоновать все используемые библиотеки. Я не буду вдаваться в подробности, только отмечу, что эта часть команды обеспечивает статическую компоновку, однако я советую заглянуть в документацию с описанием `cgo`;

[1] Честно говоря, эти образы можно сжать до «всего» 240 и 337 Мбайт соответственно.

- `GOOS=linux` сообщает компилятору о необходимости сгенерировать двоичный файл для Linux, выполнив кросс-компиляцию, если необходимо;
- `-a` требует от компилятора пересобрать любые обновившиеся пакеты;
- `-o kvs` определяет имя для выходного двоичного файла.

После выполнения этой команды у вас должен появиться статически скомпонованный двоичный файл для Linux. Проверить соответствие перечисленным выше критериям можно с помощью команды `file`:

```
$ file kvs
kvs: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), statically linked,
not stripped
```


Двоичные файлы для Linux будут работать в контейнере Linux, даже если он запущен под управлением Docker в MacOS или Windows, но они не будут работать в MacOS или Windows без контейнера.

Отлично! Теперь соберем образ контейнера и посмотрим, что получится:

```
$ docker build --tag kvs .
...вывод опущен.

$ docker images
REPOSITORY     TAG      IMAGE ID         CREATED              SIZE
kvs            latest   7b1fb6fa93e3     About a minute ago   6.88MB
node           15       ebcfbb59a4bd     7 days ago           936MB
python         3.9      2a93c239d591     8 days ago           885MB
openjdk        15       7666c92f41b0     2 weeks ago          486MB
```

Меньше 7 Мбайт! Это примерно на два порядка меньше, чем более массивные образы для других языков. Это может оказаться весьма кстати, когда потребуется масштабировать приложение и загружать образ на пару сотен узлов несколько раз в день.

Но работает ли он? Давайте выясним:

```
$ docker run --detach --publish 8080:8080 kvs
4a05617539125f7f28357d3310759c2ef388f456b07ea0763350a78da661afd3

$ curl -X PUT -d 'Hello, key-value store!' -v http://localhost:8080/v1/key-a
> PUT /v1/key-a HTTP/1.1
< HTTP/1.1 201 Created

$ curl http://localhost:8080/v1/key-a
Hello, key-value store!
```

Похоже, что работает!

Итак, теперь мы имеем простой и красивый Dockerfile, создающий образ с предварительно скомпилированным двоичным файлом. К сожалению, это означает, что вы (или ваша система непрерывной интеграции) должны заново собрать двоичный файл перед каждой сборкой образа контейнера. Это не очень обременительное требование, но оно означает, что вам нужно установить Go на ваши серверы сборки. Это тоже не страшно, но мы определенно можем добиться большего.

Итерация 2: многоэтапная сборка

В предыдущем разделе мы создали простой файл Dockerfile, который упаковывает существующий двоичный файл для Linux в «пустой» образ. А можно ли выполнить *все* необходимое для сборки образа – компиляцию исходного кода на Go и прочее – в Docker?

Одно из решений – использование образа `golang` в качестве родительского. В этом случае Dockerfile сможет скомпилировать код на Go и запустить полученный двоичный файл во время развертывания. Такой образ можно собрать даже на хостах, где не установлен компилятор Go, но получившийся образ будет обременен дополнительными 862 Мбайтами (размер образа `golang:1,16`) совершенно ненужного механизма компиляции.

Другое решение – использование двух файлов Dockerfile: один для сборки двоичного файла и другой для сборки контейнера на основе результатов первой сборки. Это намного ближе к тому, что нам хотелось бы получить, но для этого требуются два разных файла Dockerfile, которые должны использоваться последовательно или управляться отдельным сценарием.

Однако самое удачное решение стало возможно с появлением поддержки многоэтапной сборки, которая позволяет объединить несколько разных сборок – даже с совершенно разными базовыми образами – так, чтобы результаты выполнения одного этапа можно было выборочно копировать в другой, отбрасывая все ненужное. Для реализации этого решения определим сборку с двумя этапами: этап «build» будет генерировать двоичный файл Go, а этап «image» будет использовать этот двоичный файл для создания окончательного образа.

Для этого используем несколько инструкций `FROM` в Dockerfile, каждая из которых будет отмечать начало нового этапа. Каждому этапу можно дать произвольное имя. Например, вот как можно дать имя `build` нашему первому этапу:

```
FROM golang:1.16 as build
```

После создания этапов с именами можно использовать инструкцию `COPY`, чтобы скопировать любой артефакт *с любого предыдущего этапа*. На последнем этапе может иметься такая инструкция, которая копирует файл `/src/kvs` из этапа `build` в текущий рабочий каталог:

```
COPY --from=build /src/kvs .
```

Объединив все вместе, получаем законченный файл Dockerfile:

```
# Этап 1: Компиляция двоичного файла в контейнеризованном окружении Golang
#
FROM golang:1.16 as build

# Скопировать исходные файлы с хоста
COPY . /src

# Назначить рабочим каталог с исходным кодом
WORKDIR /src
```

```
# Собрать двоичный файл!
RUN CGO_ENABLED=0 GOOS=linux go build -o kvs

# Этап 2: Сборка образа со службой хранилища пар ключ/значение
#
# Использовать образ "scratch", не содержащий распространяемых файлов
FROM scratch

# Скопировать двоичный файл из контейнера build
COPY --from=build /src/kvs .

# Если предполагается использовать TLS, скопировать файлы .pem
COPY --from=build /src/*.pem .

# Сообщить фреймворку Docker, что служба будет использовать порт 8080
EXPOSE 8080

# Команда, которая должна быть выполнена при запуске контейнера
CMD ["/kvs"]
```

Теперь, получив законченный файл Dockerfile, можно создать образ, как мы уже делали это выше. На этот раз отметим его тегом `multipart`, чтобы потом сравнить два образа:

```
$ docker build --tag kvs:multipart .
...вывод опущен.

$ docker images
REPOSITORY     TAG           IMAGE ID         CREATED              SIZE
kvs            latest        7b1fb6fa93e3     2 hours ago          6.88MB
kvs            multipart     b83b9e479ae7     4 minutes ago        6.56MB
```

Это обнадеживает! Теперь у нас есть единственный файл Dockerfile, который компилирует исходный код на Go – независимо от того, установлен ли компилятор Go на сервере сборки, – и передает полученный двоичный файл в этап `FROM scratch` для создания очень, очень маленького образа, не содержащего ничего, кроме службы хранилища пар ключ/значение.

Но не останавливайтесь на достигнутом. При желании вы можете добавить другие этапы, например этап тестирования, выполняющий любые модульные тесты перед этапом сборки окончательного образа. Однако я не буду обсуждать эту тему дальше, потому что в ней нет ничего нового, но я призываю вас попробовать сделать это у себя.

Сохранение данных контейнера вовне

Контейнеры должны быть эфемерными, и любой контейнер должен разрабатываться и запускаться с учетом того, что он может быть (и будет) уничтожен в любое время и унесет с собой все свои данные. Для ясности отмечу, что это – хорошее качество контейнеров. Но иногда желательно, чтобы данные сохранялись между перезапусками контейнера.

Например, возможность монтировать внешние файловые системы непосредственно в файловую систему контейнера может помочь отделить конфигурационные файлы от образов и избавить от необходимости перестраивать их всякий раз, когда меняются только настройки. Это очень мощная стратегия и, вероятно, наиболее распространенный вариант сохранения данных контейнера вовне. На самом деле она настолько распространена, что Kubernetes даже предлагает для этой цели отдельный тип ресурса `ConfigMap`.

Точно так же может появиться желание сохранить данные, сгенерированные в контейнере, чтобы они сохранились после его остановки. Сохранение данных на хосте может оказаться отличной стратегией, например, для разогрева кешей. Однако важно помнить об одной из реалий облачной инфраструктуры: ничто не является постоянным, даже серверы. Не храните на хосте ничего, что не должно быть потеряно навсегда.

К счастью, в отличие от фреймворка Docker, ограничивающего круг возможностей переноса данных только локальными дисками[1], системы управления контейнерами, такие как Kubernetes, предоставляют различные абстракции (https://oreil.ly/vBXfA), позволяющие сохранять данные даже в случае потери хоста.

К сожалению, для полного освещения этой темы пришлось бы написать целую книгу, поэтому мы не будем обсуждать Kubernetes. Но я настоятельно рекомендую вам внимательно изучить превосходную документацию Kubernetes (https://oreil.ly/wxlmg) и не менее превосходную книгу Брендана Бернса (Brendan Burns), Джо Беда (Joe Beda) и Келси Хайтауэр (Kelsey Hightower) *Kubernetes: Up and Running* (O'Reilly; https://oreil.ly/e6rve).

Итоги

Это была длинная глава, и мы затронули в ней множество разных тем. Давайте вспомним, что мы узнали!

- Опираясь на базовые принципы, мы спроектировали и реализовали простое монолитное хранилище пар ключ/значение, используя пакеты `net/http` и `gorilla/mux` для создания службы RESTful на основе функций, предоставляемых небольшой, независимой и легко тестируемой библиотекой Go.
- Использовали мощные возможности интерфейса Go для создания двух совершенно разных реализаций регистратора транзакций, одна из которых основана на локальных файлах и использует `os.File`, а также пакеты `fmt` и `bufio`; а другая – на базе данных Postgres и использует пакеты `database/sql` и драйвер Postgres `github.com/lib/pq`.
- Мы обсудили важность обеспечения безопасности, исследовали некоторые основы протокола TLS как части более широкой стратегии безопасности и внедрили поддержку HTTPS в нашу службу.

[1] Я намеренно не упоминаю такие решения, как Amazon Elastic Block Store, которые тоже могут помочь, но имеют свои собственные проблемы.

- Наконец, мы рассмотрели контейнеризацию, одну из основных облачных технологий, и узнали, как создавать образы, запускать контейнеры и управлять ими. Мы даже поместили в контейнеры не только наше приложение, но и процесс его сборки.

В следующих главах мы продолжим расширение нашей службы хранилища пар ключ/значение и попутно будем знакомиться с новыми идеями и понятиями, так что оставайтесь с нами. Скоро станет еще интереснее.

Часть III

ОБЛАЧНЫЕ АТРИБУТЫ

Глава 6

Все дело в надежности

> Важнейшим свойством программы является выполнение ею намерений пользователя[1].
>
> – Ч. Э. Р. Хоар (C. A. R. Hoare), *Communications of the ACM* (October 1969)

Профессор сэр Чарльз Энтони Ричард (Тони) Хоар (Charles Antony Richard (Tony) Hoare) – блестящий ученый. Он изобрел алгоритм быстрой сортировки, создал логику Хоара для рассуждений о правильности компьютерных программ и формальный язык «взаимодействия последовательных процессов» (Communicating Sequential Processes, CSP), ставший основой модели конкуренции в Go. Да, и еще он разработал парадигму структурного программирования[2], которая составляет основу всех современных языков программирования, широко используемых в настоящее время. Он также изобрел пустую ссылку. Но, пожалуйста, не обвиняйте его в этом. В 2009 году он публично извинился[3] за это, назвав данное изобретение своей «ошибкой на миллиард долларов».

Тони Хоар буквально изобрел программирование в том виде, в каком мы его знаем. Поэтому, когда он говорит, что важнейшим свойством программы является выполнение ею намерений пользователя, то можете принять это на веру. Подумайте об этом: Хоар, в частности (и совершенно справедливо), указывает, что правильность работы программы определяется по совпадению с намерениями ее *пользователей*, а не *создателей*. Как плохо, что намерения пользователей программы не всегда совпадают с намерениями ее создателя!

В свете этого утверждения логично предположить, что первое ожидание пользователя относительно программы – *она должна работать*. Но что означает «работать»? На самом деле это довольно широкий вопрос, лежащий в основе проектирования облачных приложений. Первая цель данной главы –

---

1 Hoare, C.A.R. «An Axiomatic Basis for Computer Programming». *Communications of the ACM*, vol. 12, no. 10, October 1969, pp. 576–583. https://oreil.ly/jOwO9.

2 Когда Эдсгер В. Дейкстра (Edsger W. Dijkstra) написал статью о вреде инструкции GOTO, он ссылался на работу Хоара, посвященную структурному программированию.

3 Hoare, Tony. «Null References: The Billion Dollar Mistake». *InfoQ.com*. 25 August 2009. https://oreil.ly/4QWS8.

познакомиться с этой идеей и в процессе знакомства ввести такие понятия, как «надежность» и «безотказность», которые можно использовать для более точного описания (и удовлетворения) ожиданий пользователей. Наконец, мы коротко рассмотрим ряд практик, обычно используемых в облачной разработке, чтобы гарантировать соответствие служб ожиданиям пользователей. Мы подробно обсудим каждую из них в оставшейся части книги.

В чем суть облачных вычислений?

В главе 1 мы потратили несколько страниц на определение «по-настоящему облачной службы», начав с определения фонда Cloud Native Computing Foundation и затем перечислив свойства идеальной облачной службы. Затем еще несколько страниц потратили на обсуждение требований, которые привели к широкому распространению облачных технологий.

Однако мы почти не говорили о причинах, *почему* следует использовать именно облачные технологии. Почему появилась идея облачных вычислений? Зачем вообще нужны облачные системы? Какова их цель? Что в них особенного? Почему это должно нас волновать?

Итак, почему *появилась* технология облачных вычислений? На самом деле ответ на этот вопрос довольно прост: все дело в надежности. В первой части этой главы мы подробно рассмотрим понятие надежности: что это такое, почему это важно и какое место занимает надежность в шаблонах и методах, которые мы называем облачными.

Все дело в надежности

Холли Камминс (Holly Cummins), руководитель учебного центра IBM Garage всемирного сообщества разработчиков, однажды сказала, что «если термин “облачное окружение” и может быть синонимом, то только синонимом понятию идемпотентности»[1]. Камминс – гениальный человек и высказала много блестящих мыслей[2], но мне кажется, что она отразила только половину картины. Я думаю, что идемпотентность очень важна – возможно, даже необходима для облачных вычислений, – но одной ее недостаточно.

История развития программного обеспечения, особенно сетевого, – это история борьбы за оправдание ожиданий все более искушенных пользователей. Давно прошли те времена, когда службу можно было остановить ночью «для обслуживания». Современные пользователи все больше полагаются на службы и ожидают, что они будут доступны постоянно и оперативно реагировать на их запросы. Помните, когда в последний раз вы пытались запустить фильм на Netflix и это заняло больше пяти секунд?

---

[1] *Cloud Native Is About Culture, Not Containers*. Cummins, Holly. Cloud Native London 2018.

[2] Если у вас появится возможность побеседовать с ней, не отказывайтесь от нее!

Пользователей не волнует, что службы требуют ухода. Они не будут терпеливо ждать, пока вы отыщете этот загадочный источник задержки. Они просто хотят быстрее досмотреть второй сезон сериала «Во все тяжкие»[1].

Все шаблоны и приемы, которые мы ассоциируем с облачными технологиями, – *каждый из них* – существуют, чтобы дать возможность развертывать, эксплуатировать и масштабировать надежные службы в ненадежном окружении, которые сделают пользователей счастливыми.

Иначе говоря, я думаю, что если термин «облачное окружение» и может быть синонимом, то только синонимом понятию «надежность».

Что такое надежность, и почему она так важна?

Я выбрал слово «надежность» не просто так. На самом деле это одно из базовых понятий в *системной инженерии*, где работают очень умные люди и которые говорят очень умные вещи о проектировании сложных систем и управлении ими. Понятие надежности в контексте вычислений было впервые строго определено Жан-Клодом Лапри (Jean-Claude Laprie) около 35 лет назад[2] как соответствие ожиданиям пользователей. Оригинальное определение Лапри изменялось и расширялось на протяжении многих лет разными авторами, но вот мое любимое:

> Надежность компьютерной системы – это способность избегать более частых или более серьезных отказов и более длительных простоев, чем это приемлемо с точки зрения пользователей[3].
>
> – Фундаментальные концепции надежности компьютерных систем (2001 г.)

Иначе говоря, надежная система последовательно делает то, чего ожидают от нее пользователи, и ее можно быстро исправить, когда происходит сбой.

Согласно этому определению, система является надежной, только когда ей можно *обоснованно* доверять. Очевидно, что систему нельзя считать надежной, если она или любой ее компонент может выйти из строя в любой момент времени, или если ей потребуются часы для восстановления после сбоя. Даже проработав без сбоев в течение нескольких месяцев, ненадежная система может оказаться в шаге от катастрофы: на удачу нельзя полагаться.

К сожалению, сложно объективно оценить «ожидания пользователей». По этой причине, как показано на рис. 6.1, надежность – это зонтичная концепция, охватывающая несколько более конкретных атрибутов, поддающихся

---

[1] Помните, что Уолт тогда сделал с Джейн? Это было так запутано.

[2] Laprie, J-C. «Dependable Computing and Fault Tolerance: Concepts and Terminology». *FTCS-15 The 15th Int'l Symposium on Fault-Tolerant Computing*, June 1985, pp. 2–11. https://oreil.ly/UZFFY.

[3] A. Avižienis, J. Laprie, and B. Randell. «Fundamental Concepts of Computer System Dependability». *Research Report No. 1145, LAAS-CNRS*, April 2001. https://oreil.ly/4YXd1.

количественной оценке, – доступность, безотказность и ремонтопригодность, – каждый из которых подвержен похожим угрозам, которые можно преодолеть похожими средствами.

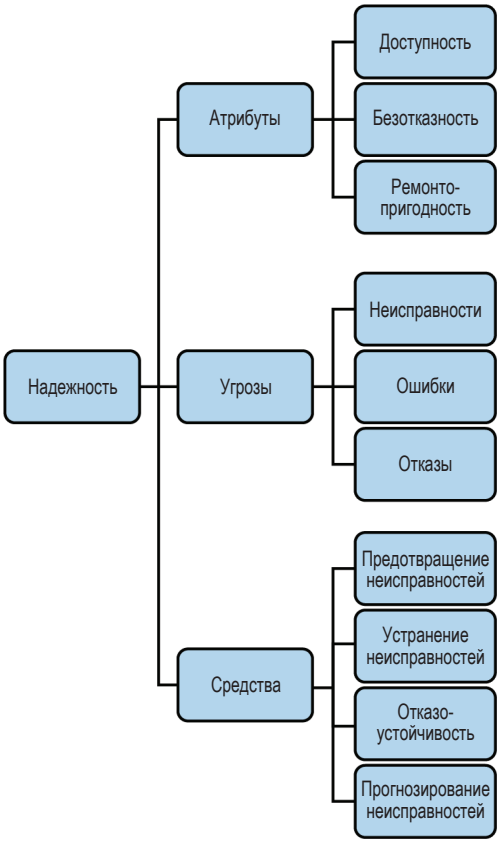


Рис. 6.1 ❖ Системные атрибуты и средства, способствующие надежности

Несмотря на то что понятие «надежность» само по себе может быть несколько неоднозначным и субъективным, атрибуты, которые способствуют надежности, поддаются количественной оценке и могут быть достаточно полезными:

Доступность
: Способность системы выполнять свои функции в каждый конкретный момент времени. Обычно доступность выражается как вероятность успешной обработки запроса системой и определяется как время безотказной работы, деленное на общее время.

Безотказность
: Способность системы выполнять свои функции в течение заданного интервала времени. Безотказность часто выражается как среднее время на-

работки на отказ (общее время, деленное на количество отказов) или как частота отказов (количество отказов, деленное на общее время).

Ремонтопригодность
: Возможность модификации и ремонта системы. Существует множество косвенных показателей ремонтопригодности, начиная от вычислений цикломатической сложности и заканчивая отслеживанием количества времени, необходимого для изменения поведения системы в соответствии с новыми требованиями или для восстановления ее функционального состояния.

Позднее авторы расширили определение надежности, которое дал Лапри, добавив несколько свойств, связанных с безопасностью, включая безопасность, конфиденциальность и целостность. После некоторых раздумий я опустил их, но не потому, что безопасность не важна (она *ОЧЕНЬ* важна!), а лишь для краткости. Для более или менее полного обсуждения безопасности потребовалось бы написать отдельную книгу.

Надежность – это не безотказность

Если вам приходилось читать книги издательства O'Reilly по проектированию безотказности сайтов (Site Reliability Engineering, SRE)[1], то вы уже многое знаете о безотказности. Однако, как показано на рис. 6.1, безотказность – это лишь одно из свойств, способствующих общей надежности.

Но если это правда, то почему безотказность стала стандартной оценкой функционирования служб? Почему есть «инженеры по безотказности сайта», но нет «инженеров по надежности сайта»?

На эти вопросы можно ответить по-разному, но суть заключается в том, что определение «надежности» является чисто качественным. Надежность нельзя измерить количественно, а когда что-то нельзя измерить, очень трудно построить набор правил.

С другой стороны, безотказность можно оценить количественно. При наличии четкого определения[2], что для системы означает «корректное» функционирование, относительно несложно вычислить «безотказность» этой системы, что делает ее мощным (хотя и косвенным) показателем оценки пользовательского опыта (впечатлений пользователя от взаимодействия с системой).

Надежность обеспечивается не только операторами

Когда сетевые службы только появились, задачей разработчиков было создание служб, а системных администраторов («операторов») – развертывание этих служб на серверах и поддержка их работы. Какое-то время такое

---

[1] Если нет, то рекомендую *Site Reliability Engineering: How Google Runs Production Systems* (https://oreil.ly/OJn99). Это очень хорошая книга. (*Бейер Бетси, Джоунс Крис*. Site Reliability Engineering. Надежность и безотказность как в Google. СПб.: Питер, 2019. ISBN: 978-5-4461-0976-0. – *Прим. перев.*)

[2] Во многих организациях для этого используются цели уровня обслуживания (Service-Level Objectives, SLO).

разделение труда давало положительные результаты, но имело неприятный побочный эффект: основное свое внимание разработчики уделяли разработке новых возможностей, иногда в ущерб стабильности и корректной работе.

К счастью, за последние десять лет, одновременно с появлением методологии DevOps[1], появилась новая волна технологий, позволяющих полностью изменить подход к работе специалистов, занятых во всей технологической цепочке.

С точки зрения эксплуатации и обслуживания, появление технологий «инфраструктура как услуга» (Infrastructure as a Service, IaaS) и «платформа как услуга» (Platform as a Service, PaaS), и инструментов, таких как Terraform и Ansible, работа с инфраструктурой стала больше похожа на написание программного обеспечения.

С точки зрения разработки, популяризация таких технологий, как контейнеры и бессерверные вычисления, дала разработчикам новый набор возможностей управления сопровождением и эксплуатацией, особенно в отношении виртуализации и развертывания.

В результате некогда строгая граница между программным обеспечением и инфраструктурой становится все более размытой. Можно даже утверждать, что с развитием абстракций инфраструктуры, таких как виртуализация, использование фреймворков оркестровки контейнеров, например, Kubernetes, и программно-определяемого поведения, как в сервисных сетках (service mesh), мы можем оказаться в точке, где они сливаются. Теперь все суть есть программное обеспечение.

Постоянно растущий спрос на надежность обслуживания привел к появлению нового поколения облачных технологий. Влияние этих новых технологий и их возможностей оказалось весьма значительным, и традиционные роли разработчиков и операторов меняются в соответствии с ними. Разрозненность исчезает, и быстрое производство надежных и высококачественных служб все чаще становится результатом совместных усилий проектировщиков, разработчиков и специалистов по сопровождению.

Достижение надежности

Теперь перейдем к главному. Если вы зашли так далеко, поздравляю.

До сих пор мы обсуждали определение «надежности», данное Лапри, которое можно перефразировать как «счастливые пользователи», и обсудили атрибуты – доступность, безотказность и ремонтопригодность, – способствующие этому. Это все хорошо, но без действенных советов, как добиться надежности, вся дискуссия носит чисто академический характер.

---

[1] Расшифровывается как «Development Operations» – интеграция разработки и эксплуатации; методология разработки программного обеспечения, нацеленная на общение, взаимодействие и интеграцию специалистов по разработке и информационно-технологическому обслуживанию программного обеспечения. – *Прим. перев.*

Лапри тоже так думал и определил четыре обширные категории методов, которые можно использовать для повышения надежности системы (или которые, в случае их неиспользования, могут ее снизить):

Предотвращение неисправностей
: Методы предотвращения неисправностей используются при конструировании системы, чтобы предотвратить возникновение или появление неисправностей.

Отказоустойчивость
: Методы обеспечения отказоустойчивости используются при проектировании и реализации системы, чтобы предотвратить отказ службы в случае отказа отдельных ее компонентов.

Устранение неисправностей
: Методы устранения неисправностей используются для уменьшения количества и серьезности отказов.

Прогнозирование неисправностей
: Методы прогнозирования неисправностей используются для определения наличия неисправностей, обнаружения отказов и оценки последствий отказов.

Как показано на рис. 6.2, эти четыре категории на удивление хорошо соответствуют пяти атрибутам облачных окружений, которые были представлены в главе 1.

Рис. 6.2 ❖ Четыре средства достижения надежности и соответствующие им атрибуты облачных окружений

Предотвращение неисправностей и отказоустойчивость образуют два нижних уровня пирамиды, соответствующих масштабируемости, слабой связанности и устойчивости. Проектирование системы с учетом масштабируемости предотвращает появление множества ошибок, типичных для облачных приложений, а методы обеспечения отказоустойчивости позволяют

системе преодолевать отказы, когда они возникают. Методы обеспечения слабой связанности попадают в обе категории: они предотвращают отказы и повышают отказоустойчивость службы. Можно сказать, что вместе они вносят вклад в то, что Лапри называет *гарантией надежности*: средств, благодаря которым система сохраняет способность выполнять свои функции.

Методы и архитектурные решения, способствующие управляемости, помогают создавать системы, которые можно легко модифицировать, что упрощает процесс устранения неисправностей при их выявлении. Точно так же наблюдаемость естественным образом способствует прогнозированию неисправностей в системе. Вместе методы устранения и прогнозирования неисправностей способствуют тому, что Лапри назвал *подтверждением надежности*: средства, с помощью которых достигается уверенность в способности системы выполнять свои функции.

Рассмотрим последствия этих отношений: то, что 35 лет назад было чисто академическим упражнением, по сути, было заново открыто как естественное следствие многолетнего накопления опыта создания безотказных промышленных систем. Надежность прошла полный круг.

В следующих разделах мы исследуем эти взаимосвязи более подробно и кратко ознакомимся с содержимым последующих глав, в которых мы обсудим, почему эти две, казалось бы, несопоставимые системы в действительности довольно близки друг к другу.

Предотвращение неисправностей

В основании пирамиды «средств достижения надежности» лежат методы, направленные на предотвращение возникновения или появления неисправностей. Как могут подтвердить опытные программисты, многие (если не большинство) классы ошибок и неисправностей можно предсказать и предотвратить на самых ранних этапах разработки. Таким образом, многие методы предотвращения неисправностей вступают в игру на этапе проектирования и реализации службы.

Рекомендуемые практики программирования

Предотвращение неисправностей – одна из основных целей программной инженерии и явная цель любой методологии разработки, от парного программирования до разработки через тестирование и практики обзора кода. Многие из этих методов можно объединить в так называемые «рекомендуемые практики программирования», о которых написано бесчисленное количество замечательных книг и статей, поэтому мы не будем подробно останавливаться на них.

Особенности языка

Выбор языка тоже может сильно повлиять на способность предотвращать или исправлять неисправности. Многие особенности языка, приветствуемые

некоторыми программистами, такие как динамическая типизация, арифметика указателей, ручное управление памятью и генерируемые исключения (и это лишь малая часть), легко могут привести к непреднамеренному поведению, которое трудно найти и исправить, и иногда даже к злонамеренному использованию.

Все это сильно повлияло на многие архитектурные решения при создании языка Go, в результате чего появился строго типизированный язык со сборкой мусора. Чтобы узнать, почему Go особенно хорошо подходит для разработки облачных служб, загляните в главу 2.

Масштабируемость

Идея масштабируемости была кратко представлена еще в главе 1, где она определялась как способность системы продолжать предоставлять услуги в условиях значительного изменения нагрузки.

Там же были описаны два разных подхода к масштабированию – вертикальное масштабирование путем изменения объема доступных ресурсов и горизонтальное масштабирование путем добавления (или удаления) экземпляров службы, – а также некоторые плюсы и минусы каждого из них.

В главе 7 мы более подробно рассмотрим каждый из этих подходов, а также их подводные камни и недостатки. Кроме того, мы много поговорим о проблемах, связанных с состоянием[1]. На данный момент, однако, достаточно сказать, что необходимость масштабирования службы добавляет немало накладных расходов, включая, помимо прочего, стоимость, сложность и отладку.

Масштабирование ресурсов часто неизбежно, но обычно лучше (и дешевле!), учитывая эффективность выполнения и алгоритмическое масштабирование, не поддаваться искушению использовать аппаратное обеспечение для решения проблемы и откладывать реализацию масштабирования на потом. Поэтому мы рассмотрим ряд особенностей и инструментов Go, которые позволят нам выявлять и устранять распространенные проблемы, такие как утечки памяти и конфликты блокировок, имеющие тенденцию мешать масштабированию систем.

Слабая связанность

Слабая связанность, определение которой мы дали в разделе «Слабая связанность» в главе 1, – это свойство системы и стратегия проектирования, обеспечивающие минимальную осведомленность компонентов системы друг о друге. Слабая связанность служб может иметь огромное – и часто недооцениваемое – влияние на способность системы масштабироваться, а также изолировать и выдерживать сбои отдельных компонентов.

В самом начале, когда только появились микросервисы, у них было большое число противников, которые указывали на сложность развертывания и обслуживания систем на основе микросервисов как свидетельство того,

---

[1] Состояние приложения и без того сложно обслуживать, а если его реализовать неправильно, то оно может усложнить масштабирование.

что такие архитектуры слишком сложны, чтобы быть жизнеспособными. Я не согласен с ними, но вижу, откуда берутся эти доводы, учитывая, насколько легко создать *распределенный монолит*. Отличительной чертой распределенного монолита является тесная связь между его компонентами, в результате чего приложение получает всю сложность микросервисов, плюс все запутанные зависимости типичного монолита. Если вы развертываете большое количество служб вместе или в результате ошибки происходит каскадный сбой, вызывающий остановку всей системы, то, скорее всего, вы имеете дело с распределенным монолитом.

Говорить о слабо связанных системах проще, чем реализовать их, но это вполне возможно при соблюдении дисциплины и разумном проведении границ. В главе 8 мы посмотрим, как использовать контракты обмена данными для установления этих границ, познакомимся с некоторыми моделями синхронных и асинхронных взаимодействий, а также исследуем архитектурные шаблоны и пакеты, используемые для их реализации и предотвращения создания ужасного распределенного монолита.

Отказоустойчивость

Отказоустойчивость имеет ряд синонимов – самовосстановление, самоисправление, устойчивость: все они описывают способность системы обнаруживать ошибки и предотвращать их каскадное распространение и превращение в полномасштабный отказ. Обычно отказоустойчивость делится на две части: *обнаружение ошибок*, когда ошибка обнаруживается во время нормальной работы, и *восстановление*, когда система возвращается в состояние, в котором ее можно снова активировать.

Наиболее распространенной, пожалуй, стратегией обеспечения устойчивости является избыточность: дублирование критических компонентов (запуск нескольких экземпляров служб) или вызовов функций (повторные попытки отправить запрос). Это обширная и очень интересная область со множеством подводных камней, которые мы рассмотрим в главе 9.

Устранение неисправностей

Устранение неисправностей, третье из четырех средств обеспечения надежности, – это процесс уменьшения количества и серьезности скрытых недостатков программного обеспечения, которые могут вызывать ошибки, еще до того, как они проявятся в виде ошибок.

Даже в идеальных условиях может возникнуть множество причин, по которым система может ошибиться или начать проявлять непредусмотренное поведение. Она может не выполнить ожидаемое действие или выполнить совершенно неправильное действие, возможно, по злому умыслу. Однако в реальной жизни условия редко бывают идеальными.

Многие неисправности можно идентифицировать путем тестирования, которое позволяет убедиться, что система (или, по крайней мере, ее компоненты) действует, как ожидалось, в известных условиях.

Но как насчет неизвестных условий? Требования меняются, и реальный мир не заботится о том, чтобы соответствовать вашим условиям тестирования. К счастью, приложив определенные усилия, систему можно спроектировать так, что она будет достаточно управляемой, чтобы ее поведение можно было корректировать для обеспечения безопасности, бесперебойной работы и соответствия меняющимся требованиям.

Мы кратко обсудим их дальше.

Проверка и тестирование

Есть ровно четыре способа найти скрытые программные ошибки в коде: тестирование, тестирование, тестирование и невезение.

Это была шутка, но в каждой шутке есть доля правды: если вы не обнаружите свои программные ошибки, то это сделают ваши пользователи. Если повезет. Если нет, то их найдут плохие парни, которые попытаются ими воспользоваться.

Если без шуток, то есть два общих подхода к поиску ошибок в программном обеспечении в процессе разработки:

Статический анализ
: Автоматический анализ кода на основе правил, производимый без фактического выполнения программ. Статический анализ полезен на ранних стадиях и может использоваться для поиска ошибок и брешей в системе безопасности с применением согласованных методов, независимо от уровня знаний или усилий.

Динамический анализ
: Проверка правильности системы или подсистемы путем ее выполнения в контролируемых условиях и оценки поведения. Чаще называется просто «тестирование».

Ключом к тестированию является наличие программного обеспечения, *пригодного для тестирования* за счет минимизации *степеней свободы* – диапазона возможных состояний – его компонентов. Функции с высокой степенью пригодности для тестирования решают единственную задачу, для одних и тех же входных данных всегда возвращают один и тот же результат и имеют незначительные *побочные эффекты* или вообще не имеют их; то есть они не изменяют переменные за пределами своей области видимости. Простите меня за занудство, но такой подход сужает *пространство поиска* – набор всех возможных решений – каждой функции.

Тестирование – важный шаг в разработке программного обеспечения, которым слишком часто пренебрегают. Создатели Go поняли это и встроили модульное тестирование и тестирование производительности в сам язык в виде команды `go test` и пакета тестирования (https://oreil.ly/PrhXq). К сожалению, глубокое погружение в теорию тестирования выходит далеко за рамки этой книги, и все же мы поговорим немного о тестировании в главе 9.

Управляемость

Ошибки возникают, когда поведение системы не совпадает с требованиями. Но что происходит, когда требования меняются?

Такая характеристика облачных систем, как *управляемость*, впервые представленная в разделе «Управляемость» главы 1, позволяет настраивать поведение системы без изменения кода. Управляемая система, по сути, имеет «ручки настройки», которые позволяют контролировать в реальном времени безопасность системы, бесперебойность ее работы и соответствие меняющимся требованиям.

Управляемость может принимать различные формы, включая (но не ограничиваясь!) регулировку и настройку потребления ресурсов, применение исправлений безопасности на лету, *флаги* для включения или отключения особенностей и даже загрузку плагинов для изменения поведения.

Очевидно, что управляемость – это обширная тема, поэтому мы рассмотрим лишь несколько механизмов, предлагаемых языком Go, в главе 10.

Прогнозирование неисправностей

На вершине нашей пирамиды «средств достижения надежности» (рис. 6.2) находится *прогнозирование неисправностей*, которое основывается на знаниях и решениях, полученных и реализованных на нижних уровнях и позволяющих оценить текущее число неисправностей, их распространенность в будущем и вероятные последствия.

Слишком часто прогнозирование строится на основе предположений и интуиции, что обычно приводит к неудачам, когда исходное предположение перестает быть верным. К более систематическим подходам можно отнести: анализ видов и последствий неисправностей и стресс-тестирование, которые здорово помогают понять возможные режимы отказа системы.

В системе, спроектированной с поддержкой *наблюдаемости*, которую мы подробно обсудим в главе 11, есть возможность следить за индикаторами режимов отказа, чтобы их можно было спрогнозировать и исправить до того, как они проявятся в виде сбоев. Более того, когда происходят неожиданные отказы – а они обязательно произойдут, – системы с высокой степенью наблюдаемости позволяют быстро идентифицировать, изолировать и исправить лежащие в их основе причины.

Непреходящая актуальность методологии «Двенадцать факторов»

В начале 2010-х разработчики компании Heroku, занимающейся поддержкой технологии «платформа как услуга» (Platform as a Service, PaaS), пионера облачных вычислений, осознали, что они снова и снова сталкиваются с одними и теми же недостатками при разработке веб-приложений.

Руководствуясь списком системных (на их взгляд) проблем в современных приложениях, они опубликовали документ «Двенадцать факторов» (The Twelve Factor App). Он включал набор из двенадцати правил и рекомендаций, составляющих методологию разработки веб-приложений и, в более широком смысле, облачных приложений (хотя в то время термин «облачные» еще не вошел в широкое употребление). Методология заключалась в создании веб-приложений, которые[1]:

- используют декларативные форматы для автоматизации настройки, помогающие минимизировать время и затраты на присоединение новых разработчиков к проекту;
- используют чистые контракты для взаимодействия с базовой операционной системой, обеспечивающие максимальную переносимость;
- подходят для развертывания в современных облачных платформах, устраняющих необходимость в серверах и системном администрировании;
- минимизируют расхождения между разработкой и производством, обеспечивая непрерывное развертывание для максимальной гибкости;
- поддерживают масштабирование без значительных изменений в наборах используемых инструментов, архитектуры или методов разработки.

«Двенадцать факторов» и свойства, которые пропагандирует этот документ, не получили полной оценки в 2011 году, когда были впервые опубликованы, но по мере того, как сложности облачной разработки стали более понятными, их начали указывать как руководство для разработки любой службы, предназначенной для работы в облаке.

I. Кодовая база

> Одна кодовая база в системе управления версиями, множество развертываний.
>
> – The Twelve-Factor App

Для любой конкретной службы должна быть ровно одна кодовая база, которая будет использоваться при создании любого количества неизменяемых выпусков для развертывания в нескольких окружениях. В число таких окружений обычно входят производственный сайт и один или несколько промежуточных сайтов и сайтов разработки.

Наличие нескольких служб, использующих один и тот же код, как правило, приводит к стиранию границ между модулями и постепенному дрейфу в сторону монолита, что затрудняет внесение изменений в одну часть службы, не оказывающих неожиданного влияния на другую часть (или другую службу!). Общий код следует реорганизовать в библиотеки, которые могут храниться в системе управления версиями отдельно и подключаться с помощью диспетчера зависимостей.

---

[1] *Адам Уиггинс* (Wiggins, Adam). Двенадцать факторов. 2011. https://12factor.net.

Распределение одной службы по нескольким репозиториям делает практически невозможным автоматическую сборку и развертывание этой службы.

II. Зависимости

> Явно объявляйте и изолируйте зависимости.
>
> – The Twelve-Factor App

Для любой данной версии кодовой базы команды `go build`, `go test` и `go run` должны действовать детерминированно: всегда давать одинаковый результат, и полученный продукт всегда должен одинаково реагировать на одни и те же входные данные.

Но что, если зависимость – импортируемый пакет или установленный системный инструмент, которым программист не управляет, – изменится так, что нарушит сборку, внесет ошибку или станет несовместимой со службой?

Большинство языков программирования предлагают систему упаковки для распространения вспомогательных библиотек, и Go в том числе[1]. Используя механизм модулей Go (https://oreil.ly/68ds1) для объявления всех зависимостей, можно гарантировать, что импортированные пакеты не изменятся и не нарушат сборку.

В свете этого хотелось бы отметить, что службам следует также избегать использования функции `Command` из пакета `os/exec` для запуска внешних инструментов, таких как ImageMagick или `curl`.

Да, ваш целевой инструмент может быть доступен во всех (или в большинстве) системах, но *невозможно гарантировать*, что будет присутствовать и останется полностью совместимым со службой везде, где она может работать в настоящем или будущем. В идеале, если вашей службе требуется внешний инструмент, то он должен быть включен в состав службы и, соответственно, в ее репозиторий.

III. Конфигурация

> Храните конфигурацию в среде выполнения.
>
> – The Twelve-Factor App

Конфигурация – это все, что может различаться в разных окружениях (промежуточное, производственное, разработки и т. д.), и она должна четко отделяться от кода. Ни при каких обстоятельствах конфигурация приложения не должна встраиваться в код.

К элементам конфигурации могут относиться, например:

- URL-адреса и описатели баз данных или других вышестоящих служб, даже если они в ближайшее время не должны измениться;
- *любые* секреты, например пароли или учетные данные для доступа к внешним службам;

[1] Хотя этого пришлось ждать довольно долго!

- значения, зависящие от окружения, например каноническое имя хоста для развертывания.

Распространенным способом отделения конфигурации от кода является ее *выделение* в некоторый конфигурационный файл – часто в формате YAML[1], – который может или не может храниться в репозитории вместе с кодом. Это, безусловно, лучше, чем хранить конфигурацию в коде, но далеко не идеально.

Во-первых, если файл конфигурации хранится вне репозитория, его слишком легко случайно сохранить в репозиторий. Кроме того, такие файлы имеют тенденцию размножаться в виде разных версий для разных окружений, хранящихся в разных местах, что затрудняет просмотр конфигураций и управление ими.

Как вариант в вашем репозитории *могут иметься* разные версии конфигураций для разных окружений, но это довольно неуклюжее решение, которое часто требует применения некоторых неудобных уловок для работы с репозиторием.

Не бывает частично скомпрометированных секретов

Стоит подчеркнуть, что в отличие от настроек конфигурации, которые никогда не должны присутствовать в коде, пароли и другие конфиденциальные секреты *вообще никогда и ни при каких обстоятельствах* не должны присутствовать в коде. Есть риск по неосмотрительности поделиться этими секретами со всем миром.

Как только секрет будет раскрыт, он перестанет быть секретом. Частично скомпрометированных секретов не бывает.

Всегда относитесь к своему репозиторию – и к коду, который он хранит, – так, как будто он может быть опубликован в любое время. Что, конечно же, возможно.

Вместо конфигураций в коде или даже во внешних конфигурационных файлах в документе «Двенадцать факторов» рекомендуется хранить конфигурации в переменных окружения. Такой подход имеет ряд преимуществ:

- он является стандартным и в значительной степени не зависит от операционной системы и языка;
- переменные окружения легко изменять без изменения кода;
- переменные окружения легко внедряются в контейнеры.

В Go есть несколько инструментов для этого.

Первый и самый простой – пакет `os`, который предлагает функцию `os.Getenv`:

```
name := os.Getenv("NAME")
place := os.Getenv("CITY")

fmt.Printf("%s lives in %s.\n", name, place)
```

[1] Худший язык в мире для оформления конфигурации (кроме всех остальных).

Для более сложных вариантов конфигурации доступно несколько отличных пакетов. Из них особенной популярностью пользуется `spf13/viper` (https://oreil.ly/8giE4). Вот пример использования Viper:

```
viper.BindEnv("id")          // Преобразует в верхний регистр автоматически
viper.SetDefault("id", "13") // Значение по умолчанию "13"

id1 := viper.GetInt("id")
fmt.Println(id1)             // 13

os.Setenv("ID", "50")        // Обычно выполняется за пределами приложения!

id2 := viper.GetInt("id")
fmt.Println(id2)             // 50
```

Кроме того, Viper предлагает возможности, отсутствующие в стандартных пакетах, такие как определение значений по умолчанию, типизированные переменные и чтение флагов командной строки, разные форматы конфигурационных файлов и даже удаленные системы конфигураций, такие как etcd и Consul.

Более подробно о Viper и других вопросах конфигурации мы поговорим в главе 10.

IV. Сторонние службы

> Считайте сторонние службы подключаемыми ресурсами.
>
> – The Twelve-Factor App

Сторонняя служба – это любая нижестоящая зависимость, которую данная служба использует в своей работе (см. врезку «Вышестоящие и нижестоящие зависимости» в главе 1). Служба не должна делать различий между вспомогательными службами одного типа. Не имеет значения, будет ли это внутренняя служба, управляемая той же организацией, или удаленная служба, управляемая третьей стороной.

Каждая нижестоящая сторонняя служба должна рассматриваться просто как еще один ресурс, к которому можно обратиться с помощью настраиваемого URL-адреса или какого-либо другого дескриптора ресурса, как показано на рис. 6.3. Все ресурсы следует рассматривать как одинаково подверженные *заблуждениям распределенных вычислений* (см. главу 4).

Иначе говоря, к базе данных MySQL, управляемой системными администраторами вашей команды, следует относиться точно так же, как к экземпляру RDS, управляемому AWS. То же касается *любой* нижестоящей службы, где бы она ни работала – в центре обработки данных в другом полушарии или в контейнере Docker на том же сервере.

Службу, способную использовать любой другой ресурс того же типа – действующий внутри той же компании или вовне, – стоит лишь изменить значение конфигурации, проще развертывать в разных окружениях, проще тестировать и проще поддерживать.

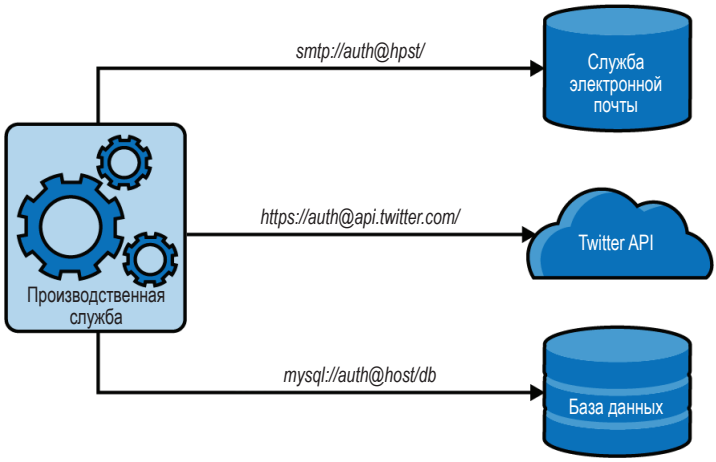


Рис. 6.3 ❖ Каждая нижестоящая сторонняя служба должна рассматриваться просто как еще один ресурс, доступный посредством настраиваемого URL-адреса или какого-либо другого дескриптора ресурса, и все ресурсы одинаково подвержены заблуждениям распределенных вычислений

V. Сборка, выпуск, выполнение

> Строго разделяйте стадии сборки и запуска.
>
> – The Twelve-Factor App

Каждое развертывание (не связанное с разработкой) – сочетание конкретной версии кода и конфигурации – должно быть неизменным и иметь уникальную метку. Также должна иметься возможность точного воссоздания развертывания, если (не дай бог) потребуется откатиться к более ранней версии.

Обычно развертывание выполняется в три этапа, показанных на рис. 6.4 и описанных ниже.

Этап сборки

На этапе сборки автоматизированный процесс извлекает определенную версию кода с его зависимостями и компилирует выполняемый артефакт, который мы называем *сборкой*. Каждая сборка всегда должна иметь уникальный идентификатор, обычно включающий отметку времени или увеличивающийся номер сборки.

Этап выпуска

На этапе выпуска конкретная сборка объединяется с конфигурацией, предназначенной для целевого развертывания. Полученный *выпуск* готов к немедленному запуску в среде выполнения. Как и сборки, выпуски тоже должны иметь уникальный идентификатор. Важно отметить, что при создании выпусков с одной и той же версией сборки повторная сборка кода не должна выполняться: для обеспечения постоянства окружения каждая конфигурация, зависящая от окружения, должна использовать одну и ту же сборку.

Этап запуска

На этапе запуска выпуск доставляется в среду развертывания и запускается.

В идеале новая сборка будет создаваться автоматически при развертывании нового кода.

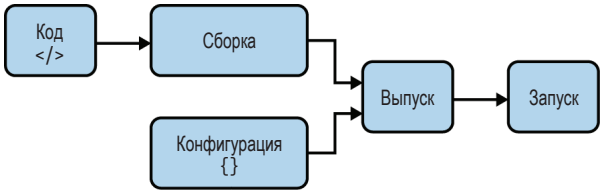


Рис. 6.4 ❖ Процесс развертывания кода в окружении (не для разработки) должен выполняться в три этапа: сборка, выпуск и запуск

VI. Процессы

> Запускайте приложение как один или несколько процессов, не сохраняющих внутреннего состояния (stateless).
>
> – The Twelve-Factor App

Процессы, составляющие службу, не должны иметь внутреннего или разделяемого состояния. Любые данные, которые необходимо сохранить, должны храниться в сторонней службе, например в базе данных или внешнем кеше.

Мы уже говорили о состоянии и поговорим еще в следующей главе, поэтому не будем углубляться в этот вопрос здесь.

Однако любопытствующие прямо сейчас могут прочитать раздел «С состоянием и без состояния» главы 7.

VII. Изоляция данных

> Каждая служба должна управлять только своими данными.
>
> – Cloud Native, *Data Isolation*

Каждая служба должна быть полностью *автономной*. То есть она должна управлять только своими данными и предоставлять доступ к ним только через API, предназначенный для этой цели. Если вам это кажется знакомым, хорошо! На самом деле это один из основных принципов микросервисов, который мы обсудим более подробно в разделе «Архитектура системы микросервисов» главы 7.

Очень часто такие службы реализуют API типа запрос/ответ, на манер RESTful API, или протокол RPC, принимающий запросы через определенный порт, но также могут принимать форму асинхронной службы, реагирующей на события и использующей шаблон обмена сообщениями типа публикация/подписка. Оба этих шаблона будут подробно описаны в главе 8.

Историческая справка

В действительности седьмой фактор в документе «Двенадцать факторов» называется «Привязка портов» (Port Binding), который гласит: «экспортируйте службы через привязку портов»[1].

В то время этот совет действительно имел смысл, но данный заголовок скрывает главную мысль: служба должна инкапсулировать и управлять своими собственными данными и предоставлять эти данные только через API.

Многие (или даже большинство) веб-приложения фактически предоставляют доступ к своим API через порты, однако растущая популярность технологии «функции как услуга» (Functions as a Service, FaaS) и архитектур, управляемых событиями, означает, что в настоящее время это не всегда так.

Поэтому вместо оригинального фактора я решил указать более актуальное (и более правильное) обобщение, предложенное Борисом Шоллем (Boris Scholl), Трентом Свенсоном (Trent Swanson) и Питером Яусовеком (Peter Jausovec) в их книге *Cloud Native: Using Containers, Functions, and Data to Build Next-Generation Applications* (O'Reilly; https://oreil.ly/uvvsa).

И наконец, хотя это нельзя увидеть в мире Go, некоторые языки и фреймворки позволяют внедрять серверы приложений в среду выполнения для создания веб-службы. Такая практика ограничивает возможности тестирования и переносимость, нарушая изоляцию данных и независимость от окружения, и *не рекомендуется* к использованию.

VIII. Масштабируемость

> Масштабируйте приложение, используя модель процессов.
>
> – The Twelve-Factor App

Службы должны поддерживать возможность масштабирования по горизонтали за счет добавления дополнительных экземпляров.

В этой книге мы довольно много говорим о масштабируемости. Ей даже посвящена вся седьмая глава. И это не случайно: важность масштабируемости невозможно переоценить.

Конечно, удобнее просто нарастить мощность одного сервера, на котором действует ваша служба, и это оправданно в (очень) краткосрочной перспективе, но в долгосрочной перспективе вертикальное масштабирование – проигрышная стратегия. Если вам повезет, то вы достигнете точки, когда вертикальное масштабирование окажется невозможным. Вероятнее всего, ваш отдельный сервер будет испытывать скачки нагрузки с более широким размахом, чем вы сможете масштабировать, или просто выйдет из строя[2]. Оба сценария ведут к недовольству пользователей. Мы еще вернемся к проблеме масштабируемости в главе 7.

---

[1] *Адам Уиггинс* (Wiggins, Adam). Привязка портов // Двенадцать факторов. 2011. https://12factor.net/ru/port-binding.

[2] Может быть, даже ночью.

IX. Живучесть

> Максимизируйте живучесть за счет быстрого запуска и корректного завершения работы.
>
> – The Twelve-Factor App

Облачные окружения непостоянны: виртуальные серверы имеют забавное свойство исчезать в самое неподходящее время. Службы должны учитывать это: экземпляры служб должны допускать возможность запуска и остановки – намеренно или нет – в любое время.

Службы должны стремиться запускаться максимально быстро, чтобы сократить время, необходимое для развертывания (или повторного развертывания), и обеспечить эластичное масштабирование. Go, не имеющий виртуальной машины и других значительных накладных расходов, особенно хорошо справляется с этим.

Контейнеры обеспечивают быстрый запуск и тоже очень полезны с этой точки зрения, но необходимо следить за тем, чтобы образы оставались небольшими для уменьшения накладных расходов на передачу данных, возникающих при каждом первоначальном развертывании нового образа. Это еще одна область, в которой Go показывает себя с самой лучшей стороны: его самодостаточные двоичные файлы, как правило, можно заключить в образы `SCRATCH` без дополнительной среды выполнения или других внешних зависимостей. Мы продемонстрировали это в предыдущей главе в разделе «Контейнеризация хранилища пар ключ/значение».

Службы также должны корректно завершаться при получении сигнала `SIGTERM`, сохраняя все данные, которые необходимо сохранить, закрывая все сетевые соединения и завершая любую незавершенную работу или возвращая текущее задание в рабочую очередь.

X. Сходство окружений разработки/эксплуатации

> Сохраняйте окружения разработки, эксплуатации и любые другие промежуточные окружения максимально похожими.
>
> – The Twelve-Factor App

Окружения разработки и эксплуатации должны иметь минимально возможные различия. В число этих различий, конечно, входят различия в коде, но не только:

Расхождения в коде
: Ветви разработки должны быть короткими и недолговечными, их следует тестировать и передавать в производство как можно быстрее. Это поможет свести к минимуму функциональные различия между окружениями и снизит риск откатов после неудачного развертывания.

Расхождения в стеках
: Желательно не использовать в окружениях разработки и эксплуатации разные компоненты (скажем, SQLite в OS X и MySQL в Linux), окружения

должны оставаться как можно более похожими. Легковесные контейнеры – отличный инструмент для этой цели. Они помогут свести к минимуму вероятность появления неудобных различий между почти одинаковыми реализациями, способных испортить вам настроение.

Расхождения между людьми
: Когда-то разделение труда между программистами, которые пишут код, и операторами, которые его развертывают, было обычным делом, но такое разделение порождало конфликты и контрпродуктивные отношения. Привлечение авторов кода к развертыванию и возложение на них ответственности за его поведение в производственном окружении помогает преодолеть разрозненность разработки и эксплуатации и стимулирует достижение стабильности и высокой скорости работы.

Все вместе эти подходы помогают сократить разрыв между разработкой и производством, что, в свою очередь, способствует быстрому, автоматизированному и непрерывному развертыванию.

XI. Журналирование

> Рассматривайте журналы как потоки событий.
>
> – The Twelve-Factor App

Журналы – нескончаемый поток сознания службы – невероятно полезны, особенно в распределенной среде. Обеспечивая видимость поведения работающего приложения, хороший журнал может существенно упростить задачу обнаружения и диагностики неправильного поведения.

Обычно службы журналируют события в файлы на локальном диске. Однако в облачных окружениях такое решение затруднит поиск ценной информации из-за неудобного доступа и невозможности агрегирования. В динамических окружениях, таких как Kubernetes, экземпляры служб (и файлы журналов) могут даже исчезнуть к тому моменту, когда вы решите их просмотреть.

По этим причинам облачная служба должна интерпретировать журналы как обычные потоки событий, отправляя информацию о каждом событии непосредственно в стандартный вывод. Она не должна заботиться о деталях реализации, таких как маршрутизация или сохранение событий, – исполнитель сам должен решить, что с этой информацией делать.

Такой подход кажется простым (и, возможно, несколько нелогичным), но дает большую свободу.

Во время разработки в локальной системе программист может наблюдать за потоком событий в терминале и оценивать поведение службы. В процессе эксплуатации выходной поток может перехватываться средой выполнения и направляться в одно или в несколько мест, например в систему индексации журналов, такую как Elasticsearch, Logstash и Kibana (ELK) или Splunk для просмотра и анализа, или в хранилище данных для долговременного хранения.

Подробнее о журналах и журналировании в контексте наблюдаемости мы поговорим в главе 11.

XII. Задачи администрирования

> Выполняйте задачи администрирования/управления с помощью разовых процессов.
>
> – The Twelve-Factor App

Из всех двенадцати факторов именно этот больше всего свидетельствует о возрасте документа. Во-первых, он явно рекомендует использовать командную оболочку для выполнения задач вручную.

Сразу оговорюсь, что *внесение изменений в экземпляр сервера вручную создает «снежинку»*. Это плохо. См. врезку «Серверы-снежинки» ниже.

Если ваше окружение дает возможность войти в командную оболочку, то вы должны учитывать, что она может (и в конечном итоге будет) быть уничтожена и воссоздана в любой момент.

Но давайте пока отложим в сторону этот аспект и рассмотрим первоначальный замысел: административные и управленческие задачи должны выполняться как разовые процессы. Это правило можно интерпретировать двояко, и каждая интерпретация требует своего подхода:

- если ваша задача является административным процессом, например выполняющим восстановление данных или миграцию базы данных, то он должен выполняться как короткоживущий процесс. Контейнеры и функции – отличные средства для таких целей;
- если вы запускаете процесс для обновления службы или среды выполнения, то вместо этого следует изменить сценарии сборки/настройки службы или среды соответственно.

Серверы-снежинки

Поддержание работоспособности серверов может быть сложной задачей. В 3 часа ночи, когда что-то пошло не так, вас будет преследовать неукротимое желание побыстрее внести изменения и вернуться в постель. Поздравляем, вы только что создали *сервер-снежинку*: экземпляр сервера с изменениями, придающими ему уникальное и обычно недокументированное поведение.

Даже незначительные и, казалось бы, безобидные изменения могут привести к серьезным проблемам. Даже если изменения задокументированы, что случается редко, такие серверы сложно воспроизвести в точности, особенно когда требуется синхронизировать весь кластер. Это может привести к неприятным последствиям, когда после повторного развертывания службы на новом оборудовании вдруг выяснится, что по непонятным причинам она не работает.

Более того, поскольку после таких изменений окружение тестирования перестанет соответствовать производственному окружению, вы потеряете уверенность, что ваша среда разработки надежно и точно воспроизводит производственное окружение.

По этой причине серверы и контейнеры следует рассматривать как нечто *неизменяемое*. Если что-то нужно обновить, исправить или изменить, то изменения должны вноситься путем обновления соответствующих сценариев сборки, выпекания[1] нового образа и подготовки новых экземпляров сервера или контейнера для замены старых.

Как говорится, с экземплярами следует обращаться как с «рогатым скотом, а не как с домашними любимцами».

Итоги

В этой главе мы рассмотрели вопрос «Что такое облачная система?». Типичный ответ – «компьютерная система, работающая в облаке». Но слово «работа» может означать все, что угодно. Конечно, хотелось бы иметь более конкретный ответ.

Поэтому мы обратились к таким мыслителям, как Тони Хоар (Tony Hoare) и Жан-Клод Лапри (Jean-Claude Laprie), которые сформулировали первую часть ответа: *надежность*. То есть, если перефразировать, облачными называют компьютерные системы, которые ведут себя ожидаемым для пользователей образом, несмотря на то что действуют в принципиально ненадежном окружении.

Очевидно, что говорить о характеристиках облачных систем проще, чем реализовать их, поэтому мы рассмотрели три способа достижения желаемого:

- академические «средства надежности» Лапри, которые включают предотвращение, терпимость, устранение и прогнозирование неисправностей;
- «Двенадцать факторов» Адама Уиггинса (Adam Wiggins) – документ, определяющий более требовательный (и немного устаревший, местами) подход;
- наши собственные «облачные атрибуты», основанные на определении Cloud Native Computing Foundation, представленном в главе 1, вокруг которых организована вся книга.

По сути, эта глава была лишь кратким обзором теории, и тем не менее здесь содержится много важной фундаментальной информации, описывающей мотивы и средства, используемые для достижения того, что мы называем «облачностью».

[1] Термин «выпекание» (backing) иногда используется для обозначения процесса создания нового образа контейнера или сервера.

Глава 7

Масштабируемость

> На самом деле лучшие программы пишутся на бумаге. Ввод их в компьютер – лишь техническая деталь[1].
>
> – Макс Канат-Александер (Max Kanat-Alexander),
> *Code Simplicity: The Fundamentals of Software*

Летом 2016 года я присоединился к небольшой компании, оцифровывавшей формы и различные документы, выпускавшиеся местными властями. Их основное приложение находилось в состоянии, типичном для стартапов на ранних стадиях развития, поэтому мы приступили к работе и уже к осени сумели поместить его в контейнер, сформировать инфраструктуру и полностью автоматизировать его развертывание.

Одним из наших клиентов был небольшой прибрежный город на юго-востоке Вирджинии, поэтому, когда ураган Мэтью – первый атлантический ураган 5-й категории, возникший за почти десятилетие, – обрушился на побережье недалеко оттуда, местные власти послушно объявили чрезвычайное положение и использовали нашу систему для создания необходимых документов, которые должны были заполняться гражданами, разместили ее в социальных сетях, и все полмиллиона человек вошли в систему одновременно.

Когда сработал сигнал тревоги, дежурный проверил параметры и обнаружил, что нагрузка на агрегированный процессор достигла 100 %, и сотни тысяч запросов были прерваны по тайм-ауту.

Мы добавили ноль в число желаемого количества серверов, включили в планы на будущее реализацию автоматического масштабирования и вернулись к работе. В течение 24 часов наплыв посетителей схлынул, и мы уменьшили количество действующих серверов.

Какой урок мы извлекли из случившегося, кроме преимуществ автоматического масштабирования[2]?

Во-первых, мы уяснили, что без возможности масштабирования наша система едва ли смогла бы обслужить такой наплыв пользователей. Но возможность добавлять ресурсы по требованию помогла справиться с нагрузкой, намного превышающей ту, которую мы ожидали. Как дополнительное пре-

---

[1] Kanat-Alexander, Max. *Code Simplicity: The Science of Software Design*. O'Reilly Media, 23 March 2012.

[2] Честно говоря, если бы у нас было автоматическое масштабирование, я бы, наверное, и не вспомнил, что это произошло.

имущество, если какой-то сервер выходил из строя, его работу можно было разделить между оставшимися.

Во-вторых, иметь гораздо больше ресурсов, чем необходимо, не только расточительно, но и дорого. Возможность уменьшить количество экземпляров при падении спроса помогает экономить и платить только за те ресурсы, которые действительно нужны. Большой плюс для стартапа с ограниченным бюджетом.

К сожалению, из-за того, что немасштабируемые службы могут казаться отлично функционирующими в начальных условиях, поддержка масштабирования не всегда рассматривается при проектировании. Имеющихся возможностей может быть вполне достаточно в краткосрочной перспективе, но в общем случае службы, неспособные масштабироваться дальше первоначальных ожиданий, имеют ограниченную ценность. Более того, реорганизовать службу для поддержки масштабируемости зачастую чрезвычайно сложно, поэтому их проектирование с учетом такой возможности может сэкономить время и деньги в долгосрочной перспективе.

Однако эта книга в первую очередь посвящена языку Go, по крайней мере в большей степени, чем инфраструктурным или архитектурным решениям. Да, мы будем обсуждать такие аспекты, как масштабируемая архитектура и шаблоны обмена сообщениями, но большая часть этой главы все же будет сосредоточена на демонстрации примеров использования Go для создания служб, опирающихся на другую (неинфраструктурную) часть уравнения масштабируемости: эффективность[1].

Что такое масштабируемость?

