Мэтт Батчер
Мэтт Фарина

Go
на практике

MANNING

ДМК ПРЕСС ИЗДАТЕЛЬСТВО

Мэтт Батчер, Мэтт Фарина

Go на практике

Matt Butcher, Matt Farina

GO IN PRACTICE

MANNING
SHELTER ISLAND

Мэтт Батчер, Мэтт Фарина

GO НА ПРАКТИКЕ

ДМК ПРЕСС
ИЗДАТЕЛЬСТВО
Москва, 2017

УДК 004.438Go
ББК 32.973.26-018.1
Б28

Батчер М., Фарина М.
Б28 Go на практике / пер. с англ. Р. Н. Рагимова; науч. ред. А. Н. Киселев. – М.: ДМК Пресс, 2017. – 374 с.: ил.

ISBN 978-5-97060-477-9

Go – превосходный системный язык. Созданный для удобной разработки современных приложений с параллельной обработкой, Go предоставляет встроенный набор инструментов для быстрого создания облачных, системных и веб-приложений. Знакомые с такими языками, как Java или C#, быстро освоят Go – достаточно лишь немного попрактиковаться, чтобы научиться писать профессиональный код.

Книга содержит решения десятков типовых задач в ключевых областях. Следуя стилю сборника рецептов – проблема/решение/обсуждение, – это практическое руководство опирается на основополагающие концепции языка Go и знакомит с конкретными приемами использования Go в облаке, тестирования и отладки, маршрутизации, а также создания веб-служб, сетевых и многих других приложений.

Издание адресовано опытным разработчикам, уже начавшим изучать язык Go и желающим научиться эффективно использовать его в своей профессиональной деятельности.

УДК 004.438Go
ББК 32.973.26-018.1

ISBN 978-1-63343-007-5 (анг.)

ISBN 978-5-97060-477-9 (рус.)

Содержание

Предисловие

Когда я услышал, что Мэтт Фарина и Мэтт Батчер начали работу над новой книгой о языке Go, это меня очень взволновало. Они оба много лет были важнейшими действующими лицами в экосистеме Go, обладают большим опытом работы и способны добавить в аромат содержания этой книги запахи специй прошлых учебных пособий. Эта книга должна стать преемницей книги «Go in Action», развивающей заложенные там основы и переводящей их в практическое русло.

Книга разбита на четыре простые в освоении части, каждая из которых имеет собственную направленность. Часть 1 освежает в памяти основные идеи языка Go. Если вы спешите и уже обладаете навыками, позволяющими уверенно писать код на языке Go, можете смело пропустить этот раздел, хотя я не рекомендую этого делать. Знакомясь с окончательным вариантом рукописи, я обнаружил в ней самородки такого размера, что, полагаю, главы этой части будут полезны всем читателям.

Часть 2 погружает читателя в недра механики управления Go-приложениями. Глава об ошибках является одним из лучших описаний Go-ошибок из всех, прочитанных мной прежде, а глава, посвященная отладке и тестированию, содержит массу полезной информации об этом важном этапе разработки, помогающем поднять приложение с уровня, требующего доказательства идеи, до уровня надежной производственной системы.

В третьей части вы узнаете о способах создания пользовательских интерфейсов. Глава по шаблонам является отличным руководством по самой сложной, как многие полагают, части экосистемы Go. Она знакомит с практическими приемами многократного использования шаблонов и создания выразительных веб-интерфейсов. Приведенные примеры соответствуют уровню книги, поскольку трудно найти примеры использования шаблонов, легко переносимые в реальные приложения. Затем вы увидите, как создавать и использовать REST API, и познакомитесь с хитростями управления версиями этого API.

Заключительная часть книги посвящена теме функциональной совместимости, необходимой практически любому современному приложению. Она позволяет глубоко погрузиться в облачную инфраструктуру и увидеть, как язык Go вписывается в модель облачных

вычислений. Заканчивается эта часть широким обзором микрослужб и методов взаимодействий между службами.

Кем бы вы ни были, новичком, только что познакомившимся с языком Go, или профессионалом с многолетним опытом, эта книга даст вам жизненно необходимые знания, которые помогут вам поднять ваши приложения на новый уровень. Авторы проделали большую работу по представлению сложной информации в согласованной манере, позволяющей ее легко усвоить. Я искренне рад публикации этой книги и тому вкладу, которое она привнесет в Go-сообщество. Я надеюсь, что вы получите то же удовольствие от ее прочтения, что и я.

— *Брайан Кетелсен* (*Brian Ketelsen*),
один их авторов книги «Go in action»,
один из основателей «Gopher academy»

Введение

При первом знакомстве с языком Go мы сразу оценили его большой потенциал. У нас появилось желание писать на нем приложения. Но это был новый язык, а во многих компаниях опасаются использовать новые языки программирования.

Это особенно касается тех компаний, где потенциал языка Go нашел бы широкое применение. Новым языкам приходится добиваться доверия, признания и принятия. Сотни тысяч разработчиков заняты в бизнесе, лидеры которого долго колеблются, перед тем как попробовать новый язык, а разработчикам, чтобы понять его выгоды и применить в разработке приложений, необходимо достаточно хорошо изучить язык.

Продвижению нового языка способствуют проекты с открытым исходным кодом, конференции, курсы и книги. Цель этой книги – помочь в изучении языка Go всем желающими, внести наш вклад в развитие Go-сообщества и разъяснить перспективы применения языка Go руководству компаний, занятых в том числе разработкой программного обеспечения.

Начиная работу над книгой, мы представляли ее целиком посвященной применению Go в разработке облачных приложений. Мы уже несколько лет работаем в сфере облачных вычислений, а Go – это язык, специально созданный для нее. Начав сотрудничать с издательством Manning, мы сочли возможным выйти за пределы облачных технологий и дополнительно охватить некоторые полезные и практичные шаблоны программирования. В результате центр внимания книги сместился из сферы облачных вычислений в сферу практического применения Go. Тем не менее корнями она по-прежнему уходит в облачные технологии.

Книга «Go на практике» – это наша попытка помочь разработчикам познакомиться с языком для продуктивного его использования, а также помочь развитию сообщества и способствовать улучшению разрабатываемого программного обеспечения.

Благодарности

На написание этой книги мы потратили около двух лет, но ее завершение стало бы невозможным без поддержки наших семей. Они поддерживали нас с раннего утра до позднего вечера и в выходные дни, когда мы целиком уходили в работу. Они были рядом, когда мы не писали, но были поглощены разрешением связанных с книгой проблем.

Хорошие программы не пишутся в вакууме. Мы также признательны членам Go-сообщества, столь щедро отдавшим свое время созданию языка, его библиотек и процветанию его экосистемы. Увлекательно быть частью такого разнообразного и динамичного сообщества разработчиков. Мы, в частности, благодарим, Роба Пайка (Rob Pike), Брайана Кетелсена (Brian Ketelsen) и Дэйва Чейни (Dave Cheney), которые помогли нам на начальном этапе изучения Go. Они отлично справляются с ролью эмиссаров языка. Особая благодарность Брайану за предисловие к книге и одобрение нашей работы.

Мы ценим помощь множества людей, которые пожертвовали своим временем и приложили усилия к созданию этой книги. Это был трудный путь. Мы благодарны многим внимательным читателям, в том числе и нашим рецензентам, обнаружившим многочисленные ошибки, что позволило их своевременно исправить.

Мы хотели бы поблагодарить всех сотрудников издательства Manning, особенно нашего редактора Сусанну Клайн (Susanna Kline), научных редакторов Айвана Киркпатрика (Ivan Kirkpatrick), Кима Шириера (Kim Shrier), Гленна Бернсайда (Glenn Burnside) и Алена Коуниот (Alain Couniot), технического корректора Джеймса Фрасче (James Frasché), а также всех, кто трудился над нашей книгой. Большое спасибо рецензентам, нашедшим время на чтение нашей рукописи на различных стадиях ее готовности и написание отзывов, оказавших нам неоценимую помощь: Энтони Крампу (Anthony Cramp), Остину Риендо (Austin Riendeau), Брэндону Титусу (Brandon Titus), Дугу Спарлингу (Doug Sparling), Фердинандо Сантакосе (Ferdinando Santacroce), Гари А. Стаффорду (Gary A. Stafford), Джиму Амрхайну (Jim Amrhein), Кевину Мартину (Kevin Martin), Натану Дэвису (Nathan Davies), Квинтину Смиту (Quintin Smith), Сэму Зайделу (Sam Zaydel) и Уэсу Шэддиксу (Wes Shaddix).

Наконец, мы благодарны сообществу Glide, сформировавшемуся за время работы над созданием диспетчера пакетов верхнего уровня для Go. Благодарим вас за вашу поддержку.

Я начал писать эту книгу, работая в компании Revolv, продолжил после приобретения ее компанией Google/Nest и закончил в компании Deis. Спасибо им всем за поддержку в создании этой книги. Благодарю Брайана Хардока (Brian Hardock), Кристиана Кавалли (Cristian Cavalli), Ланна Мартина (Lann Martin) и Криса Чинга (Chris Ching), всех, с кем я советовался. Мэтт Боерсма (Matt Boersma) написал полезные отзывы о нескольких главах. Кент Rancourt (Kent Rancourt) и Аарон Шлезингер (Aaron Schlesinger) подали идею для нескольких примеров в книге. Мэтт Фишер (Matt Fisher), Сиварам Мозики (Sivaram Mothiki), Кирзан Мала (Keerthan Mala), Хельги Порбьёрнссон (Helgi Þorbjörnsson) (да, Хельги, я скопировал и вставил это), Гейб Монрой (Gabe Monroy), Крис Армстронг (Chris Armstrong), Сэм Бойер (Sam Boyer), Джефф Блейел (Jeff Bleiel), Джошуа Андерсон (Joshua Anderson), Римас Мосевикиус (Rimas Mocevicius), Джек Фрэнсис (Jack Francis) и Джош Лэйн (Josh Lane) – все вы (вольно или невольно) оказали влияние на определенные фрагменты этой книги. Нельзя недооценивать влияние Мишеля Ноорали (Michelle Noorali) и Адама Риза (Adam Reese). Я многому научился, наблюдая за работой нескольких Ruby-разработчиков, знатоков Go. И спасибо Энджи (Angie), Аннабель (Annabelle), Клэр (Claire) и Кэтрин (Katherine) за их постоянную поддержку и понимание.