Как вы помните, идея масштабируемости впервые была представлена еще в главе 1, где она определялась как способность системы продолжать функционировать в условиях значительных колебаний спроса. Согласно этому определению, система может считаться масштабируемой, если ее не нужно перепроектировать для корректной работы во время резкого увеличения нагрузки.

Обратите внимание, что это определение[2] вообще ничего не говорит о добавлении физических ресурсов. Скорее, оно указывает на способность системы справляться с большими колебаниями спроса. Здесь «масштабируется» величина спроса. Добавление ресурсов является одним из вполне приемлемых способов достижения масштабируемости, но это не совсем то же самое,

---

[1] Если вы хотите узнать больше об облачной инфраструктуре и архитектуре, то я советую обратиться к книгам, посвященным этой тематике. В частности, я могу порекомендовать книгу Джастина Гаррисона (Justin Garrison) и Криса Новы (Kris Nova) *Cloud Native Infrastructure* и книгу Пини Резника (Pini Reznik), Джеми Добсона (Jamie Dobson) и Мишель Дженоу (Michelle Gienow) *Cloud Native Transformation* (обе выпущены издательством O'Reilly Media).

[2] Это мое собственное определение. Я признаю, что оно расходится с другими общепринятыми определениями.

что масштабируемость. Чтобы еще больше запутать вас, добавлю, что слово «масштабирование» также может применяться к системе, и в этом случае оно *действительно* означает изменение количества выделенных ресурсов.

Так как же справиться с высоким спросом, не добавляя ресурсов? Как будет рассказываться в разделе «Отложенное масштабирование: эффективность» ниже, системы, построенные с прицелом на *эффективность*, по своей сути более масштабируемы благодаря их способности изящно справляться с высоким спросом, без необходимости немедленно наращивать ресурсы в ответ на каждый всплеск спроса и занимать избыточные ресурсы «на всякий случай».

Различные формы масштабирования

К сожалению, даже самые передовые стратегии увеличения эффективности имеют предел, и рано или поздно вам придется заняться вопросом масштабирования службы, чтобы предоставить дополнительные ресурсы. Это можно сделать двумя способами (см. рис. 7.1), каждый из которых имеет свои плюсы и минусы.

Вертикальное масштабирование
: Систему можно *масштабировать по вертикали* путем выделения дополнительного объема ресурсов. В общедоступном облаке существующий сервер можно довольно легко масштабировать по вертикали, просто изменив размер экземпляра, но только если имеются более крупные типы экземпляров (и деньги на вашем счету для его оплаты).

Горизонтальное масштабирование
: Систему можно *масштабировать по горизонтали* путем запуска дополнительных экземпляров системы или службы, чтобы ограничить нагрузку на каждый конкретный сервер. Системы, использующие эту стратегию, часто способны справляться с самыми высокими нагрузками, но, как будет показано в разделе «С состоянием и без состояния» ниже, наличие состояния может сделать реализацию этой стратегии трудной или даже невозможной.

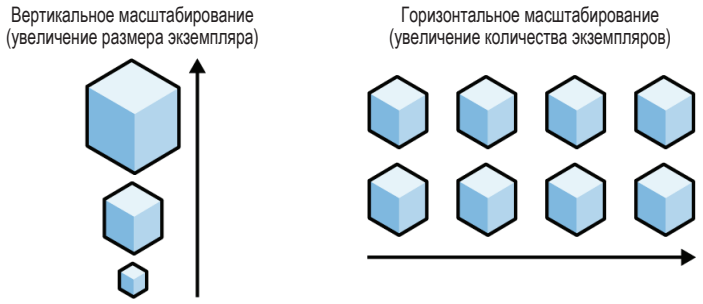


Рис. 7.1 ❖ Вертикальное масштабирование – эффективное решение в краткосрочной перспективе; горизонтальное масштабирование технически сложнее, но является более выигрышной стратегией в долгосрочной перспективе

Эти два термина часто используются для описания способов масштабирования: взять всю систему и просто сделать ее *больше*. Однако существует множество других стратегий масштабирования.

Наиболее распространенной из них является, пожалуй, *функциональная декомпозиция*, с которой вы, несомненно, уже знакомы, даже если не знали, что она так называется. Функциональная декомпозиция предполагает разделение сложных систем на множество более мелких функциональных блоков, которые можно независимо оптимизировать, использовать и масштабировать. Вы можете распознать в этой стратегии обобщение ряда передовых практик проектирования – от простых программ до обширных распределенных систем.

Другой подход, широко используемый в системах с большими объемами данных, особенно в базах данных, – это *сегментирование* (sharding). Системы, использующие эту стратегию, распределяют нагрузку, разделяя данные на разделы, называемые *сегментами* (shards), каждый из которых содержит определенную часть большего набора данных. Пример этой стратегии представлен в разделе «Минимизация простоев на блокировках с использованием сегментирования» ниже.

Четыре основных узких места

По мере увеличения требований к системе неизбежно наступает момент, когда какого-то одного ресурса оказывается недостаточно, и эта нехватка эффективно препятствует дальнейшим усилиям по масштабированию. Данный ресурс становится *узким местом*.

Чтобы вернуть систему в работоспособное состояние, требуется выявить и устранить узкие места. Это можно сделать в краткосрочной перспективе, увеличив объем недостающего ресурса, то есть прибегнуть к вертикальному масштабированию, например добавив больше памяти или установив более мощные процессоры. Как вы, возможно, помните из обсуждения в разделе «Различные формы масштабирования», это решение не всегда возможно (или нерентабельно), и на него нельзя полагаться вечно.

Однако часто узкое место можно устранить, уменьшив нагрузку на компонент за счет использования другого ресурса, которого в системе в избытке. База данных может избежать интенсивного дискового ввода/вывода за счет кеширования данных в ОЗУ; и наоборот, служба, потребляющая оперативную память в огромных количествах, может выгружать данные на диск. Горизонтальное масштабирование тоже не является панацеей: увеличение количества действующих экземпляров влечет за собой дополнительные накладные расходы на связь, что создает дополнительную нагрузку на сеть. Даже системы с высокой степенью конкуренции могут пасть жертвами нехватки ресурсов для выполнения внутренних операций, поскольку спрос на них возрастает, и в игру вступают такие явления, как конфликт блокировок. Эффективное использование ресурсов часто означает компромисс.

Конечно, устранение узких мест требует сначала идентифицировать ограничивающий компонент, и хотя существует множество различных ресурсов, которые могут выступать в качестве целей для масштабирования – будь то фактическое масштабирование ресурса или его более эффективное использование, – такие усилия, как правило, сосредоточены всего на четырех ресурсах:

Процессор
: Количество операций в единицу времени, которые может выполнить центральный процессор системы, является узким местом для многих систем. Стратегии масштабирования вычислительной мощности включают кеширование результатов дорогостоящих детерминированных операций (ценой увеличения потребления памяти) или просто увеличение мощности или количества процессоров (ценой увеличения объема сетевого ввода/вывода).

Память
: Объем данных, которые можно сохранить в основной памяти. Современные системы могут хранить невероятные объемы данных, порядка десятков и даже сотен гигабайт, но даже этой емкости может не хватить, особенно если система обрабатывает гигантские объемы данных, хранимых в памяти, чтобы преодолеть ограничения скорости дискового ввода/вывода. В числе стратегий масштабирования, используемых в этом случае, можно назвать выгрузку данных из памяти на диск (ценой увеличения объемов дискового ввода/вывода) или во внешний выделенный кеш (ценой увеличения объема сетевого ввода/вывода) либо просто увеличение объема доступной памяти.

Дисковый ввод/вывод
: Скорость, с которой данные могут читаться и записываться на жесткий диск или другой носитель данных. Дисковый ввод/вывод – типичное узкое место в системах с большим уровнем параллелизма, которые интенсивно читают и записывают на диск, например в базах данных. В числе стратегий масштабирования, используемых в этом случае, можно назвать кеширование данных в ОЗУ (ценой увеличения потребления памяти) или использование внешнего выделенного кеша (ценой увеличения объема сетевого ввода/вывода).

Сетевой ввод/вывод
: Скорость, с какой данные могут передаваться по сети, либо из определенной точки, либо в совокупности. Сетевой ввод/вывод напрямую зависит от *объема данных*, который можно передать по сети в единицу времени. Стратегии масштабирования сетевого ввода/вывода часто ограничены[1], зато сетевой ввод/вывод хорошо поддается различным стратегиям оптимизации, которые мы вскоре обсудим.

---

[1] Некоторые поставщики услуг облачных вычислений устанавливают более низкие ограничения на сетевой ввод/вывод для небольших экземпляров. В иных случаях увеличение размера экземпляра может увеличить эти ограничения.

По мере роста нагрузки на систему она почти наверняка будет испытывать нехватку хотя бы одного из этих ресурсов. И несмотря на наличие стратегий повышения эффективности, которые можно применять, рост эффективности, как правило, происходит за счет более интенсивного использования одного или нескольких других ресурсов, поэтому рано или поздно вы обнаружите, что ваша система снова оказалась в узком месте из-за нехватки другого ресурса.

С состоянием и без состояния

Мы кратко коснулись отсутствия состояния во врезке «Состояние приложения и ресурса» главы 5, где описали состояние приложения – данных на стороне сервера, определяющих состояние приложения или используемых клиентом, – как нечто, чего желательно избегать, если это возможно. А сейчас давайте более внимательно разберем, что такое состояние, почему оно может вызывать проблемы и что можно сделать, чтобы их избежать.

Как оказывается, понятие «состояние» на удивление сложно определить, поэтому я постараюсь сделать все самостоятельно. Для целей этой книги я определю состояние как набор переменных приложения, которые в случае изменения влияют на его поведение[1].

Состояние приложения и состояние ресурса

Большинство приложений имеют состояние в той или иной форме, но не все состояния одинаковы. Они бывают двух видов, один из которых гораздо менее желателен, чем другой.

Первое – это *состояние приложения*, которое возникает всякий раз, когда приложению требуется запомнить событие локально. Когда кто-то говорит, что приложение *поддерживает состояние*, то обычно речь идет о такого рода локальном состоянии. Слово «локальное» здесь ключевое.

Второе – это *состояние ресурса*, которое одинаково для всех клиентов и никак не связано с действиями клиентов, то есть это не данные, хранящиеся во внешнем хранилище или управляемые с помощью конфигурации. Это может показаться странным, но утверждение о том, что приложение *не имеет состояния*, не означает, что у приложения нет никаких данных; просто оно спроектировано так, что не содержит постоянно хранимых локальных данных. Его единственное состояние – это состояние ресурса, часто потому, что все состояние приложения хранится в каком-то внешнем хранилище данных.

Чтобы понять разницу между этими двумя видами состояний, представьте службу, которая хранит информацию о сеансах клиентов, связывая ее с некоторым прикладным контекстом. Если данные сеансов хранятся приложением локально, то они будут считаться «состоянием приложения». Но если данные

[1] Если у вас есть определение получше, пришлите его мне. Я уже думаю о втором издании.

хранятся во внешней базе данных, то их можно рассматривать данными удаленного ресурса и, соответственно, считать «состоянием ресурса».

Состояние приложения – нечто вроде «препятствия масштабируемости». Несколько одновременно выполняющихся экземпляров службы с состоянием быстро обнаружат, что их индивидуальные состояния расходятся из-за разных входных данных. Привязка к серверу помогает обойти это конкретное условие, гарантируя, что каждый клиентский запрос будет направляться одному и тому же серверу, но эта стратегия представляет значительный риск для данных, потому что отказ любого отдельного сервера может привести к потере данных.

Преимущества отсутствия состояния

До сих пор мы обсуждали различия между состоянием приложения и состоянием ресурса и даже предположили – без особых доказательств (пока), – что состояние приложения – это нечто плохое. Отсутствие состояния, напротив, дает некоторые очень заметные преимущества:

Масштабируемость
: Наиболее очевидным и часто упоминаемым преимуществом является способность приложения без состояния обрабатывать каждый запрос или взаимодействие независимо от предыдущих запросов либо взаимодействий. Это означает, что любой экземпляр службы может обрабатывать любой запрос, что позволяет масштабировать приложение в любую сторону или перезапускать без потери данных, необходимых для обработки любых сеансов или запросов. Это особенно важное свойство для автоматического масштабирования службы, потому что экземпляры, узлы или поды (группы контейнеров), на которых размещается служба, могут (и обычно будут) создаваться и уничтожаться без предупреждения.

Долговечность
: Данные, хранящиеся строго в одном месте (например, в одном экземпляре службы), могут быть (и в какой-то момент будут) потеряны, когда этот экземпляр исчезнет по любой причине. Помните: все, что есть в «облаке», со временем испаряется.

Простота
: Службы, не имеющие состояния приложения, освобождаются от необходимости… эм-м… управлять своим состоянием[1]. Отсутствие необходимости синхронизировать состояние, следить за его целостностью и при необходимости восстанавливать[2] делает API менее сложным и, следовательно, более простым в проектировании, создании и сопровождении.

---

[1] Я знаю, что слишком часто употребляю слово «состояние». Но поверьте, писать книги – сложно.

[2] Вспомните наше обсуждение идемпотентности.

Возможность кеширования
API, предлагаемые службами без состояния, относительно легко адаптировать для поддержки кеширования. Если служба знает, что в ответ на определенный запрос всегда будет возвращаться один и тот же результат, независимо от того, кто и когда его посылает, то этот результат можно сохранить в кеше и в будущем извлекать его оттуда, что поможет повысить эффективность и сократить время ответа.

Эти преимущества могут показаться совершенно разными, и все же они частично совпадают в том, что они предоставляют. В частности, отсутствие состояния делает службы проще и безопаснее для сборки, развертывания и обслуживания.

Отложенное масштабирование: эффективность

В контексте облачных вычислений мы обычно рассматриваем масштабируемость с точки зрения способности системы добавлять сетевые и вычислительные ресурсы. При этом обычно игнорируется роль *эффективности* в масштабируемости. В частности, способность системы справляться с возросшей нагрузкой *без* добавления дополнительных ресурсов.

Можно даже утверждать, что большинство программистов начинают заботиться об эффективности программы только с увеличением спроса на услугу. Если язык отличается относительно высокими накладными расходами на конкуренцию, что особенно характерно для языков с динамической типизацией, то приложения на нем будут потреблять доступную память или вычислительные ресурсы намного быстрее, чем приложения на менее дорогостоящем в этом отношении языке, и, следовательно, потребуется больше ресурсов и больше событий масштабирования, чтобы справиться с той же нагрузкой.

Это было одним из основных соображений при разработке модели конкуренции в языке Go, сопрограммы которого являются не потоками выполнения, а легковесными процедурами, мультиплексируемыми с использованием нескольких потоков операционной системы. Запуск каждой сопрограммы обходится чуть дороже, чем простое выделение места в стеке, что потенциально позволяет создавать миллионы одновременно выполняющихся сопрограмм.

Далее в этом разделе мы рассмотрим набор средств и инструментов Go, которые позволяют избежать распространенных проблем масштабирования, таких как утечки памяти и конфликты блокировок, а также выявлять и исправлять их.

Эффективное кеширование с использованием кеша LRU

Кеширование в памяти – очень гибкая стратегия повышения эффективности, помогающая уменьшить нагрузку на что угодно, от процессора до дискового

или сетевого ввода/вывода, и даже просто уменьшить задержки, связанные с удаленными или иными медленными операциями.

Идея кеширования выглядит простой. У вас есть что-то, что вы хотите запомнить, например результат дорогостоящих (но детерминированных) вычислений, и вы помещаете это что-то в ассоциативный массив для использования в дальнейшем. Верно?

Что ж, кеширование можно реализовать и так, но вскоре вы начнете сталкиваться с проблемами. Что происходит при увеличении количества ядер в процессоре и сопрограмм? Поскольку вы не учитывали возможность конкурентного использования кеша, то вскоре обнаружите, что операции изменения накладываются друг на друга и это приводит к некоторым неприятным последствиям. Кроме того, поскольку вы забыли предусмотреть удаление из кеша ставших ненужными значений, он будет продолжать расти, пока не займет всю память.

Итак, нам нужен кеш, который:

- поддерживает конкурентные операции чтения, записи и удаления;
- хорошо масштабируется, так как количество ядер и сопрограмм увеличивается;
- не будет неограниченно расти и не исчерпает всю доступную память.

Одним из распространенных решений этой задачи является кеш LRU (Least Recently Used – наиболее давно использовавшийся) – замечательная структура данных, которая помнит, как давно «использовался» (читался или записывался) каждый ее ключ. Когда новое значение добавляется в кеш, объем которого достиг ранее заданного порога, то кеш может «выселить» (удалить) наиболее давно использовавшийся элемент.

Подробное обсуждение реализации кеша LRU выходит за рамки этой книги, но я скажу, что она довольно остроумная. Как показано на рис. 7.2, кеш LRU содержит двусвязный список (в котором фактически хранятся значения) и ассоциативный массив, связывающий каждый ключ с узлом в списке. Каждый раз, когда происходит обращение к ключу (то есть выполняется операция чтения или записи), соответствующий узел перемещается в конец списка, соответственно наиболее давно использовавшийся узел всегда находится в начале.

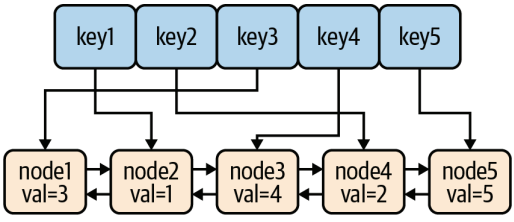


Рис. 7.2 ❖ Кеш LRU содержит ассоциативный массив и двусвязный список. Такая организация позволяет удалять наиболее устаревшие элементы при достижении максимального объема кеша

В Go доступна пара реализаций кеша LRU, но не в базовых библиотеках (пока). Наиболее, пожалуй, популярная находится в библиотеке golang/group-

cache (https://oreil.ly/Q5pzk). Однако я предпочитаю расширение HashiCorp с открытым исходным кодом для `groupcache`, `hashicorp/golang-lru` (https://oreil.ly/25ESk), которое имеет подробную документацию и поддерживает `sync.RWMutexes` для обеспечения безопасности в конкурентном окружении.

Библиотека HashiCorp содержит две функции конструкторов, каждая из которых возвращает указатель типа `*Cache` и ошибку `error`:

```
// Функция New создает кеш LRU с заданной емкостью.
func New(size int) (*Cache, error)

// Функция NewWithEvict создает кеш LRU с заданной емкостью, а также принимает
// ссылку на функцию-обработчик "удаления элемента из кеша", которая
// вызывается, когда кеш удаляет устаревший элемент.
func NewWithEvict(size int,
    onEvicted func(key interface{}, value interface{})) (*Cache, error)
```

Структура `*Cache` имеет несколько присоединенных методов. Вот некоторые наиболее полезные из них:

```
// Функция Add добавляет значение в кеш и возвращает true, если
// в процессе добавления пришлось удалить какой-то устаревший элемент.
func (c *Cache) Add(key, value interface{}) (evicted bool)

// Проверяет наличие указанного ключа key в кеше
// (не влияет на устаревание элемента).
func (c *Cache) Contains(key interface{}) bool

// Функция Get отыскивает ключ key и, если он существует,
// возвращает (value, true). Если искомый ключ отсутствует в кеше,
// то возвращается (nil, false).
func (c *Cache) Get(key interface{}) (value interface{}, ok bool)

// Функция Len возвращает количество элементов в кеше.
func (c *Cache) Len() int

// Функция Remove удаляет указанный ключ key из кеша.
func (c *Cache) Remove(key interface{}) (present bool)
```

Полный список методов вы найдете в GoDocs (https://oreil.ly/ODcff).

В следующем примере создается и используется кеш LRU с емкостью в два элемента. Чтобы наглядно показать, когда происходит выселение элементов из кеша, была добавлена функция обратного вызова, которая выводит на экран некоторую информацию в этот момент. Обратите внимание, что в примере переменная кеша инициализируется в функции `init` – специальной функции, которая автоматически вызывается перед функцией `main` и после того, как переменные получат свои значения от своих инициализаторов:

```
package main

import (
    "fmt"
    lru "github.com/hashicorp/golang-lru"
```

```
)

var cache *lru.Cache

func init() {
    cache, _ = lru.NewWithEvict(2,
        func(key interface{}, value interface{}) {
            fmt.Printf("Evicted: key=%v value=%v\n", key, value)
        },
    )
}

func main() {
    cache.Add(1, "a")         // добавит ключ 1
    cache.Add(2, "b")         // добавит ключ 2; теперь кеш заполнен полностью

    fmt.Println(cache.Get(1)) // "a true"; теперь 1 -- наименее давно
                              // использовавшееся значение

    cache.Add(3, "c")         // добавит ключ 3, ключ 2 будет выселен

    fmt.Println(cache.Get(2)) // "<nil> false" (не найдено)
}
```

В этом примере создается кеш с емкостью в два элемента, то есть добавление третьего значения приведет к удалению наиболее давно использовавшегося.

После добавления в кеш элементов `{1: "a"}` и `{2: "b"}` вызывается `cache.Get(1)`, в результате чего элемент `{1: "a"}` становится менее устаревшим, чем `{2: "b"}`. Поэтому при добавлении нового элемента `{3: "c"}` из кеша удаляется элемент `{2: "b"}`, и последующий вызов `cache.Get(2)` возвращает элемент со значением `nil`.

Запустите эту программу, чтобы увидеть, как она действует. Она должна вывести:

```
$ go run lru.go
a true
Evicted: key=2 value=b
<nil> false
```

Кеш LRU – отличная структура данных для использования в роли глобального кеша в большинстве случаев, но у него есть одно ограничение: при очень высоком уровне конкуренции – порядка нескольких миллионов операций в секунду – начинают возникать задержки из-за конкуренции.

К сожалению, на момент написания этих строк в Go все еще не было реализации механизма кеширования с очень высокой пропускной способностью[1].

---

[1] Если вы хотите узнать больше о высокопроизводительном кешировании в Go, то я советую обратить внимание на отличную публикацию Маниша Рай Джайна (Manish Rai Jain) с названием «The State of Caching in Go» в блоге Dgraph (https://dgraph.io/blog/post/caching-in-go/).

Эффективная синхронизация

Как гласит афоризм Go: «Не общайтесь, разделяя память. Разделяйте память, общаясь». Иначе говоря, для взаимодействий вместо разделяемых структур данных лучше использовать каналы.

Это довольно мощная концепция. В конце концов, примитивы конкуренции в Go – сопрограммы и каналы – обеспечивают мощный и выразительный механизм синхронизации, и использование каналов в сопрограммах для обмена ссылками на структуры данных часто позволяет полностью отказаться от блокировок.

(Если вы подзабыли детали, касающиеся каналов и сопрограмм, то вернитесь к разделу «Сопрограммы» главы 3. Я подожду.)

Тем не менее Go предоставляет более традиционные механизмы блокировки в виде пакета `sync`. Но если каналы настолько хороши, то зачем использовать что-то вроде `sync.Mutex` и когда правильнее его использовать?

Как оказывается, каналы *чрезвычайно* полезны, но они не являются панацеей. Каналы особенно хороши при работе со множеством дискретных значений и лучше всего подходят для передачи прав собственности на данные, распределения единиц работы или асинхронной отправки результатов. Мьютексы же идеально подходят для синхронизации доступа к кешам и другим большим структурам данных, имеющим свое состояние.

В конце концов, ни один инструмент не может решить всех проблем. В конечном счете лучший вариант – использовать то, что окажется наиболее выразительным и/или самым простым.

Разделяйте память, общаясь

Организовать многопоточное выполнение просто, но правильно использовать блокировки – сложно.

В этом разделе рассмотрим классический пример, первоначально представленный в статье Эндрю Герранда (Andrew Gerrand) «Share Memory By Communicating» в блоге *Go Blog*[1], чтобы продемонстрировать эту истину и показать, как каналы Go могут сделать конкуренцию более безопасной и простой для понимания.

Представьте гипотетическую программу, которая опрашивает серверы в списке адресов URL, отправляя каждому запрос `GET` и ожидая ответа. Загвоздка в том, что на ожидание ответа на каждый запрос может потребоваться довольно много времени: от миллисекунд до секунд (или больше), в зависимости от особенностей удаленной службы. Определенно эта операция может выиграть, если реализовать ее на конкурентный манер, не так ли?

В традиционной многопоточной среде, использующей блокировки для синхронизации, данные можно структурировать примерно так:

---

[1] Gerrand, Andrew. «Share Memory By Communicating». *The Go Blog*, 13 July 2010. https://oreil.ly/GTURp. Этот раздел в значительной степени опирается на работу, созданную в Google (https://oreil.ly/D8ntT), которая используется в соответствии с условиями лицензии Creative Commons 4.0 Attribution License.

```
type Resource struct {
    url        string
    polling    bool
    lastPolled int64
}

type Resources struct {
    data []*Resource
    lock *sync.Mutex
}
```

Как видите, вместо среза строк с адресами URL здесь используются две структуры – `Resource` и `Resources`, – каждая из которых снабжена рядом структур синхронизации, в дополнение к строкам URL, которые нам действительно нужны.

Для организации опроса в традиционной многопоточной программе может использоваться функция `Poller`, как показано ниже, выполняющаяся в нескольких потоках:

```
func Poller(res *Resources) {
    for {
        // Получить ресурс Resource, наиболее давно опрашивавшийся,
        // и поставить признак, что к нему был отправлен запрос
        res.lock.Lock()

        var r *Resource

        for _, v := range res.data {
            if v.polling {
                continue
            }
            if r == nil || v.lastPolled < r.lastPolled {
                r = v
            }
        }

        if r != nil {
            r.polling = true
        }

        res.lock.Unlock()

        if r == nil {
            continue
        }

        // Послать запрос по URL

        // Обновить флаг polling ресурса и записать
        // время отправки запроса в lastPolled
        res.lock.Lock()
        r.polling = false
        r.lastPolled = time.Nanoseconds()
```

```
        res.lock.Unlock()
    }
}
```

Эта реализация работает, но ее можно улучшить. Это длинный код, его сложно читать, о нем трудно рассуждать, и это еще притом, что в него не включена ни логика опроса URL, ни логика обработки ситуации исчерпания пула ресурсов.

Теперь давайте рассмотрим аналогичную функцию, реализованную с использованием каналов Go. В следующем примере структура `Resource` сократилась до своего основного компонента (строки URL), а функция `Poller` получает экземпляр `Resource` из входного канала и отправляет его в выходной канал, когда работа с ним будет завершена:

```
type Resource string

func Poller(in, out chan *Resource) {
    for r := range in {
        // Послать запрос по URL

        // Отправить обработанный экземпляр Resource в выходной канал
        out <- r
    }
}
```

Это так... просто. Мы полностью избавились от блокировок в `Poller` и от служебных данных в структуре `Resource`. Фактически у нас осталось только все самое важное.

А что, если нам потребуется запустить несколько процессов, вызывающих `Poller`? Разве не это мы пытаемся сделать? Ответ потрясающе прост: используйте сопрограммы. Взгляните на следующий код:

```
for i := 0; i < numPollers; i++ {
    go Poller(in, out)
}
```

Запустив `numPollers` сопрограмм, мы создали `numPollers`, процессов, выполняющихся конкурентно, каждый из которых использует одни и те же каналы.

В предыдущих примерах многое было опущено, чтобы не потерялись наиболее важные моменты. Полный идиоматический код программы на Go, в которой используются эти идеи, вы найдете в Codewalk, в статье «Share Memory By Communicating» (https://oreil.ly/HF1Ay).

Уменьшение простоев на блокировках с помощью буферизованных каналов

В какой-то момент, читая эту главу, вы могли подумать: «Конечно, каналы – это здорово, но операция записи в каналы все еще блокирует вызывающий процесс». В конце концов, каждая операция отправки в канал блокируется

до тех пор, пока с другой стороны не появится процесс, готовый принять отправляемые данные, верно? В большинстве случаев это действительно так. По крайней мере, при использовании небуферизованных каналов.

Однако, как рассказывалось в разделе «Буферизация каналов» главы 3, можно создавать каналы, имеющие внутренний буфер сообщений. Операции отправки в такие буферизованные каналы блокируются, только когда буфер заполнен, а операции чтения из канала – только когда буфер пуст.

Напомню, что буферизованный канал можно создать, передав в функцию `make` дополнительный параметр `capacity`, определяющий размер буфера:

```
ch := make(chan type, capacity)
```

Буферизованные каналы особенно удобны для обработки неравномерных нагрузок. Мы уже использовали эту стратегию в главе 5, когда инициализировали `FileTransactionLogger`. Если проанализировать логику, по крупицам разбросанную в той главе, то получим следующее:

```
type FileTransactionLogger struct {
    events       chan<- Event // Канал только для записи; для передачи событий
    lastSequence uint64       // Последний использованный порядковый номер
}

func (l *FileTransactionLogger) WritePut(key, value string) {
    l.events <- Event{EventType: EventPut, Key: key, Value: value}
}

func (l *FileTransactionLogger) Run() {
    l.events = make(chan Event, 16) // Создать канал событий
    go func() {
        for e := range events {     // Извлечь следующее событие Event
            l.lastSequence++        // Увеличить порядковый номер
        }
    }()
}
```

В этом фрагменте у нас есть функция `WritePut`, которую можно вызвать для отправки в канал событий `events` сообщения, полученного в цикле `for` внутри сопрограммы, созданной в функции `Run`. Если бы `events` был стандартным каналом, то каждая операция отправки блокировалась бы до тех пор, пока анонимная сопрограмма не выполнит операцию приема. В большинстве случаев это нормально, но если операции записи будут следовать чаще, чем сопрограмма будет успевать читать, то вышестоящий клиент будет блокироваться.

Используя буферизованный канал, мы дали возможность обрабатывать небольшие пакеты запросов (до 16), следующих друг за другом. Однако здесь важно отметить, что 17-я операция записи *будет* заблокирована.

Также важно учитывать, что использование буферизованных каналов создает риск потери данных, если программа завершится до того, как какие-либо сопрограммы смогут очистить буфер.

Уменьшение простоев на блокировках с помощью сегментирования

Какими бы прекрасными ни были каналы, но, как мы отметили в разделе «Эффективная синхронизация» выше, они не решают *всех* проблем. Типичным примером может служить большая центральная структура данных, такая как кеш, которую нелегко разложить на отдельные единицы работы[1].

Когда разделяемые структуры данных одновременно используются несколькими процессами, принято использовать механизм блокировки, например мьютексы, предлагаемые пакетом `sync`, как было показано в разделе «Добавление в структуру данных поддержки использования в конкурентном окружении» главы 5. Так мы могли бы создать структуру с ассоциативным массивом и встроенным `sync.RWMutex`:

```
var cache = struct {
    sync.RWMutex
    data map[string]string
}{data: make(map[string]string)}
```

Когда процедуре понадобится выполнить запись в кеш, она установит блокировку вызовом `cache.Lock` и затем, выполнив запись, снимет ее вызовом `cache.Unlock`. Операцию записи можно было бы даже оформить в виде отдельной функции:

```
func ThreadSafeWrite(key, value string) {
    cache.Lock()           // Установить блокировку записи
    cache.data[key] = value
    cache.Unlock()         // Снять блокировку записи
}
```

По замыслу, этот прием должен разрешать запись только одной процедуре, владеющей блокировкой. Он отлично работает. Однако, как мы обсуждали в главе 4, с увеличением числа конкурентных процессов, работающих с этими же данными, увеличивается и среднее время, которое процессы тратят на ожидание снятия блокировок. Возможно, вы помните название этого неприятного явления: конфликт блокировок.

В некоторых случаях эту проблему можно решить, масштабируя число экземпляров, но масштабирование увеличивает сложность и задержку из-за необходимости использовать распределенные блокировки для обеспечения согласованности. Альтернативной стратегией уменьшения конфликтов блокировок, защищающих доступ к общим структурам данных в экземпляре службы, является *вертикальное сегментирование*, когда большая структура данных делится на две или более структур, каждая из которых представля-

---

[1] Мы смогли бы решить проблему взаимодействий с кешем, используя каналы, но нам едва ли удалось бы сделать это решение более простым, чем с применением блокировки.

ет часть целого. При использовании этой стратегии блокировать требуется только часть общей структуры, что снижает общую конкуренцию за блокировки.

Вопросы вертикального сегментирования мы подробно обсуждали в разделе «Сегментирование» главы 4. Если вы хотите освежить в памяти теорию или реализацию вертикального сегментирования, то можете вернуться к тому разделу и вновь прочитать его.

Утечки памяти могут вызвать... фатальную ошибку исчерпания памяти во время выполнения

Утечки памяти – это класс ошибок, при которых память не освобождается, когда надобность в ней отпадает. Эти ошибки могут быть довольно незаметными и особенно характерны для таких языков, как C++, в которых управление памятью осуществляется вручную. Автоматический механизм сборки мусора, безусловно, помогает, освобождая память, занятую объектами, которые больше не используются программой, но даже языки со сборкой мусора, такие как Go, не защищены от утечек памяти. Структуры данных все еще могут расти неограниченно, сопрограммы могут накапливаться, если их вовремя не останавливать, и даже таймеры `time.Ticker` могут ускользнуть от вас.

В этом разделе мы рассмотрим несколько распространенных причин утечек памяти, характерных для Go, и способы их устранения.

Утечки сопрограмм

У меня нет фактических цифр по этому вопросу[1], но, основываясь исключительно на своем личном опыте, я склонен подозревать, что сопрограммы являются самым большим источником утечек памяти в Go.

Каждый раз, когда запускается сопрограмма, ей выделяется небольшой фрагмент памяти для стека – 2048 байт, – который может динамически увеличиваться или уменьшаться по мере выполнения в соответствии с потребностями процесса. Точный максимальный размер стека зависит от многих факторов[2], но по сути он зависит от объема доступной физической памяти.

Обычно, когда сопрограмма завершается, ее стек либо освобождается, либо откладывается для повторного использования[3]. Однако не все сопрограммы завершаются. Например:

```
func leaky() {
    ch := make(chan string)
```

---

[1] Если у вас они есть, сообщите мне!

[2] Дэйв Чейни (Dave Cheney) написал отличную статью на эту тему под названием «Why is a Goroutine's stack infinite?» (https://oreil.ly/PUCLF), которую я рекомендую прочесть всем, кого интересует динамика выделения памяти сопрограммам.

[3] Есть очень хорошая статья Винсента Бланшона (Vincent Blanchon) на тему утилизации сопрограмм под названием «How Does Go Recycle Goroutines?» (https://oreil.ly/GnoV2).

```
    go func() {
        s := <-ch
        fmt.Println("Message:", s)
    }()
}
```

В этом примере функция `leaky` создает канал и запускает сопрограмму, которая читает данные из этого канала. Функция `leaky` завершается без ошибок, но если присмотреться, то можно заметить, что в канал `ch` вообще ничего не отправляется, поэтому сопрограмма никогда не завершится, и ее стек никогда не будет освобожден. Есть и еще один побочный эффект: поскольку сопрограмма ссылается на `ch`, этот экземпляр не будет удален сборщиком мусора.

Это самая настоящая утечка памяти. Если такая функция вызывается регулярно, то общий объем потребляемой памяти будет постепенно увеличиваться до полного исчерпания.

Конечно, это искусственный пример, но есть веские причины, по которым программист может создать долго работающие сопрограммы, поэтому часто сложно понять, был ли такой процесс создан намеренно или это случилось по ошибке.

И что нам с этим делать? Дэйв Чейни (Dave Cheney) предлагает отличный совет: «никогда не запускайте сопрограмму, не зная, как она остановится… Каждый раз, используя ключевое слово `go` для запуска сопрограммы, вы должны знать, как и когда эта сопрограмма завершится. Если вы не знаете ответов на эти вопросы, то такая сопрограмма становится потенциальной утечкой памяти»[1].

Этот совет может показаться очевидным и даже банальным, но он невероятно важен. Слишком легко написать функцию, допускающую утечку сопрограмм, и эти утечки порой очень сложно выявить.

Вечно тикающие таймеры

Очень часто в программах бывает нужно отмерять время, чтобы выполнить некоторый код в определенный момент в будущем или через определенные интервалы времени.

Пакет `time` предоставляет два инструмента для этой цели: `time.Timer`, срабатывающий в определенный момент в будущем, и `time.Ticker`, срабатывающий многократно через заданные интервалы.

Однако если `time.Timer` имеет конечный срок использования, то `time.Ticker` не имеет такого ограничения. `time.Ticker` может тикать вечно. Возможно, вы уже поняли, к чему я клоню.

И `time.Timer`, и `time.Ticker` используют схожие механизмы: оба предоставляют канал, в который отправляется значение при каждом срабатывании. Вот пример использования обоих объектов:

```
func timely() {
    timer := time.NewTimer(5 * time.Second)
```

[1] Cheney, Dave. «Never Start a Goroutine without Knowing How It Will Stop». *dave.cheney.net*, 22 Dec. 2016. https://oreil.ly/VUlrY.

```
    ticker := time.NewTicker(1 * time.Second)

    done := make(chan bool)

    go func() {
        for {
            select {
            case <-ticker.C:
                fmt.Println("Tick!")
            case <-done:
                return
            }
        }
    }()

    <-timer.C
    fmt.Println("It's time!")
    close(done)
}
```

Функция `timely` запускает сопрограмму, которая повторяет итерации цикла снова и снова через равные промежутки времени, получая сигналы от `ticker`, поступающие каждую секунду, или из канала `done`, который завершает сопрограмму. Инструкция `<-timer.C` блокируется до тех пор, пока не сработает 5-секундный таймер, вызывающий срабатывание условия `case <-done`, по которому сопрограмма завершает цикл.

Функция `timely` завершается без ошибок, как и ожидалось, и сопрограмма как будто имеет жестко установленное условие завершения, поэтому вам простительно, если вы подумали, что все в порядке. Здесь кроется одна хитрая ошибка: экземпляры `time.Ticker` содержат активную сопрограмму, которую нельзя остановить напрямую. Поскольку в этом примере таймер нигде не останавливается, `timely` допускает утечку памяти.

Решение: всегда останавливайте таймеры. Для этой цели отлично подходит `defer`:

```
func timelyFixed() {
    timer := time.NewTimer(5 * time.Second)
    ticker := time.NewTicker(1 * time.Second)
    defer ticker.Stop()     // Гарантировать остановку ticker!

    done := make(chan bool)

    go func() {
        for {
            select {
            case <-ticker.C:
                fmt.Println("Tick!")
            case <-done:
                return
            }
        }
    }()
```

```
    <-timer.C
    fmt.Println("It's time!")
    close(done)
}
```

Вызов `ticker.Stop()` останавливает таймер `Ticker`, что позволяет сборщику мусора подобрать его и предотвратить утечку памяти.

В заключение об эффективности

В этом разделе мы рассмотрели несколько распространенных методов повышения эффективности программ: от использования кеша LRU вместо ассоциативного массива для ограничения объема потребляемой памяти и до приемов эффективной синхронизации процессов и предотвращения утечек памяти. Может показаться, что эти разделы имеют мало общего, но все они важны для создания масштабируемых программ.

Конечно, есть бесчисленное множество других методов, о которых я тоже хотел бы рассказать, но, к сожалению, время и пространство накладывают фундаментальные ограничения.

В следующем разделе мы вновь поменяем тему и рассмотрим некоторые распространенные архитектуры служб и их влияние на масштабируемость. Они в меньшей степени ориентированы на Go, но имеют решающее значение для исследования масштабируемости, особенно в контексте облачных вычислений.

Архитектуры служб

Идея *микросервисов* впервые появилась в начале 2010-х годов как уточнение и упрощение более ранней архитектуры на основе служб (Service-Oriented Architecture, SOA) и как ответ на *монолиты* – серверные приложения в форме одного большого выполняемого файла, – которые тогда были наиболее распространенной архитектурной моделью[1].

В то время идея архитектуры микросервисов – отдельного приложения, состоящего из нескольких небольших служб, каждая из которых выполняется в отдельном процессе и взаимодействует с другими службами приложения посредством легковесных механизмов, – была революционной. В отличие от монолитов, требующих повторной сборки и развертывания всего приложения при любых изменениях в системе, микросервисы можно было развертывать независимо, используя полностью автоматизированные механизмы. Казалось бы, в этой идее нет ничего особенного, но она оказала (и продолжает оказывать) огромное влияние.

Если попросить программистов сравнить монолиты с микросервисами, то большинство из них вам ответит, что монолиты – медленные, тяжеловесные и раздутые, а микросервисы маленькие, гибкие и шустрые. Однако широкие

[1] Надо отметить, что они еще не ушли в прошлое.

обобщения почти всегда ошибочны, поэтому давайте спросим себя, правда ли это и могут ли монолиты иногда быть правильным выбором.

Для начала давайте определим, что подразумевается под монолитами и микросервисами.

Архитектура монолитной системы

В *монолитной архитектуре* все функционально различимые аспекты службы объединены вместе. Типичным примером является веб-приложение, в котором пользовательский интерфейс, уровень данных и бизнес-логика взаимосвязаны и часто находятся на одном сервере.

Традиционно корпоративные приложения состоят из трех основных частей, как показано на рис. 7.3: клиентский интерфейс, действующий на компьютере пользователя, реляционная база данных, в которой хранятся все данные приложения, и серверное приложение, обрабатывающее ввод пользователя, выполняющее бизнес-логику и использующее базу данных.

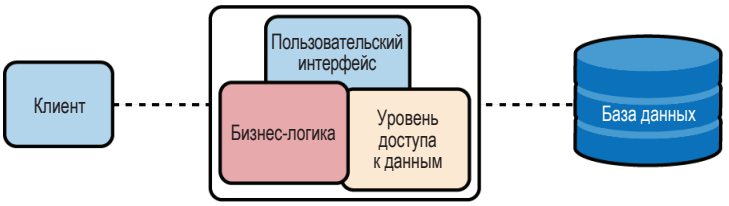


Рис. 7.3 ❖ В монолитной архитектуре все функционально различимые аспекты службы объединены вместе

В то время такая организация имела смысл. Вся бизнес-логика выполнялась в одном процессе, что упрощало разработку, и ее даже можно было масштабировать, запуская несколько монолитов за балансировщиком нагрузки, обычно используя привязку сеансов, чтобы запросы от одного и того же пользователя обрабатывались одним и тем же сервером. Все было прекрасно, и в течение многих лет это был самый распространенный способ создания веб-приложений.

Даже в наши дни для относительно небольших или простых приложений эта архитектура подходит как нельзя лучше (хотя я по-прежнему настоятельно рекомендую службы без состояния вместо привязки к серверу).

Однако с увеличением поддерживаемых возможностей и общей сложности монолита начинают возникать трудности:

- монолиты обычно развертываются как единый артефакт, поэтому, чтобы внести даже небольшое изменение, требуется выполнить сборку, тестирование и развертывание новой версии всего монолита;
- несмотря даже на самые лучшие намерения, монолитный код имеет тенденцию к уменьшению модульности с течением времени, что затрудняет внесение изменений в одну часть службы без риска непреднамеренно затронуть другую;

- масштабирование приложения означает создание копий всего приложения, а не только частей, нуждающихся в масштабировании.

Чем больше и сложнее монолит, тем выраженнее эти эффекты. К началу и середине 2000-х эти проблемы были уже хорошо известны, и программисты начали экспериментировать с разделением своих больших сложных служб на более мелкие компоненты, развертываемые и масштабируемые независимо. К 2012 году у этого подхода даже появилось название: архитектура микросервисов.

Архитектура системы микросервисов

Определяющей характеристикой *архитектуры микросервисов* является служба с функциональными компонентами, разделенными на более мелкие службы, которые можно независимо развивать, тестировать, развертывать и масштабировать.

Это проиллюстрировано на рис. 7.4, где показана служба пользовательского интерфейса (это может быть веб-приложение или общедоступный API), которая взаимодействует с клиентами, но не выполняет бизнес-логику локально, а отправляет вторичные запросы одной или нескольким серверным службам для выполнения некоторых специфических операций. Эти службы, в свою очередь, могут обращаться к другим службам.

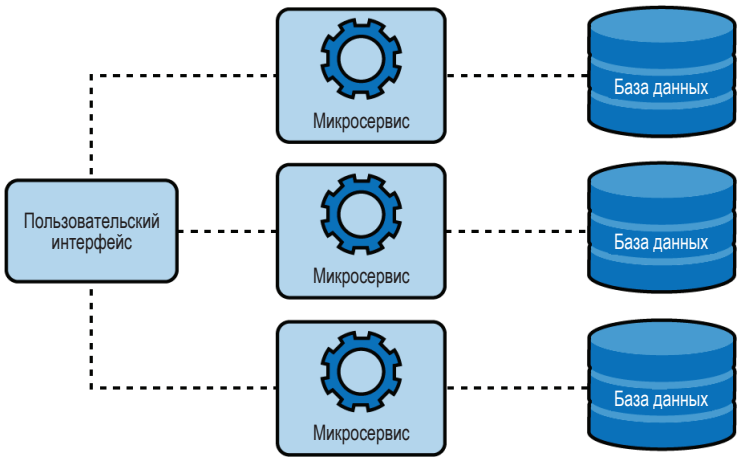


Рис. 7.4 ❖ В архитектуре микросервисов функциональные компоненты делятся на ряд небольших дискретных служб

Архитектура микросервисов имеет ряд преимуществ по сравнению с монолитом, но необходимо также учитывать значительные затраты. С одной стороны, микросервисы предлагают ряд неоспоримых преимуществ:

- четкое разделение задач обеспечивает и усиливает модульность, что очень удобно для больших команд или групп из нескольких команд;

- микросервисы могут развертываться независимо, что упрощает управление ими и дает возможность изолировать ошибки и сбои;
- в системе микросервисов для реализации разных служб можно использовать разные технологии – язык, среду разработки, хранилище данных и т. д., – которые лучше соответствуют их функциям.

Эти преимущества не следует недооценивать: модульность и функциональная изоляция микросервисов позволяют создавать компоненты, более удобные в обслуживании, чем монолит с той же функциональностью. Получившаяся система не только проще в развертывании и управлении, но и в изучении и расширении большим числом программистов и команд.

Сочетание разных технологий может показаться привлекательным, но не следует забывать о сдержанности. Каждая технология добавляет новые требования к инструментам и опыту программистов. Всегда следует внимательно рассматривать плюсы и минусы внедрения новой технологии – любой новой технологии[1].

Дискретный характер микросервисов значительно упрощает их обслуживание, развертывание и масштабирование, по сравнению с монолитами. Однако, несмотря на реальные преимущества, которые могут принести реальные выгоды, микросервисы имеют и недостатки:

- распределенная природа микросервисов делает их подверженными заблуждениям распределенных вычислений (см. главу 4), что значительно усложняет их программирование и отладку;
- совместное использование службами любого состояния часто бывает очень трудно реализовать;
- развертывание и управление несколькими службами может быть довольно сложной задачей, как правило, требующей высокого уровня подготовки специалистов.

Итак, что вы выберете, учитывая все перечисленное? Относительную простоту монолита или гибкость и масштабируемость микросервисов? Возможно, вы заметили, что большинство преимуществ микросервисов начинают приносить выгоды с увеличением размера приложения или числа команд, работающих над ним. По этой причине многие авторы рекомендуют начинать с монолита и уже потом выполнять его декомпозицию.

От себя лично замечу, что я не видел, чтобы в какой-либо организации удалось успешно разбить на части большой монолит, но я видел множество таких попыток. Нельзя сказать, что это невозможно, просто это сложно. Я не могу сказать, стоит ли сразу начинать разработку системы как системы микросервисов или лучше начать с монолита и потом разбивать его на части. Если бы я попробовал сделать это, то наверняка получил бы много гневных писем. Но пожалуйста, что бы вы ни делали, избегайте сохранения состояния.

[1] Даже языка Go.

Бессерверные архитектуры

Бессерверные вычисления – довольно популярная тема в сфере веб-приложений, и на ее рекламу было потрачено много (цифровых) чернил. Большая часть этой шумихи создана основными поставщиками облачных услуг, которые вложили значительные средства в бессерверную архитектуру, но не вся.

Что такое бессерверные вычисления на самом деле?

Что ж, как это часто бывает, ответ зависит от того, кого вы спрашиваете. Однако в этой книге мы определим их как форму вычислений, в которой некоторая серверная логика, написанная программистом, прозрачно выполняется в управляемом эфемерном окружении в ответ на некоторый предопределенный сигнал. Иногда технологию бессерверных вычислений называют «функции как услуга» (Functions as a Service, FaaS). Все основные поставщики облачных услуг предлагают реализации FaaS, такие как AWS Lambda или GCP Cloud Functions.

Бессерверные функции обладают большой гибкостью, и их можно включить во многие архитектуры. Фактически, как мы вскоре увидим, можно даже построить целые бессерверные архитектуры, которые вообще не используют традиционные службы, а целиком и полностью состоят из ресурсов FaaS и сторонних управляемых служб.

Не доверяйте шумихе

Я могу показаться старым седым динозавром, но я научился с осторожностью относиться к новым технологиям, которые, по сути, никто не понимает, но которые претендуют на роль панацеи от всех проблем.

По данным исследовательской и консалтинговой компании Gartner, специализирующейся на изучении тенденций в области информационных технологий, бессерверная инфраструктура находится на «пике завышенных ожиданий»[1] в «цикле рекламной шумихи» (https://oreil.ly/fNuG8) или приближается к нему. Рано или поздно последует «впадина разочарования».

Со временем люди начнут понимать, в чем настоящая полезность этой технологии и в каких случаях следует ее использовать, и тогда начнется «подъем просвещения» и будет достигнуто «плато производительности». Я на собственном горьком опыте убедился, что обычно лучше подождать, пока технология войдет в эти две более поздние фазы, прежде чем вкладывать значительные средства в ее использование.

И тем не менее бессерверные вычисления интригуют и кажутся подходящими для некоторых случаев использования.

Достоинства и недостатки бессерверных вычислений

Так же как в случае любого другого архитектурного решения, выбор в пользу бессерверной архитектуры (частично или полностью) следует тщательно

[1] Bowers, Daniel, et al. «Hype Cycle for Compute Infrastructure, 2019». *Gartner, Gartner Research*, 26 July 2019, https://oreil.ly/3gkJh.

взвесить со всеми доступными вариантами. Бессерверные вычисления дают некоторые преимущества – одни очевидны (нет серверов, которыми нужно управлять!), другие нет (экономия затрат и энергии), – но они сильно отличаются от традиционных архитектур и имеют свой набор недостатков.

Итак, будем взвешивать. Начнем с преимуществ.

Оперативное управление

Возможно, наиболее очевидным преимуществом бессерверных архитектур являются значительно меньшие эксплуатационные расходы[1]. Нет серверов, которые нужно настраивать и обслуживать, нет необходимости покупать лицензии, и нет программного обеспечения, требующего установки.

Масштабируемость

При использовании бессерверных функций именно провайдер, а не пользователь, отвечает за масштабирование для удовлетворения спроса. Благодаря этому разработчик может тратить меньше времени и сил на изучение и реализацию правил масштабирования.

Снижение затрат

Поставщики услуг FaaS обычно используют модель оплаты за использованные ресурсы, когда плата взимается только за процессорное время и память, потраченные на выполнение функции. Такая модель может быть значительно выгоднее, чем развертывание традиционных служб на серверах.

Продуктивность

В модели FaaS единица работы – это функция, управляемая событиями. Такая модель поощряет образ мышления «функция прежде всего», а это приводит к тому, что код зачастую становится проще, читабельнее и легче тестируется.

Но розы – это не только цветы. У бессерверных архитектур есть несколько реальных недостатков, которые необходимо учитывать.

Задержка при запуске

Когда функция вызывается в первый раз, ей нужно время на «раскрутку». Обычно это занимает меньше секунды, но в некоторых случаях может добавить 10 или более секунд ко времени обработки начальных запросов. Эта задержка известна как задержка *холодного старта*. Кроме того, если функция не вызывается в течение нескольких минут – точное время зависит от провайдера, – она «замедляется», из-за чего при повторном вызове снова возникает задержка холодного старта. Обычно это не проблема, если функция редко простаивает, но может стать серьезной проблемой, если нагрузка имеет скачкообразный характер.

Наблюдаемость

Многие поставщики облачных услуг обеспечивают базовый мониторинг своих предложений FaaS, но обычно он довольно примитивен. Несмотря

[1] Это прямо вытекает из названия!

на то что сторонние поставщики работают над заполнением пустоты, качество и количество данных, характеризующих работу эфемерных функций, часто ниже и меньше, чем хотелось бы.

Тестирование
: Модульное тестирование бессерверных функций организуется довольно просто, но интеграционное тестирование, напротив, достаточно сложно. Часто бывает трудно или невозможно смоделировать бессерверную среду, а вспомогательные имитации в лучшем случае являются приблизительными.

Расходы
: Модель оплаты только за использованные ресурсы может быть очень выгодной, когда спрос невелик, но по достижении некоторого уровня ситуация меняется. Фактически очень высокие нагрузки могут обходиться довольно дорого.

Ясно, что здесь есть над чем подумать – с обеих сторон, – и хотя в настоящее время вокруг бессерверных вычислений много шумихи, я считаю, что они заслуживают некоторого внимания. Бессерверные вычисления обещают (и в значительной степени обеспечивают) масштабируемость и снижение затрат, но у них есть немало подводных камней, включая, помимо всего прочего, проблемы с тестированием и отладкой, не говоря уже о повышенной нагрузке на тех, кто сопровождает бессерверные функции из-за худшей наблюдаемости[1]!

Наконец, как мы увидим в следующем разделе, бессерверные архитектуры также требуют уделять больше внимания предварительному планированию, чем традиционные архитектуры. Кто-то может назвать это достоинством, но это требование способно значительно усложнить работу.

Бессерверные службы

Как упоминалось выше, технология функции как услуга (FaaS) достаточно гибкая, чтобы служить основой для всей бессерверной архитектуры, в которой вообще не используются традиционные службы и которая полностью построена на ресурсах FaaS и сторонних управляемых службах.

Возьмем для примера знакомую трехуровневую систему, в которой клиент отправляет запрос службе, а та, в свою очередь, взаимодействует с базой данных. Хорошим примером является хранилище пар ключ/значение, которое мы начали разрабатывать в главе 5 и чья (по общему признанию примитивная) монолитная архитектура может выглядеть примерно так, как показано на рис. 7.5.

Чтобы преобразовать этот монолит в бессерверную архитектуру, нам понадобится *шлюз API* (API gateway): управляемая служба, определяющая некоторые конечные точки HTTP и направляющая запросы к каждой конечной точке в определенные ресурсы – обычно функции FaaS, – которые обраба-

[1] Как бы нам этого ни хотелось, но не существует такого понятия, как NoOps.

тывают запросы и возвращают ответы. При использовании данного подхода архитектура нашего хранилища пар ключ/значение может выглядеть примерно так, как показано на рис. 7.6.

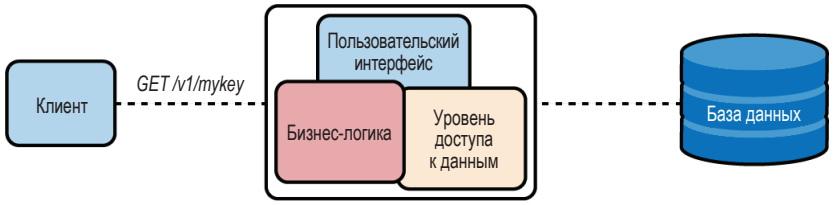


Рис. 7.5 ❖ Монолитная архитектура примитивного хранилища пар ключ/значение

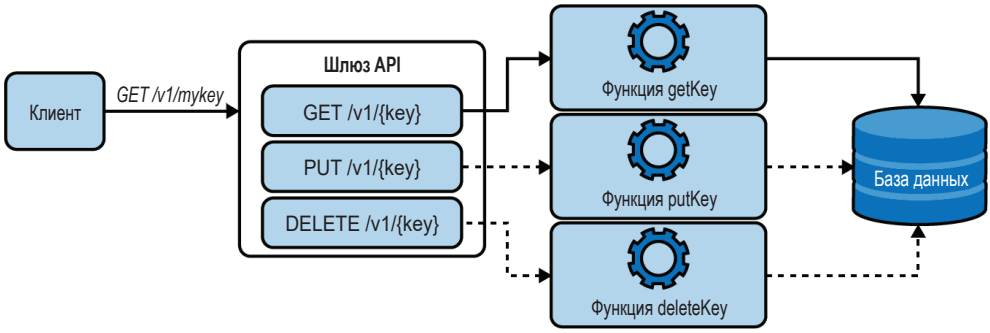


Рис. 7.6 ❖ Шлюз API пересылает HTTP-запросы бессерверным функциям-обработчикам

В этом примере мы заменили монолит шлюзом API, который поддерживает три конечные точки: `GET /v1/{key}`, `PUT /v1/{key}` и `DELETE /v1/{key}` (компонент `{key}` указывает, что эта часть пути будет соответствовать любой строке и ссылаться на значение как на ключ).

Шлюз API настроен так, что запросы к каждой из трех конечных точек направляются другим функциям-обработчикам – `getKey`, `putKey` и `deleteKey` соответственно, – выполняющим всю логику обработки запроса и взаимодействия с базой данных.

Конечно, это максимально упрощенное приложение, в котором отсутствуют такие вещи, как аутентификация (которая может быть реализована с использованием ряда замечательных сторонних служб, таких как Auth0 или Okta), но некоторые идеи заметны сразу.

Во-первых, здесь большее количество движущихся частей, которые нужно обдумать, что требует более тщательного предварительного планирования и тестирования. Например, как поступить, если в функции-обработчике возникнет ошибка? Что делать с запросом? Отправить его куда-то еще или, может быть, поместить в очередь недоставленных сообщений для повторной попытки обработки?

Не стоит недооценивать этого увеличения сложности! Замена внутрипро-

цессных взаимодействий распределенными имеет тенденцию создавать множество проблем, которых просто не существует в первых. Относительно простую проблему можно по неосторожности превратить в чрезвычайно сложную. Сложность убивает; простота стимулирует.

Во-вторых, с появлением различных компонентов возникает потребность в более сложном распределенном мониторинге, чем в монолитной системе или в небольшой системе микросервисов. Из-за того, что FaaS сильно зависит от облачного провайдера, организовать такой мониторинг может быть сложно или, по крайней мере, неудобно.

Наконец, эфемерный характер FaaS означает, что ВСЕ состояния, даже краткосрочные оптимизации, такие как кеши, следует перенести в базу данных, во внешний кеш (например, Redis) или в сетевое хранилище файлов/объектов (например, S3). Конечно, можно утверждать, что это хорошо, но это действительно добавляет сложности.

Итоги

Писать эту главу было очень сложно, но не потому, что нечего было сказать, а потому, что масштабируемость – это такая обширная тема, и я мог бы рассказать о множестве разных вещей. Мне пришлось выбирать, неделями обдумывая, что включить в книгу.

В итоге мне пришлось выбросить некоторые совершенно замечательные сведения по архитектуре, которые, если честно, просто не подходили для этой книги. К счастью, мне удалось спасти раздел по обмену сообщениями, который в итоге был перенесен в главу 8. Я думаю, что в любом случае там ему самое место.

В те недели я много думал о том, что такое масштабируемость на самом деле и какую роль в ней играет эффективность. И теперь, оглядываясь назад, я считаю, что решение потратить так много времени на программные, а не на инфраструктурные проблемы масштабирования, было правильным.

Я думаю, что конечный результат получился неплохим. Мне удалось охватить очень многое:

- мы рассмотрели различные направления масштабирования и выяснили, что масштабирование часто является лучшей долгосрочной стратегией;
- мы обсудили наличие и отсутствие состояния, а также почему состояние приложения по сути является препятствием для масштабируемости;
- мы изучили несколько стратегий эффективного кеширования и предотвращения утечек памяти;
- мы сравнили и противопоставили монолитную, микросервисную и бессерверную архитектуры.

Это довольно много, и хотя мне жаль, что у меня не было возможности углубиться в некоторые детали, я рад, что смог коснуться всего этого.

Глава 8

Слабая связанность

> Мы строим наши компьютеры так же, как строим наши города, – постепенно, без плана и на руинах[1].
>
> – Эллен Ульман (Ellen Ullman),
> *The Dumbing-down of Programming* (май 1998)

Связанность – одна из тех увлекательных тем, которые кажутся простыми в теории, но довольно сложны на практике. Как вы узнаете далее, существует множество способов введения связанности в систему, а это значит, что связанность – еще одна *обширная* тема. Соответственно, в этой главе мы охватим множество аспектов связанности.

Прежде всего мы познакомимся с самим этим предметом, исследуем понятие «связанности» и обсудим относительные преимущества «слабой» связанности перед «тесной» связанностью. Мы поговорим о некоторых наиболее распространенных механизмах связанности и о том, как некоторые виды тесной связи могут привести к ужасающему «распределенному монолиту».

Далее мы поговорим о взаимодействиях между службами и о том, что хрупкие протоколы обмена почти всегда порождают тесную связанность в распределенных системах. Мы рассмотрим некоторые из распространенных протоколов, которые используются в настоящее время и помогают ослабить связанность двух служб.

В третьей части мы немного изменим направление и перейдем от распределенных систем к реализации самих служб. Мы будем говорить о службах как об артефактах, которые могут быть связаны в результате смешивания реализаций и нарушения принципа разделения ответственности, и представим плагины (подключаемые модули) как способ динамического расширения возможностей реализации.

В заключение мы познакомимся с гексагональной архитектурой, которая ставит слабую связанность во главу угла философии проектирования.

В этой главе мы постараемся найти баланс между теорией, архитектурой и реализацией. Большая часть главы будет посвящена весьма любопытному обсуждению множества различных стратегий управления связанностью, особенно (но не только) в распределенном контексте, и демонстрации приемов расширения нашего примера хранилища пар ключ/значение.

---

[1] Ullman, Ellen. «The Dumbing-down of Programming». *Salon*, 12 мая 1998. https://oreil.ly/Eib3K.

Тесная связанность

Термин «связанность» описывает степень знания компонентами особенностей внутреннего устройства друг друга. Например, клиент, отправляющий запросы службе, по определению связан с этой службой. Однако степень данной связанности может значительно варьироваться между крайностями.

«Тесно связанные» компоненты имеют полное представление о других компонентах. Возможно, обеим сторонам требуется для взаимодействий одна и та же версия разделяемой библиотеки, а может быть, клиенту необходимо знать архитектуру сервера или схему базы данных. Стремясь максимально оптимизировать взаимодействия, легко создать тесно связанные системы, но у них есть огромный недостаток: чем теснее связаны два компонента, тем выше вероятность, что изменение одного компонента повлечет необходимость изменения другого. В результате тесно связанные системы теряют многие преимущества архитектуры микросервисов.

Напротив, «слабо связанные» компоненты имеют самый минимум знаний друг о друге. Они относительно независимы и обычно взаимодействуют через устойчивую к изменениям абстракцию. Системы, рассчитанные на слабую связанность, требуют более тщательного предварительного планирования, но такие системы проще развивать, повторно развертывать и даже полностью переписывать, потому что все эти шаги не оказывают значительного влияния на системы, которые от них зависят.

Проще говоря, если вы захотите узнать, насколько тесно связаны компоненты вашей системы, то спросите себя, сколько и каких изменений можно внести в один компонент, которые не окажут отрицательного влияния на другой.

Наличие некоторой связанности не всегда плохо, особенно на ранних этапах разработки. Может возникнуть соблазн чрезмерно абстрагировать и усложнить взаимодействия, но преждевременная оптимизация по-прежнему является корнем всех зол.