– *Мэтт Батчер* (*Matt Butcher*)

Я хотел бы поблагодарить мою красивую и удивительную жену Кристину, а также наших прекрасных дочерей, Изабеллу и Обри, за их любовь и поддержку.

Я писал эту книгу, работая в компании Hewlett Packard Enterprise, ранее называющейся Hewlett-Packard. Работая в компании Hewlett Packard Enterprise, я многому научился, общаясь с теми, кто был гораздо умнее меня. В частности, я должен поблагодарить Раджива Пандей (Rajeev Pandey), Брайана Акера (Brian Aker), Стива Маклеллана (Steve McLellan), Эрин Handgen (Erin Handgen), Эрика Густафсона (Eric Gustafson), Майка Хагедорна (Mike Hagedorn), Сьюзан Балле (Susan Balle), Дэвида Гравеса

(David Graves) и многих других. Они учили меня писать приложения и управлять ими, и это, безусловно, отразилось на данной книге.

Многие другие также оказали влияние на определенные фрагменты этой книги, иногда даже сами не осознавая этого. Благодарю Тима Плетчера (Tim Pletcher), Джейсона Буберела (Jason Buberel), Сэм Бойера (Sam Boyer), Ларри Гарфилда (Larry Garfield) и всех тех, кого я мог забыть.

Наконец, я хочу поблагодарить Мэтта Батчера. Я никогда не задумывался над тем, чтобы стать автором книги, пока ты не предложил мне это. Спасибо!

– *Мэтт Фарина* (*Matt Farina*)

О книге

Книга «Go на практике» описывает практические приемы программирования на языке Go. Разработчики, знакомые с основами Go, найдут в ней шаблоны и методики создания Go-приложений. Каждая глава затрагивает определенную тему (как, например, глава 10 «Взаимодействие облачных служб») и исследует различные технологии, относящиеся к ней.

Как организована книга

Одиннадцать глав разделены на четыре части.

Часть I «Основные понятия и принципы» закладывает фундамент для будущих приложений. Глава 1 описывает основы языка Go и будет полезна всем, кто еще не знаком с ним или хотел бы узнать о нем больше. В главе 2 рассказывается о создании консольных приложений и серверов, а в главе 3 – об организации параллельных вычислений в Go.

Часть II «Надежные приложения» включает главы 4 и 5, охватывающие темы ошибок, аварий, отладки и тестирования. Цель этого раздела – рассказать о создании надежных приложений, способных автоматически справляться с возникающими проблемами.

Часть III «Интерфейсы приложений» содержит три главы и охватывает диапазон тем, от генерации разметки HTML и обслуживания ресурсов до реализации различных API и работы с ними. Многие Go-приложения поддерживают взаимодействие через веб-интерфейс и REST API. Эти главы охватывают приемы, помогающие в их конструировании.

Часть IV «Облачные приложения» содержит остальные главы, посвященные облачным вычислениям и генерации кода. Язык Go разрабатывался с учетом нужд облачных технологий. В этой части демонстрируются приемы, позволяющие работать с облачными службами и приложениями и даже с микрослужбами в них. Кроме того, она охватывает приемы генерации кода и метапрограммирование.

В книге описывается 70 приемов, и в каждом случае сначала ставится задача, затем дается решение и в заключение обсуждаются причины выбора тех или иных подходов.

Соглашения и загрузка примеров кода

Весь исходный код в книге оформлен моноширинным шрифтом, `как этот текст`, чтобы выделить его в окружающем тексте. Большинство листингов сопровождаются указателями на ключевые понятия или пронумерованными маркерами для ссылки на них в тексте с пояснениями к коду.

Исходный код примеров можно загрузить на сайте издательства, по адресу: www.manning.com/books/go-in-practice, или из репозитория GitHub: github.com/Masterminds/go-in-practice.

Авторский интернет-форум

Приобретая книгу «Go на практике», вы получаете свободный доступ к закрытому веб-форуму издательства Manning Publications, где можно оставить отзыв о книге, задать технические вопросы и получить помощь авторов или других пользователей. Чтобы получить доступ к форуму и зарегистрироваться на нем, перейдите на страницу www.manning.com/books/go-in-practice. Здесь вы найдете информацию о том, как пользоваться форумом после регистрации, какого рода помощь можно получить и правила поведения на форуме.

Издательство Manning, идя навстречу пожеланиям своих читателей, предоставляет площадку для конструктивного диалога между читателями, а также между читателями и авторами. Это не обязывает авторов к участию в нем – любое их участие является добровольным (и никак не оплачивается). Задавайте авторам сложные вопросы, чтобы заинтересовать их!

Авторский интернет-форум и архивы предыдущих дискуссий будут доступны на веб-сайте издателя, пока книга продолжает печататься.

Об авторах

Мэтт Батчер – архитектор программного обеспечения в компании Deis, где занимается проектами с открытым исходным кодом. Написал несколько книг и множество статей. Имеет докторскую степень и преподает на факультете информационных технологий в университете Лойола (Чикаго, США). Мэтт увлечен созданием сильных команд и разработкой элегантных решений сложных проблем.

Мэтт Фарина работает в должности ведущего инженера группы передовых технологий в компании Hewlett Packard Enterprise. Технический писатель, лектор и регулярно вносит свой вклад в проекты с открытым исходным кодом. Занимается разработкой программного обеспечения уже более четверти века. Любит решать проблемы, применяя не только новейшие, но и самые обычные технологии, о которых часто забывают.

Об изображении на обложке

Рисунок на обложке книги «Go на практике» называется «Типичная одежда русской крестьянки, 1768 год». Иллюстрация взята из книги «Collection of the Dresses of Different Nations, Ancient and Modern» (Коллекция костюмов разных народов, античных и современных) Томаса Джефферис (Thomas Jefferys), опубликованной в Лондоне между 1757 и 1772 годом. В подписи к странице было указано, что она представляет выполненную вручную каллиграфическую цветную гравюру, обработанную аравийской камедью.

Томас Джефферис (1719–1771) носил звание «Географ короля Георга III». Английский картограф, был ведущим поставщиком карт того времени. Он выгравировал и напечатал множество карт для нужд правительства, других официальных органов и широкий спектр коммерческих карт и атласов, в частности Северной Америки. Будучи картографом, интересовался местной одеждой народов, населяющих разные земли, и собрал блестящую коллекцию различных платьев в четырех томах.

Очарование далеких земель и дальних путешествий для удовольствия было относительно новым явлением в конце XVIII века, и коллекции, такие как эта, были весьма популярны, знакомя с внешним видом жителей других стран. Разнообразие рисунков, собранных Джефферисом, свидетельствует о проявлении народами мира около 200 лет яркой индивидуальности и уникальности. С тех пор стиль одежды сильно изменился, и исчезло разнообразие, характеризующее различные области и страны. Теперь трудно отличить по одежде даже жителей разных континентов. Если взглянуть на это с оптимистической точки зрения, мы пожертвовали культурным и внешним разнообразием в угоду разнообразию личной жизни, или в угоду более разнообразной и интересной интеллектуальной и технической деятельности.

В наше время, когда трудно отличить одну техническую книгу от другой, издательство Manning проявило инициативу и деловую сметку, украшая обложки книг изображениями, основанными на богатом разнообразии жизненного уклада народов двухвековой давности, придав новую жизнь рисункам Джефферис.

Часть I

ОСНОВНЫЕ ПОНЯТИЯ И ПРИНЦИПЫ

Эта часть книги содержит базовые сведения о языке Go и описание фундаментальных принципов разработки приложений на нем. Глава 1 начинается с обзора языка Go для тех, кто еще с ним не знаком.

Главы 2 и 3 охватывают основные компоненты приложений. Глава 2 описывает основы разработки приложений, включая консольные и серверные, и приемы их настройки. Глава 3 рассматривает использование сопрограмм Go. Сопрограммы являются одним из наиболее мощных и полезных элементов языка Go. Они широко используются в Go-приложениях, и вы часто будете сталкиваться с ними на протяжении всей книги.

Глава 1

Введение в язык Go

В этой главе рассматриваются следующие темы:

- *введение в язык Go;*
- *место языка Go в ландшафте языков программирования;*
- *подготовка к работе с языком Go.*

С течением времени меняется подход к разработке и запуску программного обеспечения. Инновации трансформируют понятие о вычислительной среде, где выполняются программы. Чтобы полностью использовать преимущества нововведений, необходимы языки и инструменты, изначально их поддерживающие.

Во времена, когда создавалось большинство популярных ныне языков программирования и поддерживающих их инструментов, существовали только одноядерные процессоры, соответственно, они проектировались с прицелом на работу в однопроцессорной среде. В настоящее время настольные компьютеры, серверы и даже телефоны оснащены многоядерными процессорами. На любом из них можно выполнять программы с параллельными операциями.

Меняются инструменты разработки приложений. Увеличение сложности программного обеспечения требует окружений, способных быстро собирать код и эффективно его выполнять. Тестирование больших и сложных приложений должно выполняться быстро, чтобы не тормозить процесс разработки. Многие приложения используют библиотеки. Благодаря решению проблемы нехватки дискового пространства появилась возможность поддерживать несколько версий библиотек.

Меняется подход к предоставлению инфраструктуры и программного обеспечения. Использование групп серверов, размещаемых в одном месте, и простое выделение виртуальных частных серверов стали нормой. Прежде масштабирование служб, как правило, озна-

чало необходимость инвестиций в собственное оборудование, включая средства балансировки нагрузки, серверы и хранилища. Закупка всего перечисленного, установка и ввод в эксплуатацию занимали от нескольких недель до нескольких месяцев. Теперь все это можно получить в облаке за несколько секунд или минут.

Эта глава служит введением в язык программирования Go для тех, кто не сталкивался с ним раньше. Она содержит сведения о языке, средствах поддержки, его месте в ряду других языков, установке и начале работы с ним.

1.1. Что представляет собой язык Go?

Язык Go, который часто называют *golang*, чтобы упростить поиск сведений о нем в Интернете, – это статически типизированный и компилируемый язык программирования с открытым исходным кодом, разработку которого начинала компания Google. Роберт Грисемер (Robert Griesemer), Роб Пайк (Rob Pike) и Кен Томпсон (Ken Thompson) сделали попытку создать язык для современных программных систем, способных решать проблемы, с которыми они столкнулись при масштабировании больших систем.

Вместо попытки достичь теоретического совершенства создатели языка Go отталкивались от ситуаций, часто возникающих на практике. Их вдохновлял опыт ведущих языков, созданных прежде, таких как C, Pascal, Smalltalk, Newsqueak, C#, JavaScript, Python, Java и других.