Связанность в различных вычислительных контекстах

Термин «связанность» в контексте компьютерных вычислений появился намного раньше, чем микросервисы и архитектура на основе служб, и в течение многих лет использовался для описания степени знания одного компонента о другом.

В программировании код считается тесно связанным, когда зависимый класс прямо ссылается на конкретную реализацию, а не на абстракцию (например, интерфейс; см. раздел «Интерфейсы» в главе 3). В Go это может быть функция, которая использует `os.File`, тогда как вполне могла бы использовать `io.Reader`.

Можно сказать, что многопроцессорные системы, взаимодействующие посредством совместно используемой памяти, тесно связаны. В слабо связанной системе компоненты взаимодействуют через систему передачи сообщений (см. раздел «Эффективная синхронизация» в главе 7, чтобы узнать, как Go решает эту проблему с помощью каналов).

Важно отметить, что иногда действительно может иметься веская причина для создания тесной связи между определенными компонентами. Устранение абстракций

и других промежуточных уровней часто позволяет снизить накладные расходы, что может оказаться полезной оптимизацией, если скорость является критическим требованием.

Поскольку эта книга в основном посвящена распределенным архитектурам, то мы сосредоточимся на связанности служб, которые обмениваются данными по сети, но имейте в виду, что есть и другие пути формирования тесной связанности между программным обеспечением и ресурсами.

Множество форм тесной связанности

Круг способов образования тесной связи между компонентами в распределенной сети чрезвычайно широк. Однако, несмотря на общий фундаментальный недостаток тесно связанных компонентов, все они зависят от некоторого свойства другого компонента, которое, как они ошибочно полагают, не изменится, – большинство способов можно сгруппировать в несколько классов, в зависимости от ресурса, с которым образуется связь.

Хрупкие протоколы обмена

Помните простой протокол доступа к объектам (Simple Object Access Protocol, SOAP)? Скорее всего, нет[1]. SOAP как протокол обмена сообщениями был разработан в конце 1990-х годов и предназначался для обеспечения расширяемости и независимости от реализации. Службы с поддержкой SOAP предоставляли *контракт*, которому клиенты должны были следовать при оформлении своих запросов[2]. Идея контракта была своего рода прорывом в то время, но реализация SOAP оказалась чрезвычайно хрупкой: если контракт каким-либо образом изменялся, приходилось вносить соответствующие изменения в клиентские приложения. То есть клиенты SOAP были тесно связаны со своими службами.

Людям не потребовалось много времени, чтобы понять, что это проблема, и протокол SOAP быстро был забыт. На смену ему пришел протокол REST, который, впрочем, несмотря на значительные улучшения, тоже может привести к образованию тесной связи. В 2016 году компания Google выпустила gRPC (gRPC Remote Procedure Calls[3] – вызов удаленных процедур gRPC) – фреймворк с открытым исходным кодом, обладающий рядом полезных возможностей, в том числе, что особенно важно, допускающий слабую связанность между компонентами.

Мы обсудим некоторые из этих более современных вариантов в разделе «Взаимодействия между службами» ниже, где будет показано, как использовать пакет `net/http` для создания клиента REST/HTTP и расширения нашего хранилища пар ключ/значение с помощью gRPC.

---

[1] Это моя лужайка!

[2] На языке XML, никак не меньше. В то время мы не знали ничего лучше.

[3] В Google даже аббревиатуры рекурсивные.

Общие зависимости

В 2016 году Бен Кристенсен (Ben Christensen) из Facebook выступил с докладом (https://oreil.ly/ZX2Oe) на саммите Microservices Practitioner Summit, где рассказал об еще одном механизме образования тесной связи между распределенными службами и ввел термин «распределенный монолит».

Бен описал антишаблон, когда службы *вынуждены* использовать определенные библиотеки – и определенные их версии – для взаимодействий друг с другом. Такие системы настолько обременены зависимостями, что обновление этих библиотек может потребовать синхронного обновления всех служб. Подобная общая зависимость тесно связывает все службы.

Распределенные монолиты

В главе 7 мы убедились в том, что монолиты, по крайней мере для сложных систем со множеством различных функций, – (как правило) менее подходящий, а микросервисы (как правило) – более подходящий вариант[1]. Конечно, говорить об этом проще, чем сделать, отчасти потому, что чрезвычайно легко создать *распределенный монолит*: систему на основе микросервисов, содержащую тесно связанные службы.

В распределенном монолите даже небольшие изменения в одной службе могут потребовать изменений в других, вызывая непредвиденные последствия. Службы часто невозможно развернуть независимо, поэтому развертывание приходится тщательно организовывать, потому что ошибки в одном компоненте могут вызвать сбой всей системы, а откаты функционально невозможны.

Иначе говоря, распределенный монолит – это система, которая сочетает в себе накладные расходы на управление и сложность множества служб с зависимостями и связями монолита, теряя при этом многие преимущества микросервисов. Избегайте создания распределенного монолита любой ценой.

Общий момент времени

Часто системы проектируются так, что клиенты ожидают немедленного ответа от служб. Системы, использующие шаблон обмена сообщениями *запрос/ответ*, неявно предполагают, что служба всегда готова незамедлительно ответить. Но если это не так, то попытка запроса потерпит неудачу. Можно сказать, что клиент и служба *тесно связаны во времени*.

Однако связанность во времени не всегда плоха. Более того, она может быть желательна, особенно когда человек ждет быстрого ответа. Мы даже увидим, как создать такого клиента, в разделе «Шаблон обмена сообщениями запрос/ответ» ниже.

Но если на отправку ответа нет ограничения по времени, то безопаснее было бы отправлять сообщения в промежуточную очередь, откуда получатели смогут извлечь их, когда будут готовы. Это – шаблон обмена сообщениями типа *публикация/подписка*.

---

[1] На самом деле это весьма дискуссионный вопрос. См. раздел «Архитектуры служб» в главе 7.

Фиксированные адреса

Одна из характерных черт микросервисов – потребность во взаимодействиях. Но для этого им сначала нужно найти друг друга. Процесс поиска служб в сети называется *обнаружением служб*.

Традиционно службы располагались в относительно фиксированных, хорошо известных местах в сети, и их можно было обнаружить, обратившись к некоему централизованному реестру. Первоначально реестры имели форму поддерживаемых вручную файлов `hosts.txt`, но по мере увеличения масштаба сети стали применяться система доменных имен DNS и адреса URL.

Традиционная система DNS хорошо подходит для поиска долгоживущих служб, местоположение которых в сети редко меняется, но возросшая популярность эфемерных приложений на основе микросервисов создала мир, в котором продолжительность жизни экземпляров службы часто измеряется секундами или минутами, а не месяцами или годами. В таких динамических окружениях адреса URL и традиционная служба DNS становятся просто еще одной формой тесной связи.

Потребность в гибком механизме динамического обнаружения служб привела к появлению совершенно новых стратегий, таких как *сервисная сетка* (service mesh), согласно которым выделяется отдельный слой, упрощающий взаимодействия между службами и ресурсами в распределенной системе.

К сожалению, мы не сможем охватить в этой книге захватывающие и быстро развивающиеся темы обнаружения служб и сервисных сеток. Однако сфера сервисных сеток богата множеством зрелых проектов с открытым исходным кодом и активными сообществами, таких как Envoy, Linkerd и Istio, и даже коммерческими предложениями, такими как Hashicorp Consul.

Взаимодействия между службами

Взаимодействия и передача сообщений – критически важная функция распределенных систем, и все распределенные системы зависят от поддержки обмена сообщения в той или иной форме, с помощью которой службы получают инструкции и указания, обмениваются информацией и возвращают результаты. Но, как вы понимаете, бессмысленно посылать сообщение, если получатель не сможет понять его.

Чтобы службы могли взаимодействовать, они должны установить явный или неявный *контракт*, определяющий структуру сообщений. Такой контракт необходим, но при этом он эффективно связывает компоненты, которые от него зависят.

На самом деле очень легко создать тесную связь, которая отразится на способности безопасно изменять протокол. Протокол может допускать изменения, сохраняя прямую и обратную совместимость, как, например, протокол буферов и gRPC, или, напротив, даже незначительные изменения в контракте могут нарушить обмен данными, как в случае с протоколом SOAP.

Конечно, протокол обмена и его контракт – не единственная переменная во взаимодействиях между службами. Кроме него, существуют два широких класса шаблонов обмена сообщениями:

Запрос/ответ (синхронный)
: Двусторонний обмен сообщениями, когда отправитель (клиент) посылает запрос получателю (службе) и ожидает ответа. Типичным примером может служить протокол HTML.

Публикация/подписка (асинхронный)
: Односторонний обмен сообщениями, когда отправитель (издатель) посылает сообщение в шину событий или очередь сообщений, а не напрямую конкретному получателю. Сообщения могут извлекаться асинхронно и обрабатываться одной или несколькими службами (подписчиками).

Каждый из этих шаблонов имеет множество реализаций и конкретных сценариев использования со своими достоинствами и недостатками. Я не смогу охватить все возможные нюансы, но постараюсь дать общий обзор и некоторые указания о том, как эти шаблоны можно реализовать на языке Go.

Шаблон обмена сообщениями запрос/ответ

Как следует из названия, системы, использующие шаблон обмена сообщениями *запрос/ответ*, или шаблон *синхронного обмена*, обмениваются данными, используя серию скоординированных запросов и ответов, где запрашивающая сторона (или клиент) отправляет запрос получателю (или службе) и ожидает, пока получатель вернет ответ (см. рис. 8.1).

Самым очевидным примером реализации этого шаблона является протокол HTTP, который настолько распространен и хорошо зарекомендовал себя, что был дополнен новыми функциями и теперь лежит в основе распространенных протоколов обмена сообщениями, таких как REST и GraphQL.

Рис. 8.1 ❖ Системы, использующие шаблон обмена сообщениями *запрос/ответ*, обмениваются данными, используя серию скоординированных запросов и ответов

Преимущество шаблона запрос/ответ в том, что его относительно легко осмыслить и легко реализовать. Он долгое время считался шаблоном обмена сообщениями по умолчанию, особенно для общедоступных служб. Однако это также шаблон обмена «точка-точка», то есть во взаимодействии участвуют ровно один отправитель и один получатель. Кроме того, он требует, чтобы запрашивающий процесс приостанавливался до получения ответа.

Все вместе эти свойства делают шаблон запрос/ответ хорошим выбором для прямого обмена между двумя конечными точками, когда получения ответа можно ожидать в достаточно скором времени, но он плохо подходит для сценариев, когда сообщение должно отправляться нескольким получателям или когда на подготовку ответа может потребоваться больше времени, чем запрашивающая сторона может позволить себе потратить на ожидание.

Распространенные реализации шаблона запрос/ответ

За прошедшие годы было разработано множество протоколов, реализующих шаблон запрос/ответ для самых разных целей. Со временем ситуация устоялась, и остались три основные реализации.

REST
: Вы, вероятно, уже хорошо знакомы с протоколом REST, который мы подробно обсуждали в разделе «Создание HTTP-сервера с использованием net/http» главы 5. В REST есть некоторые интересные возможности. Он удобен и прост в реализации, что делает его хорошим выбором для внешних служб (именно поэтому мы выбрали его в главе 5). Мы еще обсудим его в разделе «Отправка HTTP-запросов с использованием net/http» ниже.

Вызов удаленных процедур (*Remote Procedure Calls, RPC*)
: Фреймворки вызова удаленных процедур (RPC) позволяют программам запускать процедуры в другом адресном пространстве, часто на другом компьютере. В Go имеется стандартная реализация RPC в форме пакета `net/rpc`. Есть также две крупные реализации RPC, поддерживающие разные языки программирования: Apache Thrift и gRPC. Несмотря на сходство целей и архитектуры, gRPC, похоже, пользуется большей популярностью в сообществе. Более подробно мы обсудим gRPC в разделе «Вызов удаленных процедур с использованием gRPC» ниже.

GraphQL
: Относительно недавно появившийся на сцене, GraphQL – это язык запросов и управляющих воздействий, который часто считают альтернативой REST. Он особенно эффективен при работе со сложными наборами данных. Мы не будем углубляться в обсуждение GraphQL в этой книге, но я настоятельно рекомендую изучить его (https://graphql.org) и использовать, когда в следующий раз вы приступите к созданию нового API для взаимодействия с внешним миром.

Отправка HTTP-запросов с использованием net/http

HTTP является, пожалуй, самым распространенным протоколом типа запрос/ответ, особенно для взаимодействий с общедоступными службами, в основе

которых лежат популярные форматы API, такие как REST и GraphQL. Для взаимодействия с HTTP-службой вам понадобится возможность программно отправлять запросы и получать ответы.

К счастью, в состав стандартной библиотеки Go входит пакет `net/http` с превосходными реализациями HTTP-клиента и сервера. Возможно, вы помните пакет `net/http` по разделу «Создание HTTP-сервера с использованием net/http» в главе 5, где мы использовали его для создания первой версии нашего хранилища пар ключ/значение.

Кроме всего прочего, пакет `net/http` предлагает вспомогательные функции, реализующие HTTP-методы GET, HEAD и POST. Ниже показаны сигнатуры двух первых из них, `http.Get` и `http.Head`:

```
// Функция Get посылает запрос GET по указанному адресу URL
func Get(url string) (*http.Response, error)

// Функция Head посылает запрос HEAD по указанному адресу URL
func Head(url string) (*http.Response, error)
```

Это очень простые функции, и используются они одинаково: обе принимают строку с адресом URL, и обе возвращают значение ошибки и указатель на структуру `http.Response`.

Структура `http.Response` особенно полезна, потому что содержит ценную информацию об ответе службы на запрос, включая код состояния и тело ответа.

Вот небольшая часть структуры `http.Response`:

```
type Response struct {
    Status string         // например, "200 OK"
    StatusCode int        // например, 200

    // Header отображает ключи с именами заголовков в значения.
    Header Header

    // Body представляет тело ответа.
    Body io.ReadCloser

    // ContentLength хранит длину содержимого ответа.
    // Значение -1 указывает, что длина неизвестна.
    ContentLength int64

    // Request -- это запрос, в ответ на который был отправлен этот ответ.
    Request *Request
}
```

В структуре есть очень ценная информация! Особый интерес представляет поле `Body`, содержащее тело HTTP-ответа. Это интерфейс `ReadCloser`, который сообщает нам, что тело ответа будет передаваться в потоковом режиме в процессе чтения и у него есть метод `Close`, который мы должны вызвать.

Далее мы посмотрим, как использовать вспомогательную функцию `Get`, как закрыть тело ответа и как использовать `io.ReadAll` для получения *всего* тела ответа в виде строки (если вы предпочитаете такой подход):

```
package main

import (
    "fmt"
    "io"
    "net/http"
)

func main() {
    resp, err := http.Get("http://example.com") // Послать HTTP-запрос GET
    if err != nil {
        panic(err)
    }
    defer resp.Body.Close()            // Не забыть закрыть ответ!

    body, err := io.ReadAll(resp.Body) // Прочитать тело как срез []byte
    if err != nil {
        panic(err)
    }

    fmt.Println(string(body))
}
```

В этом примере мы используем функцию `http.Get` для отправки запроса GET по адресу `http://example.com`, а в ответ получаем указатель на структуру `http.Response` и значение ошибки.

Как упоминалось выше, тело HTTP-ответа доступно в поле `resp.Body`, которое реализует интерфейс `io.ReadCloser`. Обратите внимание, как сразу после получения ответа был запланирован вызов `resp.Body.Close()`. Это очень важно: если не закрыть тело ответа, то может возникнуть утечка памяти.

Поскольку `Body` реализует интерфейс `io.Reader`, мы можем использовать ряд стандартных средств для извлечения данных. В подобном случае используется очень надежный метод `io.ReadAll`, который возвращает все тело ответа в виде среза `[]byte`.

Никогда не забывайте вызывать метод `Close()`, чтобы закрыть тело ответа! Невыполнение этого условия может привести к утечкам памяти.

Итак, мы познакомились с функциями `Get` и `Head`, но как отправлять POST-запросы? К счастью, для этого есть аналогичные функции. Их две: `http.Post` и `http.PostForm`. Вот сигнатуры этих функций:

```
// Функция Post посылает запрос POST по указанному адресу
func Post(url, contentType string, body io.Reader) (*Response, error)

// Функция PostForm посылает запрос POST по указанному адресу с данными
// в виде пар ключ/значение, закодированными в формате URL
func PostForm(url string, data url.Values) (*Response, error)
```

Первая из них, функция `Post`, принимает экземпляр `io.Reader`, который представляет тело сообщения, например файл в формате JSON. Ниже показано, как выполнить запрос POST и послать данные в формате JSON:

```
package main

import (
    "fmt"
    "io"
    "net/http"
    "strings"
)

const json = `{ "name":"Matt", "age":44 }` // Данные в формате JSON

func main() {
    in := strings.NewReader(json)      // Заключить JSON в экземпляр io.Reader

    // Отправить HTTP-запрос POST, передав в заголовке content-type
    // строку "text/json"
    resp, err := http.Post("http://example.com/upload", "text/json", in)
    if err != nil {
        panic(err)
    }
    defer resp.Body.Close()            // Закрыть тело ответа!

    message, err := io.ReadAll(resp.Body)
    if err != nil {
        panic(err)
    }

    fmt.Printf(string(message))
}
```

Ловушка вспомогательных функций

Мы называем функции `Get`, `Head`, `Post` и `PostForm` «вспомогательными», но что это означает?

Как оказывается, за кулисами каждая из них вызывает соответствующий метод экземпляра по умолчанию `*http.Client` потокобезопасного типа, который Go использует для управления внутренними механизмами взаимодействий по HTTP.

Например, вспомогательная функция `Get` на самом деле просто вызывает метод `http.Client.Get` клиента по умолчанию:

```
func Get(url string) (resp *Response, err error) {
    return DefaultClient.Get(url)
}
```

Как видите, вызывая `http.Get`, вы фактически используете `http.DefaultClient`. А поскольку `http.Client` безопасен в конкурентном окружении, существовать может только один экземпляр в виде предопределенной переменной пакета.

Исходный код инициализации переменной `DefaultClient` довольно прост, он создает пустой экземпляр `http.Client`:

```
var DefaultClient = &Client{}
```

Это вполне нормальное решение. Однако здесь кроется потенциальная проблема, связанная с тайм-аутами. Методы экземпляра `http.Client` могут устанавливать

тайм-ауты, прерывающие длительные запросы. Это очень полезно. К сожалению, значение тайм-аута по умолчанию равно 0, что Go интерпретирует как «без тайм-аута».

Итак, у HTTP-клиента по умолчанию в языке Go никогда не истечет время ожидания. Это проблема? Обычно нет, но что получится, если клиент подключится к серверу, который не отвечает и не закрывает соединение? В результате появится особенно неприятная и недетерминированная утечка памяти.

Можно ли исправить эту проблему? Как оказывается, `http.Client` поддерживает тайм-ауты, нужно просто включить эту функцию, создав своего клиента и установив тайм-аут:

```
var client = &http.Client{
    Timeout: time.Second * 10,
}
response, err := client.Get(url)
```

Более подробную информацию о `http.Client` и доступных настройках вы найдете в документации к пакету `net/http` (https://oreil.ly/91haC).

Вызов удаленных процедур с использованием gRPC

gRPC – это эффективный многоязычный фреймворк обмена данными, изначально разработанный в Google для замены *Stubby*, универсального фреймворка RPC, использовавшегося внутри компании Google более десяти лет. Его исходный код был открыт в 2015 году под названием gRPC, а в 2017 году он был передан в фонд Cloud Native Computing Foundation.

В отличие от REST, представляющего собой набор передовых методов и приемов, gRPC – это полнофункциональный фреймворк обмена данными, который, как и другие фреймворки RPC, такие как SOAP, Apache Thrift, Java RMI и CORBA, позволяет клиенту выполнять конкретные функции, реализованные в разных системах, как если бы они были локальными функциями.

Этот подход имеет ряд преимуществ перед REST, включая:

Лаконичность
: Сообщения имеют более компактный формат и требуют меньше сетевого ввода/вывода.

Скорость
: Двоичный формат обмена намного быстрее интерпретируется и преобразуется.

Строгая типизация
: Сообщения строго типизированы, что устраняет необходимость дополнительной интерпретации и избавляет от множества распространенных ошибок.

Многофункциональность
Фреймворк имеет ряд встроенных функций, таких как аутентификация, шифрование, тайм-аут и сжатие (и это лишь некоторые из них), которые в противном случае пришлось бы реализовать самостоятельно.

Нельзя сказать, что gRPC – всегда лучший выбор. По сравнению с REST:

Действует на основе контракта
Контракты делают фреймворк gRPC менее пригодным для взаимодействий с внешними службами.

Двоичный формат
Данные gRPC не читаются человеком, что затрудняет их проверку и отладку.

gRPC – обширная и богатая тема, которую невозможно полностью передать в этом скромном разделе. Желающим узнать больше я рекомендую официальное введение в gRPC «Introduction to gRPC» (https://oreil.ly/10q7G) и превосходную книгу Касуна Индрасири (Kasun Indrasiri) и Данеша Куруппу (Danesh Kuruppu) *gRPC: Up and Running*[1] (O'Reilly Media; https://oreil.ly/Dxhjo).

Определение интерфейса с использованием протокола буферов

Подобно большинству фреймворков RPC, фреймворк gRPC требует определить *интерфейс службы*. По умолчанию для этой цели gRPC использует *протокол буферов* (https://oreil.ly/JKoyj), хотя при желании можно использовать альтернативный язык определения интерфейса, например JSON.

Чтобы определить интерфейс службы, автор использует схему протокола буферов и в файле `.proto` описывает методы службы, которые могут вызываться удаленно клиентами. Затем описание *компилируется* в интерфейс для конкретного языка (в нашем случае для Go).

Как показано на рис. 8.2, серверы gRPC реализуют получившийся исходный код для обработки клиентских вызовов, в то время как на стороне клиента используется заглушка с теми же методами, что и на стороне сервера.

Сейчас все это кажется очень необычным и абстрактным. Но продолжайте читать, чтобы узнать больше!

---

[1] *Индрасири Касун, Курупу Данеш*. gRPC. Запуск и эксплуатация облачных приложений. Go и Java для Docker и Kubernetes. СПб.: Питер, 2021. ISBN: 978-5-4461-1737-6. – *Прим. перев.*

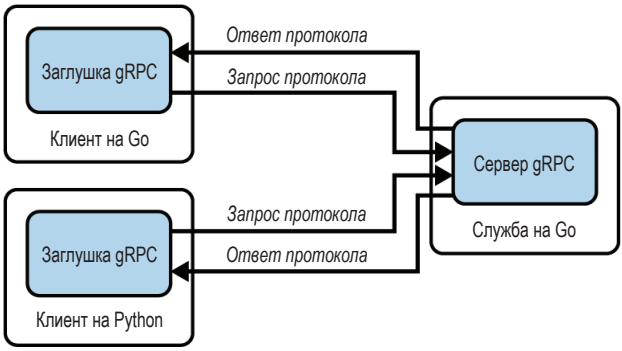


Рис. 8.2 ❖ По умолчанию gRPC использует протокол буферов в качестве языка определения интерфейса и формата обмена сообщениями; серверы и клиенты могут быть написаны на любом поддерживаемом языке (https://oreil.ly/N0uWc)

Установка компилятора протокола буферов

Прежде чем продолжить, установим компилятор протокола буферов `protoc` и плагин протокола буферов для Go. Мы будем использовать их для компиляции файлов `.proto` в исходный код интерфейса службы Go.

1. Если вы применяете Linux или MacOS, то самый простой способ установить `protoc` – воспользоваться диспетчером пакетов. Установить компилятор в Debian Linux можно с помощью `apt` или `apt-get`:

   ```
   $ apt install -y protobuf-compiler
   $ protoc --version
   ```

   Установить `protoc` в MacOS проще всего с помощью Homebrew:

   ```
   $ brew install protobuf
   $ protoc --version
   ```

2. Выполните следующую команду, чтобы установить плагин протокола буферов для Go:

   ```
   $ go install google.golang.org/protobuf/cmd/protoc-gen-go
   ```

   Плагин компилятора `protoc-gen-go` будет установлен в `$GOBIN` (по умолчанию `$GOPATH/bin`). Этот каталог должен быть включен в список каталогов в переменной окружения `$PATH`, чтобы `protoc` мог найти его.

В этой книге используется протокол буферов версии 3. Не забудьте проверить свою версию компилятора `protoc` после установки, чтобы убедиться, что он имеет версию 3 или выше.

Если вы используете другую операционную систему, или ваш диспетчер пакетов устанавливает старую версию, или если вы просто хотите установить самую последнюю и самую лучшую версию, то тогда вам следует зайти на сайт

gRPC и прочитать инструкции по установке предварительно скомпилированных двоичных файлов компилятора протокола буферов (https://oreil.ly/b6RAD).

Определение структуры сообщения

Протокол буферов – это не зависящий от языка механизм сериализации структурированных данных. Его можно рассматривать как двоичную версию XML[1]. Данные протокола буферов структурируются в *сообщения*, где каждое сообщение является небольшой записью с серией пар имя/значение, которые называют *полями*.

Первым шагом при работе с протоколом буферов необходимо определить структуру сообщения в файле `.proto`, как показано ниже:

Пример 8.1 ❖ Пример содержимого файла .proto. Определения сообщений описывают полезную нагрузку, которую несут вызовы удаленных процедур

```
syntax = "proto3";

option go_package = "github.com/cloud-native-go/ch08/point";

// Point представляет точку на 2-мерной плоскости с подписью
message Point {
  int32 x = 1;
  int32 y = 2;
  string label = 3;
}

// Line содержит начальную и конечную точки Point
message Line {
  Point start = 1;
  Point end = 2;
  string label = 3;
}

// Polyline содержит произвольное количество (в том числе и 0) точек Point
message Polyline {
  repeated Point point = 1;
  string label = 2;
}
```

Возможно, вы заметили, что синтаксис протокола буферов напоминает C/C++, с его точкой с запятой и синтаксисом комментариев.

Первая строка в файле указывает, что используется синтаксис `proto3`: если этого не сделать, то компилятор протокола буферов будет считать, что используется синтаксис `proto2`. Это должна быть первая непустая строка в файле, не являющаяся комментарием.

Вторая строка начинается с ключевого слова `option` и определяет полный путь к пакету Go, куда будет помещен сгенерированный код.

---

[1] Если вы увлекаетесь подобными вещами.

Далее следуют определения, описывающие структуру сообщений полезной нагрузки. В этом примере у нас определено три сообщения в порядке возрастания сложности:

- сообщение `Point` содержит целочисленные значения `x` и `y` и строку `label`;
- сообщение `Line` содержит два значения `Point`;
- в сообщении `Polyline` используется ключевое слово `repeat`, чтобы указать, что это сообщение может содержать любое количество значений `Point`.

Каждое сообщение содержит ноль или более полей с именем и типом. Обратите внимание, что каждое поле имеет *номер поля*, уникальный для этого типа сообщения. Номера используются для идентификации полей в двоичном формате сообщения и не должны изменяться после передачи определений в использование.

Если это вызывает у вас ассоциацию с «тесной связанностью», то вы честно заслужили золотую звезду за внимание. Именно по этой причине протокол буферов обеспечивает явную поддержку обновления типов сообщений (https://oreil.ly/IeyL2), включая маркировку поля как зарезервированного (https://oreil.ly/I1Jiu), чтобы его нельзя было случайно использовать повторно.

Этот пример невероятно прост, но пусть эта простота не вводит вас в заблуждение: протокол буферов допускает очень сложные определения. Более подробную информацию об этом ищите в руководстве по языку протокола буферов (https://oreil.ly/UDl65).

Структура сообщений для взаимодействий с хранилищем пар ключ/значение

Итак, как с помощью протокола буферов и gRPC расширить пример хранилища пар ключ/значение, начатый в главе 5?

Допустим, что мы решили реализовать gRPC-эквиваленты для функций `Get`, `Put` и `Delete`, уже доступных с помощью методов RESTful. Форматы сообщений для этой цели могут выглядеть, как показано в следующем файле `.proto`:

Пример 8.2 ❖ keyvalue.proto – сообщения для взаимодействий с хранилищем пар ключ/значение

```
syntax = "proto3";

option go_package = "github.com/cloud-native-go/ch08/keyvalue";

// GetRequest представляет запрос к хранилищу пар ключ/значение для
// получения значения по указанному ключу
message GetRequest {
  string key = 1;
}

// GetResponse представляет ответ хранилища пар ключ/значение,
// в котором возвращается запрошенное значение
```

```
message GetResponse {
  string value = 1;
}

// PutRequest представляет запрос к хранилищу пар ключ/значение для
// сохранения указанного значения с заданным ключом
message PutRequest {
  string key = 1;
  string value = 2;
}

// PutResponse представляет ответ хранилища пар ключ/значение
// на операцию Put.
message PutResponse {}

// DeleteRequest представляет запрос к хранилищу пар ключ/значение для
// удаления элемента с указанным ключом
message DeleteRequest {
  string key = 1;
}

// DeleteResponse представляет ответ хранилища пар ключ/значение
// на операцию Delete
message DeleteResponse {}
```

Не позволяйте именам определений сообщений вводить вас в заблуждение: они представляют *сообщения* (существительные), которые будут передаваться функциям (глаголам) и возвращаться от них. Сами функции мы определим в следующем разделе.

В файле *.proto*, который мы назвали *keyvalue.proto*, есть три определения с именами `Request*`, описывающие сообщения, которые будут отправляться клиентом на сервер, и три определения с именами `*Response`, описывающие сообщения, посылаемые сервером клиенту.

Возможно, вы заметили, что мы не включили значения ошибок или состояний в определения ответов. Как будет показано в разделе «Реализация клиента gRPC» ниже, в них нет необходимости, потому что они возвращаются функциями gRPC на стороне клиента.

Определение методов службы

Итак, теперь, завершив определение сообщений, мы должны описать методы, которые будут их использовать.

Для этого мы расширим файл *keyvalue.proto* и добавим в него определение интерфейса службы с помощью ключевого слова `rpc`. Компиляция измененного файла `.proto` сгенерирует код на Go, который включает определение интерфейса службы и клиентские заглушки.

Пример 8.3 ❖ keyvalue.proto – процедуры для службы хранилища пар ключ/значение

```
service KeyValue {
  rpc Get(GetRequest) returns (GetResponse);
```

```
  rpc Put(PutRequest) returns (PutResponse);
  rpc Delete(DeleteRequest) returns (DeleteResponse);
}
```

В отличие от сообщений, которые были определены в примере 8.2, определения rpc представляют функции (глаголы), которые будут отправлять и получать сообщения (существительные).

В этом примере мы добавили в нашу службу три метода:

- `Get`, принимающий `GetRequest` и возвращающий `GetResponse`;
- `Put`, принимающий `PutRequest` и возвращающий `PutResponse`;
- `Delete`, принимающий `DeleteRequest` и возвращающий `DeleteResponse`.

Обратите внимание, что здесь мы только определяем интерфейс, но не реализуем сами функции. Мы сделаем это позже.

Все перечисленные методы являются примерами *унарных определений RPC*, в которых клиент отправляет серверу один запрос и получает обратно один ответ. Это простейший из четырех типов методов служб. Также поддерживаются различные режимы потоковой передачи, но мы не будем обсуждать их в этом коротком разделе. Более подробное описание вы найдете в документации gRPC (https://oreil.ly/rs3dN).

Компиляция протокола буферов

Далее, создав файл *.proto* с определениями сообщений и служб, мы должны сгенерировать классы, которые нам понадобятся для отправки и получения сообщений. Для этого следует запустить компилятор `protoc` и передать ему наш файл *keyvalue.proto*.

Если вы еще не установили компилятор `protoc` и плагин протокола буферов для Go, то сделайте это прямо сейчас, следуя инструкциям в разделе «Установка компилятора протокола» выше.

Теперь можно запустить компилятор, указав каталог с исходным кодом (`$SOURCE_DIR`) нашего приложения (по умолчанию текущий каталог), целевой каталог (`$DEST_DIR`; часто совпадает с `$SOURCE_DIR`) и путь к файлу *keystore.proto*. Поскольку нам нужен исходный код на языке Go, добавим параметр `--go_out`. Компилятор `protoc` также поддерживает аналогичные параметры, генерирующие исходный код на других поддерживаемых языках.

В данном случае вызов протокола выглядит так:

```
$ protoc --proto_path=$SOURCE_DIR \
    --go_out=$DEST_DIR --go_opt=paths=source_relative \
    --go-grpc_out=$DEST_DIR --go-grpc_opt=paths=source_relative \
    $SOURCE_DIR/keyvalue.proto
```

Флаги `go_opt` и `go-grpc_opt` сообщают компилятору `protoc`, что он должен поместить выходные файлы в тот же каталог, где находится входной файл. Результатом компиляции нашего файла *keyvalue.proto* являются два файла: *keyvalue.pb.go* и *keyvalue_grpc.pb.go*.

Без этих флагов компилятор поместил бы выходные файлы в каталог, указанный в параметре `go_package`. Например, в результате компиляции нашего

файла *keyvalue.proto* был бы создан файл с именем `github.com/cloud-native-go/ch08/keyvalue/keyvalue.pb.go`.

Реализация службы gRPC

Чтобы реализовать сервер gRPC, нужно добавить реализацию сгенерированного интерфейса службы, который определяет API сервера службы хранилища пар ключ/значение. В нашем примере интерфейс `KeyValueServer` находится в файле *keyvalue_grpc.pb.go*:

```
type KeyValueServer interface {
    Get(context.Context, *GetRequest) (*GetResponse, error)
    Put(context.Context, *PutRequest) (*PutResponse, error)
    Delete(context.Context, *DeleteRequest) (*PutResponse, error)
}
```

Как видите, интерфейс `KeyValueServer` определяет методы `Get`, `Put` и `Delete`: все они принимают `context.Context` и указатель на запрос и возвращают указатель на ответ и ошибку.

Побочным эффектом простоты интерфейса является простота имитации запросов и ответов для тестирования сервера gRPC.

Для реализации нашего сервера используем сгенерированную структуру, включающую реализацию по умолчанию для интерфейса `KeyValueServer`, которая в нашем случае называется `UnimplementedKeyValueServer`. Такое имя дано потому, что структура включает «нереализованные» версии по умолчанию всех наших прикрепленных методов:

```
type UnimplementedKeyValueServer struct {}

func (*UnimplementedKeyValueServer) Get(context.Context, *GetRequest)
        (*GetResponse, error) {

    return nil, status.Errorf(codes.Unimplemented, "method not implemented")
}
```

Встраивая `UnimplementedKeyValueServer`, можно реализовать наш gRPC-сервер, как показано далее. В листинге ниже показан только метод `Get`. Методы `Put` и `Delete` опущены для краткости:

```
package main

import (
    "context"
    "log"
    "net"

    pb "github.com/cloud-native-go/ch08/keyvalue"
    "google.golang.org/grpc"
)
```

```
// Структура server используется в реализации KeyValueServer. Она ДОЛЖНА
// встраивать сгенерированную структуру pb.UnimplementedKeyValueServer
type server struct {
    pb.UnimplementedKeyValueServer
}

func (s *server) Get(ctx context.Context, r *pb.GetRequest)
        (*pb.GetResponse, error) {

    log.Printf("Received GET key=%v", r.Key)

    // Локальная функция Get была реализована в главе 5
    value, err := Get(r.Key)

    // Вернуть полученные от GetResponse указатель и признак ошибки
    return &pb.GetResponse{Value: value}, err
}

func main() {
    // Создать сервер gRPC и зарегистрировать в нем наш KeyValueServer
    s := grpc.NewServer()
    pb.RegisterKeyValueServer(s, &server{})

    // Открыть порт 50051 для приема сообщений
    lis, err := net.Listen("tcp", ":50051")
    if err != nil {
        log.Fatalf("failed to listen: %v", err)
    }

    // Начать цикл приема и обработку запросов
    if err := s.Serve(lis); err != nil {
        log.Fatalf("failed to serve: %v", err)
    }
}
```

В этом коде мы реализуем и запускаем нашу службу в четыре этапа.

1. *Создается экземпляр структуры* `server`. Наша структура `server` встраивает `pb.UnimplementedKeyValueServer`. Это должно быть сделано обязательно: gRPC требует, чтобы ваша структура сервера встраивала созданную структуру `UnimplementedXXXServer`.
2. *Определяются методы службы*. Мы реализовали методы, перечисленные в сгенерированном интерфейсе `pb.KeyValueServer`. Интересно отметить: поскольку `pb.UnimplementedKeyValueServer` включает заглушки для всех методов службы, нам не нужно реализовывать их все и сразу.
3. *Выполняется регистрация нашего сервера gRPC*. В функции `main` создается новый экземпляр структуры `server` и регистрируется во фреймворке gRPC. Примерно так же мы регистрировали функции-обработчики в разделе «Создание HTTP-сервера с использованием net/http» главы 5, только здесь мы регистрируем экземпляр целиком, а не отдельные функции.

4. *Запускается цикл приема и обработки запросов*. В заключение мы открываем порт для приема запросов[1] с помощью `net.Listen`, который затем передаем фреймворку gRPC через вызов `s.Serve`.

Можно утверждать, что gRPC предоставляет лучшее из обоих миров, давая свободу реализации любой желаемой функциональности, без необходимости писать множество тестов и проверок, как при создании служб RESTful.

Реализация клиента gRPC

Поскольку весь клиентский код генерируется автоматически, использовать его очень просто.

Сгенерированный интерфейс клиента получает имя `XXXClient`, то есть в нашем случае `KeyValueClient`, как показано ниже:

```
type KeyValueClient interface {
    Get(ctx context.Context, in *GetRequest, opts ...grpc.CallOption)
        (*GetResponse, error)

    Put(ctx context.Context, in *PutRequest, opts ...grpc.CallOption)
        (*PutResponse, error)

    Delete(ctx context.Context, in *DeleteRequest, opts ...grpc.CallOption)
        (*PutResponse, error)
}
```

Здесь присутствуют все методы, описанные в нашем исходном файле `.proto`, каждый из которых принимает указатель на запрос и возвращает указатель на ответ и ошибку.

Кроме того, все методы принимают `context.Context` (если вы запамятовали, что это такое или как используется, вернитесь к разделу «Пакет context» в главе 4) и ноль или более экземпляров `grpc.CallOption`. Экземпляры `CallOption` позволяют изменять поведение клиента. Более подробную информацию можно найти в документации по gRPC API (https://oreil.ly/t8fEz).

Следующий пример демонстрирует создание и использование клиента gRPC:

```
package main

import (
    "context"
    "log"
    "os"
    "strings"
    "time"

    pb "github.com/cloud-native-go/ch08/keyvalue"
    "google.golang.org/grpc"
)
```

[1] При желании можно использовать `FileListener` (https://oreil.ly/mViL3) или даже поток `stdio`.

```
func main() {
    // Установить соединение с сервером gRPC
    conn, err := grpc.Dial("localhost:50051",
        grpc.WithInsecure(), grpc.WithBlock(), grpc.WithTimeout(time.Second))
    if err != nil {
        log.Fatalf("did not connect: %v", err)
    }
    defer conn.Close()

    // Получить новый экземпляр клиента
    client := pb.NewKeyValueClient(conn)

    var action, key, value string

    // Ожидается что-то вроде "set foo bar"
    if len(os.Args) > 2 {
        action, key = os.Args[1], os.Args[2]
        value = strings.Join(os.Args[3:], " ")
    }

    // Установить 1-секундный тайм-аут с помощью context.
    ctx, cancel := context.WithTimeout(context.Background(), time.Second)
    defer cancel()

    // Вызвать client.Get() или client.Put().
    switch action {
    case "get":
        r, err := client.Get(ctx, &pb.GetRequest{Key: key})
        if err != nil {
            log.Fatalf("could not get value for key %s: %v\n", key, err)
        }
        log.Printf("Get %s returns: %s", key, r.Value)
    case "put":
        _, err := client.Put(ctx, &pb.PutRequest{Key: key, Value: value})
        if err != nil {
            log.Fatalf("could not put key %s: %v\n", key, err)
        }
        log.Printf("Put %s", key)
    default:
        log.Fatalf("Syntax: go run [get|put] KEY VALUE...")
    }
}
```

В этом примере мы анализируем значения командной строки, чтобы определить, какую операцию выполнить – `Get` или `Put`.

Прежде всего этот код устанавливает соединение с сервером gRPC вызовом функции `grpc.Dial`, которая принимает строку с адресом и один или несколько аргументов `grpc.DialOption`, определяющих способы установки соединения. Здесь используются следующие параметры:

- `WithInsecure`, отключает поддержку безопасности транспорта. *Не используйте небезопасные соединения в производстве*;

- `WithBlock`, блокирует `Dial` до тех пор, пока соединение не будет установлено, в противном случае установка соединения будет происходить в фоновом режиме;
- `WithTimeout`, устанавливает тайм-аут, когда функция `Dial` вызывается в блокирующем режиме, требующий от нее вернуть ошибку, если в течение указанного периода времени соединение не было установлено.

Затем он вызывает `NewKeyValueClient`, чтобы получить новый экземпляр `KeyValueClient`, и анализирует аргументы командной строки.

Наконец, опираясь на значение `action`, вызывает метод `client.Get` или `client.Put`, каждый из которых возвращает значение соответствующего типа и ошибку.

И снова эти функции выглядят и работают как обычные локальные функции. Никаких проверок возвращаемых кодов, создания собственных клиентов или других замысловатых операций не требуется.

Слабое связывание локальных ресурсов с помощью плагинов

На первый взгляд, тема слабого связывания локальных – в отличие от удаленных или распределенных – ресурсов может показаться неуместной в обсуждении «облачных» технологий. Но вас может удивить, насколько часто этот прием бывает полезным.

Например, часто бывает полезно создавать службы или инструменты, способные принимать данные из разных источников (таких как интерфейс REST, интерфейс gRPC и интерфейс чат-бота) или генерировать разные виды выходных данных (например, различные виды журналов). Как дополнительное преимущество такие модульные конструкции также могут значительно упростить имитацию ресурсов при тестировании.

Как будет показано в разделе «Гексагональная архитектура» ниже, на этой идее основываются целые программные архитектуры.

Никакое обсуждение слабой связанности не будет полным без обзора технологий подключаемых модулей – плагинов.

Подключение плагинов с помощью пакета plugin

В Go имеется встроенная система плагинов в виде стандартного пакета `plugin` (https://oreil.ly/zxt9W). Этот пакет используется для доступа к плагинам Go, но не требует, чтобы вы писали плагины самостоятельно.

Как будет показано ниже, Go предъявляет минимальные требования для создания и использования плагинов. Программе на Go даже не нужно знать, что она имеет дело с плагином, или импортировать пакет `plugin`. К плагинам в Go предъявляются три основных требования: они должны находиться

в пакете `main`, экспортировать одну или несколько функций или переменных и быть скомпилированы с флагом `-buildmode = plugin`. Вот и все.

Замечания к использованию плагинов в Go

Прежде чем углубиться в обсуждение плагинов Go, хотелось бы сделать несколько важных замечаний.

1. Начиная с версии Go 1.16.1 плагины поддерживаются только в Linux, FreeBSD и MacOS.
2. Для сборки плагина и программы, к которой подключается плагин, должна использоваться одна и та же версия Go. Плагины, скомпилированные с помощью Go 1.16.0, не будут работать в программе, скомпилированной с помощью Go 1.16.1.
3. Точно так же должны совпадать версии любых пакетов, используемых плагином и программой.
4. Наконец, при создании плагинов принудительно используется параметр `CGO_ENABLED`, что усложняет кросс-компиляцию.

Эти требования препятствуют созданию распространяемых плагинов и делают их более походящими, когда они находятся в той же кодовой базе, что и приложения, их использующие.

Словарь плагинов

Прежде чем продолжить, мы должны определить несколько терминов, относящихся к плагинам. Каждый из следующих терминов описывает конкретную идею, и каждому из них соответствует некоторый тип или функции в пакете `plugin`. Все это мы подробно рассмотрим в нашем примере.

Плагин

Плагин (подключаемый модуль) – это пакет `main`, экспортирующий одну или несколько функций и/или переменных, скомпилированный с флагом `-buildmode=plugin`. В пакете `plugin` плагины представлены типом `Plugin`.

Открытие

Открытие плагина – это процесс его загрузки в память, проверки и выявления экспортированных символов. Плагин, находящийся в известном месте в файловой системе, можно открыть вызовом функции `Open`, которая возвращает значение `*Plugin`:

```
func Open(path string) (*Plugin, error)
```

Символ

Символ в терминологии плагинов – это любая переменная или функция, экспортируемая пакетом плагина. Символы можно извлекать, выполнив операцию «поиска». В пакете `plugin` они представлены типом `Symbol`:

```
type Symbol interface{}
```

Поиск

Поиск – это процесс поиска и извлечения символа, экспортируемого плагином. Эту возможность обеспечивает метод `Lookup` из пакета `plugin`, который возвращает значение `Symbol`:

```
func (p *Plugin) Lookup(symName string) (Symbol, error)
```

В следующем разделе мы рассмотрим небольшой пример, иллюстрирующий порядок использования этих ресурсов, и подробно разберем весь процесс.

Пример плагина

Очень многое можно узнать, просто рассмотрев API, даже такой небольшой, как API пакета `plugin`. Итак, давайте напишем короткий пример: программу, которая расскажет вам о различных животных[1], реализованную с использованием плагинов.

В этом примере мы создадим три независимых пакета со следующей структурой:

```
~/cloud-native-go/ch08/go-plugin
├── duck
│   └── duck.go
├── frog
│   └── frog.go
└── main
    └── main.go
```

Каждый из файлов – `duck/duck.go` и `frog/frog.go` – содержит исходный код одного плагина. Файл `main/main.go` содержит функцию `main`, загружающую и использующую плагины, которые мы определим в файлах `frog.go` и `duck.go`.

Полный исходный код этого примера доступен в репозитории GitHub для данной книги (https://oreil.ly/9jRyU).

Интерфейс Sayer

Чтобы плагин приносил пользу, функции, обращающиеся к нему, должны знать, какие символы искать и какому контракту они соответствуют.

Один из удобных, но ни в коем случае не обязательный способ сделать это – использовать интерфейс, которому должен удовлетворять символ. В нашей конкретной реализации плагины будут экспортировать только один символ `Animal`, соответствующий следующему интерфейсу `Sayer`:

```
type Sayer interface {
    Says() string
}
```

---

[1] Да, я знаю, что тема с животными уже избита. Вы вправе осудить меня за это.

Этот интерфейс определяет только один метод, Says, который возвращает строку, обозначающую звук, воспроизводимый животным.

Код плагина

Исходный код двух наших плагинов находится в файлах duck/duck.go и frog/frog.go. В следующем листинге показано содержимое первого из них, duck/duck.go. Он демонстрирует все требования к реализации плагина:

```
package main

type duck struct{}

func (d duck) Says() string {
    return "quack!"
}

// Animal экспортируется как символ.
var Animal duck
```

Как рассказывалось в начале этого раздела, требования к плагинам в Go действительно минимальны: они должны находиться в пакете main и экспортировать одну или несколько переменных либо функций.

Плагин в предыдущем примере описывает и экспортирует только один символ – Animal, – удовлетворяющий интерфейсу Sayer. Напомню, что переменные и символы, экспортируемые плагином, можно найти и извлечь позже, во время выполнения. В нашем случае код должен будет искать экспортированный символ Animal.

В этом примере у нас есть только один символ, но их количество ничем не ограничивается. Мы можем экспортировать столько символов, сколько потребуется.

Я не буду приводить здесь содержимое файла frog/frog.go, потому что оно почти ничем не отличается. Но важно знать, что внутреннее устройство плагина не имеет значения, если оно удовлетворяет ожиданиям потребителя. Эти ожидания заключаются в следующем:

- плагин экспортирует символ Animal;
- символ Animal соответствует контракту, определяемому интерфейсом Sayer.

Сборка плагинов

Сборка плагина очень похожа на сборку любого другого пакета main в языке Go, за исключением того, что должна производиться с параметром -buildmode = plugin.

Собрать наш плагин duck/duck.go можно следующей командой:

```
$ go build -buildmode=plugin -o duck/duck.so duck/duck.go
```

В результате будет создана разделяемая библиотека (.so) в формате ELF (Executable Linkable Format – формат выполняемых и компонуемых модулей):

```
$ file duck/duck.so
duck/duck.so: Mach-O 64-bit dynamically linked shared library x86_64
```

Обычно плагины компилируются в файлы формата ELF, потому что после загрузки ядра в память они экспортируют символы таким образом, что их легко можно найти и получить к ним доступ.

Использование плагинов Go

Теперь, создав плагины, которые терпеливо ждут своего часа в файлах с расширением `.so`, мы должны написать код, который будет их загружать и использовать.

Обратите внимание, что, несмотря на то что мы получили готовые к использованию плагины, нам пока не довелось обращаться к пакету `plugin`. Однако теперь, когда пришло время использовать их, нам придется сделать это.

Процесс поиска, открытия и использования плагина выполняется в несколько этапов, которые я продемонстрирую далее.

ИМПОРТИРОВАНИЕ ПАКЕТА PLUGIN Прежде всего нужно импортировать пакет `plugin`, содержащий инструменты, необходимые для открытия и использования плагинов.

В следующем примере мы импортируем четыре пакета: `fmt`, `log`, `os` и, конечно же, `plugin`:

```
import (
    "fmt"
    "log"
    "os"
    "plugin"
)
```

ПОИСК ПЛАГИНА Чтобы загрузить плагин, нужно указать полный или относительный путь к файлу. По этой причине двоичные файлы плагинов обычно именуются в соответствии с каким-либо шаблоном и помещаются в такое место, где их легко найти, например в пути поиска команд или в другом стандартном месте.

Для простоты мы будем предполагать, что наш плагин имеет имя, соответствующее названию животного, выбранного пользователем, и находится в подкаталоге, рядом с выполняемым файлом:

```
if len(os.Args) != 2 {
    log.Fatal("usage: run main/main.go animal")
}

// Получить название животного и сконструировать путь к файлу
// разделяемой библиотеки (.so).
name := os.Args[1]
module := fmt.Sprintf("./%s/%s.so", name, name)
```

Важно отметить, что при таком подходе совсем необязательно, чтобы плагин существовал к моменту компиляции программы. Благодаря этому мы

сможем реализовать любые плагины и в любое время, когда захотим, а затем загружать их и получать к ним доступ динамически.

ОТКРЫТИЕ ПЛАГИНА Теперь, когда мы знаем, где находится плагин, можно вызвать функцию `Open`, чтобы «открыть» его, загрузить в память и отыскать нужные символы. Функция `Open` возвращает значение `*Plugin`, которое затем можно использовать для поиска любых символов, экспортируемых плагином:

```
// Открыть плагин и получить *plugin.Plugin.
p, err := plugin.Open(module)
if err != nil {
    log.Fatal(err)
}
```

Когда плагин открывается в первый раз, вызываются функции `init` всех пакетов, которые еще не являются частью программы. Функция `main` пакета в этот момент еще не запускается.

Когда плагин открывается, в память загружается каноническое представление значения `*Plugin`. Если плагин уже был открыт, то последующие вызовы `Open` просто будут возвращать то же значение `*Plugin`.

Плагин нельзя загрузить более одного раза, и его нельзя закрыть.

ПОИСК СИМВОЛА Чтобы получить доступ к переменной или функции, экспортируемой плагином как символ, нужно вызвать метод `Lookup`. К сожалению, пакет `plugin` не дает возможности перечислить все символы, экспортируемые плагином, поэтому имя символа нужно знать заранее:

```
// Lookup отыщет символ "Animal" в плагине p.
symbol, err := p.Lookup("Animal")
if err != nil {
    log.Fatal(err)
}
```

Если искомый символ присутствует в плагине `p`, то `Lookup` вернет значение символа. Если символ отсутствует, то будет возвращено значение ошибки, отличное от `nil`.

ПРОВЕРКА И ИСПОЛЬЗОВАНИЕ СИМВОЛА Теперь, получив значение символа, его можно преобразовать в нужную нам форму и использовать. Чтобы упростить эту задачу, тип `Symbol` объявлен как универсальный интерфейс `interface{}`. Вот выдержка из исходного кода `pluhin`:

```
type Symbol interface{}
```

Это означает, что, зная тип требуемого символа, можно выполнить его проверку и преобразовать значение символа в конкретный тип, чтобы затем использовать его по своему усмотрению:

```
// Убедиться, что символ соответствует интерфейсу Sayer.
animal, ok := symbol.(Sayer)
```

```
if !ok {
    log.Fatal("that's not a Sayer")
}

// Теперь можно смело использовать загруженный плагин!
fmt.Printf("A %s says: %q\n", name, animal.Says())
```

В этом примере мы убедились, что значение символа удовлетворяет интерфейсу `Sayer`, и затем вывели строку, изображающую звук, который издает животное. Если символ не удовлетворяет ожидаемому интерфейсу, то мы корректно завершаем работу.

Запуск примера

Теперь, когда у нас есть код, который открывает и использует плагин, можно попробовать запустить его, как любой другой пакет `main` в Go, передав название животного в виде аргумента:

```
$ go run main/main.go duck
A duck says: "quack!"

$ go run main/main.go frog
A frog says: "ribbit!"
```

Позднее, если понадобится, можно реализовать другие плагины и использовать их, не меняя исходный код главной программы:

```
$ go run main/main.go fox
A fox says: "ring-ding-ding-ding-dingeringeding!"
```

Система плагинов HashiCorp для Go, доступных через RPC

В 2016 году, примерно за год до появления стандартного пакета plugin, в HashiCorp была создана своя система плагинов для Go (https://oreil.ly/owKQp), которая нашла широкое применение не только внутри HashiCorp, но и в других местах.

В отличие от плагинов Go, которые компилируются в разделяемые библиотеки, плагины HashiCorp реализуются как автономные процессы, которые запускаются с помощью `exec.Command`, что дает некоторые очевидные преимущества перед разделяемыми библиотеками:

Они не могут вызвать сбой в основном процессе
: Это отдельные процессы, поэтому крах плагина не приведет к краху процесса-потребителя.

Они более гибкие в отношении версий
: Плагины Go, как известно, зависят от версии языка. Плагины HashiCorp в этом отношении менее зависимые, и единственное требование к ним –

соответствие заявленному контракту. Они также поддерживают явное управление версиями протокола.

Они относительно безопасны
: Плагины HashiCorp имеют доступ только к переданным им интерфейсам и параметрам, но им недоступно адресное пространство процесса-потребителя.

Однако у них есть несколько недостатков:

Более детализированные
: Для создания плагина HashiCorp требуется написать больше шаблонного кода, чем для создания плагина Go.

Низкая производительность
: Поскольку весь обмен данными с плагинами HashiCorp происходит посредством механизма RPC, взаимодействия с ними обычно протекают медленнее, чем с плагинами Go.

А теперь давайте посмотрим, что нужно, чтобы создать простой плагин HashiCorp.

Еще один пример плагина

Чтобы получить возможность сравнивать, рассмотрим пример плагина, который функционально идентичен плагину, созданному с помощью стандартного пакета `plugin` в разделе «Пример плагина» выше.

Как и в предыдущем примере, создадим несколько независимых пакетов со следующей структурой:

```
~/cloud-native-go/ch08/hashicorp-plugin
├── commons
│   └── commons.go
├── duck
│   └── duck.go
└── main
    └── main.go
```

Как и раньше, файл `duck/duck.go` содержит исходный код плагина, а файл `main/main.go` – функцию `main` нашего примера, которая загружает и использует плагин. Поскольку плагин и программа компилируются независимо, оба находятся в пакете `main`.

Пакет `common` – новый в этом примере. Он содержит некоторые ресурсы, которые используются плагином и потребителем, включая интерфейс службы и некоторый шаблонный код, использующий механизм RPC.

Полный исходный код этого примера доступен в репозитории GitHub для данной книги (https://oreil.ly/9jRyU).

Общий код

Пакет `commons` содержит некоторые ресурсы, которые используются как плагином, так и программой-потребителем, поэтому в нашем примере он импортируется и плагином, и самой программой.

Он содержит заглушки RPC, которые используются базовым механизмом `net/rpc` для определения абстракции службы и позволяют плагинам создавать свои реализации служб.

ИНТЕРФЕЙС SAYER Первый из этих ресурсов – интерфейс `Sayer`. Это интерфейс нашей службы, определяющий контракт, которому должны соответствовать реализации служб в плагинах.

Он идентичен интерфейсу, использованному в разделе «Интерфейс Sayer» выше:

```
type Sayer interface {
    Says() string
}
```

Интерфейс `Sayer` определяет только один метод: `Says`. Несмотря на то что этот код является общим, пока интерфейс не изменяется, общий контракт будет выполняться, а связанность будет оставаться на низком уровне.

СТРУКТУРА SAYERPLUGIN Следующий и более сложный общий ресурс – структура `SayerPlugin`. Она реализует основной интерфейс `plugin.Plugin` плагинов из пакета `github.com/hashicorp/go-plugin`.

Объявление *github.com/hashicorp/go-plugin* объявляет пакет `pluigin`, а не `go-plugin`, как можно было бы предположить. Учитывайте это обстоятельство в инструкциях импорта!

Методы `Client` и `Server` используются для описания службы, как того требует стандартный пакет `net/rpc`. Мы не будем рассматривать этот пакет здесь, но если вам интересно, то подробную информацию вы найдете в документации Go (https://oreil.ly/uoe8k):

```
type SayerPlugin struct {
    Impl Sayer
}

func (SayerPlugin) Client(b *plugin.MuxBroker, c *rpc.Client)
        (interface{}, error) {

    return &SayerRPC{client: c}, nil
}

func (p *SayerPlugin) Server(*plugin.MuxBroker) (interface{}, error) {
    return &SayerRPCServer{Impl: p.Impl}, nil
}
```

Оба метода принимают `plugin.MuxBroker`, который используется для создания мультиплексированных потоков при подключении плагина. Несмотря на очевидные достоинства, мы не будем рассматривать этот более сложный вариант использования.

РЕАЛИЗАЦИЯ КЛИЕНТА SayerRPC Метод `Client` предоставляет реализацию нашего интерфейса `Sayer` для обмена данными через клиента RPC – структуру с соответствующим названием `SayerRPC`, как показано ниже:

```
type SayerRPC struct{ client *rpc.Client }

func (g *SayerRPC) Says() string {
    var resp string

    err := g.client.Call("Plugin.Says", new(interface{}), &resp)
    if err != nil {
        panic(err)
    }

    return resp
}
```

`SayerRPC` использует фреймворк RPC для вызова удаленного метода `Says`, реализованного в плагине. Для этого вызывается метод `Call`, прикрепленный к `*rpc.Client`, которому передаются любые параметры (`Says` не имеет параметров, поэтому мы передаем пустой интерфейс `interface{}`). Полученный ответ метод `Call` помещает в строку `resp`.

НАСТРОЙКА ПОДКЛЮЧЕНИЯ `HandshakeConfig` используется плагином и потребителем, чтобы установить соединение. Если подключиться не удается, например потому, что плагин скомпилирован с другой версией протокола, то возвращается ошибка, ясно описывающая проблему. Такой подход не позволяет запускать неподходящие плагины или запускать их напрямую. Важно отметить, что эта особенность связана с обеспечением качества взаимодействий с пользователем, но не с системой безопасности:

```
var HandshakeConfig = plugin.HandshakeConfig{
    ProtocolVersion:  1,
    MagicCookieKey:   "BASIC_PLUGIN",
    MagicCookieValue: "hello",
}
```

Реализация сервера `SayerRPCServer`. Метод `Server` предоставляет реализацию сервера RPC – структуру `SayerRPCServer` – для обслуживания фактических методов, как того требует `net/rpc`:

```
type SayerRPCServer struct {
    Impl Sayer // Impl содержит фактическую реализацию
}

func (s *SayerRPCServer) Says(args interface{}, resp *string) error {
    *resp = s.Impl.Says()
```

```
    return nil
}
```

SayerRPCServer не реализует службу Sayer – ее метод Says вызывает реализацию Sayer, находящуюся в Impl, которую мы предоставим во время сборки плагина.

Реализация плагина

Теперь, написав общий код, используемый плагином и основной программой (интерфейс Sayer и заглушки RPC), можно переходить к реализации плагина. Весь код, представленный в этом разделе, находится в файле duck/duck.go.

Как и стандартные плагины Go, плагины HashiCorp компилируются в отдельные двоичные файлы, поэтому они должны находиться в пакете main. Фактически каждый плагин HashiCorp является небольшим автономным сервером RPC:

```
package main
```

Мы должны импортировать пакет commons, а также пакет hashicorp/go-plugin, на который будем ссылаться как на пакет plugin:

```
import (
    "github.com/cloud-native-go/ch08/hashicorp-plugin/commons"
    "github.com/hashicorp/go-plugin"
)
```

Внутри плагина мы должны определить фактическую реализацию. Реализация может быть какой угодно[1], лишь бы она соответствовала интерфейсу Sayer, который мы определяем в пакете commons:

```
type Duck struct{}

func (g *Duck) Says() string {
    return "Quack!"
}
```

Теперь перейдем к функции main. Во многом она является шаблонной, но от этого не менее важной:

```
func main() {
    // Создать и инициализировать реализацию службы.
    sayer := &Duck{}

    // pluginMap -- это ассоциативный массив плагинов,
    // которые мы будем распространять.
    var pluginMap = map[string]plugin.Plugin{
        "sayer": &commons.SayerPlugin{Impl: sayer},
    }
```

[1] Естественно, наш плагин будет крякать.

```
    plugin.Serve(&plugin.ServeConfig{
        HandshakeConfig: handshakeConfig,
        Plugins:         pluginMap,
    })
}
```

Функция `main` выполняет три шага. Во-первых, создает и инициализирует нашу реализацию службы, в данном случае экземпляр `*Duck`.

Затем добавляет в `pluginMap` ссылку на реализацию службы под именем «sayer». При желании мы могли бы реализовать несколько плагинов и перечислить их все здесь под разными именами.

Наконец, вызывается функция `plugin.Serve`, запускающая сервер RPC для обработки запросов, который позволит программе-потребителю устанавливать соединение и вызывать методы службы по своему усмотрению.

Процесс-потребитель

Теперь перейдем к процессу-потребителю – основной команде, действующей как клиент, который находит, загружает и запускает процессы плагинов.

Как вы увидите далее, шаги использования плагинов HashiCorp не сильно отличаются от шагов использования плагинов Go, описанных в разделе «Использование плагинов Go» выше.

ИМПОРТ ПАКЕТОВ HASHICORP/GO-PLUGIN И COMMONS Как обычно, начнем с объявления и импорта нашего пакета. Сама инструкция импорта нам малоинтересна – необходимость импортируемых пакетов должна быть очевидна при изучении кода.

Однако два пакета *действительно* интересны (но не удивительны) – это `github.com/hashicorp/go-plugin`, на который мы, повторюсь еще раз, должны ссылаться как на `plugin`, и наш пакет `commons`, который содержит интерфейс и настройки соединения, то есть то, что обязательно должно быть согласовано между потребителем и плагинами:

```
package main

import (
    "fmt"
    "log"
    "os"
    "os/exec"
    "github.com/cloud-native-go/ch08/hashicorp-plugin/commons"
    "github.com/hashicorp/go-plugin"
)
```

ПОИСК ПЛАГИНА Поскольку в этом примере плагин является внешним файлом, мы должны его найти. И снова для простоты наша реализация предполагает, что плагин имеет то же имя, что и выбранное пользователем название животного, и находится в соответствующем подкаталоге, расположенном рядом с выполняемым файлом основной программы:

```
func main() {
    if len(os.Args) != 2 {
        log.Fatal("usage: run main/main.go animal")
    }

    // Получить название животного и сконструировать путь
    // к выполняемому файлу плагина.
    name := os.Args[1]
    module := fmt.Sprintf("./%s/%s", name, name)

    // Файл существует?
    _, err := os.Stat(module)
    if os.IsNotExist(err) {
        log.Fatal("can't find an animal named", name)
    }
}
```

Отмечу еще раз, что ценность этого подхода в том, что плагины не обязательно должны существовать к моменту компиляции основной программы. В любой момент можно реализовать любые плагины и использовать их динамически по мере необходимости.