Go – не обычный статически типизированный и компилируемый язык. Его статическая типизация имеет черты, делающие ее похожей на динамическую, а скомпилированные двоичные файлы включают среду выполнения со встроенным сборщиком мусора. На структуру языка наложили отпечаток проекты, для которых он должен был использоваться в Google: большие проекты, поддерживающие масштабирование и разрабатываемые многочисленными группами разработчиков.

По сути, Go – это язык программирования, определяемый спецификациями, которые можно реализовать в любом компиляторе. Их реализация по умолчанию распространяется в виде инструмента `go`. Но Go – это не просто язык программирования. На рис. 1.1 изображена его многослойная структура.

Для разработки приложений нужен не только язык программирования, а также средства тестирования, документирования и форматирования исходного кода. Инструмент `go`, обычно используемый для

компиляции приложений, поддерживает также все вышеперечисленное. Это целый комплект инструментов для разработки приложений. Одним из наиболее важных аспектов этого комплекта является поддержка управления пакетами. Встроенная система управления пакетами, вместе с общими инструментами разработки, позволила сформировать вокруг языка программирования целую экосистему.

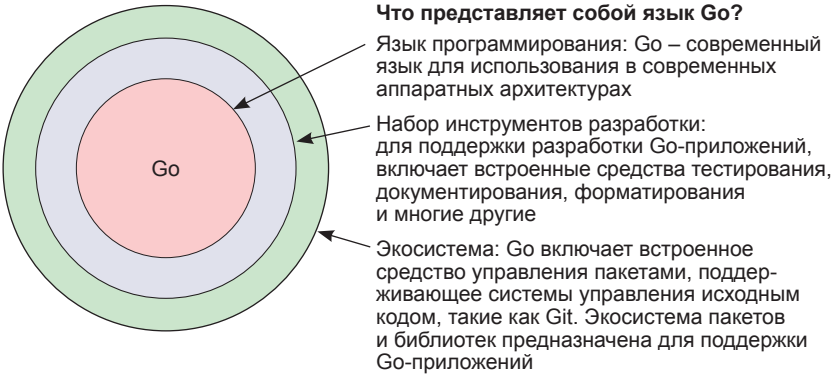


Рис. 1.1 ❖ *Слои языка Go*

Одной из определяющих особенностей Go является его простота. Когда Гисемер, Пайк и Томпсон начинали проектирование языка, новые функциональные возможности не включались в язык, пока все трое не приходили к согласию об их необходимости. Такой стиль принятия решений, а также их многолетний опыт привел к созданию простого и мощного языка. Он прост в изучении, но достаточно мощный для широкого спектра программного обеспечения.

Философию языка можно проиллюстрировать примером синтаксиса объявления переменной:

```
var i int = 2
```

Здесь создается целочисленная переменная, и ей присваивается значение 2. Поскольку имеется присваивание начального значения, определение можно сократить до:

```
var i = 2
```

При наличии начального значения компилятор автоматически определяет по нему тип. В данном случае компилятор обнаружит значение 2 и поймет, что переменная должна иметь целочисленный тип.

Но язык Go не останавливается на этом. А нужно ли само ключевое слово `var`? Нет, потому что Go поддерживает *краткое объявление переменных*:

```
i := 2
```

Это краткий эквивалент первой инструкции определения переменной. Он более чем вдвое короче, легко читается, и все это за счет автоматического определения компилятором недостающих частей.

Простота Go означает, что он поддерживает не все возможности, имеющиеся в других языках. Например, в языке Go отсутствуют тернарный оператор (обычно это `?:`) и обобщенные типы. Не содержит он и некоторых других функциональных возможностей, присутствующих в современных языках, что служит поводом для его критики, но это не должно служить поводом для отказа от использования языка Go. В мире программирования одна и та же задача часто может быть решена множеством способов. Даже если в Go отсутствуют какие-то возможности, имеющиеся в других языках, вместо них он предоставляет иные пути решения проблем, ничуть не хуже.

Несмотря на простоту ядра языка Go, встроенная система управления пакетами позволяет добавлять в него дополнительные возможности. Многие из недостающих элементов можно подключить как сторонние пакеты и включить в приложение.

Минимальный размер и сложность дают свои преимущества. Язык можно быстро освоить, и он хорошо запоминается. Это важное преимущество, когда требуется быстро исследовать чужой код.

1.2. Примечательные особенности языка Go

Поскольку язык Go разрабатывался, исходя из практических нужд, он обладает рядом особенностей, достойных особого упоминания. Все вместе эти полезные характеристики образуют строительные блоки, из которых конструируются Go-приложения.

1.2.1. Возврат нескольких значений

Одна из первых особенностей, которую вы узнаете, начав изучать Go, – функции и методы могут возвращать несколько значений. Большинство языков программирования поддерживает возврат из

функции только одного значения. Если требуется вернуть несколько значений, они встраиваются в кортеж, хэш или любое другое значение составного типа, возвращаемое функцией. Go – один из немногих языков, естественным образом поддерживающих возврат нескольких значений. Эта возможность используется в нем повсеместно, в чем легко убедиться, заглянув в исходный код библиотек и приложений, написанных на языке Go. Для примера рассмотрим следующую функцию, возвращающую две строки с именами.

Листинг 1.1 ❖ Возврат нескольких значений: `returns.go`

```
package main

import (
    "fmt"
)

func Names() (string, string) {  ← ❶ Определена как возвращающая две строки
    return "Foo", "Bar"          ← ❷ Возвращаются две строки
}

func main() {
    n1, n2 := Names()            ❸ Значения присваиваются двум переменным
    fmt.Println(n1, n2)            и выводятся

    n3, _ := Names()             ← ❹ Первое возвращаемое значение сохраняется,
    fmt.Println(n3)                  второе – отбрасывается
}
```

> **СОВЕТ** Импортируемые пакеты, что используются в этой главе, такие как `fmt`, `bufio`, `net` и другие, входят в состав стандартной библиотеки. Более подробную информацию о них, включая описание программных интерфейсов и особенностей работы, можно найти на странице: https://golang.org/pkg.

Как показано в этом примере, типы возвращаемых значений объявляются в определении функции, после списка параметров ❶. В данном случае функция возвращает два строковых значения. Оператор `return` возвращает две строки ❷, в соответствии с определением. Вызывая функцию `Names`, необходимо предоставить переменные для всех возвращаемых значений ❸. Но, если требуется сохранить только одно из возвращаемых значений, для отбрасываемого значения укажите специальную переменную `_` ❹. (Можете особенно не беспокоиться о деталях реализации в этом примере. Мы еще вернемся к понятиям, библиотекам и инструментам, использованным здесь, в следующих главах.)

Опираясь на идею возврата нескольких значений, возвращаемым значениям можно дать имена и работать с ними, как с переменными. Для иллюстрации переделаем предыдущий пример, использовав именованные возвращаемые значения.

Листинг 1.2 ❖ Именованные возвращаемые значения: `returns2.go`

```
package main

import (
    "fmt"
)

func Names() (first string, second string) {
    first = "Foo"
    second = "Bar"
    return
}

func main() {
    n1, n2 := Names()
    fmt.Println(n1, n2)
}
```

❶ Именованные возвращаемые значения

❷ Присваивание значений именованным возвращаемым переменным

❸ Оператор return вызывается без значений

❹ Значения присваиваются двум переменным

Функция `Names` возвращает именованные переменные ❶, содержащие присвоенные им значения ❷. Когда оператор `return` вызывается без перечисления возвращаемых значений ❸, он возвращает текущие значения возвращаемых именованных переменных. Код, вызывающий функцию, получает возвращаемые значения ❹ и использует их так же, как в примере, где возвращаемые значения не имели имен.

1.2.2. Современная стандартная библиотека

Современные приложения решают определенные типовые задачи, такие как сетевые операции или шифрование. И чтобы не обременять разработчиков поиском библиотек для решения этих задач, стандартная библиотека Go изначально включает такие полезные функции. Остановимся на некоторых элементах стандартной библиотеки, чтобы получить общее представление о ее содержимом.

> **ПРИМЕЧАНИЕ** Полное описание стандартной библиотеки с примерами можно найти на странице: http://golang.org/pkg/.

Сетевые операции и HTTP

Под сетевыми приложениями подразумеваются и клиентские приложения, способные подключаться к другим сетевым устройствам,

и серверные, позволяющие подключаться к себе другим приложениям (листинг 1.3). Стандартная библиотека Go легко со всем этим справляется, будь то работа по протоколу HTTP, TCP (Transmission Control Protocol), UDP (User Datagram Protocol) или с применением других типовых возможностей.

Листинг 1.3 ❖ Чтение состояния по протоколу TCP: `read_status.go`

```
package main

import (
    "bufio"
    "fmt"
    "net"
)

func main() {
    conn, _ := net.Dial("tcp", "golang.org:80")
    fmt.Fprintf(conn, "GET / HTTP/1.0\r\n\r\n")
    status, _ :=
     ➥bufio.NewReader(conn).ReadString('\n')
    fmt.Println(status)
}
```

❶ Соединение по TCP

❷ Отправка строки через соединение

❸ Вывод первой строки ответа

Непосредственное подключение к порту обеспечивает пакет `net`, в котором язык Go предоставляет типовые настройки для различных типов соединений. Функция `Dial` ❶ при подключении использует указанный тип и конечную точку. В данном случае создается TCP-соединение с портом 80 по адресу `golang.org`. После подключения посылается запрос `GET` ❷ и выводится первая строка ответа ❸.

Прием входящих запросов на соединение реализуется так же просто. Только вместо функции `Dial` используется функция `Listen` из пакета `net`, которая переводит приложение в режим приема входящих соединений.

Протокол HTTP, службы REST (Representational State Transfer) и веб-серверы широко используются для взаимодействий в сети. Для их поддержки в язык Go был включен пакет `http`, содержащий реализации клиента и сервера (пример его использования приводится в листинге 1.4). Клиент достаточно прост в использовании, поскольку отвечает обычным повседневным потребностям и допускает расширение, охватывающее сложные случаи.