СОЗДАНИЕ КЛИЕНТА ПЛАГИНА Первое отличие плагинов HashiCorp от плагинов Go заключается в способе получения реализации. Плагины Go необходимо «открывать» и искать в них требуемый символ, а плагины HashiCorp построены на основе RPC, и поэтому, чтобы их использовать, нужен клиент RPC.

На самом деле для этого нужны два клиента: `*plugin.Client`, управляющий жизненным циклом процесса плагина, и клиент протокола – реализация `plugin.ClientProtocol`, взаимодействующая с процессом плагина.

Такой неудобный API сложился по историческим причинам и используется для отделения клиента, управляющего дочерними процессами, от клиента, управляющего взаимодействиями по протоколу RPC:

```
// pluginMap -- это ассоциативный массив плагинов,
// которые мы будем распространять.
var pluginMap = map[string]plugin.Plugin{
    "sayer": &commons.SayerPlugin{},
}

// Запустить процесс плагина!
client := plugin.NewClient(&plugin.ClientConfig{
    HandshakeConfig: commons.HandshakeConfig,
    Plugins:         pluginMap,
    Cmd:             exec.Command(module),
})
defer client.Kill()

// Подключиться к плагину через RPC
rpcClient, err := client.Client()
if err != nil {
    log.Fatal(err)
}
```

Большая часть этого фрагмента состоит из определения параметров нужного нам плагина в форме `plugin.ClientConfig`. Полный список параметров настройки клиента (https://oreil.ly/z29Ys) очень длинный. В этом примере используются только три:

`HandshakeConfig`
: Конфигурация подключения. Должна соответствовать конфигурации подключения в плагине, иначе на следующем шаге будет получена ошибка.

`Plugins`
: Ассоциативный массив с именами и типами нужных плагинов.

`Cmd`
: Экземпляр `*exec.Cmd`, представляющий команду для запуска процесса плагина.

Передав все эти настройки в вызов `plugin.NewClient`, мы получаем экземпляр `*plugin.Client` и сохраняем его в переменной `client`.

После этого можно использовать `client.Client`, чтобы получить клиента протокола. В этом примере он сохраняется в переменной `rpcClient`, имя которой намекает, что данный клиент знает, как взаимодействовать с процессом плагина через RPC.

ПОДКЛЮЧЕНИЕ К ПЛАГИНУ И ИСПОЛЬЗОВАНИЕ ИНТЕРФЕЙСА Sayer Получив ссылку на клиента протокола, ее можно использовать для обращения к удаленной реализации `Sayer`:

```
// Запросить плагин с помощью клиента
raw, err := rpcClient.Dispense("sayer")
if err != nil {
    log.Fatal(err)
}

// Теперь у нас должна быть реализация Sayer! Она похожа на обычную
// реализацию интерфейса, но фактически работает через соединение RPC.
sayer := raw.(commons.Sayer)

// Теперь мы можем воспользоваться нашим плагином!
fmt.Printf("A %s says: %q\n", name, sayer.Says())
}
```

С помощью функции `Dispense` клиента протокола мы получаем нашу реализацию `Sayer` в виде `interface{}`, которую затем следует привести к типу `commons.Sayer`, после чего ее сразу же можно использовать как обычный локальный экземпляр.

По сути, переменной `sayer` присваивается экземпляр `SayerRPC`, функции которого запускают RPC-вызовы, выполняемые в адресном пространстве плагина.

В следующем разделе мы познакомимся с гексагональной архитектурой, архитектурным шаблоном, который обеспечивает слабую связанность с помощью легко заменяемых «портов и адаптеров».

Гексагональная архитектура

Гексагональная архитектура (также известная как шаблон «портов и адаптеров») – это архитектурный шаблон, использующий слабую связанность и *инверсию управления* в качестве центральной философии проектирования, чтобы установить четкие границы между бизнес- и периферийной логикой.

В гексагональной архитектуре основное приложение вообще ничего не знает о внешнем мире, работая исключительно через слабосвязанные *порты* и зависящие от контекста *адаптеры*.

Этот подход позволяет приложению, например, предоставлять различные API (REST, gRPC, тестовые функции и т. д.) или использовать разные источники данных (базу данных, очереди сообщений, локальные файлы и т. д.), не влияя на основную логику и не требуя значительного изменения кода.

Мне потребовалось невероятно много времени, чтобы понять, что название «гексагональная архитектура» на самом деле ничего не означает. Алистер Кокберн (Alistair Cockburn), автор гексагональной архитектуры (https://oreil.ly/sx5io), выбрал такую форму, потому что она давала ему достаточно места для иллюстрации особенностей архитектуры.

Архитектура

Как показано на рис. 8.3, гексагональная архитектура состоит из трех компонентов, концептуально расположенных внутри и вокруг центрального шестиугольника:

Основное приложение
: Собственно приложение, представленное шестиугольником. Оно содержит всю бизнес-логику, но не связано напрямую с какой-либо технологией, фреймворком или реальным устройством. Бизнес-логика не должна зависеть от того, как передаются результаты клиенту (через REST или gRPC API) и откуда извлекаются исходные данные (из базы данных или файла *.csv*). Доступ к внешнему миру должен осуществляться только через порты.

Порты и адаптеры
: Порты и адаптеры изображены на краях шестиугольника. Порты позволяют разным акторам «подключаться» и взаимодействовать с основной службой. Адаптеры могут «подключаться» к портам и передавать сигналы между основным приложением и актором.

  Например, приложение может иметь «порт данных», к которому может подключаться «адаптер данных». Один адаптер данных может записывать данные в базу данных, а другой – использовать хранилище данных в памяти или источник тестовых данных.

Акторы
: Акторами (действующими лицами) может быть все, что взаимодействует с основным приложением (пользователи, вышестоящие службы и т. д.) или

с чем взаимодействует приложение (устройства хранения, нижестоящие службы и т. д.). Они находятся вне шестиугольника.

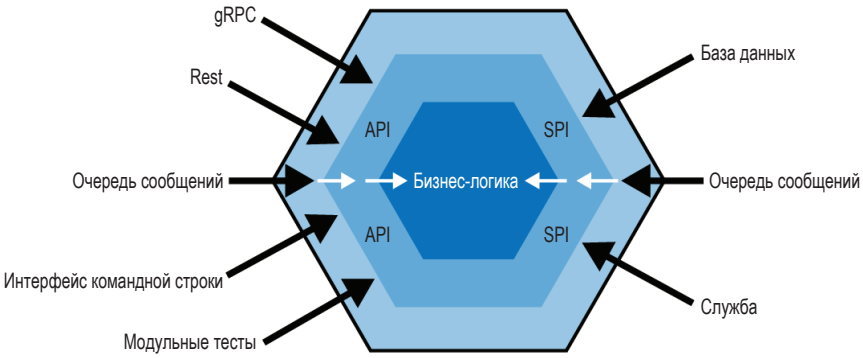


Рис. 8.3 ❖ Все зависимости в гексагональной архитектуре направлены внутрь; шестиугольники представляют бизнес-логику и уровни API, а порты и адаптеры изображены стрелками по краям шестиугольника, каждая из которых взаимодействует с определенным актором

В традиционной многоуровневой архитектуре все зависимости указывают в одном направлении, причем каждый уровень зависит от находящегося под ним.

В гексагональной архитектуре, напротив, все зависимости направлены внутрь: основная бизнес-логика ничего не знает о внешнем мире, адаптеры знают, как передавать информацию в ядро и из него, а порты знают, как взаимодействовать с акторами во внешнем мире.

Реализация гексагональной службы

Для иллюстрации мы реорганизуем нашу службу хранилища пар ключ/значение.

Как рассказывалось в главе 5, основное приложение хранилища пар ключ/значение работает с ассоциативным массивом в памяти. Доступ к этому приложению можно получить через интерфейс RESTful (или gRPC). Позже в той же главе мы реализовали регистратор транзакций, который знает, как записывать все транзакции *куда-то* и читать их обратно после запуска системы.

Здесь мы воспроизведем важные фрагменты службы, но если вам нужно освежить память и вспомнить, что было сделано, то приостановитесь и сделайте это сейчас.

К настоящему моменту мы имеем несколько реализаций разных компонентов нашей службы, которые кажутся хорошими кандидатами на порты и адаптеры в гексагональной архитектуре.

Внешний интерфейс

В разделе «Итерация 1: монолит» главы 5 мы реализовали интерфейс REST, а затем в разделе «Вызов удаленных процедур с использованием gRPC»

этой главы – отдельный интерфейс gRPC. Их можно описать с помощью одного «управляющего» порта, к которому сможем подключить любой (или оба!) как адаптер.

Регистратор транзакций
В разделе «Что такое журнал транзакций?» главы 5 мы создали две реализации журнала транзакций. Они выглядят естественным выбором на роль «управляемого» порта и адаптеров.

Несмотря на то что вся необходимая логика уже существует, нам потребуется немного реорганизовать ее, чтобы сделать эту архитектуру «гексагональной».

1. Наше исходное приложение – описанное в разделе «Итерация 0: базовая функциональность» главы 5 – использует исключительно общедоступные функции. Мы реорганизуем их в методы структуры, чтобы упростить их использование в формате «порты и адаптеры».
2. Оба интерфейса, RESTful и gRPC, уже совместимы с гексагональной архитектурой, потому что основное приложение ничего не знает о них, но они конструируются в функции `main`. Мы преобразуем их в адаптеры `FrontEnd`, в которые сможем передать наше основное приложение. Это типичный шаблон «управляющего» порта.
3. Сами регистраторы транзакций не нуждаются в реорганизации, но в настоящее время они встроены в логику внешнего интерфейса. При реорганизации основного приложения мы добавим порт регистратора транзакций, чтобы адаптер можно было передать в основную логику. Это типичный шаблон «управляемого» порта.

В следующем разделе будем брать существующие компоненты и реорганизовывать их в соответствии с принципами гексагональной архитектуры.

Реорганизация компонентов

Для этого примера поместим все компоненты в пакет `github.com/cloud-native-go/examples/ch08/hexarch`:

```
~/cloud-native-go/ch08/hexarch/
├── core
│   └── core.go
├── frontend
│   ├── grpc.go
│   └── rest.go
├── main.go
└── transact
    ├── filelogger.go
    └── pglogger.go
```

`core`
Основная логика приложения хранилища пар ключ/значение. Важно отметить, что она не имеет зависимостей за пределами стандартной библиотеки Go.

`frontend`
Содержит управляющие адаптеры интерфейсов REST и gRPC. Они зависят от основного приложения.

`transact`
Содержит управляемые адаптеры регистраторов транзакций в файл и в базу данных PostgreSQL. Также зависят от основного приложения.

`main.go`
Создает экземпляр основного приложения, в который передаются управляемые компоненты и который передается управляющим адаптерам.

Полный исходный код доступен в репозитории GitHub книги (https://oreil.ly/SsujV).

Теперь, определив общую структуру, приступим к реализации первого разъема.

Наш первый разъем

Возможно, вы помните, что мы реализовали *регистратор транзакций* для фиксации каждого изменения ресурса, чтобы в случае сбоя службы, ее повторного запуска или перехода в несогласованное состояние можно было восстановить состояние путем воспроизведения транзакций.

В разделе «Интерфейс регистратора транзакций» главы 5 мы представили обобщенный регистратор транзакций `TransactionLogger`:

```
type TransactionLogger interface {
    WriteDelete(key string)
    WritePut(key, value string)
}
```

Для краткости мы определили только методы `WriteDelete` и `WritePut`.

Общей чертой «управляемых» адаптеров является то, что *на них* воздействует основная логика, то есть основное приложение должно знать о существовании порта. Соответственно, этот код должен находиться в пакете.

Основное приложение

В первоначальной реализации в разделе «Наш суперпростой API» главы 5 регистратор транзакций использовался внешним интерфейсом. В гексагональной архитектуре мы переместим порт – в виде интерфейса `TransactionLogger` – в основное приложение:

```
package core

import (
    "errors"
    "log"
    "sync"
)
```

```
type KeyValueStore struct {
    m        map[string]string
    transact TransactionLogger
}

func NewKeyValueStore(tl TransactionLogger) *KeyValueStore {
    return &KeyValueStore{
        m:        make(map[string]string),
        transact: tl,
    }
}

func (store *KeyValueStore) Delete(key string) error {
    delete(store.m, key)
    store.transact.WriteDelete(key)
    return nil
}

func (store *KeyValueStore) Put(key string, value string) error {
    store.m[key] = value
    store.transact.WritePut(key, value)
    return nil
}
```

Сравнивая предыдущий код с исходной формой в разделе «Итерация 0: базовая функциональность» главы 5, можно заметить некоторые существенные изменения.

Во-первых, `Put` и `Delete` больше не являются чистыми функциями: теперь это методы новой структуры `KeyValueStore`, которая также имеет ассоциативный массив с данными. Мы еще добавили функцию `NewKeyValueStore`, которая инициализирует и возвращает указатель на новый экземпляр `KeyValueStore`.

Наконец, `KeyValueStore` теперь имеет `TransactionLogger`, на который воздействуют операции `Put` и `Delete`. Это наш порт.

Адаптеры TransactionLogger

В главе 5 мы создали две конкретные реализации `TransactionLogger`:

- в разделе «Реализация FileTransactionLogger» главы 5 была создана реализация на основе файлов;
- в разделе «Реализация PostgresTransactionLogger» главы 5 была создана реализация на основе PostgreSQL.

Обе они перемещены в пакет `transact`. В них практически ничего не нужно менять, за исключением того, что интерфейс `TransactionLogger` и структура `Event` теперь находятся в пакете `core`.

Но как определить, какой из них загружать? В Go нет аннотаций и других замысловатых механизмов внедрения зависимостей[1], и все же есть несколько способов сделать это.

---

[1] И слава богу!

Первый – использовать какие-либо плагины (на самом деле это основное предназначение плагинов в Go). Такой подход имеет смысл, когда желательно иметь возможность смены адаптеров без изменения кода.

Но на практике чаще используется подход на основе некоторой «фабричной» функции[1], которая вызывается функцией инициализации. Чтобы добавить адаптер, все еще требуется изменить код, но изменения ограничиваются одним легкодоступным местом. Более сложное решение – принимать параметр или читать значение конфигурации, чтобы выбрать используемый адаптер.

Вот как может выглядеть фабричная функция для создания экземпляра `TransactionLogger`:

```
func NewTransactionLogger(logger string) (core.TransactionLogger, error) {
    switch logger {
    case "file":
        return NewFileTransactionLogger(os.Getenv("TLOG_FILENAME"))

    case "postgres":
        return NewPostgresTransactionLogger(
            PostgresDbParams{
                dbName: os.Getenv("TLOG_DB_HOST"),
                host: os.Getenv("TLOG_DB_DATABASE"),
                user: os.Getenv("TLOG_DB_USERNAME"),
                password: os.Getenv("TLOG_DB_PASSWORD"),
            }
        )

    case "":
        return nil, fmt.Errorf("transaction logger type not defined")

    default:
        return nil, fmt.Errorf("no such transaction logger %s", s)
    }
}
```

В этом примере функция `NewTransactionLogger` принимает строку, определяющую желаемую реализацию, и возвращает одну из наших реализаций или ошибку. Для получения соответствующих параметров из переменных окружения используется функция `os.Getenv`.

Порт FrontEnd

Теперь перейдем к внешним интерфейсам. Как вы помните, у нас есть две реализации внешнего интерфейса:

- в разделе «Итерация 1: монолит» главы 5 мы создали интерфейс RESTful с использованием `net/http` и `gorilla/mux`;
- выше в этой главе в разделе «Вызов удаленных процедур с использованием gRPC» мы создали интерфейс RPC с использованием gRPC.

[1] Я сожалею.

Обе эти реализации включают функцию `main`, которая настраивает и запускает цикл приема запросов к службе.

Поскольку они являются «управляющими» портами, мы должны передать им основное приложение, поэтому давайте преобразуем оба интерфейса в структуры, соответствующие следующему интерфейсу:

```
package frontend

type FrontEnd interface {
    Start(kv *core.KeyValueStore) error
}
```

Интерфейс `FrontEnd` служит нашим «портом внешнего интерфейса», которому должны удовлетворять все реализации внешнего интерфейса. Метод `Start` принимает API основного приложения в форме `*core.KeyValueStore`, а также включает логику настройки, которая раньше находилась в функции `main`.

Теперь можно выполнить реорганизацию обоих интерфейсов, чтобы они соответствовали интерфейсу `FrontEnd`. Начнем с интерфейса RESTful. Как обычно, полный исходный код этого интерфейса и интерфейса gRPC доступен в репозитории GitHub для данной книги (https://oreil.ly/9jRyU):

```
package frontend

import (
    "net/http"
    "github.com/cloud-native-go/examples/ch08/hexarch/core"
    "github.com/gorilla/mux"
)

// restFrontEnd содержит ссылку на логику основного приложения
// и соответствует контракту интерфейса FrontEnd.
type restFrontEnd struct {
    store *core.KeyValueStore
}

// keyValueDeleteHandler реализует логику HTTP-метода DELETE.
func (f *restFrontEnd) keyValueDeleteHandler(w http.ResponseWriter,
        r *http.Request) {

    vars := mux.Vars(r)
    key := vars["key"]

    err := f.store.Delete(key)
    if err != nil {
        http.Error(w, err.Error(), http.StatusInternalServerError)
        return
    }
}

// ...другие функции-обработчики опущены для краткости.
```

```
// Start включает логику настройки и запуска службы,
// которая прежде находилась в функции main.
func (f *restFrontEnd) Start(store *core.KeyValueStore) error {
    // Запомнить ссылку на основное приложение.
    f.store = store

    r := mux.NewRouter()

    r.HandleFunc("/v1/{key}", f.keyValueGetHandler).Methods("GET")
    r.HandleFunc("/v1/{key}", f.keyValuePutHandler).Methods("PUT")
    r.HandleFunc("/v1/{key}", f.keyValueDeleteHandler).Methods("DELETE")

    return http.ListenAndServe(":8080", r)
}
```

Сравнив предыдущий код с кодом из раздела «Итерация 1: монолит» главы 5, можно выделить некоторые отличия:

- все функции теперь являются методами, прикрепленными к структуре `restFrontEnd`;
- все вызовы основного приложения пересекают экземпляр `store`, находящийся в структуре `restFrontEnd`;
- настройка маршрутизатора, определение функций-обработчиков и запуск сервера теперь выполняются в методе `Start`.

Аналогичные изменения нужно внести в реализацию внешнего интерфейса gRPC, чтобы сделать его совместимым с портом `FrontEnd`.

Эта новая организация упрощает выбор и подключение «внешнего адаптера», как показано ниже.

Все вместе

Вот функция `main`, в которой мы подключаем все компоненты к нашему приложению:

```
package main

import (
    "log"

    "github.com/cloud-native-go/examples/ch08/hexarch/core"
    "github.com/cloud-native-go/examples/ch08/hexarch/frontend"
    "github.com/cloud-native-go/examples/ch08/hexarch/transact"
)

func main() {
    // Создать экземпляр TransactionLogger. Этот адаптер будет включен
    // в порт TransactionLogger основного приложения.
    tl, err := transact.NewTransactionLogger(os.Getenv("TLOG_TYPE"))
    if err != nil {
        log.Fatal(err)
    }
```

```
    // Создать экземпляр Core и передать ему экземпляр TransactionLogger
    // для использования. Это пример "управляемого агента"
    store := core.NewKeyValueStore(tl)
    store.Restore()

    // Создать экземпляр frontend.
    // Это пример "управляющего агента"
    fe, err := frontend.NewFrontEnd(os.Getenv("FRONTEND_TYPE"))
    if err != nil {
        log.Fatal(err)
    }

    log.Fatal(fe.Start(store))
}
```

Здесь сначала создается регистратор транзакций в соответствии с переменной окружения `TLOG_TYPE`. Первым он создается потому, что «порт регистратора транзакций» является «управляемым», поэтому мы должны передать его при создании экземпляра приложения.

Затем создается экземпляр `KeyValueStore`, представляющий основное приложение, которое реализует API для взаимодействия с портами и адаптерами.

Далее создаются все «управляющие» адаптеры. Поскольку они используют API основного приложения, мы передаем этот API в функцию создания адаптера, а не наоборот, как это было бы с «управляемым» адаптером. Это означает, что мы могли бы также реализовать несколько внешних интерфейсов, создав новый адаптер и передав ему экземпляр `KeyValueStore`, предоставляющий API основного приложения.

В заключение вызывается метод `Start` внешнего интерфейса, который дает ему команду начать цикл приема запросов. Теперь у нас есть полноценная служба с гексагональной архитектурой!

Итоги

В этой главе мы охватили множество важных тем, но в действительности лишь вскользь перечислили различные способы образования тесных связей между компонентами и управления каждым из этих тесно связанных компонентов.

В первой половине главы мы сосредоточились на связанности, которая может возникнуть из-за особенностей взаимодействий служб. Мы поговорили о проблемах, вызываемых хрупкими протоколами обмена, такими как SOAP, и продемонстрировали решения на основе REST и gRPC, которые менее уязвимы, потому что они позволяют вносить довольно существенные изменения в серверный код без необходимости обновлять клиентов. Мы также коснулись связи «во времени», когда одна служба неявно ожидает своевременного ответа от другой, и как для решения этой проблемы можно использовать способ обмена сообщениями типа публикация/подписка.

Во второй половине мы рассмотрели некоторые способы минимизации связанности систем с локальными ресурсами. В конце концов, даже распределенные службы – это просто программы, с теми же архитектурными и прочими ограничениями, что и любые другие программы. Плагины и гексагональная архитектура – это два таких способа, основанных на принудительном разделении задач и инверсии управления.

К сожалению, нам не удалось углубиться в некоторые другие увлекательные темы, такие как обнаружение служб, и мне пришлось подвести черту, прежде чем эта тема ускользнула от меня!

Глава 9

Устойчивость

> Распределенной называется система, в которой отказ компьютера, о существовании которого вы даже не подозревали, может сделать ваш собственный компьютер бесполезным[1].
>
> – Лесли Лампорт, *DEC SRC Bulletin Board (май 1987)*

Однажды сентябрьской ночью, сразу после двух часов, часть внутренней сети Amazon тихо перестала работать[2]. Это событие было коротким и не особенно интересным, за исключением того, что оно затронуло значительное количество серверов, поддерживающих службу DynamoDB.

В большинстве случаев это не было бы такой большой проблемой. Любые серверы, затронутые сбоем, просто попытаются повторно подключиться к кластеру, получив данные о своем членстве в нем из специальной службы метаданных. Столкнувшись с невозможностью сделать это, они бы отключились на время и повторили попытку позже.

Но на этот раз, когда работа сети была восстановлена, небольшая армия серверов хранения одновременно запросила данные о своем членстве у службы метаданных, нагрузив ее так, что запросы – даже от серверов, не затронутых сбоем, – начали отбрасываться по тайм-ауту. Серверы хранения послушно реагировали на тайм-ауты, переходя в автономный режим и повторяя попытки позже, еще больше нагружая службу метаданных и вызывая отключение еще большего числа серверов. В течение нескольких минут сбой охватил весь кластер. Служба фактически перестала работать, а вместе с ней прекратил работу и ряд зависимых служб.

Хуже того, огромное количество повторных попыток – «шторм повторных попыток» – создало такую нагрузку на службу метаданных, что она перестала отвечать даже на запросы о добавлении емкости. Дежурные инженеры были вынуждены блокировать запросы к службе метаданных, лишь бы ослабить давление и получить возможность масштабировать ее вручную.

Наконец, почти через пять часов после начала сбоя в сети, вызвавшего инцидент, нормальная работа была восстановлена, и эта долгая для всех участников ночь закончилась.

---

[1] Lamport, Leslie. *DEC SRC Bulletin Board*, 28 May 1987. https://oreil.ly/nD85V.

[2] Краткое описание сбоя службы Amazon DynamoDB Service и его последствий в восточной части США. Amazon AWS, сентябрь 2015. https://oreil.ly/Y1P5S.

Почему устойчивость важна

Итак, в чем была основная причина столь масштабного сбоя в Amazon? В нарушении работы сети? В повторных попытках серверов хранения подключиться к кластеру? В большом времени ответа службы метаданных или, может быть, в ее ограниченных возможностях?

Очевидно, что происшедшее ночью имело не одну первопричину. Сбои в сложных системах никогда не происходят сразу повсюду[1]. Скорее, обычные системы терпят сбой, подобно сложным системам: сбой в одной подсистеме вызывает скрытый сбой в другой подсистеме, за которым следует еще один и еще один, и так до тех пор, пока вся система не остановится. Самое интересное, что если бы какой-то из компонентов в нашей истории – сеть, серверы хранения, служба метаданных – смог бы изолироваться от сбоев в других частях системы и восстанавить свою работоспособность, то вся система, вероятно, восстановилась бы без участия человека.

К сожалению, это лишь один пример типичного шаблона. Сбои в сложных системах распространяются сложными (и часто неожиданными) путями, но такие системы не выходят из строя сразу: они выходят из строя постепенно, по одной подсистеме за раз. По этой причине шаблоны устойчивости в сложных системах принимают форму опор и предохранительных клапанов, предназначенных для изоляции отказов вдоль границ компонентов. Изолированный сбой – это сбой, которого удалось избежать.

Это свойство, мера способности системы противостоять ошибкам и сбоям и восстанавливаться после них, называется *устойчивостью*. Систему можно считать *устойчивой*, если она способна продолжать работать правильно – возможно, менее эффективно, – вместо того чтобы полностью остановиться при выходе из строя одной из подсистем.

Устойчивость – это не безотказность

Термины *устойчивость* и *безотказность* описывают похожие понятия, которые часто путают. Но, как мы обсудим в главе 9, это не совсем одно и то же[2]:

- *устойчивость* системы – это степень, в которой она может продолжать правильно функционировать, столкнувшись с ошибками и неполадками. Устойчивость наряду с другими четырьмя облачными свойствами является лишь одним из факторов, влияющих на надежность;
- *безотказность* системы – это ее способность сохранять ожидаемое поведение в течение заданного интервала времени. Безотказность в сочетании с такими атрибутами, как доступность и ремонтопригодность, способствует общей надежности системы.

---

[1] Cook, Richard I. «How Complex Systems Fail». 1998. https://oreil.ly/WyJ4Q.

[2] Если вам интересно узнать академическую трактовку, то я настоятельно рекомендую книгу Кишора С. Триведи (Kishor S. Trivedi) и Андреа Боббио (Andrea Bobbio) «Reliability and Availability Engineering» (https://oreil.ly/80wGT).

Что подразумевается под сбоем системы?

Из-за потери гвоздя потерялась подкова,
Из-за потери подковы захромала лошадь,
Из-за захромавшей лошади враги догнали и схватили курьера,
Курьер не доставил приказ, и битва была проиграна.
И все из-за потери гвоздя, державшего подкову.

– Бенджамин Франклин (Benjamin Franklin),
Путь к богатству (1758)

Чтобы понять, что такое отказ системы, мы сначала должны определить понятие «система».

Это важно. Поверьте.

По определению, *система* – это набор компонентов, работающих вместе для достижения общей цели. Пока все понятно. Но вот важный момент: каждый компонент системы – *подсистема* – также представляет собой систему, которая, в свою очередь, состоит из еще более мелких подсистем и т. д., и т. д.

Возьмем, к примеру, автомобиль. Его двигатель – одна из десятков подсистем, но двигатель сам по себе является очень сложной системой, состоящей из множества собственных подсистем, включая подсистему охлаждения, которая содержит термостат, управляющий циркуляцией охлаждающей жидкости, и т. д. Автомобиль состоит из тысяч компонентов, подкомпонентов и подподкомпонентов. Их так много, что голова идет кругом, и каждый может отказать. Но что происходит, когда это случается?

Как упоминалось выше – и подробно обсуждалось в главе 6, – сложные системы не отказывают сразу и целиком. Процесс отказа распадается на предсказуемые шаги.

1. Все системы содержат *неисправности*, которые в мире программного обеспечения любовно называются «жучками» («багами»). Тенденция к заеданию термореле в термостате двигателя автомобиля может привести к отказу. То же самое может произойти со службой метаданных, обладающей ограниченной емкостью, из-за особенностей поведения серверов хранения, о которых рассказывалось в примере с DynamoDB выше[1]. При определенных условиях неисправность может привести к *ошибке*.
2. *Ошибка* – это любое несоответствие между предполагаемым и фактическим поведением системы. Многие ошибки можно выявить и обработать, но если ошибки не обнаруживаются, то они могут – по отдельности или в совокупности – привести к *сбою*. Заедание термореле в термостате двигателя автомобиля – это ошибка.
3. Наконец, можно сказать, что отказ системы возникает, когда она больше не может обеспечивать надлежащее обслуживание[2]. Термореле, которое перестает реагировать на высокие температуры, можно считать отказавшим. Отказ на уровне подсистемы превращается в отказ на уровне системы.

---

[1] Важно отметить, что многие недостатки очевидны только в ретроспективе.

[2] Видите? Мы наконец добрались до места.

Повторю еще раз последнее предложение: *отказ на уровне подсистемы превращается в отказ на уровне системы*. Заедание термореле приводит к сбою термостата, из-за чего нарушается циркуляция охлаждающей жидкости через радиатор, после чего температура двигателя увеличивается, из-за чего тот глохнет, и автомобиль останавливается[1].

Именно так развивается сбой в системе. Он начинается с отказа одного компонента – одной подсистемы, – что вызывает ошибку в одном или нескольких компонентах, которые с ней взаимодействуют, затем в компонентах, которые взаимодействуют с этими компонентами, и т. д., пока не произойдет отказ всей системы.

Это не просто академическое умозаключение. Знание того, как развивается отказ в сложных системах – по одному компоненту за раз, – помогает более ясно понять, как противостоять отказам: если сбой можно локализовать до того, как он распространится на всю систему, то система сможет восстановиться (или, по крайней мере, потерпеть отказ на своих условиях).

Обеспечение устойчивости

В идеальном мире можно избавить систему от всех возможных сбоев, но в нашем мире это из разряда фантастики, и пытаться добиться этого – только понапрасну потратить время и силы. Намного продуктивнее предположить, что все компоненты рано или поздно потерпят сбой – а это действительно так, – и проектировать их таким образом, чтобы они корректно реагировали на ошибки, когда те все же возникнут. То есть можно создать систему, которая будет продолжать функционировать даже после выхода из строя некоторых ее компонентов.

Есть много способов повысить устойчивость системы. Наиболее распространенным подходом является, пожалуй, создание избыточности, например, за счет развертывания нескольких одинаковых компонентов. Для изоляции определенных видов ошибок и предотвращения их распространения можно использовать специализированную логику, такую как «размыкатель цепи» и «дроссельная заслонка». Неисправные компоненты могут даже умышленно выводиться из работы, чтобы только обеспечить нормальное функционирование более крупной системы.

Устойчивость – очень богатая тема. В оставшейся части главы мы рассмотрим некоторые из подходов повышения устойчивости и многое другое.

Каскадные сбои

Уместность примера DynamoDB для нашего обсуждения обусловлена тем, что он наглядно демонстрирует разнообразие причин, по которым что-то может пойти не так.

---

[1] Давайте спросите меня, откуда я это знаю.

Возьмем, к примеру, группу серверов хранения данных, отказ в которых привел к невозможности обработки запросов в службе метаданных, что, в свою очередь, привело к отказу еще большего числа серверов хранения, что еще больше увеличило нагрузку на службу метаданных и т. д. Это отличный пример особенно распространенной причины отказа, известной как *каскадный сбой*. Начавшись, каскадный сбой имеет тенденцию к быстрому распространению, часто в течение нескольких минут.

Механизмы каскадных отказов могут отличаться, но их общей чертой является наличие некоторой положительной обратной связи. Одна часть системы испытывает локальный сбой – уменьшение емкости, увеличение задержки и т. д., – что заставляет другие компоненты пытаться компенсировать этот сбой таким способом, который усугубляет проблему, что в конечном итоге приводит к отказу всей системы.

Классической причиной каскадных сбоев является перегрузка, как показано ниже на рис. 9.1. Каскадный сбой происходит, когда один или несколько узлов, образующих систему, выходят из строя, в результате чего происходит катастрофическое увеличение нагрузки на оставшиеся узлы. Увеличение нагрузки приводит к перегрузке, что влечет их выход из строя из-за исчерпания ресурсов и остановку всей системы.

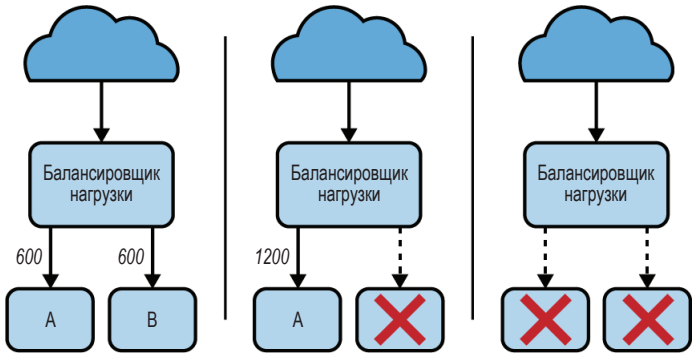


Рис. 9.1 ❖ Перегрузка сервера – частая причина каскадных сбоев; каждый сервер обслуживает по 600 запросов в секунду, поэтому когда сервер B выходит из строя, нагрузка на сервер A возрастает, и тот тоже выходит из строя

Природа положительной обратной связи часто затрудняет выход из каскадного сбоя простым увеличением емкости. Новые узлы могут быстро оказаться перегруженными, поддержав ту же обратную связь, которая первоначально привела к остановке системы. Иногда единственное решение – отключить всю службу, например явно заблокировав проблемный трафик – чтобы восстановить работоспособность, а затем постепенно нарастить нагрузку.

Но как вообще предотвратить каскадные сбои? Об этом мы поговорим в следующем разделе (и, в некоторой степени, в оставшейся части этой главы).

Предотвращение перегрузки

Каждая служба, даже хорошо спроектированная и реализованная, имеет свои функциональные ограничения. Это особенно очевидно на примере служб, предназначенных для обработки клиентских запросов[1]. Любая такая служба имеет некоторый порог – количество запросов в единицу времени, которые она способна обработать, – по достижении которого начинают возникать неприятности. Итак, как предотвратить превышение этого порога (случайное или преднамеренное!)? Отключить службу?

В конечном итоге служба, оказавшаяся в такой ситуации, не имеет другого выбора, кроме как отклонить – частично или полностью – некоторое количество запросов. Реализовать это можно с помощью двух основных стратегий:

Дросселирование
: Дросселирование (throttling) – это относительно простая стратегия, которая вступает в действие, когда запросы поступают быстрее некоторой заданной частоты, и обычно просто отклоняет их. Этот прием часто используется в качестве превентивной меры, гарантирующей, что ни один конкретный пользователь не сможет потребить ресурсов больше некоторого разумного предела.

Сброс нагрузки
: Сброс нагрузки – немного более адаптивная стратегия. Службы, использующие ее, намеренно отбрасывают некоторую часть нагрузки по мере приближения к уровню перегрузки, отклоняя запросы или переходя в режим пониженной производительности.

Эти стратегии не исключают друг друга; служба может использовать любую из них или обе сразу, в зависимости от своих потребностей.

Дросселирование

В главе 4 мы рассмотрели шаблон программирования Throttle (Дроссельная заслонка). Он работает практически так же, как дроссельная заслонка в автомобиле, за исключением того, что регулирует не количество топлива, поступающего в двигатель, а количество запросов, которые пользователь (человек или кто-то другой) может отправить службе в установленный период времени.

В разделе «Throttle (Дроссельная заслонка)» в главе 4 был представлен пример относительно простой и фактически глобальной дроссельной заслонки. Однако дросселирование также часто применяется для каждого пользователя в отдельности, чтобы реализовать что-то вроде квот и чтобы ни один пользователь не смог потребить слишком много ресурсов.

---

[1] Особенно если служба доступна из интернета.

Далее мы рассмотрим еще одну реализацию шаблона Throttle (Дроссельная заслонка), которая все так же использует корзину жетонов, но в остальном[1] сильно отличается.

Во-первых, вместо одной корзины для блокировки всех входящих запросов рассматриваемая далее реализация осуществляет регулировку для каждого отдельного пользователя, возвращая функцию, которая принимает параметр, представляющий имя пользователя или какой-то другой уникальный идентификатор.

Во-вторых, вместо попытки «воспроизвести» кешированное значение при наложении ограничения возвращаемая функция возвращает логическое значение, указывающее, что сработало ограничение. Обратите внимание, что в этом случае функция не возвращает ошибку: ограничение не является ошибкой, поэтому мы не рассматриваем ее как таковую.

Наконец, что, пожалуй, особенно интересно, эта реализация не использует таймер (`time.Ticker`) для явного добавления жетонов в корзины с некоторой регулярной частотой. Вместо этого пополнение корзины происходит в момент поступления запроса, в зависимости от времени, прошедшего после предыдущего запроса. Это означает, что реализации не нужно запускать фоновые процессы для пополнения корзин в процессе их использования, благодаря чему такой подход масштабируется более эффективно:

```
// Effector -- это функция, доступ к которой требуется регулировать.
type Effector func(context.Context) (string, error)

// Throttled служит оберткой для Effector. Она принимает те же параметры плюс
// строку "UID", идентифицирующую вызывающего, и возвращает то же значение
// плюс логическое значение true, если вызов не подвергался регулированию.
type Throttled func(context.Context, string) (bool, string, error)

// Структура bucket запоминает информацию о последнем запросе, связанном с UID.
type bucket struct {
    tokens uint
    time   time.Time
}

// Throttle принимает функцию Effector и возвращает функцию Throttled
// с корзиной, соответствующей заданному UID и заполненной максимальным
// количеством жетонов. Корзина постоянно пополняется жетонами со скоростью
// refill каждый интервал времени d.
func Throttle(e Effector, max uint, refill uint, d time.Duration) Throttled {
    // buckets отображает строки UID в конкретные корзины
    buckets := map[string]*bucket{}

    return func(ctx context.Context, uid string) (bool, string, error) {
```

[1] Из «Википедии». «Token bucket». *Wikipedia, The Free Encyclopedia*, 5 Jun. 2019. https://oreil.ly/vkOov. (Аналогичная статья в «Википедии» на русском языке: https://ru.wikipedia.org/wiki/Алгоритм_текущего_ведра – *Прим. перев.*)

```
        b := buckets[uid]

        // Это новая запись! Предполагается, что емкость >= 1.
        if b == nil {
            buckets[uid] = &bucket{tokens: max - 1, time: time.Now()}

            str, err := e(ctx)
            return true, str, err
        }

        // Подсчитать, сколько жетонов можно добавить в корзину, учитывая
        // время, прошедшее с момента предыдущего запроса.
        refillInterval := uint(time.Since(b.time) / d)
        tokensAdded := refill * refillInterval
        currentTokens := b.tokens + tokensAdded

        // Если жетонов недостаточно, вернуть false.
        if currentTokens < 1 {
            return false, "", nil
        }

        // Если корзина пополнилась, запомнить текущее время.
        // Иначе выяснить, когда в последний раз добавлялись жетоны.
        if currentTokens > max {
            b.time = time.Now()
            b.tokens = max - 1
        } else {
            deltaTokens := currentTokens - b.tokens
            deltaRefills := deltaTokens / refill
            deltaTime := time.Duration(deltaRefills) * d

            b.time = b.time.Add(deltaTime)
            b.tokens = currentTokens - 1
        }

        str, err := e(ctx)

        return true, str, err
    }
}
```

Так же как в разделе «Throttle (Дроссельная заслонка)» главы 4, функция `Throttle` принимает литерал функции, соответствующей контракту `Effector`, плюс некоторые значения, которые определяют емкость и скорость пополнения корзины жетонов.

Однако вместо того, чтобы вернуть другую функцию `Effector`, она возвращает функцию `Throttled`, которая не только обертывает функцию `Effector`, добавляя логику регулирования, но также принимает дополнительный входной параметр, представляющий уникальный идентификатор пользователя, и возвращает логическое значение, которое указывает, была ли функция подвергнута регулированию (то есть не была выполнена).

Возможно, вы найдете (или не найдете) код `Throttle` интересным, но он все еще не готов к промышленному использованию. Во-первых, он небезопасен

в конкурентном окружении. Реализация промышленного уровня, вероятно, должна защищать блокировкой доступ к значениям `record` и, возможно, к ассоциативному массиву `buckets`. Во-вторых, эта реализация не предусматривает удаление устаревших записей. Для промышленного использования, вероятно, предпочтительнее использовать что-то вроде кеша LRU, подобного тому, что был описан в разделе «Эффективное кеширование с использованием кеша LRU» в главе 7.

Далее показан упрощенный пример использования `Throttle` в веб-службе RESTful:

```
var throttled = Throttle(getHostname, 1, 1, time.Second)

func getHostname(ctx context.Context) (string, error) {
    if ctx.Err() != nil {
        return "", ctx.Err()
    }

    return os.Hostname()
}

func throttledHandler(w http.ResponseWriter, r *http.Request) {
    ok, hostname, err := throttled(r.Context(), r.RemoteAddr)

    if err != nil {
        http.Error(w, err.Error(), http.StatusInternalServerError)
        return
    }

    if !ok {
        http.Error(w, "Too many requests", http.StatusTooManyRequests)
        return
    }

    w.WriteHeader(http.StatusOK)
    w.Write([]byte(hostname))
}

func main() {
    r := mux.NewRouter()
    r.HandleFunc("/hostname", throttledHandler)
    log.Fatal(http.ListenAndServe(":8080", r))
}
```

В предыдущем примере создается небольшая веб-служба с единственной конечной точкой `/hostname`, которая возвращает имя хоста, где находится служба. В момент запуска программа создает переменную `throttled`, обертывая функцию `getHostname`, которая содержит фактическую логику службы, путем передачи ее в `Throttle`, объявленную выше.

Когда маршрутизатор получает запрос к конечной точке `/hostname`, он передает его в функцию `throttledHandler`, которая в свою очередь вызывает `throttled` и получает логическое значение с признаком регулирования, строку с именем хоста и значение ошибки. Если ошибка определена, то мы воз-

вращаем 500 Internal Server Error (внутренняя ошибка сервера), а если имело место регулирование, то 429 Too Many Requests (слишком много запросов). В остальных случаях просто возвращаем имя хоста и код состояния 200 OK.

Обратите внимание, что корзины хранятся локально, поэтому данную реализацию тоже нельзя считать готовой к использованию в промышленном окружении. Чтобы обеспечить масштабирование, можно организовать хранение записей во внешнем кеше какого-либо типа, чтобы несколько экземпляров службы могли совместно использовать их.

Сброс нагрузки

Превышение нагрузки на сервер выше некоторого предельного значения – неизбежный факт жизни, который необходимо учитывать.

Сброс нагрузки – это метод, используемый для прогнозирования момента, когда сервер приблизится к точке насыщения, и последующего уменьшения насыщения путем отбрасывания некоторой доли трафика контролируемым образом. В идеале этот метод должен предотвратить перегрузку сервера и сбой, вызывающий обслуживание с большой задержкой, или просто неконтролируемый сбой.

В отличие от дросселирования на основе квот, сброс нагрузки является методом реактивного реагирования, обычно в ответ на истощение ресурса, такого как процессор, память или размер очереди запросов.

Самая простая, пожалуй, форма сброса нагрузки – это дросселирование каждой задачи с отбрасыванием запросов, когда потребление одного или нескольких ресурсов превышает определенный порог. Например, если служба реализует конечную точку RESTful, то в ответ на перегрузку можно возвращать код HTTP 503 Service unavailable (служба недоступна). Веб-инструменты gorilla/mux, которые мы сочли очень эффективными в разделе «Создание HTTP-сервера с использованием gorilla/mux» главы 5, позволяют довольно легко реализовать это, предлагая поддержку «промежуточных» функций-обработчиков (https://oreil.ly/GTxes), которые вызываются при каждом запросе:

```
const MaxQueueDepth = 1000

// Промежуточная функция, которая вызывается при каждом запросе.
// Если размер очереди превысил некоторый порог, она возвращает код HTTP
// 503 Service unavailable.
func loadSheddingMiddleware(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        // CurrentQueueDepth здесь служит только примером.
        if CurrentQueueDepth() > MaxQueueDepth {
            log.Println("load shedding engaged")

            http.Error(w,
                err.Error(),
                http.StatusServiceUnavailable)
            return
        }

        next.ServeHTTP(w, r)
```

```
    })
}

func main() {
    r := mux.NewRouter()

    // Зарегистрировать промежуточную функцию
    r.Use(loadSheddingMiddleware)

    log.Fatal(http.ListenAndServe(":8080", r))
}
```

Промежуточные функции `gorilla/mux` вызываются при каждом запросе. Каждая из них принимает запрос, что-то с ним делает и передает его другой промежуточной функции или конечному обработчику. Это делает их идеальными для реализации сквозных задач, таких как журналирование, манипулирование заголовками, перехват `ResponseWriter`, или, как в нашем случае, для сброса нагрузки в зависимости от потребления ресурсов.

Наша промежуточная функция использует вымышленную функцию `CurrentQueueDepth()` (фактическая ее реализация будет зависеть от конкретных требований) для проверки текущего размера очереди и отклонения запросов с кодом HTTP `503 Service unavailable` (служба недоступна), если размер очереди слишком велик. Более сложные реализации могут даже выбирать, какие задачи отклонять, определяя приоритетность особенно важных запросов.

Постепенное ухудшение качества обслуживания

Прием сброса нагрузки с учетом потребления ресурсов хорошо зарекомендовал себя, но в некоторых приложениях можно действовать немного изящнее, постепенно снижая качество обслуживания при приближении к уровню перегрузки. Такое *постепенное ухудшение качества обслуживания* следует идее превентивного сброса нагрузки и сокращения работы, необходимой для удовлетворения каждого запроса, вместо простого отклонения запросов.

Способов сделать это столько же, сколько имеется служб. Конечно, постепенное ухудшение качества обслуживания можно реализовать не для каждой службы, зато есть возможность возвращать кешированные данные или получать их с использованием менее дорогостоящих, хотя и менее точных алгоритмов.

Повтори еще раз: повторные запросы

Когда в ответ на запрос возвращается ошибка или вообще ничего не возвращается, нужно просто повторить попытку, верно? Вроде как. Повторная попытка имеет смысл, но здесь очень много тонкостей.

Возьмем, к примеру, этот фрагмент кода, который я нашел в одной производственной системе:

```
res, err := SendRequest()
for err != nil {
    res, err = SendRequest()
}
```

Он выглядит соблазнительно простым, не правда ли? Он *должен* повторять неудачные запросы, и *именно это* он и будет делать. Поэтому после развертывания на нескольких сотнях серверов, когда служба, к которой эта логика отправляла запросы, перестала работать, вся система рухнула. Об этом свидетельствуют показатели обслуживания на рис. 9.2.

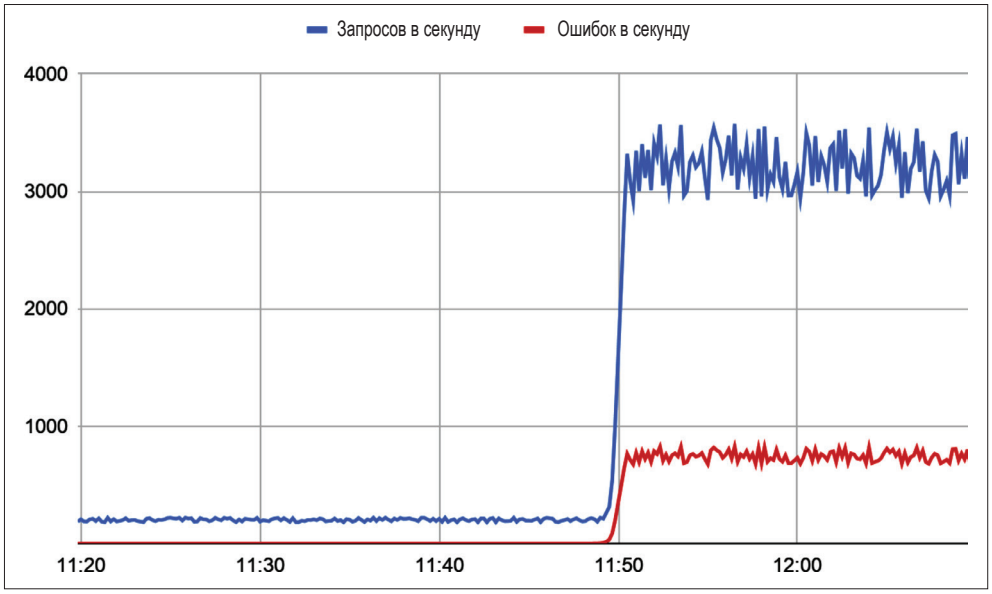


Рис. 9.2 ❖ Анатомия «шторма повторных попыток»

Похоже, что когда нижестоящая служба вышла из строя, эта служба – каждый ее экземпляр – вошла в цикл повторных попыток, выполняя тысячи запросов в секунду и забивая сеть запросами настолько сильно, что администраторы были вынуждены перезапустить всю систему.

На самом деле это очень распространенный вид каскадного сбоя, известный как *шторм повторных попыток*. Во время шторма повторных попыток благонамеренная логика, предназначенная для повышения устойчивости компонента, действует в ущерб более крупной системе. Очень часто, даже когда условия, приведшие к остановке нижестоящей службы, будут устранены, она не может возобновить работу, потому что мгновенно подвергается слишком большой нагрузке.

Но повторные попытки – это же хорошо, правда?

Да, но всякий раз, реализуя логику повтора, вы должны использовать алгоритм увеличения задержки, который мы обсудим в следующем разделе.

Алгоритмы увеличения задержки

Если запрос к нижестоящей службе по какой-то причине не был выполнен, то «лучше всего» повторить запрос. Но сколько времени желательно подождать перед повторной попыткой? Если ждать слишком долго, важная работа будет выполнена с большим опозданием. Если ждать слишком мало, то есть риск завалить запросами службу, сеть или и то, и другое.

Распространенное решение – алгоритм увеличения задержки, который вводит дополнительную задержку между повторными попытками, чтобы снизить частоту попыток до безопасной и приемлемой величины.

Существует множество разных алгоритмов увеличения задержки, самый простой из которых – выполнить короткую фиксированную задержку перед каждой повторной попыткой, а именно:

```
res, err := SendRequest()
for err != nil {
    time.Sleep(2 * time.Second)
    res, err = SendRequest()
}
```

Функция `SendRequest` в предыдущем фрагменте используется для отправки запроса и возвращает строку и признак ошибки. Если в `err` возвращается значение, отличное от `nil`, то запускается цикл, в котором производится приостановка выполнения на две секунды, а затем выполняется повторная попытка. Цикл повторяется до бесконечности, пока не будет получен ответ без ошибки.

На рис. 9.3 показан график изменения количества запросов, сгенерированных 1000 экземпляров, использующих этот код[1]. Как видите, подход с фиксированной задержкой может уменьшить количество запросов по сравнению с приемом без задержки, но общее количество запросов по-прежнему остается довольно большим.

Прием с фиксированной задержкой можно с успехом использовать, когда количество повторных попыток невелико, но он плохо масштабируется, потому что при достаточном количестве запрашивающих служб все еще может возникать перегрузка сети.

Однако не всегда можно предположить, что у какой-то конкретной службы будет небольшое количество экземпляров или даже что она будет единственной, выполняющей повторные попытки. По этой причине многие алгоритмы увеличения задержки используют экспоненциальную формулу, согласно которой длительность задержек между повторами примерно удваивается с каждой попыткой до некоторого заданного максимума.

Очень распространенная (но ошибочная, как вы скоро увидите) реализация экспоненциального увеличения задержки выглядит примерно так:

---

[1] Весь код, демонстрируемый в этом разделе, доступен в репозитории GitHub книги (https://oreil.ly/m61X7).

```
res, err := SendRequest()
base, cap := time.Second, time.Minute

for backoff := base; err != nil; backoff <<= 1 {
    if backoff > cap {
        backoff = cap
    }
    time.Sleep(backoff)
    res, err = SendRequest()
}
```

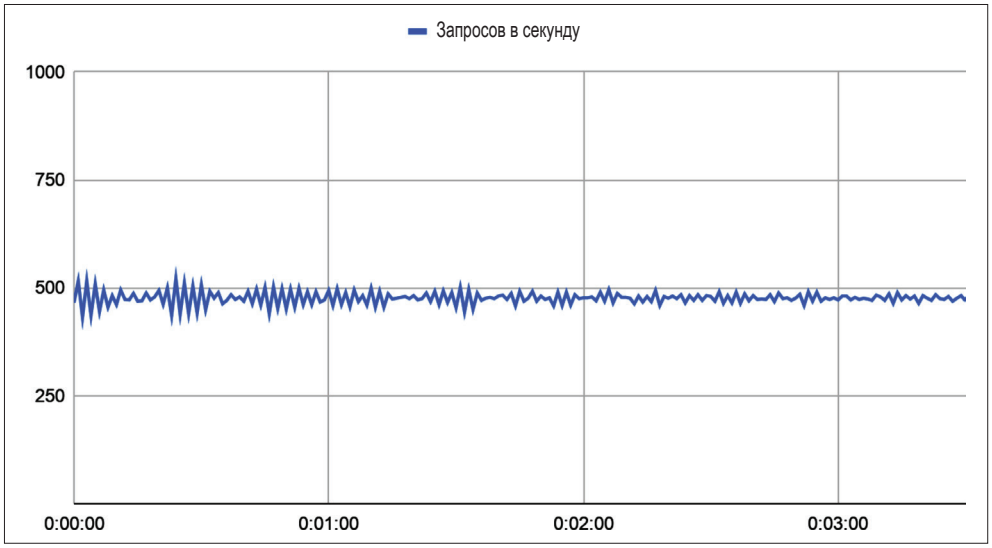


Рис. 9.3 ❖ Количество запросов в секунду, сгенерированное 1000 экземпляров службы с использованием двухсекундной задержки перед повтором

В этом фрагменте мы указываем начальную продолжительность задержки `base` и фиксированную максимальную продолжительность `cap`. Перед началом цикла переменной `backoff` присваивается значение `base` и затем в каждой итерации удваивается, пока не будет достигнуто максимальное значение `cap`.

На первый взгляд такой прием может снизить нагрузку на сеть и уменьшить количество повторных попыток. Однако моделирование все той же ситуации с 1000 узлов показывает совсем иное (см. рис. 9.4).

Как видите, такое расписание повторных попыток при наличии 1000 узлов все еще далеко от оптимального, потому что теперь повторные попытки оказываются сгруппированными и могут создать достаточно высокую нагрузку, чтобы вызвать проблемы. Таким образом, простое экспоненциальное увеличение задержки не всегда помогает так, как хотелось бы.

По всей видимости, нам нужен способ, который помог бы распределить всплески по времени, чтобы повторные попытки выполнялись с примерно

постоянной частотой. Решение состоит в том, чтобы добавить элемент случайности – *отклонения*. Добавив случайные отклонения в алгоритм вычисления задержки, получаем примерно такой код:

```
res, err := SendRequest()
base, cap := time.Second, time.Minute

for backoff := base; err != nil; backoff <<= 1 {
    if backoff > cap {
        backoff = cap
    }

    jitter := rand.Int63n(int64(backoff * 3))
    sleep := base + time.Duration(jitter)
    time.Sleep(sleep)
    res, err = SendRequest()
}
```

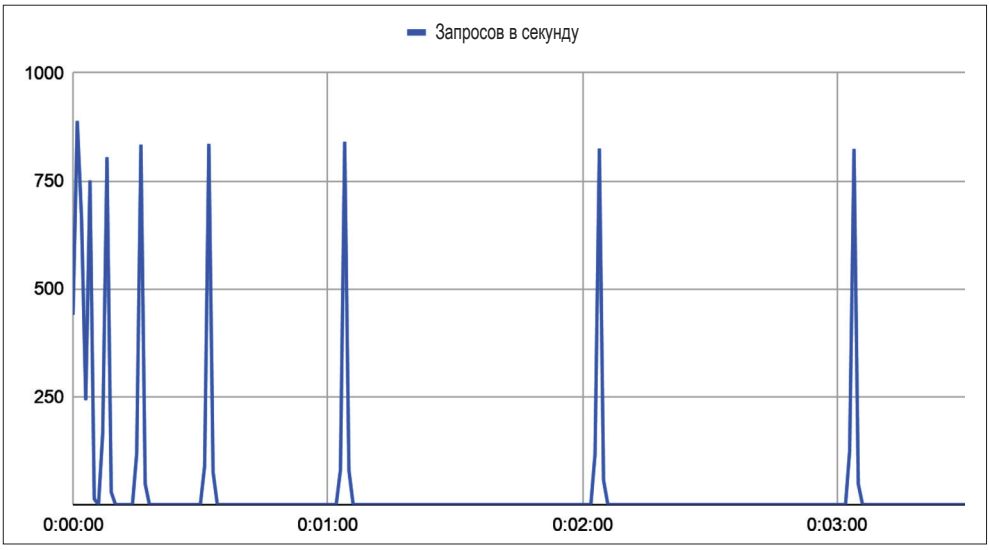


Рис. 9.4 ❖ Количество запросов в секунду, сгенерированное 1000 экземпляров службы с использованием экспоненциально увеличивающейся задержки перед повтором

Сымитировав повторные попытки, выполняемые 1000 служб, получаем картину, изображенную на рис. 9.5.

Функции в пакете `rand` создают детерминированную последовательность значений при каждом запуске программы. Если не использовать функцию `rand.Seed`, чтобы задать новое начальное значение, то они всегда будут действовать так, как если бы был выполнен вызов `rand.Seed(1)`, и всегда производить одну и ту же «случайную» последовательность чисел.

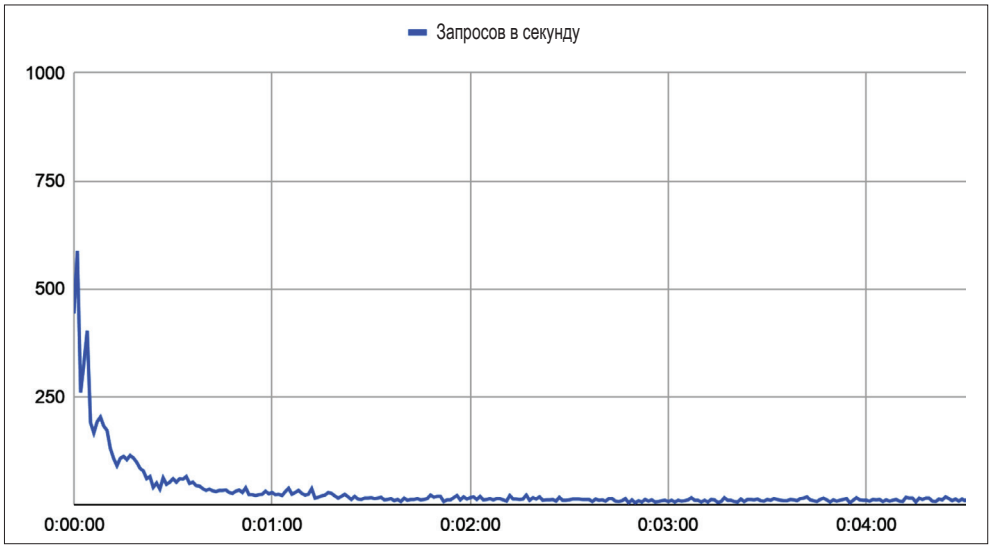


Рис. 9.5 ❖ Количество запросов в секунду, сгенерированное 1000 экземпляров службы с использованием экспоненциально увеличивающейся задержки и случайным отклонением перед повтором

При использовании экспоненциального увеличения задержки со случайным отклонением количество повторных попыток, выполняемых в течение короткого интервала, уменьшается. Это помогает избежать перегрузки вызываемой службы и распределяет запросы во времени так, что они посылаются с приблизительно постоянной скоростью.

Кто бы мог подумать, что повторная попытка выполнить запрос – это нечто большее, чем просто повторная попытка?

Размыкание цепи

Впервые шаблон Circuit Breaker (Размыкатель цепи) был представлен в главе 4 как средство изоляции потенциально неудачных вызовов методов и предотвращения более крупных или каскадных сбоев. Это определение по-прежнему остается в силе, и поскольку мы не собираемся его сильно расширять или изменять, то не будем подробно останавливаться на нем.

В общем случае шаблон Circuit Breaker (Размыкатель цепи) запоминает количество неудачных запросов нижестоящему компоненту, следующих подряд. Если счетчик отказов превышает определенный порог, цепь «размыкается», и все попытки отправить дополнительные запросы немедленно терпят неудачу (или возвращается некоторое предопределенное значение). После некоторого ожидания схема автоматически «замыкается», возвращаясь в нормальное состояние и позволяя выполнять запросы в обычном режиме.

Не все модели устойчивости являются защитными. Иногда они имеют целью реализовать добропорядочного соседа, не досаждающего окружающим своей назойливостью.

Правильное применение шаблона Circuit Breaker (Размыкатель цепи) может играть важную роль для восстановления работоспособности системы и предотвращения каскадных сбоев. В дополнение к очевидным преимуществам, таким как отсутствие затрат ресурсов или засорения сети обреченными запросами, размыкатель цепи (особенно тот, который имеет функцию увеличения задержки между попытками) может дать неисправной службе достаточно времени для восстановления и позволить ей возобновить корректную работу.

Отличия между шаблонами Circuit Breaker и Throttle

На первый взгляд шаблоны Circuit Breaker (Размыкатель цепи) и Throttle (Дроссельная заслонка) очень похожи. В конце концов, оба являются шаблонами, способствующими устойчивости и управляющими количеством вызовов в единицу времени. Но в действительности они имеют существенные отличия:

- *Circuit Breaker* (*Размыкатель цепи*) обычно применяется только к исходящим запросам. При этом его совершенно не волнует частота запросов – только количество неудавшихся запросов, и только если они идут подряд;
- *Throttle* (*Дроссельная заслонка*) действует подобно дроссельной заслонке в автомобиле, ограничивая количество запросов – независимо от успеха или неудачи – некоторой максимальной скоростью. Обычно этот шаблон применяется к входящему трафику, но, вообще говоря, нет какого-либо правила, требующего поступать именно так.

Шаблон Circuit Breaker (Размыкатель цепи) был подробно описан в главе 4, поэтому мы не будем повторно исследовать его здесь. Полное описание и примеры кода вы найдете в разделе «Circuit Breaker (Размыкатель цепи)» главы 4. Добавление случайного отклонения в алгоритм увеличения задержки я оставляю вам в качестве самостоятельного упражнения[1].

Тайм-ауты

Важность тайм-аутов не всегда очевидна. Тем не менее способность распознать ситуацию, когда запрос вряд ли будет удовлетворен, позволяет клиенту высвободить ресурсы, которые он и другие вышестоящие службы, от имени которых он может действовать, могли бы в противном случае удерживать. Это верно и для служб, которые могут простаивать до тех пор, пока клиент не сдастся.

Например, представьте простую службу, которая посылает запрос в базу данных. Если база данных внезапно замедлится, из-за чего обработка запро-

[1] Делать это здесь я посчитал излишним, но признаюсь, что, возможно, я немного обленился.

са займет несколько секунд, то запросы к службе – каждый из которых удерживает соединение с базой данных – могут накапливаться, что в конечном итоге приведет к исчерпанию пула соединений. Если база данных является общей, то такое замедление может привести к сбою других служб и в конечном итоге к каскадному сбою.

Если бы служба завершала ожидание по тайм-ауту вместо удержания соединения с базой данных, то она могла бы просто ухудшить качество обслуживания, вместо того чтобы сразу выйти из строя.

Иначе говоря, если вы думаете, что неудача неизбежна, то ее стоит поторопить.

Использование контекста Context для реализации тайм-аутов на стороне службы

В главе 4 я представил `context.Context` как идиоматическое средство в языке Go для передачи сигналов наступления крайнего срока и отмены между процессами[1]. Вернитесь назад и прочитайте еще раз раздел «Пакет context» в главе 4, чтобы освежить память.

Кроме того, возможно, вы помните, что далее в той же главе, в разделе «Timeout (Тайм-аут)», мы рассмотрели шаблон *Timeout* (*Тайм-аут*), который использует экземпляр `Context`, чтобы не только позволить процессу прервать ожидание ответа, как только станет ясно, что он может не поступить, но также чтобы уведомить другие функции с производными экземплярами `Context` о прекращении работы и освобождении любых ресурсов, которые они могут удерживать.

Эта способность отменять выполнение функций настолько эффективна, что для функций считается хорошим тоном принимать значение `Context`, если они могут работать дольше, чем вызывающий может позволить себе ждать, что верно почти всегда, когда запрос передается по сети.

По этой причине в стандартной библиотеке Go имеется множество отличных примеров функций, принимающих `Context`. Многие из них находятся в пакете `sql` и являются дополнениями к функциям, которые не принимают `Context`. Например, метод `QueryRow` структуры `DB` имеет эквивалентный метод `QueryRowContext`, принимающий экземпляр `Context`.

Допустим, функция, использующая этот прием для получения имени пользователя по его идентификатору, может выглядеть примерно так:

```
func UserName(ctx context.Context, id int) (string, error) {
    const query = "SELECT username FROM users WHERE id=?"

    dctx, cancel := context.WithTimeout(ctx, 15*time.Second)
    defer cancel()

    var username string
    err := db.QueryRowContext(dctx, query, id).Scan(&username)
```

[1] И технически значений в области видимости запроса, но правильность этого решения является спорной.

```
    return username, err
}
```

Функция `UserName` принимает `context.Context` и целочисленный идентификатор `id` и создает производный контекст `Context` с довольно длинным тайм-аутом. Этот подход гарантирует завершение функции по тайм-ауту и автоматическое освобождение любых открытых соединений через 15 секунд – дольше, чем многие клиенты, вероятно, готовы ждать, – и при этом обеспечивает реакцию на сигналы отмены от вызывающей стороны.

Отзывчивость на внешние сигналы отмены может очень пригодиться. Фреймворк `http` включает еще один замечательный пример, как показано ниже, в реализации функции-обработчика `UserGetHandler`:

```
func UserGetHandler(w http.ResponseWriter, r *http.Request) {
    vars := mux.Vars(r)
    id := vars["id"]

    // Получить контекст запроса. Через этот контекст поступает сигнал отмены.
    // Когда клиент закрывает соединение или когда метод ServeHTTP завершается,
    // запрос отменяется.
    rctx := r.Context()

    ctx, cancel := context.WithTimeout(rctx, 10*time.Second)
    defer cancel()

    username, err := UserName(ctx, id)

    switch {
    case errors.Is(err, sql.ErrNoRows):
        http.Error(w, "no such user", http.StatusNotFound)
    case errors.Is(err, context.DeadlineExceeded):
        http.Error(w, "database timeout", http.StatusGatewayTimeout)
    case err != nil:
        http.Error(w, err.Error(), http.StatusInternalServerError)
    default:
        w.Write([]byte(username))
    }
}
```

Первым делом `UserGetHandler` извлекает контекст запроса вызовом его метода `Context`. Очень удобно, что через этот контекст поступает сигнал отмены, когда клиент закрывает соединение (средствами HTTP/2) или когда метод `ServeHTTP` завершается.

На основе этого контекста создается производный контекст, явным назначением тайм-аута, по истечении которого (через 10 секунд) поступит сигнал отмены.

Поскольку производный контекст передается в функцию `UserName`, мы можем напрямую связать закрытие HTTP-запроса с закрытием соединения с базой данных: если контекст запроса закрывается, то вместе с ним закрываются все производные контексты, что в конечном итоге гарантирует освобождение всех открытых ресурсов.

Прерывание ожидания обработки клиентских запросов HTTP/REST

Во врезке «Ловушка вспомогательных функций» главы 8 я представил одну из ловушек «вспомогательных функций» в пакете `http`, таких как `http.Get` и `http.Post`: они используют тайм-аут по умолчанию. К сожалению, значение тайм-аута по умолчанию равно 0, что в языке Go интерпретируется как «без тайм-аута».

Прием установки тайм-аутов для клиентских методов, описанный в той врезке, заключался в создании собственного экземпляра `Client` с ненулевым тайм-аутом, как показано ниже:

```
var client = &http.Client{
    Timeout: time.Second * 10,
}

response, err := client.Get(url)
```

Этот прием прекрасно работает и фактически отменяет запрос точно так же, как если бы сигнал отмены поступил через его экземпляр `Context`. Но как быть, если нет иного пути, как использовать существующий или производный экземпляр `Context`? Для этого вам понадобится доступ к базовому экземпляру `Context`, который можно получить вызовом `http.NewRequestWithContext` – метода, эквивалентного методу `http.NewRequest`, который позволяет программисту указать контекст, контролирующий жизненный цикл запроса, и свой ответ.

Этот прием не сильно отличается от предыдущего. На самом деле если заглянуть в исходный код метода `Get` в `http.Client`, то можно заметить, что за кулисами он просто использует `NewRequest`:

```
func (c *Client) Get(url string) (resp *Response, err error) {
    req, err := NewRequest("GET", url, nil)
    if err != nil {
        return nil, err
    }

    return c.Do(req)
}
```

Как видите, стандартный метод `Get` вызывает `NewRequest` для создания `*Request`, передавая имя метода и URL (в последнем параметре можно передать необязательный `io.Reader` для получения тела запроса, но здесь он не нужен). Вызов функции `Do` выполняет корректный запрос.

Не считая проверки ошибок и возвращаемого значения, весь метод состоит всего из одного вызова. Казалось бы, реализация аналогичной функциональности, которая также принимает экземпляр `Context`, не представляет особого труда.

Один из способов сделать это – реализовать функцию `GetContext`, которая принимает контекст `Context`:

```
type ClientContext struct {
    http.Client
}

func (c *ClientContext) GetContext(ctx context.Context, url string)
        (resp *http.Response, err error) {

    req, err := http.NewRequestWithContext(ctx, "GET", url, nil)
    if err != nil {
        return nil, err
    }

    return c.Do(req)
}
```

Наша новая функция `GetContext` функционально идентична канонической функции `Get`, за исключением того, что принимает экземпляр `Context`, который используется для вызова `http.NewRequestWithContext` вместо `http.NewRequest`.

Порядок использования нового типа `ClientContext` очень похож на использование стандартного типа `http.Client`, за исключением того, что вместо вызова `client.Get` мы вызываем `client.GetContext` (и, конечно же, передаем экземпляр `Context`):

```
func main() {
    client := &ClientContext{}
    ctx, cancel := context.WithTimeout(context.Background(), 5*time.Second)
    defer cancel()

    response, err := client.GetContext(ctx, "http://www.example.com")
    if err != nil {
        log.Fatal(err)
    }

    bytes, _ := ioutil.ReadAll(response.Body)
    fmt.Println(string(bytes))
}
```

Но насколько правильно работает это решение? В библиотеке нет *подходящего* теста, но мы можем реализовать его самостоятельно, установив крайний срок равным 0 и запустив его:

```
$ go run .
2020/08/25 14:03:16 Get "http://www.example.com": context deadline exceeded
exit status 1
```

Похоже, что все в порядке! Отлично.

Прерывание ожидания обработки клиентских запросов gRPC

Так же как `http.Client`, клиенты gRPC по умолчанию не имеют тайм-аута, но позволяют задавать тайм-ауты явно.

Как было показано в разделе «Реализация клиента gRPC» главы 8, клиенты gRPC обычно устанавливают соединение вызовом функции `grpc.Dial`, которой можно передать список параметров соединения `grpc.DialOption`, созданный с помощью таких функций, как `grpc.WithInsecure` и `grpc.WithBlock`.

Среди этих параметров есть `grpc.WithTimeout`, который можно использовать для настройки тайм-аута соединения с клиентом:

```
opts := []grpc.DialOption{
    grpc.WithInsecure(),
    grpc.WithBlock(),
    grpc.WithTimeout(5 * time.Second),
}
conn, err := grpc.Dial(serverAddr, opts...)
```

Несмотря на кажущееся удобство, метод с использованием `grpc.WithTimeout` давно устарел, в основном из-за того, что этот механизм несовместим (и избыточен), по сравнению с более предпочтительным методом на основе контекста `Context`. Я покажу его здесь для полноты картины.

Параметр `grpc.WithTimeout` устарел и в конечном итоге будет удален. Вместо него используйте `grpc.DialContext` и `context.WithTimeout`.

Вместо этого предпочтительным методом организации тайм-аута соединения gRPC является очень удобная (для нас) функция `grpc.DialContext`, которая принимает экземпляр `context.Context`. На самом деле она несет двойную пользу, потому что методы служб gRPC в любом случае принимают экземпляр `Context`, и нам даже не нужно выполнять никаких дополнительных действий:

```
func TimeoutKeyValueGet() *pb.Response {
    // Установить 5-секундный тайм-аут с помощью контекста.
    ctx, cancel := context.WithTimeout(context.Background(), 5 * time.Second)
    defer cancel()

    // Как и прежде, мы можем настроить другие параметры.
    opts := []grpc.DialOption{grpc.WithInsecure(), grpc.WithBlock()}

    conn, err := grpc.DialContext(ctx, serverAddr, opts...)
    if err != nil {
        grpclog.Fatalf(err)
    }
    defer conn.Close()

    client := pb.NewKeyValueClient(conn)

    // Тот же контекст можно повторно использовать в клиентских вызовах.
    response, err := client.Get(ctx, &pb.GetRequest{Key: key})
    if err != nil {
        grpclog.Fatalf(err)
    }

    return response
}
```

Как видите, `TimeoutKeyValueGet` вместо `grpc.Dial` использует `grpc.DialContext`, которой передается экземпляр `context.Context` с 5-секундным тайм-аутом. В остальном список `opts` идентичен, за исключением того, что теперь в нем отсутствует `grpc.WithTimeout`.

Обратите внимание на вызов метода `client.Get`. Как упоминалось выше, методы службы gRPC принимают параметр контекста `Context`, поэтому мы просто повторно используем существующий экземпляр. Важно отметить, что повторное использование одного и того же экземпляра контекста ограничит обе операции одним и тем же тайм-аутом – время тайм-аута будет истекать независимо от того, как он используется, поэтому обязательно учитывайте это обстоятельство при планировании значений тайм-аута.

Идемпотентность

Как мы обсуждали в начале главы 4, облачные приложения по определению существуют в сетевом мире и подвержены всем его особенностям. А сети, как мы знаем, ненадежны, и сообщения, отправленные по сети, не всегда доходят до адресата вовремя (или вообще не приходят).

Более того, отправив сообщение и не получив ответа, вы не сможете узнать причину этого. Сообщение потерялось на пути к адресату? Адресат получил сообщение, но ответ потерялся на пути к вам? А может быть, все работает нормально и просто путь туда и обратно требует больше времени, чем обычно?

Единственный выход в такой ситуации – отправить сообщение еще раз. Но недостаточно скрестить пальцы и надеяться на лучшее. Важно спланировать эту неизбежность и сделать повторную отправку сообщений безопасной, спроектировав функции *идемпотентными*.

Идею идемпотентности мы коротко рассмотрели в разделе «Что такое идемпотентность, и почему это важно?» главы 5, где определили идемпотентную операцию как операцию, которая имеет такой же эффект после любого количества повторных вызовов. Идемпотентность также является важным свойством любого облачного API, которое гарантирует безопасное повторение любого взаимодействия (см. врезку «Истоки идемпотентности во Всемирной паутине» ниже, где приводятся дополнительные исторические сведения).

Конкретные средства достижения идемпотентности будут отличаться для разных служб, но есть некоторые закономерности, которые мы рассмотрим в оставшейся части этого раздела.

Истоки идемпотентности во Всемирной паутине

Идеи идемпотентности и безопасности, по крайней мере в контексте сетевых служб, были впервые определены еще в 1997 году в стандарте HTTP/1.1[1].

[1] Fielding, R., et al. «Hypertext Transfer Protocol – HTTP/1.1», *Proposed Standard*, RFC 2068, June 1997. https://oreil.ly/28rcs. (Перевод документа на русский язык можно найти по адресу: http://www.codenet.ru/webmast/http/rfc2068/. – *Прим. перев.*)

Интересно отметить, что это новаторское предложение, а также «черновой вариант» стандарта HTTP/1.0, появившийся годом ранее[1], были созданы двумя великими людьми.

Основным автором оригинального проекта HTTP/1.0 (и последним автором предложенного стандарта HTTP/1.1) был сэр Тимоти Джон Бернерс-Ли (Sir Timothy John Berners-Lee), которому приписывают изобретение Всемирной паутины, первого веб-браузера и основных протоколов и алгоритмов, позволяющих масштабировать интернет, за что он был награжден премией ACM Turing Award, рыцарским званием и различными почетными степенями.

Основным автором предложенного стандарта HTTP/1.1 (и вторым автором оригинального проекта HTTP/1.0) был Рой Филдинг (Roy Fielding), тогда аспирант Калифорнийского университета в Ирвайне. Несмотря на то что он был одним из первых авторов Всемирной паутины, Филдинг более известен своей докторской диссертацией, в которой он описал свое изобретение REST[2].

Как сделать службу идемпотентной?

Идемпотентность не встроена в логику какого-либо конкретного фреймворка. Даже в HTTP – и, как следствие, в REST – идемпотентность является соглашением и не обеспечивается явным образом. Вам ничто не мешает – по недосмотру или намеренно – создать неидемпотентную реализацию GET, если вы действительно этого захотите[3].

Одна из причин, почему бывает сложно реализовать идемпотентность, заключается в том, что она полагается на логику, встроенную в основное приложение, а не на REST или gRPC API. Например, если бы в главе 5 мы решили сделать наше хранилище пар ключ/значение совместимым с традиционными операциями CRUD (create, read, update, delete – создание, чтение, обновление, удаление) и, следовательно, неидемпотентным, то мы могли бы реализовать их примерно так:

```
var store = make(map[string]string)

func Create(key, value string) error {
    if _, ok := store[key]; ok {
        return errors.New("duplicate key")
    }

    store[key] = value
    return nil
}

func Update(key, value string) error {
    if _, ok := store[key]; !ok {
        return errors.New("no such key")
```

[1] Berners-Lee, T., et al. «Hypertext Transfer Protocol – HTTP/1.0», Informational, RFC 1945, May 1996. https://oreil.ly/zN7uo.

[2] Fielding, Roy Thomas. «Architectural Styles and the Design of Network-Based Software Architectures». UC Irvine, 2000, pp. 76–106. https://oreil.ly/swjbd.

[3] Вы – чудовище.

```
    }

    store[key] = value
    return nil
}

func Delete(key string) error {
    if _, ok := store[key]; ok {
        return errors.New("no such key")
    }

    delete(store, key)
    return nil
}
```

Эта CRUD-подобная реализация службы может быть создана с вполне благими намерениями, но если вызов какого-то из этих методов потребуется повторить, то результатом будет ошибка. Более того, в проверке текущего состояния заложена избыточная логика, которая не нужна в следующей эквивалентной идемпотентной реализации:

```
var store = make(map[string]string)

func Set(key, value string) {
    store[key] = value
}

func Delete(key string) {
    delete(store, key)
}
```

Эта версия *намного* проще во многих отношениях. Во-первых, нет необходимости в отдельных операциях «создания» и «обновления», поэтому их можно объединить в одну функцию `Set`. Кроме того, отказ от проверки текущего состояния перед каждой операцией сокращает методы, что даст дополнительные преимущества с увеличением сложности службы.

Наконец, если операцию потребуется повторить, то в таком повторе нет ничего страшного. Для функций `Set` и `Delete` несколько идентичных вызовов дадут одинаковый результат. Они идемпотентны.

А как насчет скалярных операций?

«Итак, – можете сказать вы, – это все хорошо для операций, которые либо *выполняются*, либо *не выполняются*, но как насчет более сложных операций? Например, операций со скалярными значениями?»

Справедливый вопрос. В конце концов, операция PUT либо что-то помещает на место, либо нет. Достаточно просто не возвращать ошибку при повторных попытках выполнить PUT. Отлично.

А как насчет такой операции, как «добавить 500 долларов на счет 12345»? Подобный запрос может нести полезную нагрузку в формате JSON, которая выглядит примерно так:

```
{
    "credit":{
        "accountID": 12345,
        "amount": 500
    }
}
```

Повторная попытка выполнить эту операцию приведет к добавлению лишних 500 долларов на счет 12345, и если владелец счета может не сильно возражать против этого, то банку это явно не понравится.

А теперь добавим в полезную нагрузку JSON идентификатор транзакции `transactionID` и посмотрим, что из этого получится:

```
{
    "credit":{
        "accountID": 12345,
        "amount": 500,
        "transactionID": 789
    }
}
```

Для обработки идентификатора транзакции может потребоваться выполнить дополнительную логику, но такой подход дает работоспособное решение проблемы. Запоминая идентификаторы транзакций, получатель сможет обнаруживать и отклонять повторные транзакции. Идемпотентность достигнута!

Избыточность служб

Избыточность – дублирование критически важных компонентов или функций системы с целью повышения ее безотказности – часто является первой линией обороны, когда дело доходит до повышения устойчивости к сбоям.

Мы уже обсуждали один конкретный вид избыточности – избыточность обмена сообщениями, также известную как «повторные попытки», – в разделе «Повтори еще раз: повторные запросы» выше. Однако в этом разделе мы рассмотрим ценность дублирования критически важных компонентов системы, чтобы в случае сбоя одного из них другой или другие экземпляры взяли работу на себя.

В общедоступном облаке это означает развертывание компонента на нескольких экземплярах серверов, в идеале в нескольких зонах или даже в нескольких регионах. В контейнерных платформах, таких как Kubernetes, подобное дублирование можно реализовать простой установкой количества копий в значение больше единицы.

Однако сколь бы интересной ни была эта тема, мы не будем тратить на нее слишком много времени. Репликация сервисов – это вопрос архитектуры,

который подробно освещен во многих других источниках[1]. В конце концов, это книга о языке Go. Но все же было бы упущением написать целую главу об устойчивости и даже не упомянуть об этом.

Будьте внимательны: маскировка неисправностей

Когда сбой системы компенсируется незаметно и не имеет явных проявлений, происходит *маскировка неисправностей*.

Например, представьте систему с тремя узлами, каждый из которых решает часть задач. Если один узел выйдет из строя, а другие смогут компенсировать это, то вы не заметите ничего плохого. Неисправность будет замаскирована.

Маскирование неисправностей может скрывать прогрессирующие сбои и в конечном итоге привести к потере защитной избыточности, часто с внезапным и катастрофическим исходом.

Чтобы предотвратить маскировку неисправностей, важно включать проверки работоспособности служб, которые мы обсудим в разделе «Проверка работоспособности» ниже, сообщающие информацию о работоспособности экземпляров службы.

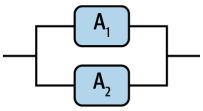
Проектирование избыточности

Усилия, затраченные на проектирование системы, поддерживающей выполнение ее служб в нескольких экземплярах, могут принести значительные выгоды. Насколько значительные? Ну… весьма значительные. Прочитайте следующую врезку, если вас интересует математика, а если нет, то просто доверьтесь мне в этом вопросе.

Безотказность в цифрах

Представьте, что вы создали службу с уровнем доступности «две девятки», или 99 %. Любой запрос к этой системе будет успешно обработан с теоретической вероятностью 0,99, что обозначается как A_s. На самом деле это далеко не лучший показатель, но в том-то и дело.

Какой уровень доступности можно обеспечить, если параллельно запустить два идентичных экземпляра службы[2]? Эту ситуацию можно изобразить так:



[1] Отличным примером может служить книга Хизера Адкинса (Heather Adkins) и других авторов *Building Secure and Reliable Systems: Best Practices for Designing, Implementing, and Maintaining Systems*.

[2] Приготовьтесь. Сейчас начнется.

Как определить уровень доступности этой системы? Фактически мы должны определить вероятность *недоступности* обоих экземпляров. Чтобы ответить на этот вопрос, возьмем произведение вероятностей отказа каждого компонента:

$$U_s = (1 - A_1) \times (1 - A_2).$$

Эту формулу легко обобщить на любое количество компонентов, работающих параллельно. То есть доступность любых N компонентов равна единице минус произведение вероятностей их недоступности:

$$A_s = 1 - \prod_{1}^{N}(1 - A_i).$$

Когда все вероятности A_i равны, то эту формулу можно упростить:

$$A_s = 1 - (1 - A_i)^N.$$

А теперь вернемся к нашему примеру. Итак, при наличии двух компонентов, каждый из которых обеспечивает уровень доступности 99 %, получаем следующее[1]:

$$A_s = 1 - (1 - 0{,}99)^2 = 0{,}9999.$$

99,99 %. Четыре девятки. Улучшение на два порядка – очень неплохо. А если добавить третий экземпляр? Немного дополнив вычисления, получаем интересные результаты, представленные в следующей таблице:

Компоненты	Доступность	Продолжительность простоя в течение года	Продолжительность простоя в течение месяца
Один компонент	99 % («2 девятки»)	3,65 дня	7,31 часа
Два параллельных компонента	99,99 % («4 девятки»)	52,60 минуты	4,38 минуты
Три параллельных компонента	99,9999 % («6 девяток»)	31,56 секунды	2,63 секунды

Невероятно, но три параллельно действующих экземпляра, каждый из которых сам по себе не особенно надежен, могут обеспечить впечатляющие 6 девяток доступности! Вот почему поставщики облачных услуг советуют клиентам развертывать по три копии своих приложений.

А если компоненты расположены последовательно, подобно балансировщику нагрузки, стоящему перед нашими компонентами? Например, так:

В этом случае если какой-то из компонентов выйдет из строя, недоступной окажется вся система. Доступность такой системы определяется как произведение вероятностей доступности отдельных компонентов:

$$A_s = \prod_{1}^{N} A_i.$$

[1] Здесь предполагается, что частота отказов компонентов абсолютно независима, что маловероятно в реальном мире. Относитесь к этому объяснению как к описанию сферического коня в вакууме.

Когда все вероятности A_i равны, эту формулу можно упростить:

$$A_s = A_i^N.$$

И что получится, если установить экземпляр балансировщика нагрузки, обеспечивающий уровень доступности 99,9999 %, перед нашим фантастическим набором служб? Как оказывается, результат совсем не впечатляет:

$$0{,}99 \times 0{,}999999 = 0{,}98999901.$$

Доступность получилась даже ниже уровня доступности балансировщика! Это важно, потому что, как оказывается:

Общая отказоустойчивость последовательной системы не может быть выше отказоустойчивости любой из последовательностей ее подсистем.

Автоматическое масштабирование

Очень часто величина нагрузки на службу меняется со временем. Типичный пример – веб-служба, обслуживающая пользователей, нагрузка на которую увеличивается днем и уменьшается ночью. Если такая служба создавалась с учетом пиковых нагрузок, то она будет понапрасну тратить вычислительные ресурсы и деньги в ночное время. Если она создавалась только с учетом ночных нагрузок, то днем она будет работать с перегрузкой.

Автоматическое масштабирование – это метод, основанный на идее балансировки нагрузки путем автоматического добавления или удаления ресурсов, таких как экземпляры облачных серверов или модули Kubernetes, для динамической корректировки емкости службы в соответствии с текущим спросом. Оно гарантирует соответствие службы различным уровням трафика, ожидаемым или нет.

Как дополнительное преимущество автоматическое масштабирование кластера может сэкономить деньги за счет распределения ресурсов в соответствии с требованиями обслуживания.

Все ведущие поставщики облачных услуг предоставляют механизм масштабирования экземпляров серверов, и большинство их собственных служб явно или неявно поддерживают автоматическое масштабирование. Платформы управления контейнерами, такие как Kubernetes, тоже поддерживают автоматическое масштабирование как в отношении количества подов (горизонтальное автоматическое масштабирование), так и в отношении ограничений доступных им ресурсов – процессоры и память (вертикальное автоматическое масштабирование).

Разные поставщики облачных услуг и платформы управления контейнерами предлагают разные механизмы автоматического масштабирования, поэтому подробное обсуждение приемов сбора метрик и настройки прогнозирования необходимости масштабирования выходит за рамки этой книги. Тем не менее отмечу некоторые ключевые моменты:

- устанавливайте разумные максимумы, чтобы необычно большие всплески трафика (или, не дай бог, каскадные сбои) не привели к полному истощению вашего бюджета. Здесь также могут пригодиться методы

дросселирования и сброса нагрузки, которые обсуждались в разделе «Предотвращение перегрузки» выше;

- минимизируйте время запуска. Если вы используете экземпляры серверов, создавайте их образы заранее, чтобы минимизировать время настройки при запуске. Для Kubernetes эта проблема менее характерна, и все же старайтесь создавать образы контейнеров как можно меньшего размера и с минимальным временем запуска;
- независимо от скорости запуска, процедура масштабирования требует времени. У вашей службы должен быть хотя бы небольшой запас для маневра без масштабирования;
- как обсуждалось в разделе «Отложенное масштабирование: эффективность» главы 7, лучшим считается такой вид масштабирования, в котором никогда не возникает необходимости.

Проверка работоспособности

В разделе «Избыточность служб» выше мы кратко обсудили ценность избыточности – дублирования критически важных компонентов или функций системы с целью повышения общей отказоустойчивости – и ее важность для повышения устойчивости всей системы.

Наличие нескольких экземпляров службы предполагает использование механизма балансировки нагрузки – сервисной сетки или выделенного балансировщика нагрузки, – но что происходит, когда экземпляр службы выходит из строя? Разумеется, что в этом случае балансировщик нагрузки не должен отправлять трафик прежним путем. И как это организовать?

Использовать *проверку работоспособности*. В простейшем и наиболее распространенном случае проверка работоспособности реализуется как конечная точка API, которую клиенты – балансировщики нагрузки, а также службы мониторинга, реестры служб и т. д. – могут использовать, чтобы убедиться в работоспособности экземпляра службы.

Например, служба может предоставлять отдельную конечную точку HTTP (например, с именем `/health` или `/healthz`), которая возвращает `200 OK`, если экземпляр работает исправно, и `503 Service Unavailable`, если это не так. Более сложные реализации могут даже возвращать разные коды состояния в разных ситуациях: реестр службы Consul компании HashiCorp интерпретирует любой код `2XX` как успешный, `429 Too Many Requests` как предупреждение, а все остальные – как отказ.

Наличие конечной точки, сообщающей клиенту об исправности (или неисправности) экземпляра службы, – это, конечно, хорошо, но возникает вопрос: что подразумевается под «работоспособностью» экземпляра?

Проверки работоспособности похожи на светофильтры. Неудачный результат проверки работоспособности означает, что служба не работает, а удачный – что служба, вероятно, «исправна». (Спасибо Синди Шридхарану (Cindy Sridharan)[1].)

[1] Sridharan, Cindy (@copyconstruct). «Health checks are like bloom filters…», 5 Aug 2018, 3:21 AM. Tweet. https://oreil.ly/Qpw3d.

Что подразумевается под «работоспособностью» экземпляра?

Мы часто используем слово «работоспособность» в контексте служб и их экземпляров, но что именно под этим подразумевается? Что ж, как это часто бывает, есть два ответа: простой и сложный – и, наверное, много промежуточных ответов.

Начнем с простого ответа. Согласно существующему определению, экземпляр считается «исправным», когда он «доступен». То есть когда он в состоянии правильно обслужить запрос.

К сожалению, не все так однозначно. А что, если сам экземпляр действует в точности как задумано, но нижестоящая служба работает некорректно? Должна ли проверка работоспособности как-то различать подобные ситуации? Если да, то должен ли балансировщик нагрузки действовать по-разному в каждом случае? Следует ли заменить экземпляр, если причина не в нем, особенно если при этом будут затронуты все экземпляры службы?

К сожалению, на эти вопросы нет простых ответов, поэтому вместо ответов я предлагаю обсудить три наиболее распространенных подхода к проверке работоспособности, а также их достоинства и недостатки. Ваши собственные реализации будут зависеть от потребностей вашей службы и желательного поведения вашего балансировщика нагрузки.

Три типа проверок работоспособности

Когда экземпляр службы выходит из строя, то это обычно происходит по одной из следующих причин:

- локальный сбой, как, например, ошибка в приложении или исчерпание ресурсов – процессоров, памяти, соединений с базой данных и т. д.;
- удаленный сбой в какой-то из зависимостей – в базе данных или другой нижестоящей службе, – влияющий на работу службы.

Эти две широкие категории отказов приводят к трем (да, трем) стратегиям проверки работоспособности, каждая из которых имеет свои маленькие плюсы и минусы.

Проверка жизнеспособности просто возвращает сигнал «успех». Она не предпринимает дополнительных попыток определить состояние службы и ничего не сообщает о службе, кроме того что она принимает запросы и доступна. Но иногда этого достаточно. Подробнее о проверке жизнеспособности мы поговорим в разделе «Проверка жизнеспособности» ниже.

Поверхностная проверка работоспособности идет дальше, чем проверка жизнеспособности, и подтверждает, что экземпляр службы работоспособен. Поверхностная проверка работоспособности проверяет только локальные ресурсы, поэтому она вряд ли обнаружит сбой сразу во многих экземплярах, но она не сможет точно сказать, насколько успешно обработает запрос каждый конкретный экземпляр службы. Подробнее о поверхностной проверке состояния мы поговорим в разделе «Поверхностная проверка работоспособности» ниже.

Глубокая проверка работоспособности позволяет получить гораздо более полное представление о работоспособности экземпляра, потому что проверяет способность экземпляра службы выполнять свою функцию, используя при этом нижестоящие ресурсы, такие как базы данных. Такие проверки могут быть весьма дорогостоящими и подвержены ложным срабатываниям. Мы подробно рассмотрим глубокую проверку работоспособности в разделе «Глубокая проверка работоспособности» ниже.

Проверка жизнеспособности

Конечная точка проверки жизнеспособности всегда возвращает значение «успех», несмотря ни на что. Такая проверка может показаться абсолютно бесполезной – в конце концов, в чем ценность проверки работоспособности, которая ничего не говорит о работоспособности, – тем не менее проверка жизнеспособности все же может дать некоторую полезную информацию, подтверждая, что:

- экземпляр службы принимает запросы и возвращает ответы;
- экземпляр доступен по сети;
- любой брандмауэр, группа безопасности или другие конфигурации определены правильно.

Конечно, такая простота имеет свои недостатки. Отсутствие активной логики проверки работоспособности не позволяет судить, действительно ли экземпляр службы способен выполнять свои функции.

С другой стороны, проверка жизнеспособности легко реализуется. Вот пример такой проверки с использованием пакета `net/http`:

```
func healthLivenessHandler(w http.ResponseWriter, r *http.Request) {
    w.WriteHeader(http.StatusOK)
    w.Write([]byte("OK"))
})

func main() {
    r := mux.NewRouter()
    http.HandleFunc("/healthz", healthLivenessHandler)
    log.Fatal(http.ListenAndServe(":8080", r))
}
```

Этот пример показывает, как мало нужно для проверки жизнеспособности. Он создает и регистрирует конечную точку `/healthz`, которая ничего не делает, кроме как возвращает код `200 OK` (и, присмотритесь внимательно, текст `OK`).

При использовании пакета `gorilla/mux` на проверку работоспособности может повлиять любая зарегистрированная промежуточная функция (например, функция сброса нагрузки, как рассказывалось в разделе «Сброс нагрузки» выше)!

Поверхностная проверка работоспособности

Поверхностная проверка работоспособности идет дальше простой проверки жизнеспособности, подтверждая, что экземпляр службы, вероятно, способен

работать, но никак не исследует работоспособность нижестоящих зависимостей службы, таких как база данных или другая служба.

Поверхностная проверка работоспособности может оценить любое количество условий, способных отрицательно сказаться на работоспособности службы, включая (но не ограничиваясь):

- достаточность основных локальных ресурсов (память, процессор, соединения с базой данных);
- возможность чтения или записи локальных данных; в ходе этой проверки проверяется наличие свободного дискового пространства, разрешений и аппаратных сбоев;
- наличие вспомогательных процессов, таких как процессы мониторинга или обновления.

Поверхностная проверка работоспособности более точная, чем проверка жизнеспособности, а ее специфика означает, что любые сбои вряд ли повлияют сразу на все множество экземпляров службы[1]. Однако поверхностная проверка подвержена ложным срабатываниям: если проверяемая служба не работает из-за какой-либо проблемы, связанной с внешним ресурсом, то поверхностная проверка ее не заметит. Приобретая конкретность, вы жертвуете чувствительностью.

В примере ниже показано, как может выглядеть поверхностная проверка работоспособности. Здесь проверяется способность службы читать и записывать данные на локальный диск:

```
func healthShallowHandler(w http.ResponseWriter, r *http.Request) {
    // Создать тестовый файл.
    // Следующая инструкция создаст файл с именем, таким как /tmp/shallow-123456
    tmpFile, err := ioutil.TempFile(os.TempDir(), "shallow-")
    if err != nil {
        http.Error(w, err.Error(), http.StatusServiceUnavailable)
        return
    }
    defer os.Remove(tmpFile.Name())

    // Убедиться, что в файл можно что-то записать.
    text := []byte("Check.")
    if _, err = tmpFile.Write(text); err != nil {
        http.Error(w, err.Error(), http.StatusServiceUnavailable)
        return
    }

    // Убедиться, что файл можно закрыть.
    if err := tmpFile.Close(); err != nil {
        http.Error(w, err.Error(), http.StatusServiceUnavailable)
        return
    }

    w.WriteHeader(http.StatusOK)
}
```

[1] Хотя мне доводилось видеть и такое.

```
func main() {
    r := mux.NewRouter()
    http.HandleFunc("/healthz", healthShallowHandler)
    log.Fatal(http.ListenAndServe(":8080", r))
}
```

Этот код одновременно проверяет наличие свободного дискового пространства, разрешений на запись и исправность оборудования, что может быть очень полезным для тестирования, особенно если службе требуется записывать данные в кеш на диске или в другие временные файлы.

Внимательный читатель может заметить, что здесь запись производится в каталог, по умолчанию используемый для временных файлов. В Linux это каталог `/tmp`, который во многих дистрибутивах реализован как RAM-диск (диск в оперативной памяти). Также такая проверка может пригодиться для тестирования, но если вы хотите проверить возможность записи на диск в Linux, то следует выбрать другой каталог, иначе вы будете проверять совсем не то, что нужно.

Глубокая проверка работоспособности

Глубокая проверка работоспособности напрямую проверяет способность службы взаимодействовать со смежными системами. Она позволяет получить еще более полное представление о работоспособности экземпляра, выявляя проблемы с зависимостями, такие как недопустимые учетные данные, потеря подключения к хранилищам данных или другие неожиданные проблемы в сети.

Однако глубокая проверка работоспособности может стоить чрезвычайно дорого, требовать много времени и нагружать зависимости, особенно если проверки выполняются слишком часто.

Не пытайтесь протестировать *каждую* зависимость в своих проверках работоспособности: сосредоточьтесь на тех, которые абсолютно необходимы для нормальной работы службы.

При тестировании нескольких нижестоящих зависимостей оценивайте их одновременно, если это возможно.

Более того, поскольку отказ зависимости интерпретируется как отказ экземпляра, глубокие проверки особенно подвержены ложным срабатываниям. Вместе с более низкой специфичностью, по сравнению с поверхностной проверкой – проблемы с зависимостями будут влиять на работу всего множества экземпляров службы, – вы получаете потенциальную возможность предупредить каскадный сбой.

При использовании глубоких проверок работоспособности рекомендуется применять такие стратегии, как размыкание цепи (которую мы рассмотрели в разделе «Размыкание цепи» выше), а ваш балансировщик нагрузки должен оставаться «открытым при отказе» (обсудим в разделе «Открытие при отказе» ниже), когда это возможно.

Вот тривиальный пример реализации глубокой проверки работоспособности, которая оценивает базу данных путем вызова гипотетической функции `GetUser` службы:

```
func healthDeepHandler(w http.ResponseWriter, r *http.Request) {
    // Получить контекст запроса и добавить 5-секундный тайм-аут
    ctx, cancel := context.WithTimeout(r.Context(), 5*time.Second)
    defer cancel()

    // service.GetUser -- это гипотетический метод службы,
    // который выполняет запрос к базе данных
    if err := service.GetUser(ctx, 0); err != nil {
        http.Error(w, err.Error(), http.StatusServiceUnavailable)
        return
    }

    w.WriteHeader(http.StatusOK)
}

func main() {
    r := mux.NewRouter()
    http.HandleFunc("/healthz", healthDeepHandler)
    log.Fatal(http.ListenAndServe(":8080", r))
}
```

В идеале тестирование зависимости должно использовать настоящую системную функцию, но при этом быть легковесным в максимально разумной степени. В этом примере предполагается, что функция `GetUser` выполняет запрос к базе данных, который удовлетворяет обоим этим критериям[1].

«Настоящие» запросы обычно предпочтительнее простой проверки доступности базы данных по двум причинам. Во-первых, они более точно соответствуют тому, что делает служба. Во-вторых, они позволяют использовать время выполнения запроса как меру работоспособности базы данных. Предыдущий пример действительно делает это, хотя и в очень упрощенной форме, устанавливая тайм-аут с помощью `Context`, но вообще в проверку можно включить более сложную логику.

Открытие при отказе

Что случится, если все ваши экземпляры одновременно решат, что они неработоспособны? При использовании глубокой проверки работоспособности такая ситуация вполне вероятна (и может даже случаться регулярно). В зависимости от настройки балансировщика нагрузки в какой-то момент может оказаться, что у вас ноль действующих экземпляров службы, обслуживающих трафик, что может вызвать сбой, способный распространиться по всей системе.

К счастью, некоторые балансировщики нагрузки справляются с этим, используя трюк «открытия при отказе». Если балансировщик нагрузки, от-

[1] Это мнимая функция, поэтому просто согласимся, что все это – правда.

крывающийся при отказе, *не имеет* работоспособных целей, то есть если проверки работоспособности *всех* его целей завершаются неудачей, то он направит трафик всем целям.

Это несколько противоречит здравому смыслу, но такой подход делает более безопасным использование глубоких проверок работоспособности, позволяя трафику продолжать движение, даже когда в нижестоящей зависимости возникают проблемы.

Итоги

Писать эту главу было очень интересно. Об устойчивости можно говорить до бесконечности, и уж тем более до бесконечности можно обсуждать приемы реализации ее поддержки. Мне пришлось сделать сложный выбор о том, что включить в эту главу, а что нет. И все же она оказалась немного длиннее, чем я предполагал, но я вполне доволен результатом. Это разумный компромисс между слишком малым и слишком большим количеством информации, а также между практическим опытом и фактическими реализациями на Go.

Мы узнали, что означает отказ (сбой) системы и как сложные системы выходят из строя (по одному компоненту за раз). Это привело нас к обсуждению особенно неприятного, но распространенного режима отказа: каскадного сбоя. В случае каскадного сбоя попытки системы восстановиться ускоряют ее коллапс. Мы рассмотрели общие меры предотвращения каскадных сбоев на стороне сервера: дросселирование и сброс нагрузки.

Повторные попытки при появлении ошибок могут способствовать повышению устойчивости службы, но, как мы видели на примере DynamoDB, при простом применении они же могут способствовать каскадным сбоям. Мы рассмотрели также меры, которые можно предпринять на стороне клиента, включая размыкание цепи, тайм-ауты и особенно алгоритмы с экспоненциальным увеличением задержки. Было показано несколько красивых графиков. Я потратил много времени на их построение.

Все это привело к обсуждению темы избыточности служб, о том, как она влияет на отказоустойчивость (с некоторыми математическими выкладками для интересующихся), а также о том, когда и как использовать автоматическое масштабирование.

Конечно, бессмысленно говорить об автоматическом масштабировании, не сказав ни слова о «работоспособности» ресурса. Мы постарались ответить на вопрос, что значит «работоспособный» экземпляр и как эту работоспособность проверить. Мы рассмотрели три вида проверок работоспособности и взвесили их плюсы и минусы, уделив особое внимание соотношению их относительной чувствительности/специфичности.

В главе 10 мы ненадолго отвлечемся от вопросов эксплуатации и перейдем к вопросам управляемости: искусству и науке замены колес на движущемся автомобиле.

Глава 10

Управляемость

> Всем известно, что отладка в два раза сложнее написания программы. То есть если вы вложили все свои знания и умения в написание программы, то как вы будете ее отлаживать[1]?
>
> – Брайан Керниган (Brian Kernighan),
> *The Elements of Programming Style* (1978)

В идеальном мире вам никогда не придется развертывать новую версию службы или (не дай бог!) останавливать всю систему, чтобы исправить или изменить ее в соответствии с новыми требованиями.

С другой стороны, в идеальном мире существовали бы единороги, и четверо из пяти стоматологов рекомендовали бы нам есть пирожные на завтрак[2].

Мы живем не в идеальном мире. Но хотя единороги, возможно, никогда не появятся[3], мы не должны мириться с миром, в котором должны обновлять свой код всякий раз, когда требуется изменить поведение системы.

Вероятно, вам все равно придется вносить изменения в код, чтобы обновить базовую логику, но системы можно строить так, чтобы вы – или, что особенно важно, кто-то еще – могли менять модели поведения без необходимости повторного кодирования развертывания.

Вы наверняка помните этот важный атрибут облачных систем, представленный еще в разделе «Управляемость» в главе 1, где мы определили его как простоту, с которой можно изменить поведение системы для обеспечения безопасности, бесперебойной работы и соответствия меняющимся требованиям.

Несмотря на кажущуюся простоту, управляемость – это нечто большее, чем вы думаете. Это не только файлы конфигурации (хотя они, безусловно, являются ее частью). В этой главе вы узнаете, что подразумевается под управляемой системой, и познакомитесь с некоторыми методами и реализациями, которые помогут вам построить систему, способную меняться почти так же быстро, как и требования к ней.

---

1 Kernighan, Brian W., and P. J. Plauger. *The Elements of Programming Style*. McGraw-Hill, 1978. (*Керниган Б., Плоджер Ф.* Элементы стиля программирования // Радио и связь. 1984. – *Прим. перев.*)

2 Staff, America's Test Kitchen. *Perfect Pie: Your Ultimate Guide to Classic and Modern Pies, Tarts, Galettes, and More*. America's Test Kitchen, 2019. https://oreil.ly/rl5TP.

3 Люди добились потрясающих успехов в генной инженерии, поэтому еще не все потеряно!

Управляемость – это не удобство сопровождения

Можно сказать, что понятия управляемости и удобства сопровождения (или ремонтопригодности) в некоторой степени «пересекаются», потому что оба связаны с легкостью модификации системы. Основное отличие между ними состоит в том, *как именно* модифицируется система:

- *управляемость* описывает простоту изменения поведения системы, обычно без изменения ее кода. Иначе говоря, насколько легко изменить систему *извне*;
- *удобство сопровождения* (*ремонтопригодность*) описывает простоту изменения или расширения возможностей программной системы или компонента, исправления ошибок или дефектов либо повышения производительности[1], обычно путем внесения изменений в код. Иначе говоря, насколько легко изменить систему *изнутри*.

Что такое управляемость, и почему она важна?

Когда речь заходит об управляемости, многие представляют себе единственную службу. Насколько легко настроить мою службу? Есть ли в ней все необходимые параметры и настройки, которые могут понадобиться?

Однако, сосредоточив все внимание на единственном компоненте, есть риск упустить из виду более важный момент. Управляемость не заканчивается на границе службы. Чтобы система была управляемой, необходимо рассматривать всю систему.

Найдите минутку и подумайте об управляемости сложной системы. Насколько легко изменить ее поведение? Можно ли изменять поведение ее компонентов независимо друг от друга? Насколько легко заменить их более новыми версиями при необходимости? Как понять, что система действительно управляемая?

Управляемость охватывает все возможные аспекты поведения системы, а ее функции можно условно разделить на четыре большие категории[2]:

Конфигурация и управление

Важно, чтобы система – и каждый ее компонент – легко настраивалась для достижения оптимальной доступности и производительности. Некоторые системы требуют регулярного или даже постоянного управления, поэтому наличие правильных «рычагов и кнопок» абсолютно необходимо. Именно на этом мы сосредоточим основное внимание в данной главе.

Мониторинг, журналирование и оповещение

Эти функции помогают следить за способностью системы выполнять свою работу и имеют решающее значение для эффективного управления ею.

---

[1] «Systems and Software Engineering: Vocabulary». ISO/IEC/IEEE 24765:2010(E), 15 Dec. 2010. https://oreil.ly/NInvC.

[2] Radle, Byron, et al. «What Is Manageability?» *NI*, National Instruments, 5 Mar. 2019. https://oreil.ly/U3d7Q.

В конце концов, как иначе можно узнать, что система требует управления? Но какими бы важными ни были эти функции для управляемости, мы не будем обсуждать их в этой главе – им будет посвящена отдельная глава 11 «Наблюдаемость».

Развертывание и обновление
Даже при отсутствии изменений в коде возможность с легкостью развернуть, обновить, откатить и масштабировать компоненты системы представляет большую ценность, особенно когда управлять приходится множеством систем. Эта легкость важна для первоначального развертывания, но ее важность продолжает увеличиваться на протяжении всего жизненного цикла системы и приносит выгоды каждый раз, когда систему необходимо обновить. Благодаря отсутствию внешних окружений времени выполнения и самодостаточности выполняемых артефактов язык Go особенно ярко выделяется в этой области.

Обнаружение и учет служб
Ключевой особенностью облачных систем является их распределенный характер. Очень важно, чтобы компоненты могли быстро и безошибочно обнаруживать друг друга. Эта функция называется *обнаружением служб*. Поскольку обнаружение служб – это архитектурная черта, а не программная, мы не будем углубляться в нее здесь.

Так как эта книга посвящена языку Go, а не архитектуре[1], основное внимание в ней уделяется реализации служб. Только поэтому – *а не потому, что это более важно,* – большая часть этой главы также будет посвящена конфигурации на уровне служб. К сожалению, подробное обсуждение этой темы выходит за рамки данной книги[2].

Управление сложными вычислительными системами – как правило, непростая задача и требует много времени, а затраты на управление ими могут намного превышать затраты на базовое оборудование и программное обеспечение. По определению, управляемая система может эффективно управляться и, следовательно, с меньшими затратами. Даже без учета затрат на управление, одно только снижение сложности может иметь огромное влияние на вероятность человеческой ошибки, облегчая и ускоряя ее исправление. Таким образом, управляемость напрямую влияет на безотказность, доступность и безопасность, что делает ее ключевым элементом надежности системы.

Настройка приложения

Самая основная функция управляемости – поддержка возможности настройки приложения. В идеальном приложении все, что может различаться в раз-

[1] По крайней мере, так я сказал моим редакторам. Привет, Амелия! Привет, Зан!

[2] Все это меня здорово огорчает. Это важные темы, но мы должны сосредоточиться.

ных окружениях – промежуточном, производственном, разработки и т. д., – должно четко отделяться от кода и может каким-то образом определяться извне.

Вы, возможно, помните, что в документе «Двенадцать факторов», описывающем набор из двенадцати правил по созданию веб-приложений и представленном в главе 6, довольно много говорится по этому поводу. Фактически третье правило (см. раздел «III. Конфигурация» в главе 6) касается исключительно конфигурации приложения. Оно гласит:

> Храните конфигурацию в среде выполнения.

Как мы уже отмечали, документ «Двенадцать факторов» настаивает на хранении всех настроек в переменных окружения. По этому поводу существует множество мнений, но за годы, прошедшие с момента его публикации, отрасль, похоже, достигла общего консенсуса в отношении того, что действительно важно.

Конфигурация должна строго отделяться от кода
: Конфигурация – все, что может различаться в разных окружениях, – всегда должна четко отделяться от кода. Конфигурация может существенно различаться в разных вариантах развертывания, а код – нет. Конфигурацию не следует встраивать в код. Никогда.

Конфигурации должны храниться в системе управления версиями
: Хранение конфигураций в системе управления версиями – отдельно от кода – позволяет при необходимости быстро откатить изменения в настройках и воссоздать или восстановить систему. Некоторые платформы развертывания, такие как Kubernetes, делают подобное разделение простым и естественным, предлагая примитивы для определения конфигурации, такие как ConfigMap.

В настоящее время еще довольно часто можно встретить приложения, настройки которых определяются через переменные окружения, но не менее часто можно увидеть приложения, настраиваемые через флаги командной строки и конфигурационные файлы разных форматов. Некоторые приложения даже поддерживают сразу несколько вариантов. В следующих разделах мы рассмотрим некоторые из них, перечислим достоинства и недостатки, а также исследуем примеры их реализации на Go.

Рекомендуемые приемы организации конфигураций

При создании приложения у вас на выбор есть множество вариантов организации, реализации и развертывания конфигураций. Однако, как показывает мой личный опыт, наилучшие долгосрочные и краткосрочные результаты дают следующие общие практики.

Храните конфигурации в системе управления версиями

Да, я повторяюсь, но это правило стоит повторить. Файлы конфигурации должны храниться в системе управления версиями и извлекаться оттуда перед развертыванием. Это позволяет просмотреть их перед развертыванием, быстро включать их в сборку и при необходимости быстро откатывать изменения. Это также может пригодиться, если (и когда) потребуется воссоздать и восстановить систему.

Не используйте свои нестандартные форматы

Используйте стандартные форматы для оформления своих файлов конфигурации, такие как JSON, YAML или TOML. Мы рассмотрим некоторые из них далее в этой главе. Если по каким-то причинам у вас возникнет *необходимость* использовать свой нестандартный формат, убедитесь, что вам по душе идея поддерживать его – и заставлять иметь дело с ним всех сопровождающих – в будущем.

Используйте нулевые значения как значения по умолчанию

Не используйте ненулевые значения по умолчанию без необходимости. Это очень хорошее правило; есть даже соответствующий постулат Go[1]. Поведение, возникающее в результате неопределенной конфигурации, должно быть приемлемым, разумным и обыденным. Простая минимальная конфигурация снижает вероятность ошибок.

Настройка с использованием переменных окружения

Как отмечалось в главе 6 и выше в этой главе, для определения значений конфигурации документ «Двенадцать факторов» рекомендует использовать переменные окружения. В этом есть определенное достоинство: переменные окружения поддерживаются повсеместно, они гарантируют, что настройки не будут случайно включены в код, а их использование обычно требует меньше кода, чем использование файла конфигурации. Они также идеально подходят для небольших приложений.

С другой стороны, процесс определения и передачи переменных окружения может быть неудобным, утомительным и многословным. Некоторые приложения поддерживают возможность определения переменных окружения в файле, но такой подход сводит на нет саму цель использования переменных окружения.

Также некоторые проблемы может создавать неявный характер переменных окружения. Поскольку нет простого способа проверить существование и поведение переменных окружения, просмотрев файл конфигурации или проверив вывод справки, приложения, реагирующие на них, иногда могут оказаться сложнее в использовании, а ошибки в них труднее отлаживать.

---

[1] Pike, Rob. «Go Proverbs». Gopherfest, 18 Nov. 2015, YouTube. https://oreil.ly/5bOxW.

Как и большинство языков высокого уровня, Go предлагает простой доступ к переменным окружения через стандартный пакет `os`, в котором для этой цели имеется функция `os.Getenv`:

```
name := os.Getenv("NAME")
place := os.Getenv("CITY")

fmt.Printf("%s lives in %s.\n", name, place)
```

Функция `os.Getenv` извлекает значение переменной окружения с указанным именем, и если переменная отсутствует, то возвращает пустую строку. Если в приложении важно отличать пустое и отсутствующее значение, то для этого можно использовать функцию `os.LookEnv`, которая вместе со значением переменной возвращает также логическое значение, сообщающее, была ли установлена данная переменная окружения (значение `false` говорит о том, что переменная не была установлена):

```
if val, ok := os.LookupEnv(key); ok {
    fmt.Printf("%s=%s\n", key, val)
} else {
    fmt.Printf("%s not set\n", key)
}
```

Эта функциональность предлагает минимум возможностей, но их вполне достаточно для многих (если не для большинства) целей. Для реализации более сложных параметров, например типизированных или имеющих значение по умолчанию, есть несколько отличных сторонних пакетов. Особенно большой популярностью пользуется библиотека Viper (`spf13/viper`), о которой мы поговорим в разделе «Viper: швейцарский армейский нож конфигурационных пакетов» ниже.

Настройка с использованием аргументов командной строки

Методы передачи настроек через аргументы командной строки определенно заслуживают внимания, по крайней мере для небольших и несложных приложений. В конце концов, эти настройки имеют более явный характер, и описание подробностей об их использовании обычно доступно через параметр `--help`.

Стандартный пакет flag

В стандартной библиотеке Go имеется пакет `flag`, реализующий простой синтаксический анализ командной строки. Пакет `flag` не отличается особым богатством возможностей, зато он довольно прост в использовании и, в отличие от `os.Getenv`, поддерживает типизацию.

Возьмем, к примеру, следующую программу, которая использует `flag` для чтения и вывода значений флагов командной строки:

```
package main

import (
    "flag"
    "fmt"
)

func main() {
    // Объявить строковый флаг со значением по умолчанию "foo"
    // и коротким описанием. Он возвращает указатель на строку.
    strp := flag.String("string", "foo", "a string")

    // Объявить числовой и логический флаги.
    intp := flag.Int("number", 42, "an integer")
    boolp := flag.Bool("boolean", false, "a boolean")

    // Вызвать flag.Parse() для синтаксического анализа командной строки.
    flag.Parse()

    // Вывести значения флагов и конечные позиционные аргументы.
    fmt.Println("string:", *strp)
    fmt.Println("integer:", *intp)
    fmt.Println("boolean:", *boolp)
    fmt.Println("args:", flag.Args())
}
```

Как видно в предыдущем примере, пакет `flag` позволяет регистрировать флаги командной строки с типами, значениями по умолчанию и краткими описаниями, а также присваивать их значения переменным. Увидеть описание этих флагов можно, запустив программу с флагом `-help`:

```
$ go run . -help
Usage of /var/folders/go-build618108403/exe/main:
  -boolean
        a boolean
  -number int
        an integer (default 42)
  -string string
        a string (default "foo")
```

В справке выводится список всех доступных флагов. Запустив программу со всеми этими флагами, мы получим следующий результат:

```
$ go run . -boolean -number 27 -string "A string." Other things.
string: A string.
integer: 27
boolean: true
args: [Other things.]
```

Все прекрасно работает! Однако пакет `flag` имеет пару проблем, ограничивающих его полезность.

Во-первых, как вы могли заметить, синтаксис определения флагов выглядит несколько… нестандартно. Многие из нас ожидают, что интерфейс

командной строки будет соответствовать стандарту аргументов GNU (https://oreil.ly/evqk4), согласно которому длинные имена параметров записываются с двумя дефисами в начале (`--version`) и имеют короткие однобуквенные эквиваленты (`-v`).

Во-вторых, пакет `flag` просто выполняет синтаксический анализ флагов (хотя, честно говоря, он не претендует на что-либо большее), и он не такой мощный, как мог бы быть. Было бы неплохо, если бы имелась возможность отображать команды в функции.

Парсер командной строки Cobra

Пакет `flags` отлично подходит для случаев, когда требуется просто проанализировать флаги, но для построения более мощного интерфейса командной строки лучше использовать что-то другое, например пакет Cobra (https://oreil.ly/4oCyH). Пакет Cobra обладает набором возможностей, которые делают его популярным выбором для создания полноценного интерфейса командной строки. Он используется в ряде известных проектов, включая Kubernetes, CockroachDB, Docker, Istio и Helm.

Помимо флагов, полностью совместимых с POSIX (короткие *и* длинные версии), Cobra также поддерживает вложенные подкоманды и автоматически генерирует вывод справки (`--help`) и подсказки для автозаполнения в различных командных оболочках. Он также интегрируется с Viper, о котором мы расскажем в разделе «Viper: швейцарский армейский нож конфигурационных пакетов» ниже.

Как нетрудно догадаться, основным недостатком Cobra является более высокая сложность по сравнению с пакетом `flag`. Реализация программы из раздела «Стандартный пакет flag» с использованием Cobra выглядит следующим образом:

```
package main

import (
    "fmt"
    "os"
    "github.com/spf13/cobra"
)

var strp string
var intp int
var boolp bool

var rootCmd = &cobra.Command{
    Use:  "flags",
    Long: "A simple flags experimentation command, built with Cobra.",
    Run:  flagsFunc,
}

func init() {
    rootCmd.Flags().StringVarP(&strp, "string", "s", "foo", "a string")
    rootCmd.Flags().IntVarP(&intp, "number", "n", 42, "an integer")
```

```
    rootCmd.Flags().BoolVarP(&boolp, "boolean", "b", false, "a boolean")
}

func flagsFunc(cmd *cobra.Command, args []string) {
    fmt.Println("string:", strp)
    fmt.Println("integer:", intp)
    fmt.Println("boolean:", boolp)
    fmt.Println("args:", args)
}

func main() {
    if err := rootCmd.Execute(); err != nil {
        fmt.Println(err)
        os.Exit(1)
    }
}
```

В отличие от версии, использующей пакет `flag`, которая просто читает некоторые флаги и выводит результаты, эта программа имеет более сложную структуру с несколькими отдельными частями.

Во-первых, целевые переменные объявляются в области видимости пакета, а не функции. Это необходимо, потому что они должны быть доступны как в функции `init`, так и в функции, реализующей логику команды.

Далее создается структура `rootCmd` типа `cobra.Command`, которая представляет корневую команду. Для представления каждой команды и подкоманды, доступной через интерфейс командной строки, должен создаваться свой экземпляр `cobra.Command`. В поле `Use` определяется короткое однострочное сообщение, описывающее команду, а в поле `Long` – длинное сообщение, отображаемое в выводе справки. Поле `Run` определяет функцию типа `func(cmd *Command, args []string)`, которая представляет фактическую реализацию команды.

Обычно команды создаются в функции `init`. В данном случае мы добавляем три флага – строку, число и логическое значение – к нашей корневой команде вместе с короткими флагами, значениями по умолчанию и описаниями.

Для каждой команды автоматически генерируется вывод справки, который можно получить с помощью флага `--help`:

```
$ go run . --help
A simple flags experimentation command, built with Cobra.

Usage:
  flags [flags]

Flags:
  -b, --boolean        a boolean
  -h, --help           help for flags
  -n, --number int     an integer (default 42)
  -s, --string string  a string (default "foo")
```

Справка не только выглядит осмысленной, но и красиво отформатирована! Но будет ли работать созданная нами команда так, как мы ожидаем? Запуск

программы (с использованием стандартного стиля оформления флагов) дает следующий результат:

```
$ go run . --boolean --number 27 --string "A string." Other things.
string: A string.
integer: 27
boolean: true
args: [Other things.]
```

Этот вывод идентичен выводу предыдущей версии; мы достигли желаемого. Но это всего лишь одна команда. Одно из преимуществ Cobra заключается в возможности определения *подкоманд*.

Что это значит? Возьмем, к примеру, команду `git`. В этом примере `git` будет корневой командой. Сама по себе она почти ничего не делает, но у нее есть ряд подкоманд – `git clone`, `git init`, `git blame` и т. д., – которые связаны между собой, но каждая представляет отдельную операцию.

Cobra позволяет реализовать такую возможность, представляя команды как древовидную структуру. Каждая команда и подкоманда (включая корневую команду) представлены отдельным экземпляром `cobra.Command`. Они связываются друг с другом с помощью функции `(c *Command) AddCommand(cmds ...*Command)`. Рассмотрим эту возможность в следующем примере, превратив команду `flags` в подкоманду новой корневой команды `cng` (от *Cloud Native Go*).

Для этого сначала нужно переименовать исходный экземпляр `rootCmd` в `flagsCmd`. Затем добавим атрибут `Short`, чтобы определить краткое описание для вывода в справке, а все остальное оставим без изменений. Но теперь нам нужна новая корневая команда, поэтому создадим и ее:

```
var flagsCmd = &cobra.Command{
    Use:   "flags",
    Short: "Experiment with flags",
    Long:  "A simple flags experimentation command, built with Cobra.",
    Run:   flagsFunc,
}

var rootCmd = &cobra.Command{
    Use:  "cng",
    Long: "A super simple command.",
}
```

Теперь у нас есть две команды: корневая команда `cng` и подкоманда `flags`. Следующим шагом добавим подкоманду `flags` в корневую команду, чтобы она находилась в дереве команд сразу под корнем. Обычно это делается в функции `init`, как показано ниже:

```
func init() {
    flagsCmd.Flags().StringVarP(&strp, "string", "s", "foo", "a string")
    flagsCmd.Flags().IntVarP(&intp, "number", "n", 42, "an integer")
    flagsCmd.Flags().BoolVarP(&boolp, "boolean", "b", false, "a boolean")

    rootCmd.AddCommand(flagsCmd)
}
```

В этой функции init мы все так же вызываем три метода Flags, за исключением того, что теперь они вызываются для flagsCmd.

Новым здесь является вызов метода AddCommand, добавляющий flagsCmd в rootCmd в качестве подкоманды. Мы можем вызвать AddCommand столько раз, сколько потребуется, с разными экземплярами Command, чтобы добавить необходимое число подкоманд (или подподкоманд, или подподподкоманд).

Теперь, когда мы сообщили пакету Cobra о новой подкоманде flags, информация о ней появится в сгенерированном выводе справки:

```
$ go run . --help
A super simple command.

Usage:
  cng [command]

Available Commands:
  flags       Experiment with flags
  help        Help about any command

Flags:
  -h, --help  help for cng

Use "cng [command] --help" for more information about a command.
```

Теперь, если верить этой справке, мы имеем корневую команду верхнего уровня cng, у которой есть две подкоманды: команда flags и автоматически сгенерированная подкоманда help, которая позволяет пользователю получить описание любой подкоманды. Например, help flags выведет описание и инструкцию по использованию для подкоманды flags:

```
$ go run . help flags
A simple flags experimentation command, built with Cobra.

Usage:
  cng flags [flags]

Flags:
  -b, --boolean        a boolean
  -h, --help           help for flags
  -n, --number int     an integer (default 42)
  -s, --string string  a string (default "foo")
```

Ну разве не здорово?

Это был простенький пример, демонстрирующий лишь некоторые возможности библиотеки Cobra, которых намного больше и вполне достаточно, чтобы создать надежный набор параметров конфигурации. Если вам интересно узнать больше о пакете Cobra и как с его помощью создавать мощные интерфейсы командной строки, загляните в его репозиторий GitHub (https://oreil.ly/oy7EN) и в документацию на GoDoc (https://oreil.ly/JOeoJ).