Листинг 1.4 ❖ HTTP-запрос GET: `http_get.go`

```
package main

import (
    "fmt"
    "io/ioutil"
    "net/http"
)

func main() {
    resp, _ :=
    ➥http.Get("http://example.com/") ← Создание HTTP-запроса GET
    body, _ :=
    ➥ioutil.ReadAll(resp.Body) ← Чтение тела ответа
    fmt.Println(string(body)) ← Вывод тела в виде строки
    resp.Body.Close() ← Закрытие соединения
}
```

Этот пример демонстрирует вывод тела простого HTTP-запроса `GET`. HTTP-клиент может намного больше, например работать с прокси, обрабатывать шифрование TLS, устанавливать заголовки, обрабатывать cookie, создавать клиентские объекты и даже подменять весь транспортный уровень.

Создание HTTP-сервера на языке Go – весьма распространенная задача. Стандартная библиотека Go обладает мощными возможностями, поддерживающими масштабирование, простыми в освоении и достаточно гибкими для применения в сложных приложениях. Созданию HTTP-сервера и работе с ним будет посвящена глава 3.

HTML

Работая над созданием веб-сервера, невозможно обойти стороной разметку HTML. Пакеты `html` и `html/template` отлично справляются с формированием веб-страниц. Пакет `html` имеет дело с экранированной и неэкранированной HTML-разметкой, а пакет `html/template` предназначен для создания многократно используемых HTML-шаблонов. Модель безопасности обработки данных прекрасно документирована, и имеется ряд вспомогательных функций для работы с HTML, JavaScript и многим другим. Система шаблонов поддерживает возможность расширения, что делает ее идеальной основой для реализации более сложных действий.

Криптография

В настоящее время криптография стала обычным компонентом приложений, используемым для работы с хэшами или шифрования конфиденциальной информации. Язык Go предоставляет часто используемые функциональные возможности, включающие поддержку MD5, нескольких версий Secure Hash Algorithm (SHA), Transport Layer Security (TLS), Data Encryption Standard (DES), Triple Data Encryption Algorithm (TDEA), Advanced Encryption Standard (AES, прежний Rijndael), Keyed-Hash Message Authentication Code (HMAC) и многих других. Кроме того, в нем имеется криптографически безопасный генератор случайных чисел.

Кодирование данных

При совместном использовании данных несколькими системами встают типичные для современных сетей задачи кодирования, такие как: обработка данных в формате base64, преобразование данных в формате JSON (JavaScript Object Notation) или XML (Extensible Markup Language) в локальные объекты.

Язык Go разрабатывался с учетом необходимости решать задачи кодирования. Внутренне Go использует только кодировку UTF-8. Это неудивительно, поскольку создатели Go прежде занимались созданием UTF-8. Но далеко не всегда при обмене между системами используется кодировка UTF-8, поэтому приходится иметь дело с самыми разными форматами данных. Для их обработки в языке Go имеются соответствующие пакеты и интерфейсы. Пакеты поддерживают такие возможности, как преобразование строк JSON в экземпляры объектов, а интерфейсы обеспечивают переключение между кодировками и добавление новых способов работы с кодировками посредством подключения внешних пакетов.

1.2.3. Параллельная обработка с помощью сопрограмм и каналов

Многоядерные процессоры стали обычным явлением. Они применяются во многих устройствах, начиная с серверов и заканчивая мобильными телефонами. Однако большинство языков программирования разрабатывалось под одноядерные процессоры, поскольку на тот момент только они и существовали.

Кроме того, среда выполнения в некоторых языках имеет глобальную блокировку, что затрудняет параллельное выполнение процедур.

Язык Go изначально ориентировался на параллельную и конкурентную обработку.

В языке Go имеются так называемые *сопрограммы*, или *go-подпрограммы*, – процедуры, способные выполняться параллельно с основной программой и другими сопрограммами. Иногда называемые *легковесными потоками*, сопрограммы управляются средой выполнения Go, которая помещает их в соответствующие потоки операционной системы и утилизирует их с помощью сборщика мусора, когда они становятся ненужными. При наличии в системе процессора с несколькими ядрами go-подпрограммы способны выполняться параллельно, поскольку различные потоки могут выполняться одновременно на разных ядрах. Для разработчика создание сопрограмм выглядит не сложнее написания обычных функций. Рисунок 1.2 иллюстрирует работу go-подпрограмм.

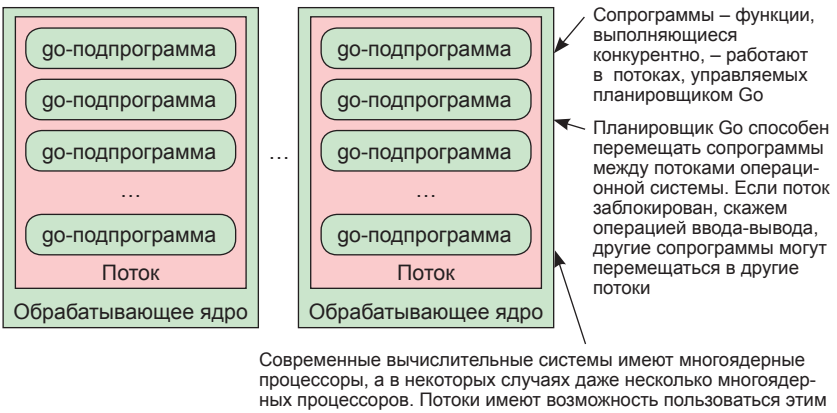


Рис. 1.2 ❖ *Go-подпрограммы выполняются в потоках, распространяемых на все доступные ядра*

Для иллюстрации рассмотрим сопрограмму, ведущую счет от 0 до 4 одновременно с тем, как основная программа печатает `Hello World`, как показано в листинге 1.5.

Листинг 1.5 ❖ Результат конкурентного вывода

```
0
1
Hello World 2
3
4
```

Такой вывод получается в результате одновременного выполнения двух функций. Соответствующий код, следующий ниже, напоминает обычную процедурную программу, с небольшим исключением.

Листинг 1.6 ❖ Конкурентный вывод

```
package main

import (
    "fmt"
    "time"
)

func count() {
    for i := 0; i < 5; i++ {
        fmt.Println(i)
        time.Sleep(time.Millisecond * 1)
   }
}

func main() {
    go count()
    time.Sleep(time.Millisecond * 2)
    fmt.Println("Hello World")
    time.Sleep(time.Millisecond * 5)
}
```

❶ Функция, выполняемая как go-подпрограмма

← ❷ Вызов go-подпрограммы

Функция `count` ❶ – это самая обычная функция, которая выводит числа от 0 до 4. Чтобы запустить ее параллельно с основной программой, используется ключевое слово `go` ❷. Благодаря этому функция `main` продолжает работать, как обычно, после вызова функции `count`. В результате функции `count` и `main` выполняться одновременно.

Передача данных между сопрограммами может осуществляться через каналы. По умолчанию они блокируют выполнение, позволяя сопрограммам синхронизироваться друг с другом. Рисунок 1.3 иллюстрирует это на простом примере.

В этом примере переменная передается от одной сопрограммы к другой через канал. Эта операция работает, даже когда сопрограммы выполняются параллельно, на различных ядрах процессора. В примере показана однонаправленная передача информации, но вообще каналы могут быть не только однонаправленными, но и двунаправленными.

В следующем листинге приводится пример использования канала.

Рис. 1.3 ❖ *Передача переменных между go-подпрограммами*

Листинг 1.7 ❖ Использование каналов: `channel.go`

```
package main import (

    "fmt"
    "time"
)

func printCount(c chan int) {  ← ❶ Канал для передачи целого значения
    num := 0
    for num >= 0 {
        num = <-c   ← ❷ Ожидание целого значения
        fmt.Print(num, " ")
    }
}

func main() {
    c := make(chan int)  ← ❸ Создание канала
    a := []int{8, 6, 7, 5, 3, 0, 9, -1}
    go printCount(c)  ← ❹ Вызов сопрограммы
    for _, v := range a {  ← ❺ Запись целого значения в канал
        c <- v
    }
    time.Sleep(time.Millisecond * 1)  ← ❻ Функция main приостанавливается
    fmt.Println("End of main")            перед завершением
}
```

В начале функции `main` создается канал `c` для передачи целого числа ❸ между сопрограммами. Функция `printCount`, вызываемая как сопрограмма, передает значение в канал ❹. Согласно определению

параметра функции printCount, канал идентифицируется как канал для передачи целых чисел ❶. В цикле for функция printCount ожидает появления в канале c целого числа ❷ и присваивает его переменной num. Тем временем функция main выполняет обход списка целых чисел и поочередно передает их в канал c ❺. После передачи в канал очередного целого числа из main ❺ оно присваивается переменной num в функции printCount ❷. Функция printCount продолжит выполнение цикла, пока в следующей итерации вновь не достигнет оператора чтения из канала ❷, и снова приостановится до появления в канале следующего целого значения. После очередной итерации в функции main и записи нового целого значения выполнение продолжится без задержек. По завершении функции main будет закончено выполнение всей программы, поэтому необходимо выполнить паузу в одну секунду ❻, чтобы позволить функции printCount завершить выполнение раньше, чем это сделает функция main. Если запустить этот код, он выведет следующее.

Листинг 1.8 ❖ Вывод с использованием канала

```
8 6 7 5 3 0 9 -1 End of main
```

Совместное использование каналов и сопрограмм обеспечивает возможности, напоминающие легковесные потоки выполнения или внутренние микрослужбы, обменивающиеся данными через специализированный прикладной программный интерфейс. Они могут объединяться в цепочки или комбинироваться различными способами.

Go-подпрограммы (сопрограммы) и каналы являются двумя самыми мощными особенностями языка Go, которые не раз будут упоминаться на протяжении всей книги. Вы увидите, как использовать их для реализации серверов, передачи сообщений и задержки выполнения задач. Также будут описаны шаблоны проектирования, основанные на сопрограммах и каналах.

1.2.4. Go – больше, чем язык

Разработка современных масштабируемых и простых в обслуживании приложений требует множества элементов. Компиляция – лишь один из этапов в этом процессе. Так было задумано изначально. Go – это больше, чем язык и компилятор. Утилита go – это целый комплекс инструментов для управления пакетами, тестирования, документирования и многого другого, помимо компиляции кода на языке Go в исполняемые файлы. Рассмотрим несколько компонентов в этом наборе.

Управление пакетами

Диспетчеры пакетов существуют для многих современных языков программирования, но у скольких из них функция управления пакетами является встроенной? В языке Go она встроенная, и тому есть две веские причины. Самая очевидная – продуктивность программиста. Вторая причина – ускорение компиляции. Управление пакетами разрабатывалось с учетом нужд компилятора. Это один из аспектов, определяющих быстроту компиляции.