Настройка с использованием файлов

И последний вариант определения конфигурации, который мы рассмотрим, – конфигурационные файлы.

Конфигурационные файлы имеют много преимуществ перед переменными окружения, особенно для сложных приложений. Настройки в них определяются более явно и понятно, и есть возможность логически группировать и аннотировать поведение. Часто, чтобы понять, как использовать конфигурационный файл, достаточно рассмотреть его структуру или пример использования.

Конфигурационные файлы особенно полезны, когда требуется управлять большим количеством параметров, что является преимуществом перед переменными окружения и флагами командной строки. В частности, использование флагов командной строки может приводить к необходимости вводить довольно длинные команды, которые трудно читать и конструировать.

Однако файлы – не лучшее решение. В зависимости от окружения их масштабное распространение в кластере может быть проблемой. Эту проблему можно ослабить, настроив единый «источник истины», такой как распределенное хранилище пар ключ/значение, например etcd или HashiCorp Consul, или центральный репозиторий исходного кода, откуда процедура развертывания будет автоматически извлекать свою конфигурацию, но это добавляет сложности и зависимости от других ресурсов.

К счастью, большинство платформ управления контейнерами предоставляют специализированные ресурсы для размещения конфигурации, такие как объект ConfigMap в Kubernetes, которые в значительной степени решают проблему распространения.

Кроме того, существуют десятки форматов, которые используются для оформления конфигурационных файлов на протяжении многих лет, но в последние годы особую популярность завоевали два формата: JSON и YAML. В следующих разделах мы подробно рассмотрим каждый из них и порядок их использования в Go.

Наша структура конфигурационных данных

Прежде чем продолжить обсуждение форматов, рассмотрим два основных способа декодирования конфигурационных файлов:

- конфигурационные ключи и значения могут отображаться в соответствующие поля структуры определенного типа. Например, конфигурация, содержащая атрибут `host: localhost`, может быть отображена в структуру со строковым полем `Host`;
- конфигурационные данные могут быть преобразованы в один или несколько, возможно вложенных, ассоциативных массивов типа `map[string]interface{}`. Этот подход можно использовать при работе с произвольными конфигурациями, но он несколько неудобен.

Если вы заранее знаете, как будет выглядеть ваша конфигурация (как это обычно бывает), то самым простым будет первый подход – отображение

конфигурации в структуру данных, специально созданную для этой цели. Конечно, можно использовать и произвольные схемы конфигурации, но это может быть очень утомительно, и поступать так не рекомендуется.

Итак, в оставшейся части данного раздела примеры наших конфигураций будут соответствовать следующей структуре `Config`:

```
type Config struct {
    Host string
    Port uint16
    Tags map[string]string
}
```

Чтобы поле структуры можно было кодировать/декодировать с помощью любого пакета кодирования, его имя *должно* начинаться с заглавной буквы, дабы указать, что оно экспортируется своим пакетом.

Все последующие примеры будут начинаться с определения экземпляра структуры `Config`, иногда с дополнительными тегами, в зависимости от формата.

Формат JSON

Формат JSON (JavaScript Object Notation – форма записи объектов JavaScript) был изобретен в начале 2000-х годов, когда возникла потребность в современном формате обмена данными взамен XML и других форматов, использовавшихся в то время. Он основан на подмножестве языка JavaScript, что делает его относительно удобочитаемым для человека и простым для анализа и создания компьютерами. Кроме того, этот формат поддерживает семантику списков и ассоциаций, отсутствующую в XML.

Каким бы распространенным и успешным ни был формат JSON, у него есть некоторые недостатки. Многие считают его менее удобным, чем YAML. Его синтаксис строг и легко может быть нарушен отсутствующей или поставленной не в том месте запятой. Более того, он не поддерживает комментарии.

Однако из форматов, представленных в этой главе, это единственный формат, для которого имеется поддержка в стандартной библиотеке Go.

Далее следует очень короткое введение в кодирование и декодирование данных в формат JSON и из него. Более полное описание формата можно найти в статье «JSON and Go» Эндрю Герранда (Andrew Gerrand) в блоге *The Go Blog* (https://oreil.ly/6Uvl2).

КОДИРОВАНИЕ В ФОРМАТ JSON Первым шагом к пониманию, как правильно декодировать JSON (или любой другой формат конфигурации), является понимание правил *кодирования* (маршаллинга) в него. Это может показаться странным, особенно в разделе, посвященном чтению файлов конфигурации, но понимание, как происходит кодирование, важно для общего понимания правил работы с форматом JSON и использования средств создания, тестирования и отладки файлов конфигурации[1].

[1] Хитрый ход, не находите?

Кодирование и декодирование JSON поддерживаются стандартным пакетом `encoding/json`, который содержит множество вспомогательных функций для кодирования, декодирования, форматирования, проверки и других операций с JSON.

Среди них функция `json.Marshal`, которая принимает значение `v` типа `interface{}` и возвращает массив `[]byte`, содержащий представление `v` в формате JSON:

```
func Marshal(v interface{}) ([]byte, error)
```

Иначе говоря, функция принимает значение и возвращает его представление в формате JSON.

Эта функция настолько же проста в использовании, как кажется. Например, если у вас есть экземпляр `Config`, то вы можете передать его в `json.Marshal` и получить представление в формате JSON:

```
c := Config{
    Host: "localhost",
    Port: 1313,
    Tags: map[string]string{"env": "dev"},
}

bytes, err := json.Marshal(c)

fmt.Println(string(bytes))
```

Если не возникнет никаких ошибок, то переменная `err` получит значение `nil`, а в `bytes` будет записан массив `[]byte` с представлением структуры в формате JSON. В данном примере `fmt.Println` выведет:

```
{"Host":"localhost","Port":1313,"Tags":{"env":"dev"}}
```

Функция `json.Marshal` производит обход значения `v` рекурсивно, поэтому любые вложенные структуры будут закодированы как вложенные фрагменты JSON.

Это было совсем не больно! Однако, создавая файл конфигурации, было бы неплохо отформатировать его для удобочитаемости. К счастью, `encoding/json` включает еще одну функцию, `json.MarshalIndent`, которая возвращает «отформатированный» текст JSON:

```
func MarshalIndent(v interface{}, prefix, indent string) ([]byte, error)
```

Порядок использования `json.MarshalIndent` почти не отличается от использования `json.Marshal`, разве что первая принимает дополнительные строковые параметры `prefix` и `indent`, назначение которых иллюстрирует следующий пример:

```
bytes, err := json.MarshalIndent(c, "", "   ")
fmt.Println(string(bytes))
```

Он выводит именно то, что мы надеялись увидеть:

```
{
   "Host": "localhost",
   "Port": 1313,
   "Tags": {
      "env": "dev"
   }
}
```

В результате получается красиво отформатированный текст JSON, понятный людям вроде меня[1]. Это очень полезный метод формирования файлов конфигурации!

ДЕКОДИРОВАНИЕ ФОРМАТА JSON Теперь, узнав, как представить структуру данных в формате JSON, давайте посмотрим, как преобразовать (выполнить демаршаллинг) JSON в существующую структуру данных.

Для этого используем функцию с подходящим названием `json.Unmarshal`:

```
func Unmarshal(data []byte, v interface{}) error
```

Функция `json.Unmarshal` анализирует текст в формате JSON, представленный в виде массива `data`, и сохраняет результат в экземпляре структуры, на который указывает `v`. Важно отметить, что если в `v` передать `nil` или что-то, что не является указателем, то `json.Unmarshal` вернет ошибку.

Но каким должен быть тип `v`? В идеале это должен быть указатель на структуру данных, поля которой точно соответствуют структуре JSON. Конечно, произвольный фрагмент JSON можно преобразовать в неструктурированный ассоциативный массив, как будет показано во врезке «Декодирование произвольного JSON» ниже, но делать это следует только в том случае, если у вас действительно нет другого выбора.

Однако, как вы вскоре увидите, если есть тип данных, отражающий структуру фрагмента JSON, то `json.Unmarshal` сможет обновить его экземпляр напрямую. Для этого сначала нужно создать экземпляр, в котором будут храниться декодированные данные:

```
c := Config{}
```

Теперь, имея хранилище, можно вызвать `json.Unmarshal` и передать ей массив `[]byte` с данными в формате JSON и указатель на `c`:

```
bytes := []byte(`{"Host":"127.0.0.1","Port":1234,"Tags":{"foo":"bar"}}`)
err := json.Unmarshal(bytes, &c)
```

Если `bytes` содержит допустимый фрагмент в формате JSON, то `err` получит значение `nil`, а данные из `bytes` будут сохранены в структуре `c`. Если теперь вывести содержимое `c`, то вы должны увидеть следующее:

```
{127.0.0.1 1234 map[foo:bar]}
```

---

[1] И вас, конечно же!

Отлично! А что случится, если структура JSON неточно соответствует типу Go? Давайте выясним это:

```
c := Config{}
bytes := []byte(`{"Host":"127.0.0.1", "Food":"Pizza"}`)
err := json.Unmarshal(bytes, &c)
```

Интересно отметить, что этот вызов не вернет ошибку, как можно было бы ожидать. Вместо этого `c` будет содержать следующие значения:

```
{127.0.0.1 0 map[]}
```

Как видите, значение `Host` было установлено, но значение `Food`, для которого не нашлось соответствия в структуре `Config`, было проигнорировано. Как оказывается, `json.Unmarshal` декодирует только те поля, которые присутствуют в целевом типе. Такое поведение особенно полезно, когда требуется выделить лишь несколько конкретных полей из большого объекта JSON.

Декодирование произвольного JSON

Как уже упоминалось в разделе «Наша структура конфигурационных данных» выше, есть возможность декодировать и использовать произвольные фрагменты JSON, но это может быть довольно утомительно, и поэтому поступать так рекомендуется, только если структура объекта JSON неизвестна заранее.

Возьмем, к примеру, следующий совершенно произвольный фрагмент JSON:

```
bytes := []byte(`{"Foo":"Bar", "Number":1313, "Tags":{"A":"B"}}`)
```

Не зная заранее его структуру, можно воспользоваться функцией `json.Unmarshal` и декодировать его в экземпляр `interface{}`:

```
var f interface{}
err := json.Unmarshal(bytes, &f)
```

Вывод полученного значения `f` с помощью `fmt.Println` даст интересный результат:

```
map[Number:1313 Foo:Bar Tags:map[A:B]]
```

Значением `f` оказался ассоциативный массив со строковыми ключами и значениями, хранящимися как пустые интерфейсы. Функционально это значение идентично следующему:

```
f := map[string]interface{}{
    "Foo": "Bar",
    "Number": 1313,
    "Tags": map[string]interface{}{"A": "B"},
}
```

Но даже притом что базовое значение имеет тип `map[string]interface{}`, переменная `f` по-прежнему имеет тип `interface{}`. Поэтому для доступа к значениям необходимо использовать приведение типа:

```
m := f.(map[string]interface{})

fmt.Printf("<%T> %v\n", m, m)
fmt.Printf("<%T> %v\n", m["Foo"], m["Foo"])
```

```
fmt.Printf("<%T> %v\n", m["Number"], m["Number"])
fmt.Printf("<%T> %v\n", m["Tags"], m["Tags"])
```

Предыдущий фрагмент выведет следующее:

```
<map[string]interface {}> map[Number:1313 Foo:Bar Tags:map[A:B]]
<string> Bar
<float64> 1313
<map[string]interface {}> map[A:B]
```

ФОРМАТИРОВАНИЕ ПОЛЕЙ С ИСПОЛЬЗОВАНИЕМ ТЕГОВ ПОЛЕЙ СТРУКТУРЫ В действительности маршаллинг выполняется с использованием механизма рефлексии, с помощью которого анализируется исходная структура данных и генерируется соответствующий ее типу фрагмент JSON. Имена полей структуры прямо используются как ключи в JSON, а значения полей структуры – как значения JSON. Демаршаллинг работает фактически так же, только в обратном порядке.

Что происходит при маршаллинге структуры с нулевым значением? Как оказывается, выполняя маршаллинг значения `Config{}`, например, вы получаете фрагмент JSON:

```
{"Host":"","Port":0,"Tags":null}
```

Честно говоря, выглядит не особенно красиво. Неужели вообще нужно выводить пустые значения?

Точно так же поля структуры должны экспортироваться для записи или чтения, то есть их имена должны начинаться с заглавной буквы. Означает ли это, что у нас нет выбора и мы вынуждены использовать заглавные буквы в именах полей?

К счастью, ответ на оба вопроса – «нет».

Go поддерживает *теги полей структуры* – короткие строки в объявлении структуры после объявления типов полей, – которые позволяют добавлять метаданные к определенным полям структур. Теги полей чаще всего используются пакетами кодирования для изменения порядка кодирования и декодирования на уровне полей.

Теги полей структуры в Go – это специальные строки, содержащие одну или несколько пар ключ/значение, заключенные в обратные кавычки:

```
type User struct {
    Name string `example:"name"`
}
```

В этом примере поле `Name` структуры снабжено тегом `example:"name"`. Доступ к этим тегам во время выполнения можно получить с помощью пакета `reflect`, но чаще всего теги используются в роли директив кодирования и декодирования.

Пакет `encoding/json` поддерживает несколько таких тегов. В типичном случае в теге поля структуры используется ключ `json` и значение, определяющее имя поля, за которым может следовать список параметров, перечисленных

через запятую. Имя может быть пустым, в таком случае имя поля не переопределяется.

Далее перечислены некоторые параметры, поддерживаемые пакетом `encoding/json`.

Настройка ключей JSON

По умолчанию имена полей структуры сопоставляются с ключами JSON с учетом регистра. Тег переопределяет имя по умолчанию, принимая в качестве такового первое (или единственное) значение в списке параметров тега.

Пример: `CustomKey string \`json:" custom_key "\``

Пропуск пустых значений

По умолчанию поля структуры всегда отображаются в JSON, даже если они пустые. Применение параметра `omitempty` приведет к тому, что поля с нулевыми значениями будут пропускаться. Обратите внимание на запятую перед `omitempty`!

Пример: `OmitEmpty string \`json:", omitempty "\``

Игнорирование полей

Если в теге поля указать параметр `-` (дефис), то такое поле всегда будет игнорироваться в процессе кодирования и декодирования.

Пример: `IgnoredName string \`json:" - "\``

Вот пример структуры, использующей все теги, перечисленные выше:

```
type Tagged struct {
    // Поле CustomKey будет включено в JSON как ключ "custom_key".
    CustomKey string `json:"custom_key"`

    // Поле OmitEmpty будет включено в JSON как ключ "OmitEmpty" (по умолчанию),
    // но только если содержит ненулевое значение.
    OmitEmpty string `json:",omitempty"`

    // Поле IgnoredName всегда будет игнорироваться.
    IgnoredName string `json:"-"`

    // Поле TwoThings будет включено в JSON как ключ "two_things",
    // но только если содержит ненулевое значение.
    TwoThings string `json:"two_things,omitempty"`
}
```

За дополнительной информацией о том, как `json.Marshal` кодирует данные, обращайтесь к описанию функции в документации на golang.org (https://oreil.ly/5QeJ4).

Формат YAML

YAML (YAML Ain't Markup Language – YAML – это не язык разметки)[1] – это расширяемый формат файлов, пользующийся популярностью в таких проектах,

[1] Серьезно, именно так и расшифровывается эта аббревиатура.

как Kubernetes, где требуются сложные иерархические конфигурации. Он очень выразительный, хотя синтаксис может быть немного хрупким, а конфигурации в этом формате могут начать терять удобочитаемость с увеличением масштаба.

В отличие от JSON, который изначально создавался как формат обмена данными, YAML по своей природе является языком определения конфигураций. Интересно отметить, что YAML 1.2 является расширенным множеством JSON, и эти два формата в значительной степени взаимно конвертируемы. Тем не менее YAML имеет некоторые преимущества перед JSON: он позволяет определениям ссылаться на самих себя, использовать встроенные блочные литералы, а также поддерживает комментарии и сложные типы данных.

В отличие от JSON, YAML не поддерживается стандартными библиотеками Go. Однако есть несколько сторонних пакетов YAML, стандартный выбор – Go-YAML (https://oreil.ly/yhERJ). Версия Go-YAML 1 начиналась в 2014 году как внутренний проект в Canonical, перед которым стояла задача переноса известной библиотеки `libyaml` с языка C на язык Go. Это исключительно зрелый и хорошо поддерживаемый проект. Его синтаксис напоминает синтаксис `encoding/json`.

КОДИРОВАНИЕ В ФОРМАТ YAML Использование Go-YAML для кодирования данных *во многом* похоже на кодирование в формат JSON. Сигнатуры функций `Marshal` обоих пакетов идентичны. Как и ее эквивалент в `encoding/json`, функция `yaml.Marshal` в Go-YAML тоже принимает значение `interface{}` и возвращает фрагмент в формате YAML в виде значения `[]byte`:

```
func Marshal(v interface{}) ([]byte, error)
```

Так же как в подразделе «Кодирование в формат JSON», я покажу процесс кодирования на примере экземпляра структуры `Config`, которую передам в `yaml.Marshal`, чтобы получить фрагмент YAML:

```
c := Config{
    Host: "localhost",
    Port: 1313,
    Tags: map[string]string{"env": "dev"},
}

bytes, err := yaml.Marshal(c)
```

И снова, если не возникнет никаких ошибок, переменная `err` получит значение `nil`, а в `bytes` будет записан массив `[]byte` с представлением структуры в формате YAML. Вывод строкового значения `bytes` даст следующий результат:

```
host: localhost
port: 1313
tags:
  env: dev
```

Так же как ее эквивалент в `encoding/json`, функция `Marshal` в Go-YAML производит обход значения `v` рекурсивно. Любые найденные составные типы –

массивы, срезы, ассоциативные массивы и структуры – будут соответствующим образом преобразованы и включены в выходной результат в виде вложенных элементов YAML.

ДЕКОДИРОВАНИЕ ФОРМАТА YAML Мы уже видели сходство функций `Marshal` в `encoding/json` и Go-YAML. Такое же сходство наблюдается и между функциями `Unmarshal` в этих двух пакетах:

```
func Unmarshal(data []byte, v interface{}) error
```

Функция `yaml.Unmarshal` анализирует данные в формате YAML и сохраняет их в структуру, на которую ссылается `v`. Если в `v` передать `nil` или другое значение, не являющееся указателем, то `yaml.Unmarshal` вернет ошибку. Сходство более чем очевидно, как можно заметить в следующем примере:

```
// Внимание: отступы должны оформляться пробелами, а не символами табуляции.
bytes := []byte(`
host: 127.0.0.1
port: 1234
tags:
    foo: bar
`)

c := Config{}
err := yaml.Unmarshal(bytes, &c)
```

Точно так же, как в подразделе «Декодирование формата JSON» выше, мы передаем функции `yaml.Unmarshal` указатель на экземпляр `Config`, поля которого соответствуют полям, найденным в YAML. Если теперь вывести содержимое `c`, то вы должны увидеть следующее:

```
{127.0.0.1 1234 map[foo:bar]}
```

Есть и другие сходства в поведении между `encoding/json` и Go-YAML:

- оба пакета игнорируют атрибуты в исходном документе, которые отсутствуют в структуре, переданной в функцию `Unmarshal`. Такое поведение особенно полезно, когда требуется выделить лишь несколько конкретных полей из большого объекта, но оно также может стать «проблемой»: если вы забудете экспортировать поле структуры, то `Unmarshal` будет просто молча игнорировать его;
- оба пакета способны выполнять демаршаллинг произвольных данных, если в `Unmarshal` передать указатель на `interface{}`. Однако, в отличие от `json.Unmarshal`, которая возвращает `map[string]interface{}`, `yaml.Unmarshal` вернет `map[interface{}]interface{}`. Разница незначительная, но она является еще одним потенциальным источником проблем!

ТЕГИ ПОЛЕЙ СТРУКТУРЫ ДЛЯ YAML В дополнение к «стандартным» тегам полей структуры – ключам, `omitempty` и `-` (дефис), – подробно описанным в подразделе «Форматирование полей с использованием тегов полей структуры» выше, Go-YAML поддерживает два дополнительных тега, используемых для маршаллинга YAML.

Потоковый стиль

Поля с параметром `flow` будут преобразовываться с использованием потокового стиля (https://oreil.ly/zyUpd). Это может пригодиться для анализа структур, последовательностей и ассоциативных массивов.

Пример: `Flow map[string]string \`yaml:"flow"\``

Встраивание структур и ассоциативных массивов

При наличии параметра `inline` все поля структур или ключи ассоциативных массивов обрабатываются, как если бы они были частью внешней структуры. Ключи ассоциативных массивов не должны конфликтовать с именами других полей структуры.

Пример: `Inline map[string]string \`yaml:",inline"\``

Вот пример структуры, использующей все теги, перечисленные выше:

```
type TaggedMore struct {
    // Поле Flow будет включено в YAML в соответствии с "потоковым" стилем
    // (удобно для структур, последовательностей и ассоциативных массивов).
    Flow map[string]string `yaml:"flow"`

    // При встраивании структуры или ассоциативного массива все поля
    // или ключи обрабатываются, как если бы они были частью внешней
    // структуры. Ключи ассоциативных массивов не должны конфликтовать
    // с именами других полей структуры.
    Inline map[string]string `yaml:",inline"`
}
```

Как видите, синтаксис тегов также не отличается от синтаксиса тегов JSON, только вместо префикса `json` в тегах Go-YAML используется префикс `yaml`.

Наблюдение за изменениями в конфигурационных файлах

При использовании конфигурационных файлов часто возникает ситуация, когда необходимо изменить настройки работающей программы. Если программа явно не наблюдает за изменениями и не загружает их, то такую программу приходится перезапускать, чтобы она прочитала новые настройки, что в лучшем случае неудобно, а в худшем – вызывает простои.

Рано или поздно вам придется решить, как ваша программа должна реагировать на такие изменения.

Первый (и самый простой) вариант – ничего не делать и просто ждать, когда программу потребуется перезапустить. На самом деле это довольно распространенный выбор, потому что он гарантирует отсутствие следов предыдущей конфигурации. Кроме того, этот подход позволяет программе «быстро выйти из строя», если в файл конфигурации закрадется ошибка: программе достаточно просто вывести гневное сообщение об ошибке и отказаться от запуска.

Однако некоторые предпочитают добавить в свою программу логику, которая обнаруживает изменения в конфигурационном файле и перезагружает его.

АВТОМАТИЧЕСКАЯ ЗАГРУЗКА ИЗМЕНИВШЕЙСЯ КОНФИГУРАЦИИ Чтобы организовать автоматическую загрузку новых настроек при каждом изменении конфигурационного файла, вам придется спланировать эту операцию заранее.

Для начала нужно предусмотреть единый глобальный экземпляр структуры с конфигурацией. В следующих примерах мы будем использовать экземпляр типа `Config`, представленного в разделе «Наша структура конфигурационных данных» выше. В более крупных проектах ее можно поместить в отдельный пакет `config`:

```
var config Config
```

Очень часто можно встретить код, в котором каждому методу и функции явно передается параметр `config`. Мне приходилось видеть такой код достаточно часто, чтобы понять, насколько этот антишаблон может усложнить жизнь. Кроме того, поскольку в такой ситуации конфигурация оказывается в N разных местах вместо одного, то это может существенно усложнить введение в действие обновленной конфигурации.

После создания глобального экземпляра `config` структуры `Config` нужно добавить логику чтения конфигурационного файла и передачи его содержимого в структуру. Вот пример реализации такой логики в виде функции `loadConfiguration`:

```
func loadConfiguration(filepath string) (Config, error) {
    dat, err := ioutil.ReadFile(filepath) // Загрузить файл как []byte
    if err != nil {
        return Config{}, err
    }

    config := Config{}

    err = yaml.Unmarshal(dat, &config)    // Выполнить маршаллинг
    if err != nil {
        return Config{}, err
    }

    return config, nil
}
```

Наша функция `loadConfiguration` действует почти так, как обсуждалось в разделе «Формат YAML» выше, за исключением того, что она использует функцию `ioutil.ReadFile` из стандартной библиотеки `io/ioutil`, чтобы загрузить содержимое файла, которое затем передается в вызов `yaml.Unmarshal`. Формат YAML был выбран совершенно произвольно[1]. Реализация изменения конфигурации в формате JSON будет выглядеть практически так же.

Теперь, написав логику загрузки конфигурационного файла в каноническую структуру, нужно добавить что-то, что будет вызывать эту логику при каждом получении уведомления об изменении файла. Для этого добавим

[1] Даже притом, что я просто *обожаю* JSON.

функцию startListening, которая будет следить за появлением данных в канале updates:

```
func startListening(updates <-chan string, errors <-chan error) {
    for {
        select {
        case filepath := <-updates:
            c, err := loadConfiguration(filepath)
            if err != nil {
                log.Println("error loading config:", err)
                continue
            }
            config = c

        case err := <-errors:
            log.Println("error watching config:", err)
        }
    }
}
```

Как видите, startListening принимает два канала: updates, через который поступает имя изменившегося файла (предположительно конфигурационного файла), и канал errors.

Она наблюдает за обоими каналами в инструкции select внутри бесконечного цикла, поэтому, если конфигурационный файл изменится, в канал updates будет отправлено его имя и затем передано функции, загружающей конфигурацию. Если loadConfiguration не вернет ошибку, отличную от nil, то возвращаемый ею экземпляр Config заменит текущий.

Теперь перейдем на уровень выше. У нас есть функция init, которая извлекает каналы из функции watchConfig и передает их в функцию startListening, которая запускается как сопрограмма:

```
func init() {
    updates, errors, err := watchConfig("config.yaml")
    if err != nil {
        panic(err)
    }

    go startListening(updates, errors)
}
```

Но что это за функция watchConfig? Подробностей мы пока не знаем, но мы разберемся с этим в следующих нескольких разделах. Пока известно только, что она реализует некоторую логику наблюдения за конфигурацией и имеет следующую сигнатуру:

```
func watchConfig(filepath string) (<-chan string, <-chan error, error)
```

Функция watchConfig, независимо от фактической реализации, возвращает два канала – строковый канал updates, через который она посылает путь к изменившемуся конфигурационному файлу, и канал errors для уведомле-

ния о недопустимой конфигурации, а также значение ошибки, сообщающее о фатальной ошибке при запуске.

Реализовать `watchConfig` можно несколькими способами, каждый из которых имеет свои плюсы и минусы. Рассмотрим два наиболее распространенных из них.

ПРОВЕРКА ИЗМЕНЕНИЙ В КОНФИГУРАЦИИ Регулярная проверка изменений в конфигурационных файлах является распространенным способом наблюдения за ними. Стандартная реализация использует таймер `time.Ticker`, при срабатывании которого вычисляется хеш конфигурационного файла, и если хеш изменился, то производится перезагрузка конфигурации.

В Go поддерживается несколько алгоритмов хеширования, которые доступны в пакете `crypto`, в своих собственных подпакетах, и удовлетворяют интерфейсам `crypto.Hash` и `io.Writer`.

Например, стандартная реализация SHA256 в Go находится в `crypto/sha256`. Чтобы воспользоваться ею, нужно вызвать функцию `sha256.New`, дабы получить новый экземпляр `sha256.Hash`, и затем записать в него исходные данные, используя методы интерфейса `io.Writer`. После этого следует вызвать метод `Sum`, чтобы получить требуемый хеш:

```
func calculateFileHash(filepath string) (string, error) {
    file, err := os.Open(filepath) // Открыть файл для чтения
    if err != nil {
        return "", err
    }
    defer file.Close()             // Гарантировать закрытие файла!

    hash := sha256.New()           // Создать экземпляр Hash из crypto/sha256
    if _, err := io.Copy(hash, file); err != nil {
        return "", err
    }

    sum := fmt.Sprintf("%x", hash.Sum(nil)) // Получить хеш

    return sum, nil
}
```

Хеш для конфигурационного файла генерируется в три этапа. Сначала мы получаем исходные данные в форме `[]byte` с помощью `io.Reader`. В этом конкретном примере используется `io.File`. Затем полученные данные копируются из `io.Reader` в экземпляр `sha256.Hash` вызовом `io.Copy`. И наконец, вызовом метода `hash.Sum` извлекается получившийся хеш.

Теперь, когда у нас есть функция `calculateFileHash`, создание нашей реализации `watchConfig` – это лишь вопрос использования таймера `time.Ticker` для проверки хеша с некоторой частотой и передачи любых положительных результатов (или ошибок) в соответствующий канал:

```
func watchConfig(filepath string) (<-chan string, <-chan error, error) {
    errs := make(chan error)
    changes := make(chan string)
```

```
    hash := ""

    go func() {
        ticker := time.NewTicker(time.Second)

        for range ticker.C {
            newhash, err := calculateFileHash(filepath)
            if err != nil {
                errs <- err
                continue
            }

            if hash != newhash {
                hash = newhash
                changes <- filepath
            }
        }
    }()

    return changes, errs, nil
}
```

Подход на основе периодической проверки имеет некоторые преимущества. Он прост в реализации, что всегда является большим плюсом, и работает в любой операционной системе. Но самое, пожалуй, интересное состоит в том, что поскольку при хешировании обрабатывается только содержимое конфигурации, этот подход можно обобщить для обнаружения изменений в таких местах, как удаленные хранилища пар ключ/значение, которые технически не являются файлами.

К сожалению, метод периодической проверки может быть довольно расточительным с точки зрения вычислений, особенно для больших файлов. Кроме того, по своей природе этот метод имеет небольшую задержку между изменением файла и обнаружением этого изменения. При работе только с локальными файлами было бы эффективнее наблюдать за уведомлениями файловой системы на уровне операционной системы, которые мы обсудим в следующем разделе.

НАБЛЮДЕНИЕ ЗА УВЕДОМЛЕНИЯМИ ФАЙЛОВОЙ СИСТЕМЫ Периодическая проверка изменений в общем и целом работает неплохо, но у этого метода есть некоторые недостатки. В зависимости от варианта использования может быть эффективнее наблюдать за уведомлениями файловой системы на уровне операционной системы.

Однако этот подход осложняется тем фактом, что каждая операционная система использует свой механизм уведомлений. К счастью, пакет fsnotify (https://oreil.ly/ziw4J) предоставляет готовую абстракцию, поддерживающую большинство операционных систем.

Чтобы организовать наблюдение за одним или несколькими файлами с помощью этого пакета, нужно вызвать функцию `fsnotify.NewWatcher` и получить новый экземпляр `fsnotify.Watcher`, а затем с помощью метода `Add` этого экземпляра зарегистрировать контролируемые файлы. `Watcher` предоставляет

два канала, Events и Errors, через которые передаются уведомления о событиях с файлами и об ошибках соответственно.

Вот, например, как мы могли бы организовать наблюдение за нашим конфигурационным файлом, если бы выбрали такой способ:

```
func watchConfigNotify(filepath string) (<-chan string, <-chan error, error) {
    changes := make(chan string)

    watcher, err := fsnotify.NewWatcher()   // Получить экземпляр fsnotify.Watcher
    if err != nil {
        return nil, nil, err
    }

    err = watcher.Add(filepath)             // Зарегистрировать наш конфигурационный
    if err != nil {                         // файл для наблюдения
        return nil, nil, err
    }

    go func() {
        changes <- filepath                 // Первая проверка ВСЕГДА сообщает о наличии
                                            // изменений

        for event := range watcher.Events { // Обход событий в watcher
            if event.Op & fsnotify.Write == fsnotify.Write {
                changes <- event.Name
            }
        }
    }()

    return changes, watcher.Errors, nil
}
```

Обратите внимание на инструкцию `event.Op & fsnotify.Write == fsnotify.Write`, в которой используется поразрядное «И» (`&`) для фильтрации событий «записи». Это делается по той простой причине, что уведомление `fsnotify.Event` потенциально может сообщать сразу о нескольких операциях, каждая из которых представлена одним битом в беззнаковом целом. Например, уведомление о выполнении сразу двух операций `fsnotify.Write` (`2`, в двоичном виде: `0b00010`) и `fsnotify.Chmod` (`16`, в двоичном виде: `0b10000`) даст в результате значение `event.Op`, равное `18` (в двоичном виде: `0b10010`). Поскольку `0b10010 & 0b00010 = 0b00010`, поразрядное «И» позволяет гарантировать, что `event.Op` включает операцию `fsnotify.Write`.

Viper: швейцарский армейский нож конфигурационных пакетов

Viper (`spf13/viper`; https://oreil.ly/pttZM) позиционируется как законченное решение для обслуживания конфигураций приложений на Go, и небезосновательно. Помимо всего прочего, эта библиотека позволяет настраивать приложение с помощью различных механизмов и форматов, в том числе с учетом приоритета.

Явно установленные значения
: Имеет приоритет над всеми другими методами и может пригодиться на этапе тестирования.

Флаги командной строки
: Библиотека Viper была разработана как дополнение к пакету Cobra, представленному в разделе «Парсер командной строки Cobra» выше.

Переменные окружения
: Viper включает полноценную поддержку переменных окружения. Важно отметить, что имена переменных в Viper интерпретируются с учетом регистра!

Конфигурационные файлы в нескольких форматах
: Из коробки Viper поддерживает форматы JSON и YAML, используя пакеты, представленные выше; а также TOML, HCL, INI, envfile и Java Properties. Кроме того, она может записывать конфигурацию в файлы для начальной загрузки и даже наблюдать за изменениями в файлах и перечитывать их.

Удаленные хранилища пар ключ/значение
: Viper может обращаться к хранилищам пар ключ/значение, таким как etcd или Consul, и наблюдать за изменениями в них.

Она также поддерживает значения по умолчанию и типизированные переменные, которые обычно не поддерживаются стандартными пакетами.

Но имейте в виду, что такое богатство возможностей превращает Viper из молотка в кувалду, которая тянет за собой множество зависимостей. Если ваша цель – небольшое и легковесное приложение, то Viper может оказаться слишком большим довеском.

Явно устанавливаемые значения в Viper

В библиотеке Viper имеется функция `viper.Set` для явной установки значений, например из флагов командной строки или внутри логики приложения. Такая возможность может очень пригодиться на этапе тестирования:

```
viper.Set("Verbose", true)
viper.Set("LogFile", LogFile)
```

Явно установленные значения имеют наивысший приоритет и переопределяют значения, установленные другими механизмами.

Работа с флагами командной строки в Viper

Библиотека Viper была разработана как дополнение к библиотеке Cobra (https://oreil.ly/67LFI), которую мы кратко обсудили в контексте конструирования интерфейсов командной строки в разделе «Парсер командной строки Cobra» выше. Тесная интеграция с Cobra упрощает привязку флагов командной строки к конфигурационным параметрам.

В Viper имеется функция `viper.BindPFlag`, которая позволяет привязать отдельный флаг командной строки к определенному ключу, и `viper.BindPFlags`, которая связывает полный набор флагов, используя в качестве ключей длинные имена флагов.

Поскольку фактическое значение конфигурационного параметра устанавливается при обращении к привязке, а не при вызове, `viper.BindPFlag` можно вызвать в функции инициализации, как показано ниже:

```
var rootCmd = &cobra.Command{ /* опущено для краткости */ }

func init() {
    rootCmd.Flags().IntP("number", "n", 42, "an integer")
    viper.BindPFlag("number", rootCmd.Flags().Lookup("number"))
}
```

В этом примере мы объявляем `&cobra.Command` и определяем целочисленный флаг с именем «number». Обратите внимание, что здесь вместо `IntVarP` используется метод `IntP`, потому что нет необходимости сохранять значение флага в переменной, когда Cobra используется таким образом. Затем, вызывая функцию `viper.BindPFlag`, мы связываем флаг «number» с одноименным конфигурационным параметром.

После привязки (и анализа флагов командной строки) значение привязанного ключа можно получить из Viper с помощью функции `viper.GetInt`:

```
n := viper.GetInt("number")
```

Работа с переменными окружения в Viper

Viper предоставляет несколько функций для работы с переменными окружения в роли источника конфигурации. Первая из них – `viper.BindEnv`. Она используется для привязки конфигурационного параметра к переменной окружения:

```
viper.BindEnv("id")                  // Связать "id" с переменной "ID"
viper.BindEnv("port", "SERVICE_PORT") // Связать "port" с переменной "SERVICE_PORT"
id := viper.GetInt("id")
id := viper.GetInt("port")
```

Если указать только ключ, то `viper.BindEnv` свяжет его с переменной окружения, имя которой совпадает с ключом. В дополнительных аргументах можно указать одну или несколько переменных окружения для привязки. В обоих случаях Viper автоматически предполагает, что имя переменной окружения записывается заглавными буквами.

В Viper имеется несколько дополнительных вспомогательных функций для работы с переменными окружения. Более подробную информацию об этом ищите в описании Viper в GoDoc (https://oreil.ly/CGpPS).

Работа с конфигурационными файлами в Viper

Из коробки библиотека Viper поддерживает форматы JSON и YAML, используя пакеты, представленные выше; а также TOML, HCL, INI, envfile и Java

Properties. Кроме того, она может записывать конфигурацию в файлы для начальной загрузки и даже наблюдать за изменениями в файлах и перечитывать их.

Обсуждение локальных конфигурационных файлов может показаться неожиданным в книге об облачных окружениях, и все же файлы по-прежнему являются широко используемой структурой в любом контексте. В конце концов, разделяемые файловые системы – будь то ConfigMaps в Kubernetes или монтируемые тома NFS – довольно распространены, и даже облачные службы могут развертываться с помощью системы управления конфигурациями, которая устанавливает локальную копию файла, доступную только для чтения всем экземплярам службы. Файл конфигурации можно даже включить или смонтировать в образ контейнера так, чтобы он выглядел – с точки зрения службы в контейнере – точно так же, как любой другой локальный файл.

ЧТЕНИЕ КОНФИГУРАЦИОННЫХ ФАЙЛОВ Чтобы с помощью Viper читать конфигурационные файлы, достаточно сообщить библиотеке имена файлов и места, где их искать. Кроме того, ей необходимо знать их типы, если эти типы нельзя определить по расширению файла. Функция `viper.ReadInConfig` отыскивает и читает конфигурационный файл, возвращая значение ошибки, если что-то пойдет не так. Вот как это делается:

```
viper.SetConfigName("config")

// Необязательно, если конфигурационный файл имеет расширение
viper.SetConfigType("yaml")

viper.AddConfigPath("/etc/service/")
viper.AddConfigPath("$HOME/.service")
viper.AddConfigPath(".")

if err := viper.ReadInConfig(); err != nil {
    panic(fmt.Errorf("fatal error reading config: %w", err))
}
```

Как видите, Viper может находить конфигурационный файл по нескольким путям. К сожалению, в настоящее время один экземпляр Viper поддерживает чтение только одного конфигурационного файла.

НАБЛЮДЕНИЕ ЗА ИЗМЕНЕНИЯМИ В КОНФИГУРАЦИОННЫХ ФАЙЛАХ В VIPER Viper из коробки позволяет приложению наблюдать за изменениями в конфигурационном файле и повторно загружать его при необходимости. То есть, чтобы изменения вступили в силу, перезапускать сервер не требуется.

По умолчанию эта функция отключена. Чтобы ее включить, нужно вызвать `viper.WatchConfig`. Кроме того, с помощью `viper.OnConfigChange` можно зарегистрировать свою функцию, которая будет вызываться при каждом изменении конфигурационного файла:

```
viper.WatchConfig()
viper.OnConfigChange(func(e fsnotify.Event) {
    fmt.Println("Config file changed:", e.Name)
})
```

Все вызовы `viper.AddConfigPath` должны выполняться перед вызовом `viper.WatchConfig`.

Интересно отметить, что за кулисами Viper использует пакет `fsnotify`, который мы подробно описали в разделе «Наблюдение за изменениями в конфигурационных файлах» выше.

Использование удаленных хранилищ пар ключ/значение в Viper

Наиболее интересной, пожалуй, особенностью библиотеки Viper является ее способность читать конфигурационные строки в любом поддерживаемом формате, находящиеся в удаленном хранилище пар ключ/значение, таком как etcd (https://etcd.io) или HashiCorp Consul (https://consul.io). Эти значения имеют приоритет над значениями по умолчанию, но переопределяются значениями конфигурации, полученными из файла, флагов командной строки или переменных окружения.

Чтобы включить поддержку удаленных хранилищ, сначала нужно импортировать пакет `viper/remote`:

```
import _ "github.com/spf13/viper/remote"
```

Затем нужно зарегистрировать удаленный источник конфигурации вызовом метода `viper.AddRemoteProvider`, сигнатура которого выглядит следующим образом:

```
func AddRemoteProvider(provider, endpoint, path string) error
```

- В параметре `provider` можно передать `etcd`, `consul` или `firestore`.
- Параметр `endpoint` – это URL удаленного ресурса. В Viper есть одна странная особенность, для провайдера `etcd` требуется включить в URL схему (`http://ip:port`), а для провайдера `consul` схема *не требуется* (достаточно указать `ip:port`).
- Параметр `path` – это путь к хранилищу пар ключ/значение, откуда следует читать конфигурацию.

Например, вот как можно прочитать конфигурацию в формате JSON из службы etcd:

```
viper.AddRemoteProvider("etcd", "http://127.0.0.1:4001","/config/service.json")
viper.SetConfigType("json")
err := viper.ReadRemoteConfig()
```

Обратите внимание: даже притом, что путь к конфигурационному файлу включает расширение, мы все равно должны вызвать `viper.SetConfigType` и явно определить формат конфигурации. Это связано с тем, что, с точки зрения Viper, ресурс является всего лишь потоком байтов, поэтому она не может автоматически определить формат[1]. На момент написания этих

[1] Или эта возможность пока не реализована. Я не знаю.

строк поддерживались форматы: *json*, *toml*, *yaml*, *yml*, *properties*, *props*, *prop*, *env* и *dotenv*.

Есть возможность зарегистрировать несколько провайдеров, и тогда поиск будет выполняться в порядке их регистрации.

Это было упрощенное введение в возможности Viper при работе с удаленными хранилищами пар ключ/значение. За дополнительной информацией о порядке использования Viper для чтения из Consul, наблюдения за изменениями или чтения зашифрованных конфигураций обращайтесь к файлу README (https://oreil.ly/1iE2y).

Значения по умолчанию в Viper

В отличие от всех других пакетов, которые мы рассмотрели в этой главе, библиотека Viper позволяет определять значения по умолчанию вызовом функции `SetDefault`.

Значения по умолчанию иногда могут быть очень полезны, но будьте осторожны, используя эту возможность. Как отмечалось в разделе «Рекомендуемые приемы организации конфигураций» выше, значимые нулевые значения обычно предпочтительнее неявных значений по умолчанию, которые при бездумном использовании могут привести к неожиданному поведению.

Вот пример, демонстрирующий определение значений по умолчанию в Viper:

```
viper.BindEnv("id")          // Будет автоматически преобразовано в верхний регистр
viper.SetDefault("id", "13") // Значение по умолчанию "13"
id1 := viper.GetInt("id")
fmt.Println(id1)             // 13
os.Setenv("ID", "50")        // Явная установка переменной окружения
id2 := viper.GetInt("id")
fmt.Println(id2)             // 50
```

Значения по умолчанию имеют самый низкий приоритет и используются, только если конфигурационный параметр не был установлен явно другими механизмами.

Управление функциональными возможностями с помощью флагов

Управление функциональными возможностями с помощью флагов (или *переключение функциональных возможностей*)[1] – это шаблон разработки программного обеспечения, предназначенный для повышения скорости и безопасно-

[1] Я также видел названия «включение функциональных возможностей», «изменение функциональных возможностей», «условные функциональные возможности» и многие другие. Однако в большинстве случаев используется термин «переключение» или «управление», вероятно, потому что другие названия просто недостаточно точно отражают суть.

сти разработки и предоставления новых возможностей, который позволяет включать или отключать определенные функции во время выполнения без развертывания нового кода.

Флаг, включающий функциональную возможность, – это, по сути, условие в коде, которое включает или отключает функцию, исходя из некоторых внешних критериев, часто (но не всегда) параметров конфигурации. Устанавливая различные конфигурационные значения, разработчик может, например, включить незаконченную функцию для тестирования и отключить ее для других пользователей.

Возможность выпуска продукта с незаконченными функциями дает ряд серьезных преимуществ.

Во-первых, флаги управления функциональными возможностями позволяют развертывать новые версии программного обеспечения без накладных расходов на создание и слияние ветвей в репозитории, которые иначе просто неизбежны. Иначе говоря, флаги управления функциональными возможностями отделяют выпуск функции от ее развертывания. Учитывая, что флаги по своей природе требуют внесения изменений в код как можно раньше, это одновременно и стимулирует, и упрощает непрерывное развертывание и доставку. В результате разработчики быстрее получают обратную связь о своем коде, что, в свою очередь, помогает им быстрее двигаться вперед, выполняя короткие и безопасные итерации.

Во-вторых, флаги управления функциональными возможностями не только упрощают тестирование функций до того, как они будут признаны готовыми к выпуску, но также позволяют включать и выключать эти функции динамически. Например, можно предусмотреть логику для создания обратной связи и объединить ее с таким шаблоном, как Circuit Breaker (Размыкатель цепи), чтобы автоматически включать или отключать флаги при определенных условиях.

Наконец, флаги управления функциональными возможностями можно даже использовать, чтобы дать доступ к новым возможностям определенному подмножеству пользователей. Этот метод, называемый *выборочным предоставлением функций* (feature gating), можно использовать в качестве альтернативы канареечным развертываниям и поэтапным развертываниям по географическим регионам. В сочетании с методами наблюдения выборочное предоставление функций может даже упростить проведение экспериментов, таких как A/B-тестирование или целевая трассировка, которые позволяют использовать определенные группы пользователей или даже отдельных клиентов в качестве тестировщиков.

Разработка флага для управления функциональной возможностью

В этом разделе мы шаг за шагом рассмотрим итеративный процесс реализации управления функциональной возможностью, взяв за основу REST-службу хранилища пар ключ/значение, которую мы создали в главе 5. Начав с базовой функции, мы пройдем несколько этапов, от версии без флагов до

версии с динамически управляемым флагом, который включается для определенного подмножества пользователей.

В качестве такой переключаемой функции мы реализуем поддержку масштабирования нашего хранилища и изменим его логику так, чтобы вместо локального ассоциативного массива оно могло использовать распределенную структуру данных.

Итерация 0: начальная реализация

Нашу первую итерацию мы начнем с изменения функции `keyValueGetHandler` из раздела «Реализация функции чтения» в главе 5. Как вы наверняка помните, `keyValueGetHandler` – это функция-обработчик HTTP-запросов, которая удовлетворяет интерфейсу `HandlerFunc`, объявленному в пакете `net/http`. Если вы забыли, что это означает, то, возможно, вам стоит вернуться к разделу «Создание HTTP-сервера с использованием net/http» главы 5.

Начнем со следующей реализации функции-обработчика, почти полностью скопированной из главы 5 (за вычетом обработки ошибок для краткости):

```
func keyValueGetHandler(w http.ResponseWriter, r *http.Request) {
    vars := mux.Vars(r)        // Извлечь ключ из запроса
    key := vars["key"]

    value, err := Get(key) // Получить значение для данного ключа
    if err != nil {        // Неожиданная ошибка!
        http.Error(w,
            err.Error(),
            http.StatusInternalServerError)
        return
    }

    w.Write([]byte(value)) // Записать значение в ответ
}
```

Как видите, в этой функции отсутствует логика переключения функциональных возможностей (и вообще чего-либо, что можно было бы переключать). Она просто получает ключ из запроса, вызовом функции `Get` извлекает значение, связанное с этим ключом, и записывает это значение в ответ.

В следующей итерации мы начнем тестировать новую функциональную возможность: распределенную структуру данных, которая заменит локальный ассоциативный массив `map[string]string` и позволит масштабировать службу.

Итерация 1: жестко запрограммированный флаг

В этой реализации мы представим, что создали нашу новую экспериментальную распределенную структуру и сделали ее доступной через функцию `NewGet`.

В первой попытке создать флаг управления функциональной возможностью мы введем условие, которое позволит использовать простое логическое значение useNewStorage для переключения между двумя реализациями:

```
// Установить в true, чтобы задействовать новую реализацию хранилища
const useNewStorage bool = false;

func keyValueGetHandler(w http.ResponseWriter, r *http.Request) {
    vars := mux.Vars(r)
    key := vars["key"]

    var value string
    var err error

    if useNewStorage {
        value, err = NewGet(key)
    } else {
        value, err = Get(key)
    }

    if err != nil {
        http.Error(w,
            err.Error(),
            http.StatusInternalServerError)
        return
    }

    w.Write([]byte(value))
}
```

Первая итерация показывает определенный прогресс, но результат еще далек от того, что нам нужно. Фиксированное условие в виде жестко запрограммированного в коде значения позволяет переключаться между реализациями, выполняя локальное тестирование, но организовать непрерывное и автоматическое тестирование будет нелегко.

Кроме того, придется повторно собирать и развертывать службу при каждом изменении используемого алгоритма, что фактически сводит на нет все преимущества наличия флага управления функциональной возможностью.

Соблюдайте правила гигиены флагов! Если вы давно не изменяли флаг управления функциональной возможностью, то, возможно, пришла пора его удалить.

Итерация 2: настраиваемый флаг

Прошло немного времени, и недостатки жестко запрограммированного флага управления функциональной возможностью стали очевидными. Во-первых, было бы очень хорошо, если бы мы могли изменять значение флага извне, чтобы протестировать оба алгоритма в наших тестах.

В этом примере мы используем Viper для привязки и чтения переменной окружения, которую теперь можно использовать для включения или отклю-

чения функции во время выполнения. Однако выбор механизма конфигурирования в данном случае не важен – важна сама возможность изменять флаг без изменения кода:

```
func keyValueGetHandler(w http.ResponseWriter, r *http.Request) {
    vars := mux.Vars(r)
    key := vars["key"]

    var value string
    var err error

    if FeatureEnabled("use-new-storage", r) {
        value, err = NewGet(key)
    } else {
        value, err = Get(key)
    }

    if err != nil {
        http.Error(w,
            err.Error(),
            http.StatusInternalServerError)
        return
    }

    w.Write([]byte(value))
}

func FeatureEnabled(flag string, r *http.Request) bool {
    return viper.GetBool(flag)
}
```

Помимо использования Viper для чтения переменной окружения, определяющей значение флага `use-new-storage`, мы также ввели новую функцию `FeatureEnabled`. На данный момент она просто вызывает `viper.GetBool(flag)`, но, что особенно важно, она также концентрирует логику чтения флага в одном месте. В чем именно преимущество такого подхода, мы увидим в следующей итерации.

Возможно, вам интересно, почему `FeatureEnabled` принимает указатель на запрос `*http.Request`. Да, пока она не использует его, но мы задействуем его в следующей итерации.

Итерация 3: динамический флаг

Теперь новая функциональная возможность развернута, но отключена с помощью флага. И хотелось бы иметь возможность протестировать ее в промышленном окружении на определенной подгруппе пользователей. Понятно, что в этом сценарии не получится использовать флаг, настраиваемый с помощью конфигурации. Поэтому мы реализуем динамический флаг, способный *самостоятельно определять*, когда включаться. Это означает, что мы должны связать флаг с функцией.

Динамические флаги как функции

Первым делом, приступая к созданию функции динамического флага, нужно определить сигнатуру функции. И желательно определить ее явно, в виде типа функции, как показано ниже, хотя это не является обязательным требованием:

```
type Enabled func(flag string, r *http.Request) (bool, error)
```

Тип функции `Enabled` является прототипом для всех наших функций динамических флагов. Он определяет функцию, которая принимает имя флага в виде строки и `*http.Request` и возвращает логическое значение, отражающее состояние флага: `true` – включен, `false` – выключен.

Реализация функции динамического флага

Опираясь на контракт, определяемый типом `Enabled`, мы можем реализовать функцию, которая будет определять, исходит ли запрос из частной сети, сравнивая адрес отправителя запроса со стандартным списком диапазонов IP-адресов, выделенных для частных сетей:

```
// Список диапазонов адресов, принадлежащих внутренним сетям.
var privateCIDRs []*net.IPNet

// Используем функцию init для инициализации среза privateCIDRs.
func init() {
    for _, cidr := range []string{
        "10.0.0.0/8",
        "172.16.0.0/12",
        "192.168.0.0/16",
    } {
        _, block, _ := net.ParseCIDR(cidr)
        privateCIDRs = append(privateCIDRs, block)
    }
}

// fromPrivateIP принимает имя флага (которое она игнорирует) и
// запрос. Если IP-адрес отправителя принадлежит диапазону частных сетей,
// согласно RFC1918, то возвращает true.
func fromPrivateIP(flag string, r *http.Request) (bool, error) {
    // Получить адрес хоста-отправителя
    remoteIP, _, err := net.SplitHostPort(r.RemoteAddr)
    if err != nil {
        return false, err
    }

    // Преобразовать строку с адресом отправителя в *net.IPNet
    ip := net.ParseIP(remoteIP)
    if ip == nil {
        return false, errors.New("couldn't parse ip")
    }
```

```
    // Адрес петлевого интерфейса считается "частным"
    if ip.IsLoopback() {
        return true, nil
    }

    // Выполнить поиск IP-адреса в списке адресов CIDR;
    // вернуть true, если найден.
    for _, block := range privateCIDRs {
        if block.Contains(ip) {
            return true, nil
        }
    }

    return false, nil
}
```

Как видите, функция `fromPrivateIP` соответствует типу `Enabled`, принимая строковое значение (имя флага) и `*http.Request` (экземпляр запроса). Она возвращает `true`, если запрос отправлен из частного диапазона IP-адресов (как определено в RFC 1918 (https://oreil.ly/lZ5PQ)).

Чтобы определить это, функция `fromPrivateIP` сначала извлекает IP-адрес отправителя из `*http.request`. После синтаксического анализа вызовом `net.SplitHostPort` и преобразования адреса в значение `*net.IP` применением `net.ParseIP` она сравнивает это значение с каждым из частных диапазонов CIDR, содержащихся в `privateCIDR`, и в случае совпадения возвращает `true`.

Эта функция вернет `true`, если запрос проходит через балансировщик нагрузки или обратный прокси. Промышленная реализация должна учитывать это и в идеале уметь определять протокол прокси.

Конечно, эта функция является всего лишь примером. Я использовал ее, потому что она относительно проста, но похожий метод можно использовать для включения или отключения флага по признаку принадлежности к географическому региону, для фиксированного процента пользователей или даже для конкретного клиента.

Поиск функции флага

Теперь, получив функцию динамического флага в форме `fromPrivateIP`, мы должны реализовать некоторый механизм связывания этой функции с флагом. Самый простой способ сделать это – использовать ассоциативный массив, отображающий строковые имена флагов в функции `Enabled`:

```
var enabledFunctions map[string]Enabled

func init() {
    enabledFunctions = map[string]Enabled{}
    enabledFunctions["use-new-storage"] = fromPrivateIP
}
```

Такое использование ассоциативного массива для косвенной ссылки на функции дает большую гибкость. Можно даже связать функцию с несколькими флагами, если понадобится, что может пригодиться, если нужно будет обеспечить активность связанных функциональных возможностей при одних и тех же условиях.

Вы наверняка заметили, что для заполнения ассоциативного массива `enabledFunctions` в этом примере используется функция `init`. Но постойте, у нас же уже есть функция `init`?

Да есть, и в этом решении нет никакой ошибки. Функция `init` – особенная: при желании можно определить множество функций `init`.

Функция маршрутизации

Наконец, мы можем объединить все вместе.

Для этого реорганизуем функцию `FeatureEnabled`, которая будет искать подходящую функцию динамического флага, вызывать ее и возвращать результат:

```
func FeatureEnabled(flag string, r *http.Request) bool {
    // Явно установленные флаги имеют высший приоритет
    if viper.IsSet(flag) {
        return viper.GetBool(flag)
    }

    // Получить функцию флага, если таковая имеется.
    // Если нет, то вернуть false
    enabledFunc, exists := enabledFunctions[flag]
    if !exists {
        return false
    }

    // Функция флага получена: вызвать ее
    // и вернуть результат
    result, err := enabledFunc(flag, r)
    if err != nil {
        log.Println(err)
        return false
    }

    return result
}
```

Теперь `FeatureEnabled` превратилась в полноценную функцию маршрутизатора, которая может динамически управлять выполнением того или иного пути в коде, в соответствии с явными настройками флагов и результатами, возвращаемыми функциями флагов. В этой реализации явно установленные флаги имеют приоритет над всем остальным. Это позволяет автоматическим тестам проверять обе функциональные возможности.

Наша реализация использует простой поиск в памяти для определения конкретного поведения, но то же самое легко можно реализовать с помощью

базы данных или другого источника, или даже сложной управляемой службы, такой как LaunchDarkly. Однако имейте в виду, что эти решения вводят новые зависимости.

> **Флаги управления функциональными возможностями как услуга?**
>
> Если вас интересует реализация более сложных динамических флагов управления функциональными возможностями, но вы (вероятно, не просто так) предпочитаете не использовать собственные решения, то взгляните в сторону LaunchDarkly (https://oreil.ly/0xKeq) – отличной службы, предоставляющей «флаги управления функциональными возможностями как услугу».

Итоги

Управляемость – не самая привлекательная тема в мире облачных вычислений, да и в любом другом мире, но мне очень понравилось, как мы лихо справились с деталями в этой главе.

Мы рассмотрели основные моменты различных стилей организации конфигурации, включая переменные окружения, флаги командной строки и файлы разных форматов. Мы даже исследовали пару стратегий обнаружения изменений в конфигурации для ее перезагрузки. А еще познакомились с библиотекой Viper, которая поддерживает все это и многое другое.

Я чувствую, что некоторые темы были освещены недостаточно глубоко, и я мог бы рассказать о них более подробно, если бы не был ограничен пространством и временем. Например, флаги управления функциональными возможностями – это довольно обширная тема, и я определенно хотел бы иметь возможность исследовать ее шире и глубже. Некоторые темы, такие как развертывание и обнаружение служб, мы вообще не смогли охватить. Думаю, нам есть на что рассчитывать в следующем издании, верно?

Мне очень понравилась эта глава, но еще больше мне понравилась глава 11, в которой мы углубимся в наблюдаемость в целом и OpenTelemetry в частности.

В заключение хочу дать вам один совет: всегда будьте самим собой и помните, что удача улыбается трудолюбивым.

Глава 11

Наблюдаемость

> Данные – это не информация, информация – это не знания, знания – это не понимание, понимание – это не мудрость[1].
>
> – Клиффорд Столл (Clifford Stoll),
> *High-Tech Heretic: Reflections of a Computer Contrarian*

«Облачные вычисления» – все еще довольно новая концепция. Насколько я могу судить, термин «облачный» начал входить в обиход только после основания Cloud Native Computing Foundation в середине 2015 года[2].

Мы в отрасли все еще пытаемся понять, что именно означает понятие «облачный», и с появлением новых услуг, регулярно запускаемых ведущими провайдерами облачных вычислений – каждая из которых предлагает более высокий уровень абстракции, чем предшествующие, – даже то небольшое согласие, достигнутое нами, меняется со временем.

Однако ясно одно: функции (и отказы) сетевого и аппаратного уровней все больше абстрагируются и заменяются вызовами и событиями API. С каждым днем мы приближаемся к миру, в котором *все* определяется программным обеспечением. Все наши проблемы становятся проблемами программного обеспечения.

Мы, безусловно, жертвуем изрядной долей контроля над платформами, на которых работает наше программное обеспечение, зато мы выигрываем с точки зрения общей управляемости и надежности[3], что позволяет нам сосредоточить все наше ограниченное время и внимание на нашем программном обеспечении. Также это означает, что большинство причин отказов теперь кроются в наших собственных службах и во взаимодействиях между ними. Никакие причудливые фреймворки или протоколы не могут решить проблему плохого программного обеспечения. Как я уже говорил в главе 1, приложение, сляпанное кое-как, после переноса в Kubernetes не улучшится.

В этом прекрасном новом мире с высокой степенью распределенности многое усложняется. Программное обеспечение становится более сложным,

---

[1] Stoll, Clifford. *High-Tech Heretic: Reflections of a Computer Contrarian*. Random House, September 2000.

[2] Интересно отметить, что в то же время вышла платформа AWS Lambda – «функции как услуга» (Functions as a Service, FaaS). Совпадение? Может быть.

[3] Конечно же, при условии, что все наши сетевые настройки и настройки платформы верны!

платформы усложняются, а вместе они становятся *по-настоящему* сложными, и часто мы даже понятия не имеем о том, что происходит внутри. Обеспечение прозрачности наших служб стало приобретать особую важность, и единственное, что мы *действительно* знаем, – существующие инструменты и методы мониторинга просто не справляются с этой задачей. Нам нужно что-то новое. Не просто новая технология или новый набор методов, а совершенно новый способ мышления и понимания наших систем.

Что такое наблюдаемость?

Вокруг наблюдаемости сейчас много шумихи. Это модная и важная тема. Но что такое наблюдаемость на самом деле? Чем она отличается (и чем похожа) от традиционного мониторинга и оповещения с помощью журналов, показателей и трассировки? И самое главное, как «реализовать наблюдаемость»?

Наблюдаемость – это не просто новомодное словечко, хотя эта мысль напрашивается сама собой, учитывая то внимание, которое к ней привлекают.

На самом деле все довольно просто. Наблюдаемость – это свойство системы, ничем не отличающееся от устойчивости или управляемости, которое отражает, насколько точно можно судить о внутреннем состоянии системы, опираясь на информацию, выводимую наружу. Систему можно считать *наблюдаемой*, если она позволяет быстро и последовательно получать ответы на все новые вопросы, имея минимальный объем предварительных знаний, и без необходимости перестраивать существующий или создавать новый код. Наблюдаемая система позволяет задавать вопросы, о которых вы еще не задумывались.

В конечном счете наблюдаемость – это нечто большее, чем инструмент, который некоторые провайдеры могут пытаться продать вам, называя его наблюдаемостью. Нельзя «купить наблюдаемость», так же как нельзя купить «безотказность». Никакие инструменты не сделают вашу систему наблюдаемой только потому, что вы их используете, точно так же, как молоток сам по себе не сделает конструкцию моста прочной. Инструменты могут помочь в этом, но только вы решаете, как их применять.

Конечно, говорить проще, чем делать. Встраивание наблюдаемости в сложную систему требует отказаться от поиска «известных неизвестных» и принять тот факт, что мы часто даже не можем полностью понять ее состояние в данный конкретный момент времени. Понимание *всех возможных* состояний отказа (или его отсутствия) в сложной системе практически невозможно. Первым шагом к достижению наблюдаемости является прекращение поиска конкретных ожидаемых режимов отказа – «известных неизвестных», – как если бы их вообще не было.

Зачем нужна наблюдаемость?

Наблюдаемость – это естественная эволюция традиционного мониторинга, обусловленная новыми сложностями, возникающими в облачных архитектурах.

Первая из них – огромный масштаб многих современных облачных систем, содержащих слишком много *всякой всячины*, чтобы мы с нашим ограниченным человеческим мозгом могли уделить им должное внимание. Все данные, генерируемые несколькими взаимосвязанными системами, работающими одновременно, предоставляют больше сведений, чем мы можем осознать, больше данных, чем мы можем воспринять, и больше взаимосвязей, чем мы можем разобрать.

Но особенно важно то, что природа облачных систем коренным образом отличается от более традиционных архитектур, существовавших не так давно. Их контекстные и функциональные требования различны, они поразному функционируют и терпят неудачу, и гарантии, которые они должны предоставлять, тоже различны.

Как организовать мониторинг распределенных систем, учитывая эфемерность современных приложений и окружений, в которых они находятся? Как выявить дефект в отдельном компоненте в сложной распределенной системе? Эти задачи и пытается решить «наблюдаемость».

Чем наблюдаемость отличается от «традиционного» мониторинга?

На первый взгляд грань между мониторингом и наблюдаемостью выглядит размытой. В конце концов, и мониторинг, и наблюдаемость должны давать возможность задавать вопросы системе. Разница заключается в типах вопросов, которые можно задать.

Мониторинг традиционно фокусируется на вопросах, ответы на которые помогут определить или предсказать некоторые ожидаемые или ранее имевшие место режимы отказа. Иначе говоря, мониторинг сосредоточен на «известных неизвестных». Предполагается, что система будет вести себя – и, следовательно, выходить из строя – определенным и предсказуемым образом. Когда обнаруживается новый режим отказа – обычно на собственном опыте, – его симптомы добавляются в пакет мониторинга, и процесс начинается сначала.

Этот подход хорошо зарекомендовал себя в относительно простых системах, но он имеет некоторые проблемы. Во-первых, чтобы задать системе новый вопрос, часто необходимо написать и развернуть новый код. Это негибко, немасштабируемо и очень раздражает.

Во-вторых, после достижения определенного уровня сложности количество «неизвестных неизвестных» в системе начинает превосходить количество «известных неизвестных». Отказы чаще непредсказуемы, реже предсказуемы и почти всегда являются результатом целой последовательности сбоев. Мониторинг всех возможных отказов становится практически невозможным.

Мониторинг – это ваши *действия в системе*, которые предпринимаются с целью узнать, работает она или не работает. Методы наблюдения, напротив, основной упор делают на понимание системы, позволяющее соотносить

события и поведение. Наблюдаемость – это *свойство системы*, которое позволяет спросить, почему она не работает.

«Три столпа наблюдаемости»

Три столпа наблюдаемости – это собирательное название, под которым иногда упоминаются три наиболее распространенных (и основополагающих) инструмента наблюдаемости – журналирование, метрики и трассировка. Вот некоторые подробности об этих трех компонентах в том порядке, в каком мы будем их обсуждать.

Трассировка
: Трассировка (или *распределенная трассировка*) помогает определить распространение запроса через (обычно распределенную) систему и преобразовать весь сквозной поток запросов в ориентированный ациклический граф (https://ru.wikipedia.org/wiki/Ориентированный_ациклический_граф), называемый *трассировкой*. Анализ трассировок может дать представление о том, как взаимодействуют компоненты системы, что позволяет выявить сбои и проблемы с производительностью.
Более подробно трассировка будет рассматриваться в разделе «Трассировка».

Метрики
: Под метриками подразумевается набор числовых данных, представляющих состояние различных компонентов системы в определенные моменты времени. Коллекции данных, представляющие наблюдаемые характеристики одного и того же объекта в разное время, особенно полезны для визуализации и математического анализа и могут использоваться для выявления тенденций и аномалий, прогнозирования будущего поведения.
Подробнее о метриках мы поговорим в разделе «Метрики».

Журналирование
: Журналирование – это процесс добавления неизменяемых записей о важных событиях в журнал для последующего просмотра или анализа. Журналы могут иметь самую разную форму, от постоянно дополняемого файла на диске до полнотекстовой поисковой системы, такой как Elasticsearch (https://oreil.ly/Hf4Pn). Журналы предоставляют ценную контекстную информацию о событиях, протекающих в приложениях. При этом важно, чтобы записи в журнале были правильно структурированы; невыполнение этого условия может резко ограничить их полезность.
Более подробно журналирование рассматривается в разделе «Журналирование».

Каждый из этих методов полезен сам по себе, но по-настоящему наблюдаемая система переплетает их, так что каждый может ссылаться на другие. Например, метрики могут использоваться для выявления подмножества

трассировок некорректного поведения, а эти трассировки могут ссылаться на журналы, способные помочь найти основную причину.

Самая главная мысль этой главы, которую следует запомнить: наблюдаемость – это *просто свойство системы*, такое же как устойчивость или управляемость, и никакие инструменты, фреймворки или производители не смогут «дать вам» наблюдаемость. Так называемые «три столпа» – это всего лишь методы, которые можно использовать для формирования данного свойства.

«Три столпа»

Название «Три столпа наблюдаемости» часто критикуют – я думаю, что не без оснований, – потому что его легко интерпретировать как предполагающее, что эти три «столпа» позволяют «делать наблюдаемость». Простое журналирование, сбор метрик и трассировка не обязательно сделают систему более наблюдаемой, но каждый из них является мощным инструментом, которые все вместе, при правильном понимании и использовании, могут обеспечить полное представление о внутреннем состоянии вашей системы.

Другой критический аргумент заключается в том, что этот термин подразумевает, что наблюдаемость – это комбинация трех очень разных инструментов, просто дающих разное представление об одном и том же, что в конечном итоге увеличивает уникальную возможность понять состояние системы. Но именно благодаря интеграции этих трех подходов становится возможным сделать первые шаги к наблюдаемости.

OpenTelemetry

На момент написания этих строк проект OpenTelemetry (или «OTel», как его называют крутые парни[1]) являлся одним из примерно четырех десятков проектов-участников фонда облачных вычислений Cloud Native Computing Foundation (CNCF) и, пожалуй, одним из самых интересных.

В отличие от большинства проектов CNCF, OpenTelemetry не является службой. Скорее, это попытка стандартизации выражения, сбора и передачи данных телеметрии – трассировок, метрик и (в конечном итоге) журналов. Его многочисленные репозитории (https://oreil.ly/GpGD5) включают набор спецификаций, а также API и эталонные реализации на разных языках, в том числе Go (https://oreil.ly/vSO7k)[2].

Пространство инструментов переполнено; за эти годы на сцене побывали десятки производителей инструментов, каждый со своими уникальными реализациями. OpenTelemetry стремится объединить это пространство – и всех производителей в нем – вокруг единой независимой спецификации, которая стандартизирует сбор и отправку данных телеметрии. Ранее тоже

---

[1] Я не отношусь к их числу.

[2] Существуют также реализации на Python, Java, JavaScript, .NET, C++, Rust, PHP, Erlang/Elixir, Ruby и Swift.

предпринимались попытки стандартизации. Фактически OpenTelemetry – это слияние двух более ранних проектов: OpenTracing и OpenCensus, – которые он объединяет в единый набор стандартов, независимых от производителей.

В этой главе мы рассмотрим каждый из «трех столпов», лежащих в основе их концепции, а также порядок использования OpenTelemetry в вашем коде для пересылки полученной телеметрии на выбранный вами сервер. При этом важно отметить, что OpenTelemetry – это обширная тема, заслуживающая отдельной книги, но я постараюсь сделать все возможное, чтобы достаточно осветить ее и дать вам практическое введение в нее. На момент написания этих строк не было никаких исчерпывающих ресурсов по OpenTelemetry, но я собрал все, что мог, из примеров и статей (изрядно покопавшись в исходном коде).

Когда я писал эту главу, то узнал, что Чарити Мейджорс (Charity Majors)[1] и Лиз Фонг-Джонс (Liz Fong-Jones) работали над книгой *Observability Engineering*, которую издательство O'Reilly Media планирует выпустить в январе 2022 года.

Компоненты OpenTelemetry

OpenTelemetry расширяет и унифицирует предыдущие попытки создания стандартов телеметрии, отчасти за счет включения абстракций и точек расширения в SDK, к которым вы можете подключать свои реализации. Это позволяет, например, реализовать свои экспортеры, способные взаимодействовать с поставщиком услуг по вашему выбору.

Для достижения такого уровня модульности в OpenTelemetry предусмотрены следующие основные компоненты.

Спецификации
: Спецификации OpenTelemetry описывают требования и ожидания для всех API, SDK и протоколов данных.

API
: Интерфейсы и реализации, основанные на спецификациях, которые можно использовать для добавления OpenTelemetry в приложение.

SDK
: Конкретные реализации OpenTelemetry, которые располагаются между API и экспортерами, обеспечивая такие функции, как (например) трассировка состояния и упаковывание данных для передачи. SDK также предлагает ряд конфигурационных параметров для таких действий, как фильтрация запросов и выборка транзакций.

---

[1] Если прежде вы не посещали блог Чарити Мейджорс (Charity Majors; https://charity.wtf/), то я советую сделать это прямо сейчас. Это не женщина, а вместилище гениальности и опыта, перемешанных с радугой, мультяшными единорогами и грубоватой лексикой.

Экспортеры
: Плагины из SDK для отправки данных в определенное место, которое может быть локальным файлом журнала (или стандартным выводом) или удаленным хранилищем, например Jaeger (https://oreil.ly/uMAfg), или коммерческим решением, таким как Honeycomb (https://oreil.ly/cBlnX), или Lightstep (https://oreil.ly/KScdI). Экспортеры отделяют сбор информации от хранилищ, что дает возможность изменять место хранения без изменения вашего кода.

Коллектор
: Необязательная, но очень полезная служба, которая может получать и обрабатывать данные телеметрии перед их пересылкой в один или в несколько пунктов назначения. Его можно запускать как сопутствующий процесс вместе с вашим приложением или как автономный прокси-сервер, что обеспечивает большую гибкость в организации отправки телеметрии приложения. Это может особенно пригодиться в жестко контролируемых окружениях, так характерных для многих предприятий.

Возможно, вы заметили отсутствие серверной части OpenTelemetry. Ее действительно не существует. OpenTelemetry занимается только сбором, обработкой и отправкой данных телеметрии и полагается на то, что вы предоставите серверную часть для получения и хранения данных.

В OpenTelemetry есть и другие компоненты, но вышеперечисленные можно считать основными. Взаимосвязи между ними показаны на рис. 11.1.

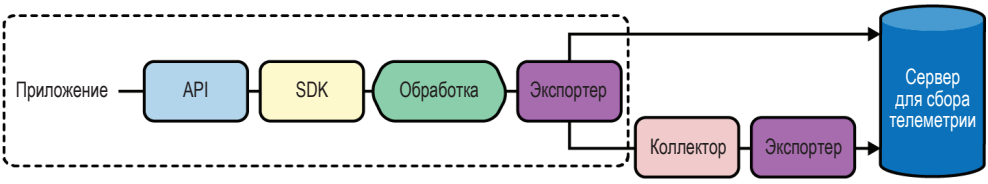


Рис. 11.1 ❖ Высокоуровневое представление основных компонентов OpenTelemetry для сбора (API), обработки (SDK) и экспорта (экспортер и коллекторы) данных; предоставление серверной части возлагается на вас

Наконец, основная цель проекта – широкая поддержка самых разных языков программирования. На момент написания этих строк проект OpenTelemetry предоставлял API и SDK для Go, Python, Java, Ruby, Erlang, PHP, JavaScript, .NET, Rust, C++ и Swift.

Трассировка

В этой книге мы много говорили о преимуществах микросервисных архитектур и распределенных систем. Но досадная реальность – которая, как я уверен, уже стала очевидной – заключается в том, что такие архитектуры создают множество новых и необычных проблем.

Мы уже говорили, что устранение сбоя в распределенной системе подобно раскрытию тайны убийства, что показывает всю сложность выяснения источника проблем, когда *что-то где-то* не работает, прежде чем перейти к нему и исправить.

Это именно та проблема, для решения которой была изобретена *трассировка*. Позволяя наблюдать за распространением запросов в системе – даже через границы процессов и сетей, – трассировка помогает (например) выявить сбои компонентов и узкие места в производительности и проанализировать зависимости служб.

Трассировка обычно обсуждается в контексте распределенных систем, но также может помочь с решением проблем и в сложном монолитном приложении, особенно если оно борется за обладание такими ресурсами, как сеть, диск или мьютексы.

В этом разделе мы подробно рассмотрим трассировку, ее основные концепции и как использовать OpenTelemetry для инструментирования кода и пересылки полученной телеметрии на сервер.

К сожалению, нехватка времени и места позволяет нам лишь слегка углубиться в эту тему. Но если вы хотите узнать больше о трассировке, то я рекомендую книгу Остина Паркера (Austin Parker), Дэниела Спунхауэра (Daniel Spoonhower), Джонатана Мейса (Jonathan Mace), Бена Сигельмана (Ben Sigelman) и Ребекки Айзекс (Rebecca Isaacs) *Distributed Tracing in Practice* (O'Reilly; https://oreil.ly/vzJMP).

Концепции трассировки

Для обсуждения трассировки необходимо знать две фундаментальные концепции: *операции* (span) и *треки* (trace).

Операции
: Операция описывает единицу работы, выполняемую запросом, такую как ветвление в потоке выполнения или переход по сети, при распространении по системе. У каждой операции есть соответствующее название, время начала и продолжительность. Они могут быть (и обычно бывают) вложенными и упорядоченными, моделируя причинно-следственные связи.

Трек
: Трек представляет все события (в виде операций), которые выполняются при прохождении запроса через систему. Трек можно рассматривать как ориентированный ациклический граф операций или как «трассировку стека», в которой каждая операция представляет работу, выполненную одним компонентом.

Эта взаимосвязь между треком и операциями проиллюстрирована на рис. 11.2, где показаны два разных представления одного и того же запроса, когда он проходит через пять разных служб, выполняя пять операций.

Когда запрос попадает в первую службу, в трек добавляется первая, *корневая операция*, которая образует первый узел в треке. Корневой операции

автоматически назначается глобально уникальный идентификатор трека, который передается вместе с каждым последующим переходом в жизненном цикле запроса. Следующая точка инструментирования создает новое представление операции с указанным идентификатором трека, вставляя или как-то иначе обогащая метаданные, связанные с запросом, перед отправкой идентификатора трека вместе с запросом далее.

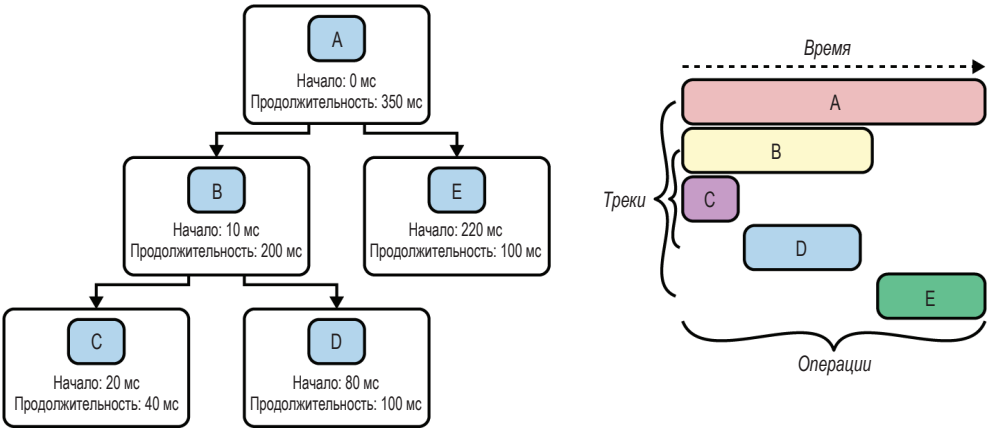


Рис. 11.2 ❖ Два представления трека запроса, проходящего через пять служб, выполняющих пять операций; обычно треки визуализируются как ориентированные графы (слева) и гистограммы (справа) с осью времени, показывающей время начала и продолжительность операций

Каждый шаг в потоке представлен одной операцией. Когда поток выполнения достигает точки инструментирования в одной из этих служб, создается соответствующая запись с любыми метаданными. Эти записи обычно асинхронно сохраняются на диск перед отправкой коллектору, который затем может реконструировать поток выполнения на основе разных записей, созданных разными частями системы.

На рис. 11.2 показаны два наиболее распространенных способа визуализации трека, содержащего пять операций, обозначенных буквами от A до E, в том порядке, в котором они были выполнены. Слева трек представлен в форме ориентированного ациклического графа; корневая операция A начинается в момент времени 0 и длится 350 мс, пока не будет получен ответ от последней службы E. Справа тот же трек представлен в виде гистограммы с осью времени, в которой положение и длина столбика отражают время начала и продолжительность операций соответственно.

Трассировка с использованием OpenTelemetry

Использование OpenTelemetry предполагает инструментирование кода, которое выполняется в два этапа: настройка и собственно инструментирова-

ние. Это верно независимо от целей инструментирования – для трассировки или сбора метрик (или того и другого), – хотя оба этих сценария имеют некоторые отличия. В обоих случаях этап настройки выполняется в программе ровно один раз, обычно в функции `main`, и включает следующие шаги.

1. Первым делом извлекаются и настраиваются экспортеры, соответствующие серверной части, куда будут передаваться данные. Экспортеры реализуют интерфейс `SpanExporter` (который в OpenTelemetry v0.17.0 находится в пакете `go.opentelemetry.io/otel/sdk/export/trace`, часто импортирующемся под псевдонимом `export`). Как рассказывается в разделе «Создание экспортеров трассировки» ниже, в составе OpenTelemetry имеется несколько стандартных экспортеров, но для многих серверов хранения телеметрии существуют свои реализации.
2. Перед инструментированием кода для трассировки экспортеры – а также соответствующие параметры конфигурации – передаются в SDK для создания «провайдера трассировки», который, как рассказывается в разделе «Создание провайдера трассировки» ниже, будет служить основной точкой входа для API трассировки OpenTelemetry на протяжении всего жизненного цикла вашей программы.
3. После создания провайдера трассировки рекомендуется установить его в качестве «глобального» провайдера трассировки. Как рассказывается в разделе «Настройка глобального провайдера трассировки» ниже, это позволит обнаруживать его с помощью функции `otel.GetTracerProvider`, которая помогает библиотекам и другим зависимостям, также использующим API OpenTelemetry, находить SDK и передавать данные телеметрии.

По завершении настройки остается выполнить всего несколько шагов, чтобы инструментировать код.

1. Перед инструментированием выбранной функции нужно получить экземпляр `Tracer`, который играет центральную роль в получении информации о треках и операциях от (обычно глобального) провайдера трассировки. Мы обсудим это более подробно в разделе «Получение экземпляра трассировщика» ниже.
2. После получения дескриптора `Tracer` его можно использовать для создания и запуска экземпляра `Span`, который вы будете использовать для инструментирования своего кода. Мы расскажем об этом более подробно в разделе «Начальная и конечная операции» ниже.
3. Наконец, в операции можно также добавлять свои метаданные, включая удобочитаемые сообщения с отметками времени, называемые *событиями*, и пары ключ/значение, называемые *атрибутами*. Мы рассмотрим метаданные операций в разделе «Установка метаданных операции» ниже.

Пакеты OpenTelemetry для трассировки

В фреймворке OpenTelemetry очень много пакетов. К счастью, для целей этого раздела нам потребуется только часть из них.

Примеры в этом разделе были созданы с использованием версии OpenTelemetry v0.17.0, которая была самой свежей на момент написания этих строк. Если вы решите опробовать у себя примеры кода, представленные в этом разделе, то вам нужно будет импортировать следующие пакеты из этой версии:

```
import (
    "go.opentelemetry.io/contrib/instrumentation/net/http/otelhttp"
    "go.opentelemetry.io/otel"
    "go.opentelemetry.io/otel/exporters/stdout"
    "go.opentelemetry.io/otel/exporters/trace/jaeger"
    "go.opentelemetry.io/otel/label"
    export "go.opentelemetry.io/otel/sdk/export/trace"
    sdktrace "go.opentelemetry.io/otel/sdk/trace"
    "go.opentelemetry.io/otel/trace"
)
```

Как обычно, все примеры кода доступны в репозитории GitHub для этой книги (https://oreil.ly/SznMj).

Создание экспортеров трассировки

Первое, что нужно сделать при использовании OpenTelemetry, – создать и настроить экспортеры. Экспортеры трассировки реализуют интерфейс `SpanExporter`, который в OpenTelemetry v0.17.0 находится в пакете `go.opentelemetry.io/otel/sdk/export/trace`, часто импортируемый под псевдонимом `export`, чтобы уменьшить вероятность конфликтов имен пакетов.

Как рассказывалось в разделе «Компоненты OpenTelemetry» выше, экспортеры OpenTelemetry – это плагины, которые знают, как преобразовывать метрики или трассировки и отправлять их в определенное место, которое может быть локальным файлом журнала (или стандартным выводом) или удаленным хранилищем (например, Jaeger, или коммерческим решением, таким как Honeycomb или Lightstep).

Если вы захотите сделать что-нибудь полезное с собранными данными, то вам понадобится как минимум один экспортер. Обычно достаточно одного, но при необходимости их может быть сколько угодно. Экспортеры создаются и настраиваются один раз при запуске программы перед передачей в OpenTelemetry SDK. Более подробно об этом будет рассказано в разделе «Создание провайдера трассировки» ниже.

В состав OpenTelemetry входит несколько стандартных экспортеров как для трассировки, так и для метрик. Два из них описываются далее.

КОНСОЛЬНЫЙ ЭКСПОРТЕР Консольный экспортер позволяет отправлять данные телеметрии в формате JSON в стандартный вывод. Это очень удобно для отладки или записи в файл журнала. Примечательная особенность консольного экспортера в том, что его также можно использовать для экспорта метрик телеметрии, как будет показано в разделе «Метрики» ниже.

Чтобы создать консольный экспортер, достаточно вызвать метод `stdout.NewExporter`, который в OpenTelemetry v0.17.0 находится в пакете `go.opentelemetry.io/otel/exporters/stdout`.

Подобно большинству функций создания экспортеров, `stdout.NewExporter` – это функция с переменным числом параметров. Она может принимать ноль или более параметров конфигурации. Мы продемонстрируем один из них, управляющий форматированием JSON:

```
stdExporter, err := stdout.NewExporter(
    stdout.WithPrettyPrint(),
)
```

В этом фрагменте использована функция `stdout.NewExporter`, которая возвращает экспортер и значение ошибки. Как выглядит результат, вы увидите, когда мы запустим наш пример в разделе «Собираем все вместе: трассировка» на стр. 373.

Дополнительную информацию о консольном экспортере вы найдете на его странице в документации OpenTelemetry (https://oreil.ly/PEfAI).

ЭКСПОРТЕР ДЛЯ JAEGER Консольный экспортер удобно использовать для журналирования и отладки, но в OpenTelemetry имеется также ряд других экспортеров, предназначенных для пересылки данных в специализированные службы. Одним из таких экспортеров является экспортер для Jaeger.

Экспортер для Jaeger (как следует из названия) преобразует данные телеметрии для передачи в распределенную систему трассировки Jaeger (https://oreil.ly/uMAfg). Получить экземпляр этого экспортера можно с помощью функции `jaeger.NewRawExporter`, как показано ниже:

```
jaegerEndpoint := "http://localhost:14268/api/traces"
serviceName := "fibonacci"

jaegerExporter, err := jaeger.NewRawExporter(
    jaeger.WithCollectorEndpoint(jaegerEndpoint),
    jaeger.WithProcess(jaeger.Process{
        ServiceName: serviceName,
    }),
)
```

В OpenTelemetry v0.17.0 экспортер для Jaeger находится в пакете `go.opentelemetry.io/otel/exporter/trace/jaeger`.

Возможно, вы заметили, что функция `jaeger.NewRawExporter` во многом похожа на `stdout.NewExporter` в том смысле, что она тоже принимает переменное число параметров и возвращает `export.SpanExporter` (экспортер для Jaeger) и значение ошибки.

В вызов `jaeger.NewRawExporter` можно передать следующие параметры:

- `jaeger.WithCollectorEndpoint`, который определяет URL конечной точки HTTP-коллектора целевого процесса Jaeger;
- `jaeger.WithProcess`, который позволяет задать информацию об экспортирующем процессе, в данном случае имя службы.

Существуют и другие конфигурационные параметры, но здесь для краткости я перечислил только два из них. За подробностями обращайтесь к соответствующей странице в документации OpenTelemetry (https://oreil.ly/dOpd5).

Что такое Jaeger?

Jaeger (https://oreil/ly/uMAfg) – это распределенная система трассировки с открытым исходным кодом, толчком к созданию которой стали появление основополагающей статьи в GoogleResearch в 2010 году[1], описывающей инфраструктуру трассировки распределенных систем Dapper, и более ранний проект OpenZipkin (https://zipkin.io/).

Первоначально разрабатывавшаяся (на Go) как внутренний проект в компании Uber Technologies, в ноябре 2016 года эта система была выпущена с открытым исходным кодом под лицензией Apache. В сентябре 2017 года она стала 12-м проектом фонда Cloud Native Computing Foundation и перешла в статус сертифицированного проекта в октябре 2019-го.

Среди ее возможностей – поддержка нескольких типов хранилищ (включая хранилище в памяти для тестирования) и современный веб-интерфейс.

Создание провайдера трассировки

Чтобы получить треки, сначала нужно создать и инициализировать *провайдера трассировки*, который в OpenTelemetry представлен типом `TracerProvider`. В OpenTelemetry v0.17.0 он находится в пакете `go.opentelemetry.io/otel/sdk/trace`, который обычно импортируют под псевдонимом `sdktrace`, чтобы избежать конфликтов имен.

`TracerProvider` – это экземпляр с поддержкой состояния, который служит основной точкой входа в API трассировки OpenTelemetry, включая, как будет показано в следующем разделе, доступ к `Tracer`, который, в свою очередь, служит провайдером новых экземпляров `Span`.

Создается провайдер трассировки с помощью функции `sdktrace.NewTracerProvider`:

```
tp := sdktrace.NewTracerProvider(
    sdktrace.WithSyncer(stdExporter),
    sdktrace.WithSyncer(jaegerExporter))
```

В этом примере мы передаем функции `sdktrace.NewTracerProvider` два экспортера, созданных в разделе «Создание экспортеров трассировки» выше – `stdExporter` и `jaegerExporter`, – и сообщаем SDK, как их использовать для экспорта данных телеметрии.

Есть еще несколько параметров, которые можно передать в вызов `sdktrace.NewTracerProvider`, включая определение `Batcher` или `SpanProcessor`. Их обсуждение выходит за рамки этой книги (так жаль), но вы сможете найти дополнительную информацию о них в спецификации OpenTelemetry SDK Specification (https://oreil.ly/BaL9M).

---

[1] Sigelman, Benjamin H., et al. «Dapper, a Large-Scale Distributed Systems Tracing Infrastructure». Google Technical Report, Apr. 2010. https://oreil.ly/Vh7Ig.

Настройка глобального провайдера трассировки

После создания провайдера трассировки обычно рекомендуется установить его в качестве глобального провайдера вызовом функции `SetTracerProvider`. В OpenTelemetry v0.17.0 она и все глобальные параметры OpenTelemetry находятся в пакете `go.opentelemetry.io/otel`.

Здесь на роль глобального провайдера трассировки мы назначаем провайдера `tp`, созданного в предыдущем разделе:

```
otel.SetTracerProvider(tp)
```

Установка глобального провайдера трассировки делает его обнаруживаемым с помощью функции `otel.GetTracerProvider`. Благодаря этим библиотекам и другим зависимостям, использующим OpenTelemetry API, легче обнаруживать SDK и передавать данные телеметрии:

```
gtp := otel.GetTracerProvider(tp)
```


Если явно не назначить глобального провайдера трассировки, то `otel.GetTracerProvider` вернет ничего не делающую реализацию `TracerProvider`, возвращающую ничего не делающие экземпляры `Tracer`, которые в свою очередь возвращают ничего не делающие экземпляры `Span`.

Получение экземпляра трассировщика

В OpenTelemetry `Tracer` – это специализированный тип, который хранит информацию о треке и операциях, включая текущую активную операцию. Перед инструментированием выбранной функции нужно вызвать метод `Tracer` провайдера трассировки (обычно глобального) и получить экземпляр `trace.Tracer`:

```
tr := otel.GetTracerProvider().Tracer("fibonacci")
```

Метод `Tracer` экземпляра `TracerProvider` принимает строковый параметр, определяющий имя трека. По соглашению, трекам присваиваются имена, включающие имя инструментируемого компонента, обычно библиотеки или пакета.

Теперь, получив трассировщика, его можно использовать для создания и запуска нового экземпляра операции `Span`.

Начальная и конечная операции

Имея дескриптор `Tracer`, его можно использовать для создания и запуска новых экземпляров `Span`, представляющих именованные и синхронизированные операции в рабочем процессе. Иначе говоря, экземпляр `Span` является эквивалентом одного шага в трассировке стека.

В OpenTelemetry v0.17.0 интерфейсы `Span` и `Tracer` находятся в пакете `go.opentelemetry.io/otel/trace`. Их взаимосвязь можно исследовать, заглянув в определение `Tracer`:

```
type Tracer interface {
    Start(ctx context.Context, spanName string, opts ...trace.SpanOption)
        (context.Context, trace.Span)
}
```

Да, это действительно все. `Start` – единственный метод `Tracer` – принимает три параметра: экземпляр контекста `context.Context`, используемый трассировщиком `Tracer` для хранения операций; имя новой операции, которое по соглашению обычно отражает имя функции или компонента; и ноль или более параметров настройки представления операции.

К сожалению, обсуждение доступных параметров настройки представлений операций выходит за рамки этой книги, но если вам интересно, то подробную информацию вы найдете в соответствующей документации Go (https://oreil.ly/ksmfV).

Важно отметить, что `Start` возвращает не только новый экземпляр `Span`, но и контекст `context.Context` – новый экземпляр `Context`, производный от заданного. Как будет показано ниже, это может пригодиться при создании дочерних экземпляров `Span`.

Теперь, имея все необходимое, можно приступать к инструментированию кода. Для этого нужно запросить экземпляр `Span` у трассировщика `Tracer` вызовом его метода `Start`, как показано ниже:

```
const serviceName = "foo"

func main() {
    // НАСТРОЙКА ЭКСПОРТЕРА ОПУЩЕНА ДЛЯ КРАТКОСТИ

    // Получить трассировщик Tracer из TracerProvider.
    tr := otel.GetTracerProvider().Tracer(serviceName)

    // Запустить корневую операцию; получить дочерний контекст (который теперь
    // содержит идентификатор трека) и trace.Span.
    ctx, sp := tr.Start(context.Background(), "main")
    defer sp.End() // Вызов End завершает операцию.

    SomeFunction(ctx)
}
```

В этом фрагменте мы используем метод `Start` трассировщика `Tracer`, чтобы создать и запустить новую операцию `Span`, который возвращает производный контекст и наш экземпляр `Span`. Обратите внимание, как мы гарантируем завершение `Span`, вызывая метод `End` в инструкции `defer`, так что `SomeFunction` полностью охватывается корневой операцией `Span`.

Конечно, мы также должны инструментировать функцию `SomeFunction`. Так как она получает производный контекст, полученный от исходного вызова `Start`, она может использовать его для создания своей собственной вложенной операции:

```
func SomeFunction(ctx context.Context) {
    tr := otel.GetTracerProvider().Tracer(serviceName)
```

```
    _, sp := tr.Start(ctx, "SomeFunction")
    defer sp.End()

    // Сделать здесь что-то ИНТЕРЕСНОЕ!
}
```

Единственные различия между `main` и `SomeFunction` – это имена операций и экземпляры контекста. Важно отметить, что `SomeFunction` использует экземпляр `Context`, полученный из исходного вызова `Start` в `main`.

Установка метаданных операции

И что можно сделать с полученным экземпляром `Span`?

Можно вообще ничего не делать. Главное – не забыть вызвать метод `End` (желательно в инструкции `defer`), чтобы зафиксировать хотя бы продолжительность выполнения вашей функции.

Однако ценность операции можно увеличить за счет добавления метаданных двух типов: *атрибутов* и *событий*.

АТРИБУТЫ Атрибуты – это пары ключ/значение, связанные с операциями. Их можно использовать позже для агрегирования, фильтрации и группировки треков.

Если значения для атрибутов известны заранее, их можно добавить при создании операций, передав методу `tr.Start` с использованием функции `WithAttributes`:

```
ctx, sp := tr.Start(ctx, "attributesAtCreation",
    trace.WithAttributes(
        label.String("hello", "world"), label.String("foo", "bar")))
defer sp.End()
```

Здесь мы вызываем метод `tr.Start`, запускающий новую операцию, передавая ему активный экземпляр `context.Context` и имя. Но `Start` – это функция с переменным количеством параметров, поэтому мы добавляем вызов функции `WithAttributes` для передачи двух строковых атрибутов: `hello=world` и `foo=far`.

Функция `WithAttributes` принимает тип `label.KeyValue` из пакета `go.opentelemetry.io/otel/label`. Значения этого типа можно создавать с помощью различных типизированных методов, таких как `label.String`, как показано в примере выше. Такие методы существуют для всех типов Go (и не только). Дополнительную информацию см. в документации к пакету label (https://oreil.ly/AVkTG).

Атрибуты не обязательно добавлять во время создания операции. Это можно сделать позже, до завершения жизненного цикла операции:

```
answer := LifeTheUniverseAndEverything()
span.SetAttributes(label.Int("answer", answer))
```

СОБЫТИЯ *Событие* – это удобочитаемое сообщение с отметкой времени в операции, которое представляет *что-то*, что происходит в течение времени ее выполнения.

Например, если вашей функции требуется монопольный доступ к ресурсу, защищенному мьютексом, то вам для последующего анализа могут пригодиться события, отмечающие время установки и снятия блокировки:

```
span.AddEvent("Acquiring mutex lock")
mutex.Lock()

// Сделать что-то полезное.

span.AddEvent("Releasing mutex lock")
mutex.Unlock()
```

При желании к событию можно добавить атрибуты:

```
span.AddEvent("Canceled by external signal",
    label.Int("pid", 1234),
    label.String("signal", "SIGHUP"))
```

Автоматическое инструментирование

В широком смысле автоматическое инструментирование – это инструментирование произвольного, в том числе и стороннего, кода. Эта полезная возможность избавит вас от необходимости выполнять большой объем ненужной работы.

OpenTelemetry предлагает поддержку автоматического инструментирования в форме различных оберток и вспомогательных функций для разных популярных фреймворков и библиотек, включая и те, что рассматриваются в этой книге, такие как `net/http`, `gorilla/mux` и `grpc`.

Эта возможность не освобождает вас от необходимости настройки OpenTelemetry при запуске, но избавляет от выполнения некоторых действий, связанных с управлением трассировками.

АВТОМАТИЧЕСКОЕ ИНСТРУМЕНТИРОВАНИЕ NET/HTTP И GORILLA/MUX Поддержка автоматического инструментирования пакета net/http из стандартной библиотеки и стороннего пакета `gorilla/mux`, с которыми мы познакомились в главе 5, когда обсуждали создание веб-службы RESTful, в OpenTelemetry 0.17.0 обеспечивается пакетом `go.opentelemetry.io/contrib/instrumentation/net/http/otelhttp`.

Он необычно прост в использовании. Возьмем, например, стандартную для `net/http` идиому регистрации функции-обработчика в мультиплексоре[1] по умолчанию и запуска HTTP-сервера:

```
func main() {
    http.HandleFunc("/", helloGoHandler)
    log.Fatal(http.ListenAndServe(":3000", nil))
}
```

[1] Мультиплексирующем HTTP-запросы.

В OpenTelemetry можно автоматически инструментировать функцию-обработчик, передав ее в вызов функции `otelhttp.NewHandler`, которая имеет следующую сигнатуру:

```
func NewHandler(handler http.Handler, operation string, opts ...Option)
    http.Handler
```

Функция `otelhttp.NewHandler` принимает и возвращает функцию-обработчик. Он заключает переданную ей функцию-обработчик в другую функцию-обработчик, которая создает представление операции `Span` с указанным именем и параметрами и использует исходную функцию-обработчик как промежуточное звено.

Вот типичный пример применения функции `otelhttp.NewHandler`:

```
func main() {
    http.Handle("/",
        otelhttp.NewHandler(http.HandlerFunc(helloGoHandler), "root"))
    log.Fatal(http.ListenAndServe(":3000", nil))
}
```

Обратите внимание, что перед передачей в вызов `otelhttp.NewHandler` функцию-обработчик нужно преобразовать в `http.HandlerFunc`. В примерах выше в этом не было необходимости, потому что `http.HandleFunc` выполняет это действие автоматически перед вызовом `http.Handle`.

При использовании `gorilla/mux` порядок действий почти такой же, за исключением того, что вместо мультиплексора по умолчанию используется мультиплексор `gorilla`:

```
func main() {
    r := mux.NewRouter()
    r.Handle("/",
        otelhttp.NewHandler(http.HandlerFunc(helloGoHandler), "root"))
    log.Fatal(http.ListenAndServe(":3000", r))
}
```

То же самое нужно проделать с каждой функцией-обработчиком, которую вы решите инструментировать, но в любом случае общий объем кода, необходимый для инструментирования всей вашей службы, остается минимальным.

АВТОМАТИЧЕСКОЕ ИНСТРУМЕНТИРОВАНИЕ GRPC Поддержка автоматического инструментирования фреймворка gRPC, представленного в главе 8, когда мы обсуждали приемы ослабления связей между компонентами системы за счет организации обмена данными, в OpenTelemetry 0.17.0 обеспечивается пакетом `go.opentelemetry.io/contrib/instrumentation/google.golang.org/grpc/otelgrpc`[1].

[1] Это победитель в конкурсе на самое длинное имя среди пакетов, по крайней мере в этой книге.

По аналогии с `net/http`, автоматическое инструментирование gRPC реализуется просто и использует *перехватчики gRPC*. Мы не говорили выше о перехватчиках gRPC, и, к сожалению, полное их обсуждение выходит за рамки этой книги. Их можно описать как gRPC-эквивалент промежуточных функций в `gorilla/mux`, которые мы использовали в разделе «Сброс нагрузки» главы 9 для реализации автоматического сброса нагрузки.

Как следует из их названия, перехватчики gRPC могут перехватывать запросы и ответы gRPC, например для внедрения информации в запрос, коррекции ответа перед передачей его клиенту или для реализации сквозных функций, таких как авторизация, журналирование или кеширование.

Если вы хотите узнать немного больше о перехватчиках gRPC, то статья «Interceptors in gRPC-Web» в блоге gRPC (https://oreil.ly/R0MGm) послужит вам хорошим введением в эту тему. Более подробный обзор вы найдете в книге Касуна Индрасири (Kasun Indrasiri) и Данеша Куруппу (Danesh Kuruppu) *gRPC: Up and Running* (O'Reilly; https://oreil.ly/N50q7)[1].

Взглянув на фрагмент исходного кода службы из раздела «Реализация службы gRPC» главы 8, вы увидите две функции:

```
s := grpc.NewServer()
pb.RegisterKeyValueServer(s, &server{})
```

В этом фрагменте создается новый сервер gRPC, который затем передается нашему автоматически сгенерированному пакету для регистрации.

Перехватчики можно добавить в сервер gRPC с помощью функции `grpc.UnaryInterceptor` и/или `grpc.StreamInterceptor`, первый из них используется для перехвата унарных (стандартный запрос/ответ) методов служб, а второй – для перехвата потоковых методов.

Чтобы подготовить сервер gRPC к автоматическому инструментированию, нужно вызвать одну или обе эти функции и добавить один или несколько предопределенных перехватчиков OpenTelemetry, в зависимости от типов запросов, обрабатываемых вашей службой:

```
s := grpc.NewServer(
    grpc.UnaryInterceptor(otelgrpc.UnaryServerInterceptor()),
    grpc.StreamInterceptor(otelgrpc.StreamServerInterceptor()),
)

pb.RegisterKeyValueServer(s, &server{})
```

Если служба, созданная нами в главе 8, использует исключительно унарные методы, то предыдущий пример кода демонстрирует добавление перехватчиков как для унарных, так и для потоковых методов.

ПОЛУЧЕНИЕ ТЕКУЩЕЙ ОПЕРАЦИИ ИЗ КОНТЕКСТА При использовании автоматического инструментирования для каждого запроса будет автомати-

[1] *Индрасири Касун, Курупу Данеш*. gRPC. Запуск и эксплуатация облачных приложений. Go и Java для Docker и Kubernetes. СПб.: Питер, 2021. ISBN: 978-5-4461-1737-6. – *Прим. перев.*

чески создаваться новый трек. Это удобно, но это также означает, что у вас не будет под рукой текущего представления операции `Span`, куда можно было бы добавить метаданные в виде атрибутов и событий. И как быть?

Не волнуйтесь! Наш прикладной фреймворк включает представление операции в текущий контекст, поэтому его легко получить:

```
func printSpanHandler(w http.ResponseWriter, req *http.Request) {
    ctx := req.Context()                    // Получить контекст запроса

    span := trace.SpanFromContext(ctx)      // Получить текущую операцию

    fmt.Printf("current span: %v\n", span) // Почему бы не вывести ее?
}
```

А как насчет клиентов?

Для полноценной распределенной трассировки метаданные трассировки должны пересылаться через границы служб. Это означает, что инструментирование также необходимо выполнять на стороне клиента.

К счастью, OpenTelemetry поддерживает автоматическое инструментирование клиентов `http` и gRPC.

К сожалению, в этой главе недостаточно места для подробного описания автоматического инструментирования клиентов. Однако примеры доступны в репозитории GitHub для данной книги (https://oreil.ly/SznMj).

Собираем все вместе: трассировка

Давайте теперь используем все, что мы обсудили в этом разделе, и создадим небольшую веб-службу. Мы собираемся инструментировать эту службу для трассировки, поэтому в идеале она должна выполнять много вызовов функций и оставаться достаточно маленькой.

Давайте создадим службу вычисления чисел Фибоначчи. Требования к ней минимальные: она должна принимать HTTP-запрос GET для получения *n*-го числа Фибоначчи, указанного в параметре `n` в строке запроса GET. Например, запрос для получения шестого числа Фибоначчи будет выглядеть так: `http://localhost:3000?n=6`.

Служба будет состоять из трех функций.

API службы
: Эта функция будет вычислять требуемое число Фибоначчи – по запросу обработчика службы – путем рекурсивного вызова самой себя, при этом каждый вызов будет генерировать свое собственное представление операции `Span`.

Обработчик службы
: Это функция-обработчик HTTP, как определено в пакете `net/http`. Она будет использоваться так же, как в разделе «Создание HTTP-сервера с ис-

пользованием net/http» главы 5, для получения запроса клиента, вызова API службы и возврата результата.

Функция main
: В функции `main` создаются и регистрируются экспортеры OpenTelemetry, функция-обработчик службы передается в фреймворк HTTP, и запускается сервер HTTP.

API службы вычисления чисел Фибоначчи

API службы находится в самом ее сердце – это то место, где выполняются фактические вычисления. В данном случае это конкурентная реализация для вычисления *n*-го числа Фибоначчи.

Так же как любой хороший API, эта функция не знает (или не заботится) о том, как она используется, поэтому она ничего не знает ни об HTTP-запросах, ни об ответах:

```
func Fibonacci(ctx context.Context, n int) chan int {
    ch := make(chan int)

    go func() {
        tr := otel.GetTracerProvider().Tracer(serviceName)

        cctx, sp := tr.Start(ctx,
            fmt.Sprintf("Fibonacci(%d)", n),
            trace.WithAttributes(label.Int("n", n)))

        defer sp.End()

        result := 1
        if n > 1 {
            a := Fibonacci(cctx, n-1)
            b := Fibonacci(cctx, n-2)
            result = <-a + <-b
        }

        sp.SetAttributes(label.Int("result", result))

        ch <- result
    }()

    return ch
}
```

В этом примере функция `Fibonacci` не знает, как она используется, *но знает* о пакете OpenTelemetry. Функции автоматического инструментирования могут трассировать только то, что обертывается ими. Все, что находится в API, должно инструментироваться вручную.

Использование `otel.GetTracerProvider` в этой функции гарантирует получение глобального провайдера `TracerProvider`, если он был настроен потребителем. Если глобальный провайдер трассировки не назначен, этот вызов будет выполнен впустую.

В качестве дополнительного упражнения выделите минуту и добавьте поддержку отмены в контекст Context в функции Fibonacci.

Функция-обработчик службы вычисления чисел Фибоначчи

Это функция-обработчик HTTP, как определено в пакете net/http.

Она будет использоваться так же, как в разделе «Создание HTTP-сервера с использованием net/http» главы 5, для получения запроса клиента, вызова API службы и возврата результата:

```
func fibHandler(w http.ResponseWriter, req *http.Request) {
    var err error
    var n int

    if len(req.URL.Query()["n"]) != 1 {
        err = fmt.Errorf("wrong number of arguments")
    } else {
        n, err = strconv.Atoi(req.URL.Query()["n"][0])
    }

    if err != nil {
        http.Error(w, "couldn't parse index n", 400)
        return
    }

    // Получить текущий контекст из входящего запроса
    ctx := req.Context()

    // Вызвать дочернюю функцию и передать ей контекст запроса.
    result := <-Fibonacci(ctx, n)

    // Получить экземпляр Span, связанный с текущим контекстом, и
    // присоединить параметр и результат в виде атрибутов.
    if sp := trace.SpanFromContext(ctx); sp != nil {
        sp.SetAttributes(
            label.Int("parameter", n),
            label.Int("result", result))
    }

    // Послать ответ с результатом.
    fmt.Fprintln(w, result)
}
```

Обратите внимание, что функция не создает и не завершает операцию Span – механизм автоматического инструментирования сделает это сам.

Однако она *устанавливает* некоторые атрибуты текущей операции. Для этого она использует trace.SpanFromContext, чтобы получить текущую операцию из контекста запроса. Имея операцию, в нее можно добавить любые метаданные.

Функция trace.SpanFromContext вернет nil, если не сможет найти операцию Span, связанную с переданным ей контекстом.

Функция main службы

На данный момент мы выполнили всю тяжелую работу и нам осталось только настроить OpenTelemetry, зарегистрировать функцию-обработчик с помощью мультиплексора HTTP по умолчанию и запустить службу:

```
const (
    jaegerEndpoint = "http://localhost:14268/api/traces"
    serviceName = "fibonacci"
)

func main() {
    // Создать и настроить консольный экспортер
    stdExporter, err := stdout.NewExporter(
    stdout.WithPrettyPrint(),
    )
    if err != nil {
        log.Fatal(err)
    }

    // Создать и настроить экспортер для Jaeger
    jaegerExporter, err := jaeger.NewRawExporter(
        jaeger.WithCollectorEndpoint(jaegerEndpoint),
        jaeger.WithProcess(jaeger.Process{
            ServiceName: serviceName,
        }),
    )
    if err != nil {
        log.Fatal(err)
    }

    // Создать провайдера трассировки и зарегистрировать в нем
    // вновь созданных экспортеров.
    tp := sdktrace.NewTracerProvider(
        sdktrace.WithSyncer(stdExporter),
        sdktrace.WithSyncer(jaegerExporter))

    // Теперь можно зарегистрировать tp как провайдер трассировки otel.
    otel.SetTracerProvider(tp)

    // Зарегистрировать и инструментировать обработчик службы
    http.Handle("/",
        otelhttp.NewHandler(http.HandlerFunc(fibHandler), "root"))

    // Запустить службу на порту 3000
    log.Fatal(http.ListenAndServe(":3000", nil))
}
```

Как видите, метод `main` просто создает двух экспортеров (консольных и для Jaeger) и настраивает провайдера трассировки, как мы это уже делали в разделе «Создание экспортеров трассировки» выше. Обратите внимание на значение `jaegerEndpoint`, которое предполагает наличие локальной службы Jaeger. Мы запустим ее на следующем шаге.

Последние две инструкции выполняют автоматическое инструментирование и регистрацию функции-обработчика и запускают службу HTTP, как мы делали это в разделе «Автоматическое инструментирование» выше.

Запуск служб

Прежде чем продолжить, нужно запустить службу Jaeger, которая будет принимать данные телеметрии от нашего экспортера для Jaeger. За дополнительной информацией о Jaeger обращайтесь к врезке «Что такое Jaeger?» выше.

Если у вас установлен Docker, то вы можете запустить службу Jaeger следующей командой:

```
$ docker run -d --name jaeger \
  -p 16686:16686 \
  -p 14268:14268 \
  jaegertracing/all-in-one:1.21
```

После запуска службы можно обратиться к ее веб-интерфейсу, перейдя по адресу `http://localhost:16686`. Очевидно, что пока никаких данных там нет.

А теперь самое интересное: запустите свою службу, вызвав ее функцию `main`:

```
$ go run .
```

Терминал должен приостановиться. Как обычно, прервать работу службы можно комбинацией Ctrl-C.

Затем в другом терминале отправьте запрос:

```
$ curl localhost:3000?n=6
13
```

После короткого ожидания вы будете вознаграждены результатом. В данном случае числом 13.

Будьте осторожны, выбирая число для параметра `n`. Если оно окажется слишком большим, служба может потратить на вычисления очень много времени или даже завершиться с ошибкой.

Вывод консольного экспортера

Теперь, после выполнения запроса, взгляните на терминал, в котором вы запустили службу. В нем должна появиться последовательность блоков JSON, как показано ниже:

```
[
    {
        "SpanContext":{
            "TraceID":"4253c86eb68783546b8ae3b5e59b4a0c",
            "SpanID":"817822981fc2fb30",
            "TraceFlags":1
        },
        "ParentSpanID":"0000000000000000",
```

```
        "SpanKind":1,
        "Name":"main",
        "StartTime":"2020-11-27T13:50:29.739725-05:00",
        "EndTime":"2020-11-27T13:50:29.74044542-05:00",
        "Attributes":[
            {
                "Key":"n",
                "Value":{
                    "Type":"INT64",
                    "Value":6
                }
            },
            {
                "Key":"result",
                "Value":{
                    "Type":"INT64",
                    "Value":13
                }
            }
        ],
        "ChildSpanCount":1,
        "InstrumentationLibrary":{
            "Name":"fibonacci",
            "Version":""
        }
    }
]
```

Эти блоки JSON вывел консольный экспортер (в котором, как вы помните, мы настроили форматирование вывода). Число блоков должно совпадать с числом операций (span), которых довольно много.

Предыдущий пример (усеченный) взят из корневой операции. Как видите, он содержит довольно много интересной информации, включая время начала и окончания, а также идентификаторы трека и операции. Он даже включает два атрибута, явно установленных нами: входное значение `n` и результат.

Просмотр результатов в Jaeger

Теперь, когда мы сгенерировали и отправили трек в Jaeger, можно попробовать визуализировать его. Для этой цели в Jaeger имеется довольно удобный веб-интерфейс!

Чтобы убедиться в этом, откройте страницу `http://localhost:16686` в веб-браузере. Выберите службу Fibonacci в раскрывающемся списке Service (Служба) и щелкните на кнопке Find traces (Найти треки). После этого вы должны увидеть результат, как показано на рис. 11.3.

Каждый столбик на графике представляет одну операцию. Вы даже можете просмотреть данные, соответствующие определенной операции, щелкнув на ней. В ответ на экране появится та же информация, которую мы видели в выводе консольного экспортера в разделе «Вывод консольного экспортера» выше.

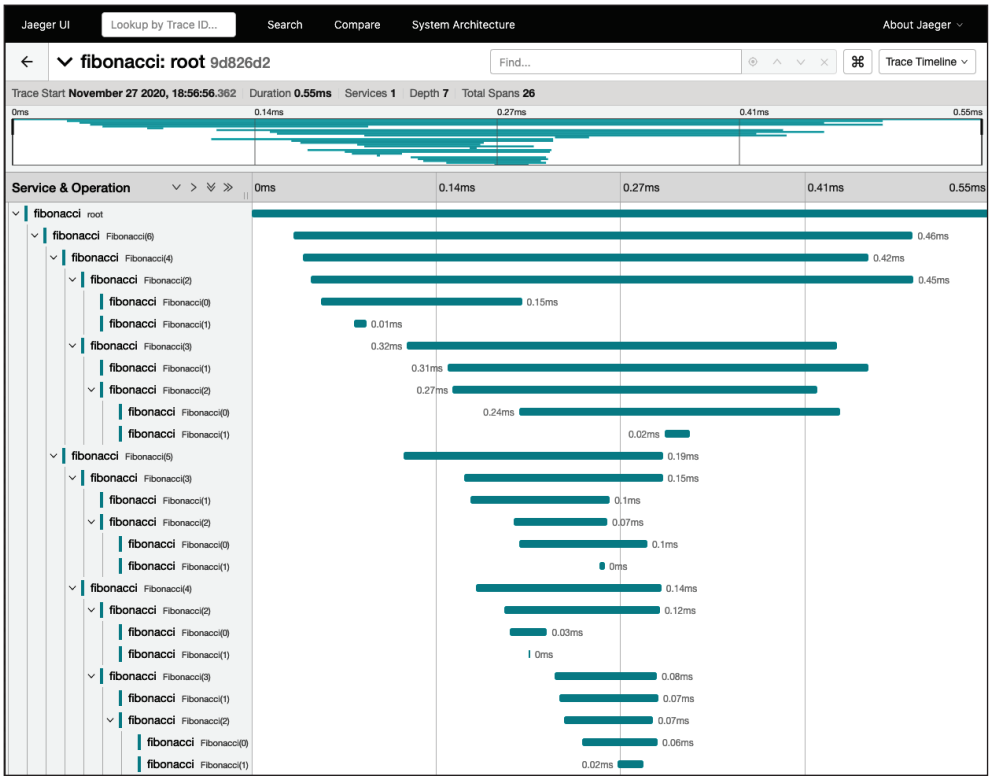


Рис. 11.3 ❖ Скриншот интерфейса Jaeger
с результатами вызова службы вычисления чисел Фибоначчи

Метрики

Метрики – это набор числовых данных, характеризующих изменение поведения компонента, процесса или действия с течением времени. Множество потенциальных источников метрик огромно и включает (но не ограничивается ими), например, параметры потребления вычислительных ресурсов (процессорного времени, памяти, дискового пространства и операций сетевого ввода/вывода), инфраструктурные показатели (количество действующих экземпляров, события автоматического масштабирования), прикладные показатели (количество запросов, количество ошибок) и бизнес-показатели (доход, количество регистраций клиентов, частота отказов, количество отмен покупок). Конечно, это лишь несколько тривиальных примеров. Для сложной системы *мощность метрик* может достигать многих тысяч или даже миллионов.

Точка данных метрики, представляющая одно наблюдение конкретного целевого аспекта (например, количество запросов, полученных конечной точкой), называется *образцом*. Каждый образец имеет имя, значение и отметку времени с точностью до миллисекунды. Также – по крайней мере,

в современных системах, таких как Prometheus (https://prometheus.io/), – множество пар ключ/значение называется *метками*.

Мощность метрик

Мощность метрик – важное понятие в наблюдаемости. Оно исходит из теории множеств, где определяется как количество элементов в множестве. Например, множество A = {1, 2, 3, 5, 8} содержит пять элементов, поэтому мощность (количество элементов) множества A равна пяти.

Позднее этот термин был заимствован разработчиками баз данных для обозначения количества различных значений в столбце таблицы[1]. Например, столбец «цвет глаз» будет иметь низкую мощность, а «имя пользователя» – довольно высокую.

Однако совсем недавно термин «мощность» начал использоваться в сфере мониторинга, где стал обозначать количество уникальных комбинаций названий и измерений метрик – количество различных значений меток, прикрепленных к определенной метрике. Мощность имеет решающее значение для наблюдаемости. Большая мощность означает, что данные можно запросить разными способами и выше вероятность того, что вы сможете задать ей вопрос, о котором раньше даже не думали.

Единственный образец имеет ограниченную ценность, но упорядоченная последовательность образцов с одинаковыми именами и метками – *временной ряд* – может быть невероятно полезной. Как показано на рис. 11.4, сбор образцов, упорядоченных во времени, позволяет визуализировать метрики путем нанесения точек данных на график, что, в свою очередь, упрощает выявление тенденций или определение аномалий и выбросов.

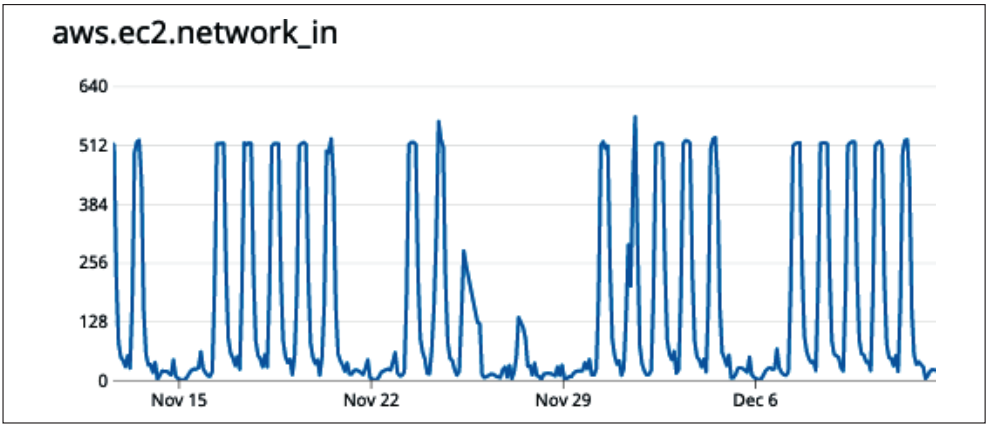


Рис. 11.4 ❖ Упорядочение образцов во времени позволяет визуализировать их в форме графика

[1] Он также может относиться к типу отношения между двумя таблицами в базе данных (т. е. «один к одному», «один ко многим» или «многие ко многим»), но это определение в данном контексте является, пожалуй, наименее актуальным.

На рис. 11.4 показан временной ряд метрики `aws.ec2.network_in` для одного экземпляра AWS EC2. Время отложено по оси X (в данном примере – один месяц с 15 ноября до 15 декабря 2020 г.). Вдоль оси Y откладывается мгновенная скорость получения сетевых данных. Визуализировав такой временной ряд, можно заметить, что трафик к экземпляру увеличивается каждый будний день.

Интересно, что период 25–27 ноября – за день до и день после Дня благодарения в Соединенных Штатах – является исключением.

Однако истинная сила метрик не в способности представлять их в графическом виде: их числовая природа позволяет использовать их для математического моделирования. Например, можно использовать тренд-анализ для обнаружения аномалий или прогнозирования будущих состояний, которые, в свою очередь, можно использовать для принятия решений или отправки предупреждений.

Два способа передачи метрик: принудительная и по запросу

Во вселенной метрик существуют две основные архитектуры: с принудительной передачей и с передачей по запросу (по характеру связи между компонентами, находящимися под наблюдением, и сервером, собирающим метрики).

В архитектуре с принудительной передачей компоненты «принудительно отправляют» свои данные центральному серверу сбора данных. В архитектуре с передачей по запросу, наоборот, сервер сбора данных активно извлекает метрики из конечных точек HTTP компонентов (или дополнительных служб, развернутых специально для этой цели, которые также ошибочно называют «экспортерами»; см. врезку «Экспортеры Prometheus» ниже), «запрашивая» их. Оба подхода показаны на рис. 11.5.

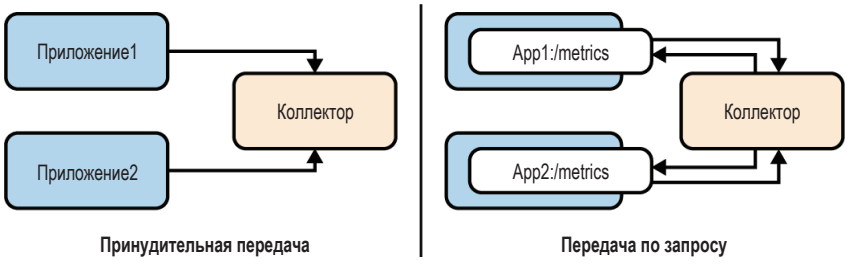


Рис. 11.5 ❖ В архитектуре с принудительной передачей метрик (слева) телеметрия принудительно отправляется в центральный сервер сбора данных; в архитектуре с передачей метрик по запросу (справа) метрики активно извлекаются коллектором из открытых конечных точек

Далее следует краткое описание этих двух подходов, а также очень ограниченный список аргументов за и против каждого подхода. К сожалению,

многие аргументы наполнены множеством нюансов – слишком тонких, чтобы вникать в них здесь, – поэтому мы ограничимся только наиболее распространенными.

Принудительная отправка метрик

В подходе с принудительной отправкой приложение само или через параллельный процесс агента периодически отправляет метрики центральному серверу сбора данных. Реализации с принудительной отправкой, такие как Ganglia, Graphite и StatsD, как правило, пользуются большей популярностью (и даже используются по умолчанию), возможно, отчасти потому, что модель принудительной передачи несколько проще для рассуждений.

Принудительная отправка обычно имеет однонаправленный характер, метрики, полученные компонентами или агентом мониторинга, отправляются центральному серверу. Это немного снижает нагрузку на сеть по сравнению с (двунаправленной) моделью передачи по запросу и может уменьшить сложность модели сетевой безопасности, потому что компоненты не должны открывать конечные точки для сбора метрик. Также модель с принудительной передачей проще в использовании, когда требуется обеспечить мониторинг очень эфемерных компонентов, таких как короткоживущие контейнеры или бессерверные функции.

Однако модель с принудительной передачей имеет некоторые недостатки. Во-первых, для ее реализации нужно знать, куда отправлять метрики. Конечно, существует множество способов сделать такую информацию доступной, но у каждого есть свои недостатки: от фиксированных адресов (которые «зашиты» в код и сложно изменяются) до поиска в DNS или в реестре служб (что может приводить к недопустимым задержкам). Масштабирование тоже иногда может превратиться в проблему, потому что большое количество компонентов, активно посылающих метрики, способно провести DDoS-атаку на сервер сбора метрик.

Передача метрик по запросу

В модели передачи по запросу сервер сбора метрик периодически (с некоторой настраиваемой частотой) обращается к конечной точке, предоставляемой компонентом или прокси-сервером и созданной специально для получения метрик. Возможно, самый известный пример системы, основанной на модели сбора метрик по запросу, – это Prometheus (https://prometheus.io/).

Метод передачи по запросу имеет ряд заметных преимуществ. Организация конечной точки для отправки метрик ослабляет связь между компонентами и коллектором, что обеспечивает все преимущества слабой связи. Например, упрощается наблюдение за службой во время разработки, и даже появляется возможность вручную проверить работоспособность компонента с помощью веб-браузера. Кроме того, с помощью модели передачи по запросу намного проще определить, продолжает ли функционировать целевая служба.

Однако этому методу сопутствует проблема обнаружения, поскольку коллектор должен каким-то образом узнать, где искать службы, откуда он должен запрашивать метрики. Эта проблема может оказаться особенно острой, если ваша система не использует механизмы динамического обнаружения служб. Дополнительным препятствием могут стать балансировщики нагрузки, перенаправляющие каждый запрос случайному экземпляру, что может значительно снизить скорость сбора информации (поскольку каждый из N экземпляров получает 1/N запросов) и запутать ситуацию (поскольку все экземпляры имеют тенденцию выглядеть как одна цель). Наконец, сбор метрик по запросу может несколько затруднить мониторинг короткоживущих эфемерных процессов, таких как бессерверные функции, что требует организации шлюза для принудительной передачи.

Что такое Prometheus?

Prometheus (https://prometheus.io/) – это набор инструментов с открытым исходным кодом для мониторинга и оповещения. Он использует модель передачи метрик по запросу через HTTP, сохраняя их как многомерные значения в своей базе данных временных рядов.

Prometheus состоит из базового сервера, который отвечает за сбор и хранение данных, а также множества дополнительных компонентов, включая шлюз для организации принудительной передачи сведений о короткоживущих заданиях и диспетчер предупреждений. Prometheus задумывался не как решение для оперативного мониторинга, тем не менее он предоставляет упрощенный веб-интерфейс и язык запросов PromQL для доступа к данным.