Идею пакетов в Go проще всего изучать на примере стандартной библиотеки (ее демонстрирует следующий листинг), которая организована на основе системы управления пакетами.

Листинг 1.9 ❖ Простой импорт пакета

```
package main

import "fmt" ← Импорт пакета fmt

func main() {
    fmt.Println("Hello World!") ← Использование функции пакета fmt
}
```

Пакеты импортируются по именам. В данном случае `fmt` – это пакет с функциями форматирования. Все имеющееся в пакете доступно при использовании имени пакета в виде префикса. Как, например, вызов `fmt.Println` в предыдущем примере.

Импортируемые пакеты можно группировать, но в этом случае они должны указываться в алфавитном порядке. После импортирования пакетов, как показано ниже, на пакет `net/http` можно ссылаться, используя префикс `http`.

```
import (
    "fmt"
    "net/http"
)
```

Механизм импортирования работает также с пакетами, не входящими в стандартную библиотеку Go, и ссылки на такие пакеты используются точно так же:

```
import (
    "golang.org/x/net/html" ← Ссылка на внешний пакет по адресу URL
    "fmt"
    "net/http"
)
```

Имена пакетов являются уникальными строками и могут быть чем угодно. Здесь для ссылки на внешний пакет использован URL-адрес. Это позволяет языку Go опознать уникальный ресурс и получить его.

```
$ go get ./...
```

Команда `go get` принимает путь к отдельному пакету, например `golang.org/x/net/html`, для загрузки отдельного пакета, или `./...`. В последнем случае Go выполнит обход базы кода и загрузит все внешние пакеты. В данном случае (см. пример выше) Go обнаружит инструкцию `import`, выделит из нее внешнюю ссылку, загрузит пакет и сделает его доступным в текущей рабочей области.

Язык Go может загружать пакеты из систем управления версиями. Он поддерживает Git, Mercurial, SVN и Bazaar, если они установлены в локальном окружении. В этом случае Go загружает базу кода из Git и проверяет последнюю версию в ветви по умолчанию.

Система управления пакетами поддерживает далеко не все, чего можно было бы пожелать. Она реализует лишь базовые возможности, которые можно использовать непосредственно или как основу для системы с более широкими возможностями.

Тестирование

Тестирование – общепринятый элемент разработки программного обеспечения, и по мнению некоторых – весьма существенный. В Go имеется полноценная система тестирования, включающая пакет в стандартной библиотеке, средство выполнения тестов из командной строки, средство оценки охвата кода тестами и средство обнаружения состояний «гонки за ресурсами».

Процесс создания и выполнения тестов достаточно прост, как показано в следующем листинге.

Листинг 1.10 ❖ Пример Hello World: `hello.go`

```
package main

import "fmt"

func getName() string {
    return "World!"
}

func main() {
    name := getName()
    fmt.Println("Hello ", name)
}
```

Начнем с варианта приложения «Hello World», содержащего функцию `getName`, которая и будет тестироваться. В соответствии с соглашением в языке Go имена файлов с тестами заканчиваются на `_test.go`. Данный суффикс сообщает среде Go, что этот файл предназначен для тестирования и должен игнорироваться при сборке приложения.

Листинг 1.11 ❖ Тест для примера Hello World: `hello_test.go`

```
package main

import "testing"

func TestName(t *testing.T) {
    name := getName()
    if name != "World!" {
        t.Error("Respone from getName is unexpected value")
    }
}
```

❶ Инструмент запуска тестов вызывает функции, начинающиеся с Test

❷ Отчет об ошибке, если тест не пройден

Если запустить команду `go test`, она выполнит функцию с именем, начинающимся с префикса `Test` ❶, – в данном случае функцию `TestName` – и передаст ей вспомогательную структуру `t`. Структура содержит несколько полезных функций, например для вывода сообщений об ошибках. За дополнительной информацией обращайтесь к документации описанием пакета `testing`. Если значение переменной `name` окажется неправильным, тест выведет сообщение об ошибке ❷.

В следующем листинге показан вывод команды `go test`, сообщающий имя пакета и результаты его тестирования. Для проверки текущего пакета и всех пакетов во вложенных подкаталогах можно использовать команду `go ./...`.

Листинг 1.12 ❖ Выполнение команды `go test`

```
$ go test
PASS
ok  go-in-practice/chapter1/hello     0.012s
```

Если метод `getName` вернет строку, отличную от `"World!"`, результат будет иным. В следующем примере система тестирования сообщает о месте, где тест выявил ошибку, имя теста, имя файла с тестом и номер строки с ошибкой. Для следующего примера мы изменили метод `getName`, чтобы он возвращал строку, отличную от `"World!"`.

Листинг 1.13 ❖ Ошибка, выявленная при тестировании командой `go test`

```
$ go test
--- FAIL: TestName (0.00 seconds)
```

```
      hello_test.go:9: Response from getName is unexpected value
FAIL
exit status 1
FAIL   go-in-practice/chapter1/hello              0.010s
```

В состав Go входят все необходимые инструменты для тестирования. Более того, Go сам использует их. Для тех, кому потребуется что-то еще, например средства поддержки разработки, основанной на поведении (Behavior-Driven Development, BDD), или нечто другое, что можно найти в других языках, существуют внешние пакеты, расширяющие встроенные функциональные возможности. В Go-тестах можно использовать весь спектр возможностей языка, включая пакеты.

Охват кода

Помимо выполнения тестов, система тестирования поддерживает создание отчетов с полной детализацией охвата кода тестами, вплоть до уровня инструкций, как показано на рис. 1.14.

> **ПРИМЕЧАНИЕ** В версии Go 1.5 команды получения оценки охвата тестами стали частью ядра Go. До версии 1.5 команда `cover` относилась к дополнительным инструментам.

Чтобы получить оценку охвата тестами, выполните следующую команду:

```
$ go test -cover
```

Добавление флага `-cover` в команду `go test` заставит ее вывести информацию об охвате тестами вместе с прочими сведениями о выполнении тестов.

Листинг 1.14 ❖ Тестирование с получением сведений об охвате тестами

```
$ go test -cover
PASS
Coverage: 33.3% of statements
ok   go-in-practice/chapter1/hello        0.011s
```

Но предоставлением этой информации система оценки охвата кода не ограничивается. Сведения об охвате можно экспортировать в файлы для использования другими инструментами. Также эти отчеты можно отображать с помощью встроенных средств. На рис. 1.4 показано, как выглядит отчет в веб-браузере, содержащий данные о тестируемых инструкциях.

Рис. 1.4 ❖ *Отчет об охвате кода тестами в веб-браузере*

Обычно такие отчеты содержат данные с детализацией, вплоть до строки. Простым примером служат инструкции `if` и `else`. Среда Go покажет, какие инструкции были выполнены, а какие остались не охваченными тестированием.

> **СОВЕТ** Более полную информацию об инструменте `cover` можно найти в блоге языка Go: http://blog.golang.org/cover.

Тестированию в языке Go придается большое значение, и мы еще уделим время этой стороне программирования в главе 4.

Форматирование

Как предпочтительнее оформлять отступы в исходном коде, символами табуляции или пробелами? Вопросы форматирования и стиля затрагиваются и обсуждаются всякий раз, когда речь заходит о соглашениях оформления кода. Сколько можно было бы сэкономить времени, отказавшись от этих дебатов? Пользователям языка Go нет смысла тратить время на обсуждение форматирования или других языковых идиом.

На странице http://golang.org/doc/effective_go.html можно ознакомиться с руководством «*Effective Go*»[1], посвященным идиоматическим особенностям языка Go. В нем описаны стили и соглашения, используемые всеми членами Go-сообщества. Эти соглашения облегчают чтение и изменение написанных на языке Go программ.

Среда Go включает встроенный инструмент форматирования, поддерживающий большинство рекомендаций по оформлению. Доста-

[1] Краткий пересказ на русском можно найти по адресу: http://golang-club.blogspot.ru/2016/03/effective-go.html. – *Прим. ред.*

точно запустить команду `go fmt` в корневом каталоге пакета, чтобы Go обошел все файлы `.go` в пакете и привел их в соответствие с каноническим стилем. В команде `go fmt` можно дополнительно указать путь к пакету или `./...` (для обхода всех подкаталогов).

Многие редакторы поддерживают автоматическое форматирование через встроенные средства или в виде дополнительных пакетов. К ним относятся: Vim, Sublime Text, Eclipse и многие другие. Например, пакет GoSublime для редактора Sublime Text производит форматирование файла при его сохранении.

1.3. Место языка Go среди других языков программирования

Популярный репозиторий GitHub хранит проекты, написанные на сотнях разных языков. Рейтинговый список TIOBE языков программирования показывает снижение популярности наиболее распространенных языков. Это дает шанс другим языкам. Давайте посмотрим, какое место занимает Go среди множества других языков программирования.

Изначально Go разрабатывался как системный язык. То, что часто называют *облачными вычислениями*, обычно относят к одной из форм системного программирования. Язык Go был разработан с учетом особенностей систем, в которых он должен был использоваться.

Системные языки должны обладать определенными особенностями. Например, Go можно применять в случаях, где обычно используются языки C или C++, но он не годится для встраиваемых систем. Среда выполнения Go обладает системой сборки мусора, которая не будет нормально работать во встраиваемых системах с ограниченными ресурсами.

Сравнение Go с другими популярными языками программирования позволит определить его положение по отношению к этим языкам. Мы считаем, что Go оптимально подходит для разработки определенных приложений, но не собираемся вести дебаты о выборе языка программирования. Для правильного выбора необходимо учитывать не только характеристики языков.

1.3.1. C и Go

Язык Go с самого начала рассматривался как альтернатива языку C для разработки приложений. Поскольку источником вдохновения

при разработке этого языка был язык C (C по-прежнему остается одним из популярных языков, если не самым популярным), имеет смысл рассмотреть сходства и различия этих языков.

Программы на обоих языках – Go и C – компилируются в машинный код целевой операционной системы и архитектуры. Оба языка схожи по своему стилю, но Go далеко выходит за рамки языка C по своим возможностям.

Язык Go имеет среду выполнения, поддерживающую такие функции, как управление потоками выполнения и сборка мусора. Разрабатывая приложения на языке Go, вы освобождаетесь от обязанности управлять потоками выполнения и выполнять операции по сборке мусора, как это принято в других языках, оснащенных сборщиком мусора. В языке C управление потоками и памятью возлагается на программиста. Вся работа с потоками и связанные с ними действия выполняются приложением. А для работы с памятью сборщик мусора в принципе не предусмотрен.

Язык C и его объектно-ориентированные производные, такие как C++, обеспечивают разработку весьма широкого спектра приложений. На языке C можно писать высокопроизводительные встраиваемые системы, крупномасштабные облачные и сложные настольные приложения. Язык Go в основном предназначен для использования в качестве системного языка и языка создания облачных платформ. Преимуществом Go является высокая продуктивность.

Среда выполнения и набор инструментов Go доступны изначально. Их возможности позволяют очень быстро разрабатывать приложения и тратить на их создание гораздо меньше усилий, чем требуется для написания сопоставимого приложения на языке C. Например, для использования всех четырех ядер процессора на сервере Go-приложение может использовать сопрограммы. В аналогичном приложении на C придется вдобавок написать код, запускающий потоки выполнения, управляющий их работой и выполняющий переключение между ними.

Компиляция приложений на C может занимать продолжительное время. Это особенно верно при наличии внешних зависимостей и необходимости их компиляции. Высокая скорость компиляции Go-приложений была одной из целей разработки языка Go, и она выполняется быстрее, чем компиляция кода на языке C. Когда приложение разрастается до таких размеров, что его компиляция начинает занимать минуты или даже часы, экономия на времени компиляции может существенно повлиять на производительность разработки.

Компиляция Go-приложений выполняется настолько быстро, что компиляция многих приложений и пакетов их зависимостей занимает считанные секунды или даже еще меньше.

C + Go = cgo

Язык Go поддерживает использование библиотек, написанных на C, в программах на Go. В состав Go входит библиотека инструментов поддержки совместимости с языком C. Эта библиотека облегчает, например, переход от строк в стиле C к строкам языка Go. Кроме того, инструменты Go позволяют компоновать программы из исходного кода на языках C и Go. Язык Go также поддерживает обертки Simplified Wrapper and Interface Generator (SWIG). Получить общее представление об имеющихся возможностях можно с помощью команд `go help c` и `go doc cgo`.

1.3.2. Java и Go

Java долгое время является одним из самых популярных языков программирования и используется для разработки широкого спектра проектов, от серверных до мобильных Android-приложений и кроссплатформенных приложений для настольных компьютеров. Go изначально разрабатывался как системный язык. С течением времени стало возможным использовать его для разработки мобильных и веб-приложений, тем не менее Go не позволяет с легкостью писать настольные приложения. Его превосходные качества проявляются только при использовании по назначению.

Учитывая популярность Java и возможность его применения для создания широкого спектра приложений, почему у кого-то может возникнуть желание пользоваться языком Go? Языки Java и Go имеют схожий синтаксис, но в действительности они сильно отличаются. Программы на Go компилируются в единственный двоичный файл, предназначенный для определенной операционной системы. Он содержит среду выполнения Go, все импортированные пакеты и само приложение, то есть все, что необходимо для выполнения программы. В Java используется другой подход. Приложения на Java требуют установки среды выполнения в операционной системе. Они упаковываются в файл, который может выполняться в любой системе с соответствующей средой выполнения. Приложения требуют наличия совместимой версии среды выполнения.

Это различие подходов, проиллюстрированное на рис. 1.5, определяет способ использования приложений. Для развертывания Go-

приложения на сервере достаточно установить один файл. Чтобы развернуть Java-приложение, вместе с самим приложением требуется установить и настроить среду выполнения Java.

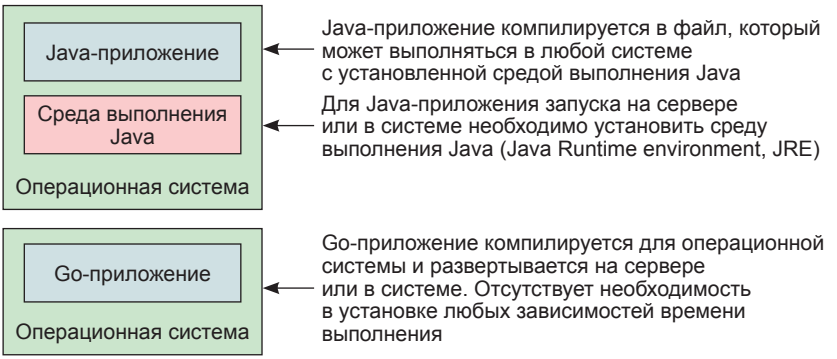


Рис. 1.5 ❖ *Выполнение программ на языках Java и Go в операционной системе*

Другое ключевое различие между языками Java и Go заключается в самом порядке выполнения приложения. Программы на языке Go компилируются в двоичный код и выполняются операционной системой. Java-приложения выполняются виртуальной машиной (VM), обычно содержащей динамический компилятор (Just-In-Time, JIT). JIT-компилятор способен анализировать выполнение кода в текущем контексте и оптимизировать его.

Возникает важный вопрос: что работает быстрее – код, выполняемый виртуальной машиной с JIT-компилятором, или предварительно скомпилированный двоичный код? Ответ на него не прост, поскольку многое зависит от JIT-компилятора, выполняемого кода и многого другого. Сравнительные тесты программ с аналогичной функциональностью не выявили явного лидера.

1.3.3. Python, PHP и Go

Python и PHP – два наиболее популярных динамических языка. Python превратился в один из самых популярных языков, изучаемых и используемых в университетах. Его можно применять в различных целях, от создания облачных приложений до разработки веб-сайтов и утилит, запускаемых из командной строки. PHP – один из самых популярных языков программирования для создания веб-сайтов. Хотя эти два языка имеют множество различий, между ними наблю-

даются определенные сходства, подчеркивающие некоторые из подходов, используемых в Go.

Python и PHP – языки с динамической типизацией, в то время как Go является языком со статической типизацией, поддерживающим некоторые черты, характерные для динамической типизации. Динамические языки осуществляют проверку типов данных во время выполнения и даже выполняют их преобразование на лету. В языках со статической типизацией проверка типов данных выполняется на основании статического анализа кода. Язык Go также поддерживает преобразование типов в определенных границах. В некоторых случаях переменные одного типа могут преобразовываться в переменные другого типа. Это может выглядеть необычным для языка со статической типизацией, но иногда очень удобно.

Обычно, когда PHP и Python используются для создания веб-сайтов или приложений, они располагаются за веб-сервером, таким как Nginx или Apache. Веб-браузер подключается к веб-серверу, обрабатывающему соединения, а веб-сервер передает информацию среде выполнения и программе, написанной на этих языках.

Go имеет встроенный веб-сервер, как показано на рис. 1.6. Такие приложения, как веб-браузеры, подключаются напрямую к Go-приложению, и оно само управляет соединением. Это обеспечивает более низкоуровневое управление и взаимодействие с подключившимися приложениями. Веб-сервер, встроенный в Go, в состоянии параллельно обрабатывать множество соединений, используя все функциональные преимущества языка.

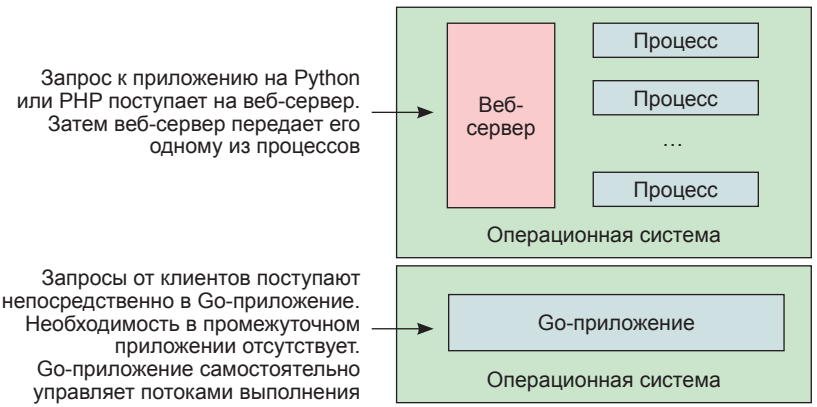


Рис. 1.6 ❖ *Прохождение запросов от клиентов к приложениям на Python, PHP и Go*

Одно из преимуществ размещения приложений на Python и PHP за веб-сервером заключается в том, что он берет на себя обслуживание потоков выполнения и параллельных подключений. В Python предусмотрена глобальная блокировка интерпретатора, позволяющая одновременно выполняться только одному потоку. PHP-приложение, как правило, выполняется от начала до конца процесса. Для одновременного доступа к приложению нескольких клиентов перед приложением должен находиться веб-сервер, запускающий обработку соединений в отдельных процессах.

Для конкурентного обслуживания подключений веб-сервер, встроенный в Go, использует сопрограммы. Среда выполнения Go распределяет сопрограммы между потоками выполнения. Более подробно об этом рассказывается в главе 3. Для обслуживания нескольких соединений запускается несколько процессов с приложениями на Python и PHP, но Go-приложение использует одну общую среду, что позволяет совместно использовать ее ресурсы там, где это имеет смысл.

Внутренняя реализация Python и PHP написана на C. Их встроенные объекты, методы и функции, реализованные на C, выполняются быстрее, чем объекты, методы и функции приложений. Исходный прикладной код преобразуется в промежуточную интерпретируемую форму. Один из советов по повышению производительности критических участков кода, написанных на Python и PHP, заключается в переводе их на язык C.

Go-приложения компилируются в выполняемый двоичный код. То есть программный код из стандартной библиотеки и из приложения компилируется в машинный код. Между ними нет никакой разницы.

1.3.4. JavaScript, Node.js и Go

Язык JavaScript был создан практически за 10 дней. Он стал одним из самых популярных языков благодаря встраиванию во все основные веб-браузеры. В последнее время JavaScript используется на серверах, в настольных приложениях и в других областях. Это стало возможным благодаря платформе Node.js.

Языки Go и JavaScript, главным образом благодаря Node.js, могут заполнять аналогичные ниши, но делают это по-разному. Исследование этих различий поможет лучше разобраться в роли языка Go.

JavaScript использует однопоточную модель. Несмотря на поддержку асинхронного ввода/вывода, который может выполняться в отдельных потоках, основная программа выполняется только в одном потоке. Когда выполнение кода в основном потоке требует зна-

чительного количества времени, он блокирует выполнение другого кода. В Go используется многопоточная модель, где среда выполнения управляет выполнением сопрограмм в отдельных потоках. Модель Go с возможностью выполнения потоков на нескольких ядрах способна более полно использовать доступное оборудование, чем однопоточная модель JavaScript.