В январе 2015 года компания SoundCloud открыла исходный код Prometheus под лицензией Apache, а в мае 2016 года Prometheus присоединился к Cloud Native Computing Foundation в качестве второго основного проекта (после Kubernetes). В августе 2018 года он получил статус сертифицированного проекта.

Какой подход лучше?

Подходы принудительной передачи и передачи по запросу кажутся полярными противоположностями друг друга, поэтому люди часто задаются вопросом, что лучше[1]. Это сложный вопрос, и, как это часто бывает при сравнении технологий, ответом является «это зависит от обстоятельств».

Конечно, это никогда не мешало мотивированным программистам резко оспаривать ту или иную сторону, но, как бы то ни было, «лучший» подход – это тот, который отвечает требованиям вашей системы. Конечно, отвечать требованиям могут оба подхода. Мы, люди с техническим складом ума, не терпим двусмысленности, но все равно упорно настаиваем на ее существовании.

Итак, я закрою этот раздел словами Брайана Бразиля (Brian Brazil), разработчика Prometheus:

[1] Что бы ни означало это «лучше».

С инженерной точки зрения вопрос о том, какой способ передачи лучше – принудительно или по запросу, – в действительности не имеет значения. Оба имеют свои преимущества и недостатки, и, приложив некоторые усилия, вы сможете реализовать любой из них[1].

Метрики в OpenTelemetry

На момент написания этих строк API метрик OpenTelemetry все еще находился в стадии альфа-версии, поэтому в нем все еще имеются некоторые проблемы, которые необходимо исправить, и несколько несоответствий с API трассировки, которые еще предстоит устранить.

При этом значительная поддержка OpenTelemetry со стороны частных лиц и сообщества в сочетании с довольно впечатляющими темпами развития делает его хорошим выбором не только для включения в эту книгу, но и как наиболее вероятного кандидата на звание золотого стандарта телеметрии в будущем.

По большей части механизм метрик в OpenTelemetry работает так же, как механизм трассировки, но они имеют достаточно различий, чтобы вызвать некоторую путаницу. Этап настройки инструментирования, как с целью трассировки, так и с целью сбора метрик, выполняется в программе ровно один раз, обычно в функции `main`, и включает следующие шаги.

1. Первым делом нужно создать и настроить экспортер, соответствующий целевому серверу. Экспортеры метрик реализуют интерфейс `metric.Exporter`, который в OpenTelemetry v0.17.0 находится в пакете `go.opentelemetry.io/otel/sdk/export/metric`. Как вы узнаете в разделе «Создание экспортеров метрик» ниже, в OpenTelemetry имеется несколько стандартных экспортеров, но, в отличие от экспортеров трассировок, экспортеры метрик пока можно использовать только по одному за раз.
2. Перед инструментированием кода для сбора метрик на основе экспортера нужно создать глобального «провайдера метрик», который будет служить основной точкой входа в API метрик OpenTelemetry на протяжении всего жизненного цикла вашей программы. Как будет показано в разделе «Установка глобального провайдера метрик» ниже, это позволит обнаруживать экспортер метрик с помощью функции `otel.GetMeterProvider`, которая помогает библиотекам и другим зависимостям, также использующим API OpenTelemetry, находить SDK и передавать данные телеметрии.
3. Если сервер для сбора метрик использует механизм передачи по запросу, такой как Prometheus, то вам нужно определить конечную точку для доступа к метрикам. В разделе «Экспортирование конечной точки метрик» вы увидите, как Prometheus использует для этого стандартный пакет `http`.

По завершении настройки остается выполнить всего несколько шагов, чтобы инструментировать код:

---

[1] Kiran, Oliver. «Exploring Prometheus Use Cases with Brian Brazil». *The New Stack Makers*, 30 Oct. 2016. https://oreil.ly/YDlek.

1. Перед инструментированием выбранной функции нужно получить экземпляр структуры `Meter`, посредством которой настраивается и передается весь набор метрик. Мы обсудим это более подробно в разделе «Получение экземпляра Meter» ниже.
2. Наконец, после получения экземпляра `Meter` его можно использовать для инструментирования кода. Сделать это можно двумя способами: явно записывать метрики или создать наблюдателей, которые автоматически и асинхронно будут выполнять сбор данных. Оба подхода мы рассмотрим в разделе «Инструменты метрик» ниже.

Пакеты OpenTelemetry для сбора метрик

В фреймворке OpenTelemetry очень много пакетов. К счастью, для целей этого раздела нам потребуется только часть из них.

Примеры в данном разделе были созданы с использованием версии OpenTelemetry v0.17.0, которая была самой свежей на момент написания этих строк. Если вы решите опробовать у себя примеры кода, представленные в этом разделе, то вам нужно будет импортировать следующие пакеты из данной версии:

```
import (
    "go.opentelemetry.io/otel"
    "go.opentelemetry.io/otel/exporters/metric/prometheus"
    "go.opentelemetry.io/otel/label"
    "go.opentelemetry.io/otel/metric"
)
```

Как обычно, все примеры кода доступны в репозитории GitHub для этой книги (https://oreil.ly/SznMj).

Создание экспортеров метрик

Как и в случае с трассировкой, первое, что нужно сделать при использовании OpenTelemetry, – создать и настроить экспортеров. Экспортеры метрик реализуют интерфейс `metric.Exporter`, который в OpenTelemetry v0.17.0 находится в пакете `go.opentelemetry.io/otel/sdk/export/metric`.

Способы создания экспортеров метрик немного различаются в зависимости от реализации, но для всех типов экспортеров характерно наличие функции-конструктора `NewExportPipeline`, по крайней мере в стандартных пакетах OpenTelemetry.

Например, чтобы получить экземпляр экспортера для Prometheus, нужно вызвать функцию `NewExportPipeline` из пакета `go.opentelemetry.io/otel/exporters/metric/prometheus`:

```
prometheusExporter, err := prometheus.NewExportPipeline(prometheus.Config{})
```

Эта инструкция создаст экспортера и настроит его с учетом направлений в аргументе `prometheus.Config`. Любые параметры, не переопределенные в `Config`, получат значения по умолчанию.

Аргумент `prometheus.Config` также позволяет указывать различные варианты поведения. К сожалению, обсуждение подробностей выходит за рамки этой книги, но если вам интересно, то можете заглянуть в код `Config` (https://oreil.ly/fKIzt) и в код клиента Prometheus на Go (https://oreil.ly/biCJn), в котором вы без труда разберетесь.

Установка глобального провайдера метрик

Для получения трассировок в OpenTelemetry имеется «провайдер трассировки», который предоставляет значения `Tracer`. Для получения метрик в OpenTelemetry имеется *провайдер метрик*, который предоставляет значения `Meter`, с помощью которых настраивается и формируется весь набор метрик.

Как вы наверняка помните, при работе с экспортерами трассировки определение глобального провайдера трассировки выполняется в два этапа: создание и настройка экземпляра провайдера трассировки, а затем назначение его на роль глобального провайдера трассировки.

Провайдер метрик работает немного иначе: вместо использования одного или нескольких экспортеров для создания и настройки провайдера (как в случае с `TracerProvider`) провайдер метрик обычно извлекается *из* экспортера метрик и затем передается непосредственно в функцию `otel.SetMeterProvider`:

```
// Получить провайдера метрик из экспортера.
mp := prometheusExporter.MeterProvider()

// Назначить его на роль глобального провайдера метрик.
otel.SetMeterProvider(mp)
```

К сожалению, из-за такой организации вы можете использовать только одного экспортера метрик за раз, потому что провайдер метрик предоставляется экспортером, а не наоборот. Очевидно, что это существенное отличие от API трассировки, и я ожидаю, что оно будет устранено по мере перехода API метрик OpenTracing в бета-версию.

Существует также вспомогательная функция `prometheus.InstallNewPipeline`, которую можно использовать вместо явного вызова `prometheus.NewExportPipeline` и `otel.SetMeterProvider`.

Экспортирование конечной точки метрик

В Prometheus используется подход к сбору метрик по запросу, поэтому данные телеметрии, которые требуется отправить, должны быть доступны через конечную точку HTTP, которую Prometheus сможет опросить.

Для создания такой конечной точки можно использовать стандартный пакет `http`, который, как мы уже не раз показывали в этой книге, требует минимальной настройки и довольно прост в использовании.

Как рассказывалось в разделе «Создание HTTP-сервера с использованием net/http» главы 5, для запуска минимального HTTP-сервера в Go требуется вызвать как минимум две функции:

- `http.Handle`, чтобы зарегистрировать функцию-обработчик, реализующую интерфейс `http.Handler`;
- `http.ListenAndServe`, чтобы запустить сервер для обслуживания поступающих запросов.

Но у экспортера OpenTelemetry для Prometheus есть в запасе довольно изящный трюк: он реализует интерфейс `http.Handler`, который позволяет передавать его напрямую в `http.Handle`, чтобы он действовал как функция-обработчик для конечной точки метрики:

```
// Зарегистрировать экспортер как обработчик для пути "/metrics".
http.Handle("/metrics", prometheusExporter)

// Запустить HTTP-сервер на порту 3000.
log.Fatal(http.ListenAndServe(":3000", nil))
```

В этом примере мы передаем экспортер для Prometheus непосредственно в `http.Handle`, чтобы зарегистрировать его как обработчик для пути `"/metrics"`. Трудно найти более удобное место.

Конечно, выбор имени конечной точки для доступа к метрикам зависит только от вас, но многие предпочитают имя `metrics`. Это же имя по умолчанию ищет Prometheus.

Экспортеры Prometheus

Экспортировать конечную точку для доступа к метрикам в приложении, которое является стандартной веб-службой, написанной на Go, относительно просто. Но что, если вам понадобится организовать сбор данных JMX из приложения на основе JVM, запрашивать метрики из базы данных PostgreSQL или получать системные метрики из развернутого экземпляра Linux или Windows?

К сожалению, не все источники данных подконтрольны вам, и очень немногие из них предоставляют свои собственные конечные точки для получения метрик.

В этом (очень распространенном) сценарии типичным решением является развертывание *экспортера Prometheus*. Экспортер Prometheus (не путать с экспортером OpenTelemetry) – это специализированный адаптер, который работает как служба, собирающая желаемые метрики и предоставляющая их через конечную точку.

На момент написания этих строк в документации Prometheus (https://oreil.ly/ZppOm) было перечислено более 200 различных экспортеров Prometheus. Кроме них, существует большое количество экспортеров, созданных сообществом и не упоминаемых в документации. На этой странице вы найдете полный и актуальный список экспортеров, а здесь я перечислю только наиболее популярные.

Node Exporter (https://oreil.ly/lq2Rv)
: Экспортирует метрики оборудования и операционной системы, предоставляемые ядрами *NIX.

Windows Exporter (https://oreil.ly/2LIFK)
: Экспортирует метрики оборудования и операционной системы, предоставляемые Windows.

JMX Exporter (https://oreil.ly/rY0jg)
: Экспортирует MBean-компоненты целевого объекта JMX (https://oreil.ly/11HLz).

PostgreSQL Exporter (https://oreil.ly/yhZrn)
Извлекает и экспортирует метрики сервера PostgreSQL.

Redis Exporter (https://oreil.ly/XRkfl)
Извлекает и экспортирует метрики сервера Redis.

Blackbox Exporter (https://oreil.ly/VuM0U)
Позволяет blackbox проверять конечные точки через HTTP, HTTPS, DNS, TCP или ICMP.

Push Gateway (https://oreil.ly/JxO3w)
Кеш метрик, который позволяет эфемерным и пакетным заданиям экспортировать свои метрики, передавая их посреднику. Технически это один из основных компонентов Prometheus, а не отдельный экспортер.

Получение экземпляра Meter

Прежде чем инструментировать операцию, нужно сначала получить экземпляр `Meter` из `MeterProvider`.

Как будет показано в разделе «Инструменты метрик» ниже, тип `metric.Meter`, который находится в пакете `go.opentelemetry.io/otel/metric`, является средством настройки всей коллекции метрик и определяет, записывать ли метрики как серии синхронных измерений или как асинхронные наблюдения.

Вот как можно получить экземпляр `Meter`:

```
meter := otel.GetMeterProvider().Meter("fibonacci")
```

Возможно, вы заметили, что эта инструкция выглядит почти так же, как инструкция получения экземпляра `Tracer` в разделе «Получение экземпляра трассировщика» выше. На самом деле функция `otel.GetMeterProvider` полностью эквивалентна функции `otel.GetTracerProvider` и работает почти так же.

Функция `otel.GetMeterProvider` возвращает зарегистрированного глобального провайдера метрик. Если программа не зарегистрировала своего глобального провайдера метрик, то возвращается провайдер по умолчанию, который перенаправляет интерфейс `Meter` в первый зарегистрированный экземпляр `Meter`.

Метод `Meter` провайдера возвращает экземпляр `metric.Meter`. Он принимает строковый параметр, представляющий имя точки инструментирования, которое по соглашению конструируется из имени инструментируемой библиотеки или пакета.

Инструменты метрик

Имея экземпляр `Meter`, можно создавать *инструменты* для проведения измерений и инструментирования кода. Однако, по аналогии со множеством разных типов метрик, имеется несколько типов инструментов. Тип используемого инструмента зависит от типа выполняемого измерения.

В общей сложности доступно 12 *видов* инструментов, каждый из которых характеризуется определенной комбинацией *синхронности*, поведения *накопления* и типа данных.

Первое из этих свойств, *синхронность*, определяет порядок сбора и передачи данных:

- *синхронные инструменты* явно вызываются пользователем, как будет показано в разделе «Синхронные инструменты» ниже;
- *асинхронные инструменты*, или *наблюдатели*, могут наблюдать за определенными характеристиками и асинхронно вызываться из SDK. Мы рассмотрим их в разделе «Асинхронные инструменты».

Второе свойство – поведение *накопления* – описывает, как инструмент следит за сбором новых данных:

- *аддитивные* инструменты отслеживают сумму, которая может произвольно увеличиваться или уменьшаться. Обычно они используются для измерения таких значений, как температура или текущее потребление памяти, а также «счетчиков», которые могут увеличиваться и уменьшаться, например количество одновременно обслуживаемых запросов;
- *аддитивные монотонные* инструменты отслеживают монотонно возрастающие (https://ru.wikipedia.org/wiki/Монотонная_функция) значения, которые могут только увеличиваться (или сбрасываться в ноль при перезапуске), подобно счетчику. К аддитивным монотонным метрикам можно отнести, например, количество обслуженных запросов, выполненных задач или ошибок;
- инструменты *группировки* предназначены для сбора данных о распределении, например для составления гистограмм. Инструмент группирования отбирает наблюдения (обычно такие характеристики, как продолжительность обработки запросов или размер ответов) и подсчитывает их количество в заданных интервалах, а также предоставляет сумму всех наблюдаемых значений.

Наконец, каждый из шести предыдущих видов инструментов поддерживает входные значения с типами `float64` или `int64`, что дает в сумме 12 видов инструментов. Каждому из них соответствует свой тип в пакете `go.opentelemetry.io/otel/metric`. Эти типы перечислены в табл. 11.1.

Таблица 11.1. 12 видов инструментов сбора метрик в OpenTelemetry по синхронности и поведению накопления

	Синхронные	Асинхронные
Аддитивные	`Float64UpDownCounter`, `Int64UpDownCounter`	`Float64UpDownSumObserver`, `Int64UpDownSumObserver`
Аддитивные, монотонные	`Float64Counter`, `Int64Counter`	`Float64SumObserver`, `Int64SumObserver`
Группировки	`Float64ValueRecorder`, `Int64ValueRecorder`	`Float64ValueObserver`, `Int64ValueObserver`

Для каждого инструмента из этих 12 типов имеется своя функция-конструктор, присоединенная к типу `metric.Meter`, и все они имеют схожие сигнатуры. Вот, например, сигнатура метода `NewInt64Counter`:

```
func (m Meter) NewInt64Counter(name string, options ...InstrumentOption)
    (Int64Counter, error)
```

Все 12 методов-конструкторов, подобно `NewInt64Counter`, принимают имя метрики в виде строки и ноль или более значений `metric.InstrumentOption`. Все возвращают экземпляр инструмента соответствующего типа с заданным именем метрики и параметрами настройки и могут вернуть ошибку, если имя пустое или по каким-то причинам признано недействительным, или если такой инструмент уже был зарегистрирован.

Вот пример функции, которая использует метод `NewInt64Counter` для получения нового экземпляра `metric.Int64Counter` из `metric.Meter`:

```
// Инструмент подсчета запросов. Действует синхронно,
// мы должны сохранить его для последующего использования.
var requests metric.Int64Counter

func buildRequestsCounter() error {
    var err error

    // Получить экземпляр Meter из провайдера метрик.
    meter := otel.GetMeterProvider().Meter(serviceName)

    // Получить инструмент Int64Counter для метрики с именем
    // "fibonacci_requests_total".
    requests, err = meter.NewInt64Counter("fibonacci_requests_total",
        metric.WithDescription("Total number of Fibonacci requests."),
    )

    return err
}
```

Обратите внимание, что в этом примере мы сохраняем ссылку на инструмент в глобальной переменной `requests`. По причинам, о которых я расскажу ниже, это особенно характерно для синхронных инструментов.

Экземпляр `metric.Int64Counter` является синхронным инструментом, но главное здесь не это, а то, что синхронные и асинхронные инструменты создаются одинаково: с помощью соответствующего метода-конструктора, присоединенного к типу `Metric`. Однако порядок их использования существенно различается, как мы увидим в следующих разделах.

СИНХРОННЫЕ ИНСТРУМЕНТЫ Первый шаг к использованию синхронного инструмента – получение экземпляра `Meter` из провайдера метрик и создание инструмента – он по большей части одинаков для синхронных и асинхронных инструментов, как вы могли видеть в предыдущем разделе.

Однако порядок использования синхронных и асинхронных инструментов отличается: синхронные инструменты должны явно вызываться в вашем коде, а это означает, что должна иметься возможность ссылаться на инструмент после его создания. Вот почему в примере выше используется глобальная переменная `requests`.

Чаще всего, пожалуй, синхронные инструменты используются для фиксации отдельных событий путем увеличения счетчика. В аддитивных инструментах для этого даже есть метод `Add`. Следующий пример демонстрирует использование инструмента `requests`, созданного выше. Здесь мы добавили

вызов request.Add в функцию Fibonacci, которая была определена в разделе «API службы вычисления чисел Фибоначчи» выше:

```
// Определить метки, чтобы упростить их использование.
var labels = []label.KeyValue{
    label.Key("application").String(serviceName),
    label.Key("container_id").String(os.Getenv("HOSTNAME")),
}

func Fibonacci(ctx context.Context, n int) chan int {
    // Вызвать метод Add экземпляра metric.Int64Counter,
    // чтобы увеличить значение счетчика.
    requests.Add(ctx, 1, labels...)

    // Остальная часть функции...
}
```

Как видите, метод requests.Add – безопасный для использования в конкурентном окружении – принимает три параметра:

- текущий контекст в виде значения context.Context. Такой параметр принимают все методы синхронных инструментов;
- число, на которое нужно увеличить счетчик. В данном случае каждый вызов Fibonacci увеличивает счетчик вызовов на единицу;
- ноль или более значений label.KeyValue, представляющих метки, которые нужно связать с точками данных. Это увеличивает мощность метрик, что невероятно полезно, как обсуждалось во врезке «Мощность метрик» выше.

Метки данных – мощное средство, позволяющее добавлять дополнительное описание в данные, помимо названия службы или экземпляра, откуда они были получены. Метки могут помочь вам задавать вопросы о ваших данных, о которых вы раньше не задумывались.

Также можно сгруппировать несколько метрик и отправить их в виде пакета. Однако этот способ работает немного иначе, чем метод Add, показанный в предыдущем примере. В частности, для каждой метрики в пакете необходимо:

1) получить значение или значения, которые нужно сохранить;
2) передать каждое значение в вызов метода Measurement соответствующего инструмента, который вернет metric.Measurement – обертку для вашей метрики с некоторыми вспомогательными метаданными;
3) передать все значения metric.Measurement в вызов метода meter.RecordBatch, который атомарно запишет весь пакет измерений.

Эти шаги демонстрирует следующий пример, в котором используется пакет runtime для получения двух значений – объема использованной памяти и количества сопрограмм в процессе – для передачи их коллектору метрик:

```
func updateMetrics(ctx context.Context) {
    // Получить экземпляр Meter из провайдера метрик.
    meter := otel.GetMeterProvider().Meter(serviceName)
```

```
    // Создать инструмент для получения объема используемой памяти
    // и количества сопрограмм. Значения ошибок игнорируются для краткости.
    mem, _ := meter.NewInt64UpDownCounter("memory_usage_bytes",
        metric.WithDescription("Amount of memory used."),
    )
    goroutines, _ := meter.NewInt64UpDownCounter("num_goroutines",
        metric.WithDescription("Number of running goroutines."),
    )

    var m runtime.MemStats

    for {
        runtime.ReadMemStats(&m)

        // Получить измеряемые значения в виде экземпляров
        // metric.Measurement.
        mMem := mem.Measurement(int64(m.Sys))
        mGoroutines := goroutines.Measurement(int64(runtime.NumGoroutine()))

        // Передать результаты измерений (а также контекст и
        // метки) в экземпляр Meter.
        meter.RecordBatch(ctx, labels, mMem, mGoroutines)
        time.Sleep(5 * time.Second)
    }
}
```

Действующая как сопрограмма, функция `updateMetrics` выполняется в два этапа: сначала она производит настройку, а затем входит в бесконечный цикл, в котором генерирует и записывает измерения.

На этапе настройки она получает экземпляр `Meter`, определяет некоторые метки и создает инструменты. Все эти шаги выполняются ровно один раз, а полученные в них значения повторно используются в цикле. Обратите внимание, что, кроме типов, инструменты имеют также названия и описания, указывающие на метрики, которые они используют.

Внутри цикла сначала вызываются функции `runtime.ReadMemStats` и `runtime.NumGoroutine` для получения метрик (объем используемой памяти и количество запущенных сопрограмм соответственно). Полученные значения передаются в вызовы метода `Measurement` инструментов для создания значений `metric.Measurement` для каждой метрики.

Затем полученные значения `Measurement` передаются в вызов метода `meter.RecordBatch`, который также принимает текущий контекст `context.Context` и любые метки для присоединения к метрикам, чтобы записать их.

АСИНХРОННЫЕ ИНСТРУМЕНТЫ Асинхронные инструменты, или *наблюдатели*, создаются и настраиваются во время запуска программы для измерения определенных свойств, а затем вызываются SDK во время сбора данных. Эти инструменты особенно удобны, когда есть значение, которое хотелось бы отслеживать без запуска своего фонового процесса записи.

Так же как синхронные инструменты, асинхронные инструменты создаются вызовом метода-конструктора, прикрепленного к экземпляру `metric.Meter`. Всего таких методов шесть: по одной версии `float64` и `int64` для каждого

из трех режимов накопления. Все шесть методов имеют похожие сигнатуры. Вот типичный пример:

```
func (m Meter) NewInt64UpDownSumObserver(name string,
    callback Int64ObserverFunc, opts ...InstrumentOption)
    (Int64UpDownSumObserver, error)
```

Как видите, `NewInt64UpDownSumObserver` принимает имя метрики в виде строки, нечто типа `Int64ObserverFunc` и ноль или более параметров инструмента (например, описание метрики). Этот метод возвращает экземпляр наблюдателя, но на самом деле он используется довольно редко, хотя может вернуть ошибку, отличную от `nil`, если передать пустое, уже зарегистрированное или недействительное по иным причинам имя.

Второй параметр – *функция обратного вызова* – является сердцем любого асинхронного инструмента. Функции обратного вызова асинхронно вызываются из SDK в процессе сбора данных. Есть два типа асинхронных функций, по одному для `int64` и `float64`, но они выглядят и работают практически одинаково:

```
type Int64ObserverFunc func(context.Context, metric.Int64ObserverResult)
```

При вызове из SDK функции обратного вызова получают текущий контекст `context.Context` и `metric.Float64ObserverResult` (для наблюдателей типа `float64`) или `metric.Int64ObserverResult` (для наблюдателей типа `int64`). Оба типа результатов имеют метод `Observe`, который можно использовать для вывода результатов.

Здесь много мелких деталей, но они довольно легко сочетаются друг с другом. Следующая функция делает именно это, определяя двух наблюдателей:

```
func buildRuntimeObservers() {
    meter := otel.GetMeterProvider().Meter(serviceName)
    m := runtime.MemStats{}

    meter.NewInt64UpDownSumObserver("memory_usage_bytes",
        func(_ context.Context, result metric.Int64ObserverResult) {
            runtime.ReadMemStats(&m)
            result.Observe(int64(m.Sys), labels...)
        },
        metric.WithDescription("Amount of memory used."),
    )

    meter.NewInt64UpDownSumObserver("num_goroutines",
        func(_ context.Context, result metric.Int64ObserverResult) {
            result.Observe(int64(runtime.NumGoroutine()), labels...)
        },
        metric.WithDescription("Number of running goroutines."),
    )
}
```

При вызове из `main` функция `buildRuntimeObservers` определяет два асинхронных инструмента – `memory_usage_bytes` и `num_goroutines`, – каждый с функ-

цией обратного вызова, которая работает точно так же, как коллектор данных в функции `updateMetrics`, которую мы определили в разделе «Синхронные инструменты» выше.

Однако в `updateMetrics` мы использовали бесконечный цикл для синхронной передачи данных. Как видите, асинхронная передача данных, не связанных с событиями, не только требует меньше работы по настройке, но и меньшего количества движущихся частей, о которых придется побеспокоиться позже, потому что нам ничего не нужно делать после того, как наблюдатели (с их функциями обратного вызова) и SDK возьмут на себя управление.

Собираем все вместе: метрики

Теперь, определив метрики и порядок их сбора, мы можем использовать их для расширения веб-службы вычисления чисел Фибоначчи, которую создали в разделе «Собираем все вместе: трассировка» выше.

Основа службы останется без изменений. Как и прежде, она будет принимать HTTP-запросы `GET` с порядковым номером числа Фибоначчи в строке запроса `GET`. Например, чтобы запросить шестое число Фибоначчи, нужно с помощью `curl` отправить службе запрос `http://localhost:3000?n=6`.

Вот конкретные изменения, которые мы внесем для сбора метрик:

- добавим в `main` вызов функции `buildRequestsCounter` для синхронной записи количества запросов и инструментируем функцию `Fibonacci` в API службы, как было описано в разделе «Синхронные инструменты» выше;
- добавим в `main` вызов функции `buildRuntimeObservers`, описанной в разделе «Асинхронные инструменты» выше, чтобы создать наблюдателей, которые будут выполнять асинхронную запись объема используемой памяти и количества активных сопрограмм.

Запуск служб

Снова запустите службу:

```
$ go run .
```

Как и прежде, терминал должен приостановиться. Остановить службу можно комбинацией Ctrl-C.

Затем запустите сервер Prometheus. Но перед этим нужно создать файл конфигурации. Prometheus имеет множество параметров настройки (https://oreil.ly/h8A7f), но нам вполне достаточно определить следующие:

```
scrape_configs:
- job_name: fibonacci
  scrape_interval: 5s
  static_configs:
  - targets: ['host.docker.internal:3000']
```

Скопируйте эти строки в файл `prometheus.yml`.

Эта конфигурация определяет единственную цель с именем `fibonacci`, которая доступна по адресу `host.docker.internal:3000`, и настраивает сбор метрик каждые пять секунд (по умолчанию он производится раз в минуту).

Что такое host.docker.internal?

Имя `host.docker.internal` – это специальное имя, определяемое в Docker Desktop для Mac и Windows, которое разрешается во внутренний IP-адрес. Он позволяет контейнеру взаимодействовать с процессами хоста.

Важно отметить, что этот адрес определяется только для удобства разработки и не будет работать в промышленном окружении в Mac и Windows за пределами Docker Desktop (то есть он не поддерживается по умолчанию в Docker в Linux).

После создания файла `prometheus.yml` можно запустить Prometheus. Самый простой способ – использовать контейнер Docker:

```
docker run -d --name prometheus \
  -p 9090:9090 \
  -v "${PWD}/prometheus.yml:/etc/prometheus/prometheus.yml" \
  prom/prometheus:v2.23.0
```

Если для разработки вы используете Linux, то добавьте в команду выше параметр `--add-host=host.docker.internal:host-gateway`. Но не используйте его в промышленном окружении.

Теперь, когда обе службы запущены, можно отправить запрос:

```
$ curl localhost:3000?n=6
13
```

За кулисами OpenTelemetry просто запишет количество вызовов (в том числе и рекурсивных) функции `Fibonacci`.

Вывод конечной точки метрик

Теперь, когда все службы работают, вы в любой момент сможете получить экспортируемые метрики с помощью стандартной утилиты `curl`, обратившись к конечной точке `/metrics`:

```
$ curl localhost:3000/metrics
# HELP fibonacci_requests_total Total number of Fibonacci requests.
# TYPE fibonacci_requests_total counter
fibonacci_requests_total{application="fibonacci",container_id="d35f0bef2ca0"} 25
# HELP memory_usage_bytes Amount of memory used.
# TYPE memory_usage_bytes gauge
memory_usage_bytes{application="fibonacci",container_id="d35f0bef2ca0"}
  7.5056128e+07
# HELP num_goroutines Number of running goroutines.
# TYPE num_goroutines gauge
num_goroutines{application="fibonacci",container_id="d35f0bef2ca0"} 6
```

Как видите, здесь присутствуют все три зарегистрированные метрики, а также указаны их типы, описания, метки и значения. Пусть вас не смущает пустое значение `container_id` – это просто означает, что служба работает не в контейнере.

Просмотр результатов в Prometheus

Теперь, когда мы запустили службу, сервер Prometheus и выполнили один-два запроса к службе, чтобы накопить некоторые данные, можно воспользоваться поддержкой визуализации метрик в Prometheus. Напомню еще раз, что Prometheus не является полноценным решением визуализации (для этого я советую использовать что-нибудь вроде Grafana (https://grafana.com)), и тем не менее он предлагает простой интерфейс для выполнения произвольных запросов.

Этот интерфейс доступен по адресу `localhost:9090`. Откройте в браузере этот адрес, и перед вами должен появиться минималистичный интерфейс с полем поиска. Чтобы увидеть, как менялось значение интересующей вас метрики с течением времени, введите ее имя в поле поиска, нажмите клавишу Enter и щелкните на вкладке Graph (График). Вы должны увидеть график, как показано на рис. 11.6.

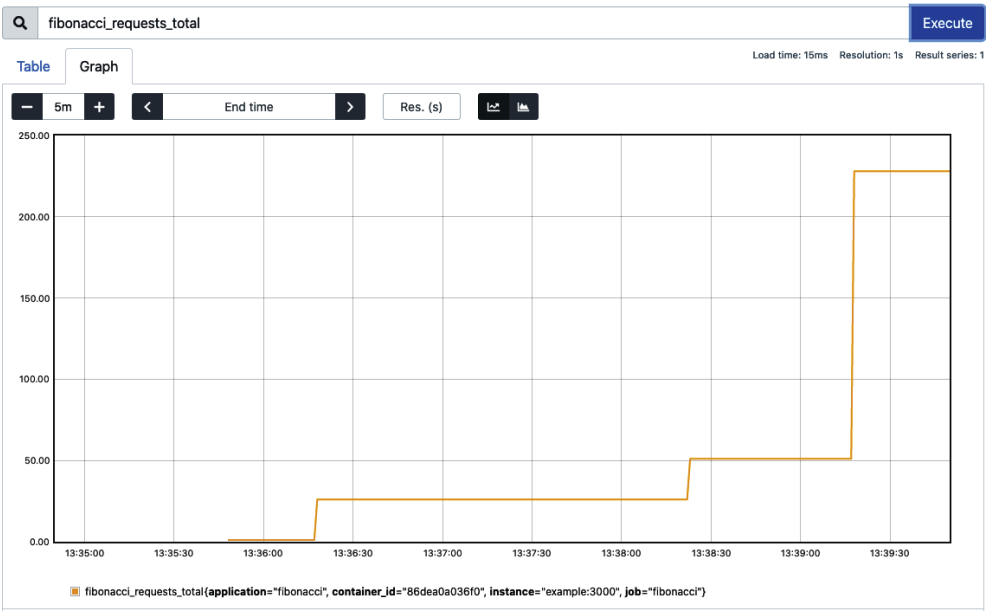


Рис. 11.6 ❖ Скриншот интерфейса Prometheus, отображающего значение метрики fibonacci_requests_total после трех обращений к службе вычисления чисел Фибоначчи

Теперь попробуйте выполнить еще несколько запросов и посмотрите, как изменится график. Можете даже посмотреть другие метрики. Пользуйтесь!

Журналирование

Журнал – это неизменяемая хронология *событий*, которые генерируются приложением с течением времени и заслуживают особого внимания. Традиционно журналы хранились в файлах, но в настоящее время в роли журналов все чаще используются хранилища данных с возможностью поиска.

Итак, что можно сказать о журналировании, кроме того что это действительно хорошая штука, которая появилась еще на заре электронно-вычислительных машин? Это альфа и омега методов наблюдаемости.

На самом деле о журналировании можно сказать много, в основном потому, что журналирование реализуется просто, но оно делает нашу жизнь сложнее, чем хотелось бы.

Из трех столпов наблюдаемости журналы, безусловно, самые простые в реализации. События, записываемые в журнал, не требуют никакой предварительной обработки, поэтому организация журналирования в простейшей его форме не сложнее добавления в код операторов вывода. Это делает журналы хорошим средством предоставления большого объема контекстных данных о работе компонента.

Но такая произвольная форма журналирования имеет и обратную сторону. Можно (и часто заманчиво), конечно, записывать в журнал все, что по вашему мнению может пригодиться позже, но из излишне подробных и неструктурированных журналов трудно извлечь полезную информацию, особенно в больших масштабах. Чтобы получить максимальную отдачу от журналирования, события должны структурироваться, а для этого придется попотеть. Структура событий должна быть хорошо продумана.

Другая, особенно недооцениваемая проблема журналирования заключается в том, что запись большого количества событий порождает большой объем дискового и/или сетевого ввода/вывода. В результате нередко на запись в журнал расходуется до половины или более доступной пропускной способности. Кроме того, нагрузка на ввод/вывод имеет тенденцию линейно масштабироваться с нагрузкой на службы: `N` пользователей, каждый из которых выполняет `M` операций, порождают `N*M` журналируемых событий с потенциально катастрофическими последствиями для масштабируемости.

Наконец, чтобы журналы приносили выгоду, их нужно обрабатывать и хранить так, чтобы они были легкодоступны. Любой, кому доводилось управлять журналами в большом масштабе, может сказать вам, что журналирование обходится довольно дорого, когда размещением и управлением журналов занимаетесь вы сами, и еще дороже, когда их размещением и управлением занимается кто-то другой.

В оставшейся части этого раздела мы сначала обсудим некоторые общие методы журналирования в масштабе, а затем посмотрим, как их реализовать в Go.

Рекомендуемые методы журналирования

Каким бы простым ни казался процесс журналирования, его очень легко реализовать так, что он усложнит жизнь всем, кто будет использовать ваши

журналы после вас. Досадные проблемы, такие как необходимость навигации по неструктурированным журналам или более высокое, чем ожидалось, потребление ресурсов, раздражающие в небольших развертываниях, могут превратиться в серьезные препятствия при масштабировании.

Как вы увидите далее, по этой и другим причинам рекомендуемые методы журналирования сосредоточены в основном на повышении качества и минимизации количества журналируемых данных.

Разумеется, что в журналы не должны попадать конфиденциальные бизнес-данные или личная информация.

Интерпретируйте журналы как потоки событий

Сколько раз вы просматривали журналы и сталкивались с непостижимым потоком сознания? Насколько такие журналы были полезными? Возможно, лучше, чем ничего, но, скорее всего, ненамного.

Не нужно рассматривать журналы как приемники данных, куда записывается все подряд и о чем можно забыть, пока что-то буквально не загорится. И определенно журналы не должны быть свалкой мусора, куда вы отправляете случайные мысли и наблюдения.

Вместо этого, как было показано в главе 6, журналы следует рассматривать как потоки событий, которые выводятся, без буферизации, непосредственно в `stdout` и `stderr`. Это правило кажется простым (и, возможно, несколько нелогичным), но такое небольшое изменение точки зрения дает большую свободу.

Освобождая код приложения от ответственности за управление журналами, вы избавляетесь от забот о таких тривиальных вещах, как маршрутизация или хранение журналов, позволяя исполнителю решать, как с ними поступить.

Такой подход открывает широкое пространство выбора в том, как управлять журналами и использовать их. В процессе разработки вы сможете следить за действиями своей службы, отправляя события прямо на локальный терминал. В промышленном окружении этот вывод может перехватываться и перенаправляться в систему индексации журналов, такую как ELK или Splunk, для просмотра и анализа или, может быть, в хранилище данных для долгосрочного хранения.

Интерпретируйте журналы как потоки событий и записывайте каждое событие непосредственно в `stdout` и `stderr`.

Структурируйте события для последующего анализа

Технически для журналирования в его простейшей и примитивной форме достаточно простых инструкций `fmt.Println`. Однако в таком случае вы получите набор неформатированных строк сомнительной полезности.

К счастью, программисты все чаще используют стандартную библиотеку `log`, простую в использовании и автоматически генерирующую отметки времени. Но какую пользу может принести терабайт или около того событий, отформатированных так:

```
2020/11/09 02:15:10AM User 12345: GET /help in 23ms
2020/11/09 02:15:11AM Database error: connection reset by peer
```

Конечно, это лучше, чем ничего, но вам все равно придется разбирать неструктурированные строки, пусть и с отметками времени. Вам все равно придется анализировать произвольный текст, чтобы извлечь значимую информацию.

Сравните этот вывод с эквивалентными сообщениями, создаваемыми структурирующим регистратором[1]:

```
{"time":1604888110, "level":"info", "method":"GET", "path":"/help",
        "duration":23, "message":"Access"}
{"time":1604888111, "level":"error", "error":"connection reset by peer",
        "database":"user", "message":"Database error"}
```

В этом примере структурированного журнала все ключевые элементы помещены в свойства объекта JavaScript. Каждый объект имеет следующие поля:

`time`
: Отметка времени, представляющая часть контекстной информации, которая имеет решающее значение для выявления проблем и определения связей между ними. Обратите внимание, что этот пример в формате JSON легче поддается анализу и требует гораздо меньше вычислительных затрат для извлечения значимой информации, чем первый, почти неструктурированный пример. Когда приходится обрабатывать журналы с миллиардами событий, такие мелочи имеют свойство накапливаться как снежный ком.

`level`
: Уровень журналирования – метка, определяющая уровень важности события. На практике часто используют следующие уровни: `INFO` (для информации), `WARN` (предупреждение) и `ERROR` (ошибка). Уровень также можно использовать для фильтрации сообщений с низким приоритетом, которые могут не иметь отношения к исследуемой проблеме.

Один или несколько контекстных элементов
: Они содержат фактическую информацию, дающую представление о состоянии приложения на момент записи сообщения. *Вся суть* журналирования событий состоит в том, чтобы правильно выразить эту контекстную информацию.

Проще говоря, структурированная форма помогает проще, быстрее и дешевле извлекать информацию, и структурированные записи намного проще искать, фильтровать и агрегировать.

Структурируйте свои журналы, оптимизируйте их для анализа компьютерами, а не для чтения людьми.

---

[1] Любые переносы строк в примере используются только для форматирования в книге. Не используйте разрывы строк в событиях, записываемых в журнал, если это возможно.

Лучше меньше, да лучше

Журналирование потребляет ресурсы. На самом деле журналирование обходится очень дорого.

Представьте, что у вас есть служба, развернутая на сервере в AWS, самом обычном сервере со стандартным универсальным диском, обеспечивающим стабильную пропускную способность 16 Мбайт/с.

Допустим, что ваша служба старается максимально подробно сообщать о протекающих в ней событиях, поэтому она тщательно регистрирует все события: каждый запрос, каждый ответ, каждое обращение к базе данных, результаты вычислений и множество других сведений, всего 16 событий по 1024 байта на каждый запрос, обрабатываемый службой. Возможно, несколько многословно, но пока ничего необычного.

Но эти затраты складываются. Например, если служба обрабатывает 512 запросов в секунду – вполне разумное число для службы, – она будет генерировать 8192 события в секунду. При объеме вывода 16 Кбайт на запрос в сумме получится 8 Мбайт журналируемых данных в секунду, или *половина пропускной способности диска*. Это довольно обременительно.

А что, если пропустить запись на диск и направлять события прямо в службу хостинга журналов? Такое решение ничуть не лучше, потому что тогда нам придется переносить и хранить наши журналы, а это довольно дорого. Если вы решите отправлять данные через интернет, например провайдеру услуг Splunk или Datadog, то вам придется заплатить своему облачному провайдеру за передачу данных. В AWS величина платы составляет 0,08 доллара США за гигабайт, что при средней скорости 8 Мбайт/с – примерно 1 Тбайт каждые полтора дня – составит почти 250 000 долларов США в год для одного экземпляра. Для пятидесяти таких экземпляров только затраты на передачу данных составят более 12 миллионов долларов.

Конечно, в этом примере не учитываются колебания нагрузки в зависимости от времени суток или дня недели. И все же он достаточно ясно показывает, что стоимость журналирования может вырасти очень быстро, поэтому регистрируйте только то, что может пригодиться, и обязательно ограничивайте объем журналирования в промышленном окружении, используя пороговые значения уровня серьезности. Обычно в качестве порога используется уровень `WARN` (предупреждение).

Динамически фильтруйте журналируемые данные

Поскольку отладочная информация обычно имеет большой объем при относительно низкой полезности, на практике ее часто исключают из журналирования в промышленном окружении путем установки уровня журнала, равного `WARN` (предупреждение). Но отладочная информация *не бесполезна*, не так ли[1]? Как оказывается, эта информация становится по-настоящему полезной, когда вы пытаетесь определить причину сбоя, а это означает, что вам

[1] Если бы она была бесполезной, зачем тогда вообще ее журналировать?

придется потратить драгоценное время для включения отладочного уровня журналирования, чтобы отыскать источник проблемы. Да, и не забудьте потом его выключить.

Благодаря применению *динамической фильтрации* журналируемых данных, когда в журнал записывается только часть событий, вы будете иметь отладочную информацию в журналах, но не слишком много, что может помочь сократить время восстановления работоспособности службы после инцидента.

Наличие отладочной информации в журналах может особенно пригодиться, когда что-то горит.

Журналирование с использованием стандартного пакета log

В Go имеется стандартный пакет журналирования с говорящим названием `log`, который включает некоторые основные функции журналирования. Это очень простой пакет, но в нем есть все, что нужно для реализации несложной стратегии журналирования.

Помимо импорта пакета `log`, для его использования не требуется никаких других настроек.

Его базовые возможности доступны в виде функций, очень похожих на функции вывода в пакете `fmt`, с которыми вы, возможно, знакомы:

```
func Print(v ...interface{})
func Printf(format string, v ...interface{})
func Println(v ...interface{})
```

Вероятно, вы заметили самое вопиющее, пожалуй, упущение в пакете `log`: он не поддерживает уровни журналирования. Однако этот недостаток компенсируется простотой и удобством использования.

Вот самый простой пример журналирования событий:

```
package main

import "log"

func main() {
    log.Print("Hello, World!")
}
```

Если запустить этот фрагмент, он выведет

```
$ go run .
2020/11/10 09:15:39 Hello, World!
```

Как видите, функция `log.Print` – подобно всем функциям журналирования – добавляет отметку времени в сообщения без какой-либо дополнительной настройки.

Специальные функции журналирования

В пакете log отсутствует поддержка уровней журналирования, зато он предлагает некоторые другие интересные функции. А именно класс вспомогательных функций, которые объединяют вывод событий в журнал с другими полезными действиями.

Первый из них – класс функций log.Fatal. Всего таких функций три, и каждая эквивалентна вызову аналогичной функции log.PrintX с последующим вызовом os.Exit(1):

```
func Fatal(v ...interface{})
func Fatalf(format string, v ...interface{})
func Fatalln(v ...interface{})
```

Также пакет log предлагает класс функций log.Panic, которые эквиваленты вызову соответствующей функции log.PrintX и последующему выполнению инструкции panic:

```
func Panic(v ...interface{})
func Panicf(format string, v ...interface{})
func Panicln(v ...interface{})
```

Оба этих класса функций используются не так часто, как функции log.Print, и обычно для обработки ошибок, когда имеет смысл сообщить об ошибке и остановиться.

Журналирование в нестандартный объект записи

По умолчанию пакет log осуществляет вывод в stderr, а можно ли выполнить вывод в другое место? Функция log.SetOutput позволяет сделать это. С ее помощью можно задать свой экземпляр io.Writer.

Это позволяет, например, записывать события в файл. Как отмечалось в разделе «Лучше меньше, да лучше» выше, записывать события в файлы обычно не рекомендуется, но при определенных обстоятельствах такая возможность может пригодиться.

Следующий пример демонстрирует этот подход: он использует os.OpenFile, чтобы открыть целевой файл, и вызывает log.SetOutput, чтобы определить его как место для записи событий:

```
package main

import (
    "log"
    "os"
)

func main() {
    // O_APPEND = добавлять записываемые данные в конец файла
    // O_CREATE = создать новый файл, если он не существует
    // O_WRONLY = открыть только для записи
    flags := os.O_APPEND | os.O_CREATE | os.O_WRONLY
```

```
    file, err := os.OpenFile("log.txt", flags, 0666)
    if err != nil {
        log.Fatal(err)
    }

    log.SetOutput(file)

    log.Println("Hello, World!")
}
```

После запуска этот код выведет запись в файл `log.txt`:

```
$ go run .; tail log.txt
2020/11/10 09:17:05 Hello, World!
```

Тот факт, что `log.SetOutput` принимает интерфейс, означает, что для вывода журналируемых записей можно поддерживать широкий спектр объектов, осуществляющих вывод, от которых требуется только, чтобы они соответствовали интерфейсу `io.Writer`. Можно даже создать реализацию `io.Writer`, которая, например, пересылает данные процессору журналов Logstash или брокеру сообщений Kafka. Возможности безграничны.

Флаги журналирования

Пакет `log` также позволяет использовать константы для добавления в журналируемые сообщения дополнительной контекстной информации, такой как имя файла, номер строки, дата и время.

Например, если добавить следующую инструкцию в предыдущий пример «Hello, World»:

```
log.SetFlags(log.Ldate | log.Ltime | log.Lshortfile)
```

то она выведет в журнал следующее сообщение:

```
2020/11/10 10:14:36 main.go:7: Hello, World!
```

Как видите, она добавляет в сообщения дату в местном часовом поясе (`log.Ldate`), время в местном часовом поясе (`log.Ltime`), а также имя файла и номер строки (`log.Lshortfile`), в котором произведен вызов функции журналирования.

Мы не можем изменить порядок следования элементов в сообщении или формат, но если вам это понадобится, то я рекомендую использовать другой фреймворк журналирования, такой как Zap.

Пакет журналирования Zap

Из трех столпов наблюдаемости журналирование меньше всего поддерживается в OpenTelemetry. Точнее, оно вообще не поддерживается, по крайней мере так было на момент написания этих строк (хотя со временем подобная поддержка появится).

Поэтому вместо обсуждения API журналирования в OpenTelemetry мы рассмотрим еще одну замечательную библиотеку – Zap (https://oreil.ly/fjMls), поддерживающую журналирование в формате JSON и оптимизированную по потреблению памяти, использованию механизма рефлексии и средств форматирования строк.

В настоящее время Zap является одним из двух самых популярных пакетов журналирования в Go, наряду с Logrus. На самом деле популярность Logrus немного выше, но я отдал предпочтение Zap, руководствуясь тремя основными факторами. Во-первых, пакет Zap известен своей скоростью работы и низким потреблением памяти (что может пригодиться при масштабировании). Во-вторых, он придерживается философии «структурированного» журналирования, что очень желательно, как утверждалось в разделе «Структурируйте события для последующего анализа» выше. Наконец, развитие пакета Logrus в настоящее время приостановилось, и в него не добавляются новые возможности.

Насколько быстро работает Zap? Очень быстро. Для примера в табл. 11.2 приводятся результаты сравнительного тестирования нескольких распространенных пакетов структурированного журналирования без включения какого-либо контекста или шаблонов в стиле `printf`.

Таблица 11.2. Результаты сравнительного тестирования пакетов структурированного журналирования без включения какого-либо контекста или шаблонов в стиле printf

Пакет	Время	Время в % от ZAP	Создано объектов
Zap	118 нс/оп	+0 %	0 объектов/оп
Zap (с расширениями)	191 нс/оп	+62 %	2 объекта/оп
Zerolog	93 нс/оп	–21 %	0 объектов/оп
Go-kit	280 нс/оп	+137 %	11 объектов/оп
Стандартная библиотека	499 нс/оп	+323 %	2 объекта/оп
Logrus	3129 нс/оп	+2552 %	24 объекта/оп
Log15	3887 нс/оп	+3194 %	23 объекта/оп

Эти числа получены с использованием набора тестов Zap (https://oreil.ly/uGbA7), но я сам проверил, обновил и выполнил эти тесты. Конечно, к результатам любого тестирования следует относиться с некоторой долей скептицизма. В этих результатах особенно сильно выделяются стандартный пакет `log`, время работы которого примерно в три раза больше времени работы стандартной версии Zap, и пакет Logrus, которому потребовалось в 25 раз больше времени, чем Zap.

Но мы же еще должны использовать поля контекста, не так ли? Как в этом случае выглядит Zap? Как показано в табл. 11.3, результаты еще более поразительные.

Отрыв Zap от Logrus увеличился до (впечатляющего) 33 раз; стандартный пакет `log` не включен в эту таблицу, потому что он не поддерживает контекстные поля.

Результаты радуют. И как использовать этот пакет?

Таблица 11.3. Результаты сравнительного тестирования пакетов структурированного журналирования с сообщениями, содержащими 10 контекстных полей

Пакет	Время	Время в % от ZAP	Создано объектов
Zap	862 нс/оп	+0 %	5 объектов/оп
Zap (с расширениями)	1250 нс/оп	+45 %	11 объектов/оп
Zerolog	4021 нс/оп	+366 %	76 объектов/оп
Go-kit	4542 нс/оп	+4277 %	105 объектов/оп
Logrus	29501 нс/оп	+3322 %	125 объектов/оп
Log15	29906 нс/оп	+3369 %	122 объекта/оп

Создание регистратора Zap

Прежде всего нужно создать регистратор типа `zap.Logger`.

Конечно, перед этим необходимо импортировать пакет Zap:

```
import "go.uber.org/zap"
```

Импортировав пакет Zap, можно создать свой экземпляр `zap.Logger`. Zap позволяет настроить несколько аспектов поведения механизма журналирования, но самый простой способ создать `zap.Logger` – использовать функции-конструкторы с предопределенными настройками: `zap.NewExample`, `zap.NewProduction` и `zap.NewDevelopment`. Каждая из них создает свой экземпляр `logger`:

```
logger, err := zap.NewProduction()
if err != nil {
    log.Fatalf("can't initialize zap logger: %v", err)
}
```

Обычно это делается в функции `init`, а экземпляр `zap.Logger` сохраняется в глобальной переменной. Регистраторы Zap можно без опаски использовать в конкурентном окружении.

Три функции конструкторов с предопределенными настройками обычно идеально подходят для небольших проектов, но в более крупных проектах могут понадобиться дополнительные настройки. Для этой цели Zap предоставляет структуру `zap.Config`. Обсуждение ее особенностей выходит за рамки этой книги, однако желающие смогут найти подробное ее описание в документации Zap (https://oreil.ly/q1mHb).

Журналирование с использованием Zap

Одна из уникальных особенностей Zap заключается в наличии двух взаимозаменяемых форм – стандартной и расширенной («sugared» – «обсахаренной»), которые несколько различаются по эффективности и удобству использования.

Стандартная реализация `zap.Logger` делает упор на высокую производительность и безопасность типов. Она немного быстрее расширенной версии `SugaredLogger` и требует гораздо меньше ресурсов, но поддерживает только

структурированное журналирование, что делает ее менее удобной в использовании:

```
logger, _ := zap.NewProduction()

// Структурированный контекст со строго типизированными полями
logger.Info("failed to fetch URL",
    zap.String("url", url),
    zap.Int("attempt", 3),
    zap.Duration("backoff", time.Second),
)
```

Вот как будет выглядеть сообщение в журнале, сгенерированное этой инструкцией:

```
{"level":"info", "msg":"failed to fetch URL",
        "url":"http://example.com", "attempt":3, "backoff":"1s"}
```

В сценариях, где высокая производительность не является абсолютно необходимой (что, вероятно, характерно для большинства случаев), можно использовать регистратор `SugaredLogger`, который легко получить из стандартного регистратора вызовом метода `Sugar`.

`SugaredLogger` тоже поддерживает структурированное журналирование, но его функции слабо типизированы, в отличие от стандартного регистратора. Несмотря на то что расширенная версия за кулисами использует механизм рефлексии, она имеет относительно неплохую производительность.

`SugaredLogger` даже имеет методы журналирования в стиле `printf`. (Но помните, что главное в журналировании – это контекст.)

Все эти функции демонстрируются в следующем примере:

```
logger, _ := zap.NewProduction()
sugar := logger.Sugar()

// Структурированный контекст со слабо типизированными парами ключ/значение.
sugar.Infow("failed to fetch URL",
    "url", url,
    "attempt", 3,
    "backoff", time.Second,
)

sugar.Infof("failed to fetch URL: %s", url)
```

Вот как будет выглядеть сообщение в журнале, сгенерированное этой инструкцией:

```
{"level":"info", "msg":"failed to fetch URL",
        "url":"http://example.com", "attempt":3, "backoff":"1s"}
{"level":"info", "msg":"failed to fetch URL: http://example.com"}
```

Не создавайте новый экземпляр `Logger` в каждой функции. Используйте глобальный экземпляр или функции `zap.L` или `zap.S`, чтобы получить глобальный стандартный или расширенный регистратор Zap соответственно.

Динамическая фильтрация журналируемых данных в Zap

Как рассказывалось в разделе «Динамически фильтруйте журналируемые данные» выше, динамическая фильтрация позволяет фильтровать журналируемые сообщения, ограничивая их количество некоторым максимальным числом в единицу времени.

Используя этот прием, можно до определенной степени управлять нагрузкой на процессор и устройства ввода/вывода, сохраняя относительную репрезентативность подмножества событий. При нацеливании на определенный класс событий большого объема и с низким приоритетом, такой как класс отладочных сообщений, динамическая фильтрация может обеспечить доступность некоторого их количества при устранении неполадок в промышленном окружении и сэкономить на пространстве, необходимом для их хранения.

Zap поддерживает динамическую фильтрацию, которую можно настроить с помощью структуры `zap.SamplingConfig`, как показано ниже:

```
type SamplingConfig struct {
    // Initial определяет предельное количество событий, журналируемых в секунду.
    Initial int

    // Thereafter определяет долю событий, которые будут журналироваться
    // после превышения предела Initial. Значение 3, например, означает,
    // что регистрироваться будет только одно событие из трех.
    Thereafter int

    // Hook (если определено) -- это функция, которая будет вызываться после
    // каждого решения "журналировать / не журналировать".
    Hook func(zapcore.Entry, zapcore.SamplingDecision)
}
```

Структура `zap.SamplingConfig` позволяет определять допустимое количество начальных событий в секунду с одним и тем же уровнем и сообщениями (`Initial`), после превышения которого регистрируется только каждое *n*-е сообщение (`Thereafter`). Остальные будут отбрасываться.

В следующем примере показано, как создать новый `zap.Logger` с использованием предварительно настроенной структуры `zap.Config`:

```
package main

import (
    "fmt"
    "go.uber.org/zap"
    "go.uber.org/zap/zapcore"
)

func init() {
    cfg := zap.NewDevelopmentConfig()
    cfg.EncoderConfig.TimeKey = ""   // Отключить вывод отметок времени

    cfg.Sampling = &zap.SamplingConfig{
        Initial:    3, // Допускается регистрировать до трех событий в секунду
        Thereafter: 3, // после превышения предела регистрировать
                       // только 1 событие из 3
```

```
        Hook: func(e zapcore.Entry, d zapcore.SamplingDecision) {
            if d == zapcore.LogDropped {
                fmt.Println("event dropped...")
            }
        },
    }

    logger, _ := cfg.Build()         // Создать новый регистратор

    zap.ReplaceGlobals(logger)       // Заменить глобальный регистратор Zap
}
```

В этом примере создается новый экземпляр `zap.Logger`, который затем устанавливается в качестве глобального регистратора Zap. Этот процесс выполняется в несколько этапов.

Сначала создается новая структура `zap.Config`. Для удобства здесь используется предопределенная функция `zap.NewDevelopmentConfig`, которая создает экземпляр `zap.Config`, настраивающий вывод в удобочитаемом формате с уровнем `DebugLevel` и выше.

При желании можно использовать функцию `zap.NewProductionConfig`, которая возвращает экземпляр `zap.Config` с порогом `InfoLevel` и настраивающий вывод в формате JSON. В особых случаях можно даже создать свой экземпляр `zap.Config` с нуля.

Затем для поля `Sampling` в структуре zap.Config создается новый экземпляр `zap.SamplingConfig`, который требует от Zap *каждую секунду* сохранять первые три любых подобных события, затем только каждое третье событие.

Функция `Hook` вызывается после каждого решения, принимаемого при динамической фильтрации. Код в примере будет выводить сообщение, обнаружив, что какое-то событие было отброшено.

Наконец, код в примере использует метод `Build` структуры `Config`, чтобы создать `zap.Logger` на ее основе, и затем вызывает `zap.ReplaceGlobals` для замены глобального регистратора в Zap. Доступ к глобальному стандартному и расширенному регистратору Zap можно получить с помощью функций `zap.L` и `zap.S` соответственно.

Но работает ли этот код так, как мы ожидаем? Что ж, давайте посмотрим:

```
func main() {
    for i := 1; i <= 10; i++ {
        zap.S().Infow(
            "Testing sampling",
            "index", i,
        )
    }
}
```

Функция в этом примере пытается вывести в журнал 10 событий, но из-за настроек динамической фильтрации в журнал должны попасть три первых события, а затем только каждое третье (6 и 9). Так ли это?

```
$ go run .
INFO zap/main.go:39 Testing sampling {"index": 1}
INFO zap/main.go:39 Testing sampling {"index": 2}
INFO zap/main.go:39 Testing sampling {"index": 3}
event dropped...
event dropped...
INFO zap/main.go:39 Testing sampling {"index": 6}
event dropped...
event dropped...
INFO zap/main.go:39 Testing sampling {"index": 9}
event dropped...
```

Результат полностью совпал с нашими ожиданиями. Очевидно, что динамическая фильтрация – очень мощный инструмент и при правильном использовании может принести значительную выгоду.

Итоги

Вокруг наблюдаемости сейчас много шумихи, и, учитывая ее обещания резко сократить цикл обратной связи с разработчиком и снова сделать сложность управляемой, легко понять, почему.

В начале этой главы я коротко рассказал о наблюдаемости и ее перспективах и еще немного о том, чего наблюдаемость *не может* дать. К сожалению, вопрос о надлежащей реализации наблюдаемости действительно очень важен, а ограничения во времени и пространстве не позволили мне рассказать о ней в том объеме, в каком хотелось бы[1]. К счастью, есть множество замечательных книг, посвященных этой теме (в первую очередь *Observability Engineering* (O'Reilly), которую написали Чарити Мейджорс (Charity Majors) и Лиз Фонг-Джонс (Liz Fong-Jones)).

Большая часть этой главы была потрачена на обсуждение трех столпов наблюдаемости, в частности на то, как реализовать их с помощью OpenTelemetry.

В общем, это была сложная глава. Наблюдаемость – обширная тема, о которой еще не так много написано вследствие ее новизны, и то же самое можно сказать о проекте OpenTelemetry. Даже его собственная документация ограничена и местами неполная. Впрочем, в этом есть свой плюс – мне пришлось потратить много времени на изучение исходного кода его реализации.

КОНЕЦ

[1] В конце концов, эта книга о Go. По крайней мере, так я постоянно говорил моим невероятно терпеливым редакторам.

Предметный указатель

Т

У

Книги издательства «ДМК ПРЕСС»
можно купить оптом и в розницу
в книготорговой компании «Галактика»
(представляет интересы издательств
«ДМК ПРЕСС», «СОЛОН ПРЕСС», «КТК Галактика»).
Адрес: г. Москва, пр. Андропова, 38;
тел.: **(499) 782-38-89**, электронная почта: **books@alians-kniga.ru**.
При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес.
Эти книги вы можете заказать и в интернет-магазине: **http://www.galaktika-dmk.com/**.

Мэтью А. Титмус

Облачный Go

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Зам. главного редактора *Сенченкова Е. А.*
Перевод *Киселев А. Н.*
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Гарнитура PT Serif. Печать цифровая.
Усл. печ. л. 33,96. Тираж 200 экз.

Веб-сайт издательства: **www.dmkpress.com**

O'REILLY®

Что общего у Docker, Kubernetes и Prometheus? Все эти облачные технологии написаны на языке программирования Go. В этой практической книге показано, как использовать сильные стороны Go для разработки масштабируемых и устойчивых облачных служб, действующих в непредсказуемом окружении. Вы узнаете, как конструировать такие приложения, познакомитесь с приемами создания низкоуровневых функций Go, шаблонами проектирования и архитектурными решениями. Каждая глава основана на предыдущей и все они последовательно рассматривают создание простого, но полнофункционального распределенного хранилища пар ключ/значение на языке Go. Вы познакомитесь с передовыми практиками использования Go в качестве языка разработки для решения задач, связанных с управлением и развертыванием облачных приложений, а также:

«Автор книги проделал большую работу, подробно описав высокоуровневую концепцию «облачных приложений» и приемы ее реализации с использованием современного языка Go. В результате получилась захватывающая и вдохновляющая книга».
Ли Атчисон,
владелец Atchison Technology LLC

«Это первая книга, из встречавшихся мне, которая с такой широтой и глубиной освещает современные приемы реализации облачных вычислений. Представленные здесь шаблоны сопровождаются наглядными примерами решения реальных задач, с которыми инженеры сталкиваются ежедневно».
Альваро Атьенза,
инженер по надежности,
Flatiron Health

- узнаете, чем облачные приложения отличаются от других программных архитектур;
- увидите, как Go помогает решать проблемы проектирования масштабируемых распределенных служб;
- познакомитесь с низкоуровневыми особенностями Go, такими как каналы и сопрограммы (горутины), позволяющими реализовать надежные облачные службы;
- узнаете, что такое «надежность» и как она связана с облачными технологиями;
- научитесь применять различные шаблоны, абстракции и инструменты для создания сложных распределенных систем и управления ими.

Мэтью А. Титмус – опытный ветеран индустрии разработки программного обеспечения, получивший дополнительное образование в области молекулярной биологии. Он был одним из первых сторонников облачных технологий и языка Go и в настоящее время увлечен решением задач повышения качества промышленных систем. Мэтью потратил много времени на обдумывание и реализацию стратегий наблюдения за распределенными системами и управления ими.

ISBN 978-5-97060-965-1

Интернет-магазин: www.dmkpress.com

Оптовая продажа: КТК «Галактика»
books@alians-kniga.ru

ДМК ПРЕСС
ИЗДАТЕЛЬСТВО
www.дмк.рф