Платформа Node.js использует движок V8, разработанный в компании Google и входящий в состав в Google Chrome. Движок V8 содержит виртуальную машину с JIT-компилятором. Концептуально это решение схоже с Java. Виртуальная машина и JIT-компилятор способны несколько повысить производительность. Движок V8 следит за длительностью выполнения программ и с течением времени повышает производительность. Например, он может распознать выполнение цикла и для увеличения производительности преобразовать часть программного кода непосредственно в машинный код. Движок V8 способен определять типы переменных, даже при том, что JavaScript является динамически типизированным языком. Чем дольше работает программа, тем больше движку V8 удается узнать о ней и применить собранную информацию в целях улучшения производительности.

Программы на Go, напротив, сразу компилируются в машинный код и выполняются со скоростью машинного кода со статической типизацией. Здесь нет нужды в JIT-компиляторе для увеличения скорости выполнения. Как и в языке C, нет необходимости в применении JIT-компиляции.

Работа с пакетами

Экосистема Node.js включает сообщество и инструментальные средства для обработки и распространения пакетов. Они могут варьироваться от библиотек до утилит командной строки и полноценных приложений. В комплект Node.js обычно входит диспетчер пакетов `npm`. Центральный репозиторий с информацией о доступных пакетах находится на сайте www.npmjs.org. Когда поступает команда загрузить пакет, диспетчер извлекает метаданные о нем из центрального репозитория и загружает пакет из исходного месторасположения.

Как упоминалось ранее, язык Go имеет собственную систему работы с пакетами. В отличие от Node.js, где метаданные и дополнительная информация располагаются в центральном репозитории, язык Go не имеет центрального хранилища, и пакеты загружаются из исходного местоположения.

1.4. Подготовка и запуск программы на языке Go

Имеется несколько вариантов знакомства с языком Go, в зависимости от ваших предпочтений.

Самый простой способ начать работу с языком Go – пройти тур на странице http://tour.golang.org, демонстрирующий использование некоторых основных функций. Что выделяет тур по языку Go в ряду типичных пособий, так это действующие примеры. Примеры можно выполнять непосредственно в браузере. При желании можно даже внести в них изменения и выполнить.

Поэкспериментировать с простыми Go-приложениями можно на странице https://play.golang.org. Эта площадка для экспериментов позволяет опробовать примеры из тура. Здесь можно проверить код и поделиться ссылкой на него. Кроме того, на площадке можно также опробовать примеры кода из книги.

1.4.1. Установка Go

Установка Go – довольно простая процедура. Вся необходимая информация о получении и установке Go приводится на странице http://golang.org/doc/install. Здесь перечисляются поддерживаемые операционные системы, аппаратное обеспечение и многое другое.

Для Microsoft Windows и Mac OS X имеются инсталляторы, которые позаботятся об установке. Этот процесс настолько же прост, как установка любой другой программы. Пользователи Homebrew на OS X могут установить Go командой `brew install go`.

Установка Go в Linux предусматривает более широкий диапазон вариантов. Установить Go можно с помощью встроенного диспетчера пакетов, такого как `apt-get` или `yum`. Но, как правило, они устанавливают не самую последнюю версию, притом, что новые версии работают быстрее и поддерживают новые функциональные возможности. Следуя инструкциям по установке Go, можно загрузить самую последнюю версию языка, разместить ее в системе и добавить пути к исполняемым файлам в переменную окружения `PATH`. В версиях Linux, поддерживающих систему управления пакетами Debian, таких как Ubuntu, установить последнюю версию Go можно с помощью `godeb`. Пояснения автора `godeb` о процедуре установки можно найти на странице http://blog.labix.org/2013/06/15/in-flight-deb-packages-of-go.

1.4.2. Работа с Git, Mercurial и другими системами управления версиями

Для работы с пакетами и внешними зависимостями, хранящимися в системах управления версиями, среда Go требует локальной установки этих систем. Go не дублирует инструментов управления конфигурацией программного обеспечения (Software Configuration Management, SCM), а просто распознает их и использует при установке.

Двумя доминирующими системами управления версиями, используемыми Go-разработчиками для работы с пакетами, являются Git и Mercurial (`hg`). Система управления версиями Git очень популярна, ею пользуются многие разработчики из Google, и создаваемые ими пакеты хранятся в GitHub. От вас требуется лишь установить Git, при этом среда Go не требует какой-то определенной версии. Подойдет любая из последних.

1.4.3. Знакомство с рабочей областью

Утилита `go` предполагает, что исходный код на языке Go хранится в рабочей области. *Рабочая область* – это иерархия, включающая каталоги `src`, `pkg` и `bin`, как показано в следующем листинге.

Листинг 1.15 ❖ Структура рабочей области

```
$GOPATH/   ← ❶ Базовый каталог или $GOPATH
        src/   ← Исходный код внешних зависимостей
            github.com/
                        Masterminds/
                                    cookoo/
                                    glide/
        bin/   ← Скомпилированные программы
            glide
        pkg/   ← Скомпилированные библиотеки
            darwin_amd64/
    github.com/
                        Masterminds/
                                              cookoo.a
```

Единственной переменной окружения, которую необходимо определить, является `$GOPATH` ❶. С ее помощью `go` определяет местоположение базового каталога рабочей области. Исходный код, включая ваш собственный и всех зависимостей, должен располагаться в каталоге `src`. Управляет этим каталогом разработчик, а инструменты Go помогают управлять кодом, хранящимся во внешних репозиториях. Двумя

другими каталогами практически всегда управляют инструменты Go. Если в каталоге проекта запустить команду `go install`, она произведет сборку выполняемых файлов и сохранит их в каталоге `bin` (как показано в листинге 1.15). В данном примере проект Glide будет скомпилирован в выполняемый файл `glide`. Для проекта Cookoo, однако, автономный выполняемый файл не будет создан. Этот проект просто содержит библиотеки, которыми смогут пользоваться другие Go-программы. Выполнение команды `go install` в этом проекте приведет к созданию в каталоге `pkg` архивного файла с расширением `.a`.

1.4.4. Работа с переменными среды

Выполняемому файлу `go` требуется только одна переменная окружения – `GOPATH`. Она должна хранить путь к каталогу рабочей области, куда будут импортироваться пакеты, сохраняться выполняемые файлы и промежуточные архивы. Следующий пример демонстрирует создание рабочей области в каталоге `go` в UNIX-подобной системе:

```
$ mkdir $HOME/go
$ export GOPATH=$HOME/go
```

Внутри каталога, на который указывает переменная среды `GOPATH`, программа `go` создаст каталог `bin` для размещения выполняемых файлов. Для большего удобства желательно добавить этот каталог в переменную среды `PATH`. Например, в UNIX-подобных системах это делается так:

```
$ export PATH=$PATH:$GOPATH/bin
```

Если вы пожелаете установить двоичные файлы в альтернативный каталог, определите путь к нему в переменной окружения `GOBIN`. Определять эту переменную необязательно.

1.5. Приложение Hello Go

В следующем листинге представлена очередная вариация стандартной программы «Hello World», которая выводит через веб-сервер фразу `Hello, my name is Inigo Montoya`.

Листинг 1.16 ❖ Вывод Hello World через веб-сервер: `inigo.go`

```
package main  ← ❶ Приложение использует пакет main

import (              ❷ Импорт необходимых пакетов
    "fmt"
    "net/http"
)
```

```
func hello(res http.ResponseWriter, req *http.Request) {
    fmt.Fprint(res, "Hello, my name is Inigo Montoya")
}
```

Обработка HTTP-запроса

```
func main() {
    http.HandleFunc("/", hello)
    http.ListenAndServe("localhost:4000", nil)
}
```

❷ Основная логика приложения

Это простое приложение состоит из трех частей. Оно начинается с объявления пакета ❶. В отличие от библиотек, краткие имена которых описывают их назначение, например `net` или `crypto`, данное приложение называется `main`. Для вывода строк и работы в режиме веб-сервера импортируются пакеты `fmt` и `http` ❷. Импорт пакетов обеспечивает их доступность в коде и в скомпилированном приложении.

Выполнение приложения начинается с вызова функции `main` ❸, не имеющей аргументов и не возвращающей значений. В первой ее строке вызывается функция `http.HandleFunc`, сообщающая веб-серверу, что при обращении к пути / должна вызываться функция `hello`. Функция `hello` реализует типичный интерфейс обработчиков. Она получает объекты с HTTP-запросом и HTTP-ответом. Затем следует вызов метода `http.ListenAndServe`, который запускает веб-сервер, принимающий запросы на порту 4000 домена `localhost`.

Запустить это приложение можно двумя способами. В следующем листинге используется команда `go run`, компилирующая приложение в каталог `temp` и запускающая его.

Листинг 1.17 ❖ Запуск на выполнение файла `inigo.go`

```
$ go run inigo.go
```

Временный файл автоматически удаляется после завершения приложения. Это удобно при разработке постоянно тестируемых новых версий приложений.

После запуска приложения можно открыть веб-браузер и перейти по адресу: http://localhost:4000, чтобы посмотреть ответ, как показано на рис. 1.7.

Приложение можно также собрать и запустить другим способом, как показано в следующем листинге.

Листинг 1.18 ❖ Сборка `inigo.go`

```
$ go build inigo.go
$ ./inigo
```

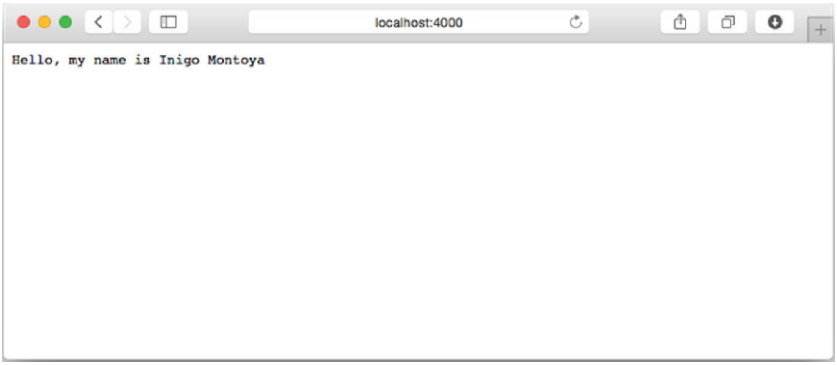


Рис. 1.7 ❖ *Просмотр вывода «Hello, my name is Inigo Montoya» в веб-браузере*

Здесь сначала выполняется сборка приложения. Если вызвать команду `go build` без имен файлов, она соберет исходный код в текущем каталоге. Если указать одно или несколько имен файлов, она скомпилирует только указанные файлы. Затем собранное приложение можно запустить.

1.6. Итоги

Язык Go предназначен для работы на современном аппаратном обеспечении и разработки современных приложений. Он использует последние технологические достижения и инструменты, упрощающие процесс разработки. В этой главе мы рассмотрели основные достоинства языка программирования Go и познакомились:

- с философией языка Go, заключающейся в простоте и расширяемости, позволившей создать удобный язык и окружающую его экосистему;
- с особенностями языка Go, помогающими использовать преимущества современного оборудования, такими как сопрограммы, обеспечивающие параллельное выполнение;
- с сопровождающим язык Go набором инструментов, включающих средства тестирования, управления пакетами, форматирования и документирования;
- с характеристиками Go в сравнении с такими языками, как C, Java, JavaScript, Python, PHP и другими, а также узнали, для каких задач он подходит лучше всего;
- с порядком установки и использования Go.

Глава 2 начинается с рассмотрения основ, необходимых для создания самых разных приложений, от консольных утилит до веб-служб. Как приложения должны обрабатывать команды, аргументы и флаги? Как правильно завершать работу веб-служб? Ответы на подобные вопросы закладывают основу для создания полноценных приложений.

Далее в книге рассматриваются практические аспекты создания и выполнения приложений, написанных на языке Go. Главы будут дополнять друг друга, пока не будет достигнут кульминационный момент, когда, объединив все рассмотренные приемы, мы создадим законченное приложение.

Глава 2

Надежная основа

В этой главе рассматриваются следующие темы:

- *использование флагов, параметров и аргументов командной строки;*
- *передача настроек в приложение;*
- *запуск и остановка веб-сервера;*
- *управление маршрутизацией для служб веб-серверов.*

Создание основы приложений – не менее важный шаг, чем любой другой. Завершение работы приложения, позволяющее не потерять данных и не вызвать трудностей в работе с ним, является примером, для чего предназначена надежная основа.

Эта глава охватывает четыре основных вопроса. Она начнется с рассмотрения консольных приложений, известных также как *CLI-приложения*. Здесь вы познакомитесь со способами обработки параметров командной строки, иногда называемых *флагами* или *опциями*, принятыми в современных приложениях для Linux и других POSIX-совместимых системах. В процессе обсуждения вы исследуете настройки, позволяющие сосредоточиться на разработке кода консольных приложений, а не на особенностях его организации.

Затем будет рассмотрено несколько способов передачи конфигурационной информации в приложения, включая популярные форматы файлов и данных, используемые для хранения конфигурационной информации.

Далее последуют серверы и практические приемы их запуска и остановки. Эти действия могут показаться элементарными, но разрешение определенных ситуаций на ранних этапах (как, например, остановка сервера до завершения сохранения данных) поможет уменьшить количество будущих проблем.

Глава заканчивается обсуждением способов сопоставления путей на серверах. Сопоставление путей с URL-адресами – типичная за-

дача веб-сайтов и серверов, решаемая средствами прикладного программного интерфейса передачи репрезентативных состояний (Representational State Transfer, REST). Здесь вы узнаете, как реализовать несколько общепринятых методов сопоставления путей.

К концу этой главы вы познакомитесь с практическими приемами создания клиентов командной строки и веб-серверов, что послужит основой для надежных Go-приложений.

2.1. CLI-приложения на Go

При использовании консольного приложения (такого как система управления версиями Git, приложение для работы с базой данных MySQL или веб-сервер Apache) аргументы и флаги командной строки являются частью его пользовательского интерфейса. Стандартная библиотека Go содержит встроенные средства для их обработки.

Оконные приложения

Go, как системный язык, не имеет встроенной поддержки для создания оконных приложений, подобных тем, что используются в Microsoft Windows или Mac OS X. В Go-сообществе велись эксперименты по их разработке, но они не имели четкой направленности.

2.1.1. Флаги командной строки

Стандартная библиотека Go включает средства обработки аргументов и флагов, реализованные в стиле операционной системы Plan 9, который отличается от стиля, принятого в системах, основанных на GNU/Linux и Berkeley Software Distribution (BSD), таких как Mac OS X и FreeBSD. Например, в Linux- и BSD-системах можно использовать команду `ls -la` для вывода списка всех файлов в каталоге. Часть `-la` команды содержит два флага, или параметра. Флаг `l` указывает команде `ls` использовать подробную форму вывода, а флаг `a` определяет необходимость включения в список скрытых файлов. Система обработки флагов в языке Go не позволяет объединять несколько флагов и рассматривает такие флаги как один флаг с именем `la`.

Команды в стиле GNU (например, команда `ls` в Linux) поддерживают длинные параметры (например, `--color`), содержащие два дефиса, чтобы указать, что строка `color` – это не пять, а один параметр.

В сети ведутся жаркие дебаты по вопросу преимуществ системы Go (или Plan 9) перед стилем, доминирующим в GNU, Linux и BSD. Честно говоря, здесь все определяется личными предпочтениями. Двойные дефисы в GNU-стиле выглядят достаточно неуклюже и часто приводят к ошибкам. Длинные опции с одним дефисом, принятые в языке Go, исключают возможность (которая только что упоминалась) группировки односимвольных параметров.

Операционная система Plan 9

Операционная система Plan 9, разработанная в Bell Labs как преемница UNIX, является операционной системой с открытым исходным кодом и разрабатывалась в период между 1980 и 2002 годом. Двое из создателей языка Go – Кен Томпсон (Ken Thompson) и Роб Пайк (Rob Pike) – входили в начальную команду и сыграли заметную роль в разработке Plan 9. Это и привело к появлению сходных черт в двух проектах.

Встроенная система обработки флагов не различает короткие и длинные флаги. В ней флаг может быть как коротким, так и длинным, и каждый флаг должен отделяться от других. Например, если выполнить команду `go help build`, она выведет список флагов, таких как `-v`, `-race`, `-x` и `-work`. Каждому параметру соответствует только один вариант написания, а не длинное или короткое его имя.

Для иллюстрации обработки флагов по умолчанию в следующем листинге демонстрируется консольное приложение, использующее пакет `flag`.

Листинг 2.1 ❖ CLI-приложение Hello World, использующее пакет `flag`

```
$ flag_cli
Hello World!
$ flag_cli -s -name Buttercup
Hola Buttercup!
$ flag_cli --spanish -name Buttercup
Hola Buttercup!
```

Все флаги отделены друг от друга пробелами и начинаются с одного или с двух дефисов (- или --), которые считаются взаимозаменяемыми. Существующий метод позволяет определять флаги с короткими или длинными именами, но, как показано в следующем листинге, этот метод действует неявно.

Листинг 2.2 ❖ Исходный код приложения Hello World, использующего пакет `flag`: `flag_cli.go`

```
package main

import (
    "flag"   ← Импорт стандартного пакета flag
    "fmt"
)

var name = flag.String("name", "World", "A name to say hello to.") ←

var spanish bool  ← ❷ Новая переменная для хранения значения флага

func init() {
    flag.BoolVar(&spanish, "spanish", false, "Use Spanish language.")
    flag.BoolVar(&spanish, "s", false, "Use Spanish language.")
}

func main() {
    flag.Parse()  ← ❹ Парсинг флагов и установка соответствующих значений в переменных
    if spanish == true {
        fmt.Printf("Hola %s!\n", *name) ← ❺ Доступ к name как к указателю
    } else {
        fmt.Printf("Hello %s!\n", *name)
    }
 }
```

❶ Создание новой переменной из флага

❸ Присваивание значения флага переменной

Здесь показаны два способа определения флагов. Первый позволяет создать переменную на основе флага. В данном примере для этого используется метод `flag.String()` ❶. Методу `flag.String` передаются: имя флага, значение по умолчанию и описание аргументов. В переменную `name` записывается адрес значения флага. Для доступа к этому значению необходимо использовать переменную `name` как указатель ❺.

Второй способ обработки флагов позволяет определить длинное и короткое имена флага. Сначала создается обычная переменная ❷ того же типа, что и значение флага. Она будет использоваться для хранения значения флага. Затем вызывается одна из функций в пакете `flag`, присваивающая значение флага существующей переменной ❸. В данном случае метод `flag.BoolVar` вызывается дважды, один раз для длинного имени флага и один раз для короткого.

> **СОВЕТ** Для флагов каждого типа существует своя функция. Ознакомиться с ними можно в описании пакета `flag` (http://golang.org/pkg/flag).

И наконец, чтобы присвоить значения флагов переменным, необходимо вызвать метод `flag.Parse()` ❹.

> **СОВЕТ** Аргументы командной строки не регистрируются пакетом `flag`, но к ним можно получить доступ с помощь функций `Args` или `Arg` из пакета `flag`.

Пакет `flag` не создает текст справки с описанием флагов, но помогает в его создании. Для этого в пакете имеются две удобные функции. Функция `PrintDefaults` генерирует текст с описанием флагов. Например, строка с описанием параметра `name` ❶ будет содержать текст:

```
-name string
    A name to say hello to. (default "World")
```

Такая аккуратность языка Go помогает информировать пользователя об особенностях использования программы.

Кроме того, в пакете `flag` имеется функция `VisitAll`, принимающая функцию обратного вызова в качестве аргумента. Она выполняет обход всех флагов и для каждого вызывает указанную функцию, что позволяет на ходу изменить текст с описанием. Например, следующий фрагмент выведет текст: `-name: A name to say hello to. (Default: 'World')`.

Листинг 2.3 ❖ Установка нестандартного текста справки для флага

```
flag.VisitAll(func(flag *flag.Flag) {
    format := "\t-%s: %s (Default: '%s')\n"
    fmt.Printf(format, flag.Name, flag.Usage, flag.DefValue)
})
```

РЕЦЕПТ 1 Аргументы командной строки в стиле GNU/UNIX

Встроенный пакет `flag` несомненно полезен и обеспечивает самые необходимые возможности, но он не поддерживает предоставления флагов, которое ожидает найти большинство пользователей. Разница в стилях взаимодействия с пользователями в Plan 9 и Linux/BSD достаточно значительна, чтобы вызвать непонимание. В основном это связано с появлением проблем при попытке совместить короткие флаги, как в команде `ls -la`.

Проблема

Те, кто работает в системах, отличных от Windows, обычно используют один из вариантов UNIX и ожидают обработки флагов в стиле UNIX. Как написать на языке Go инструмент командной строки, соот-

ветствующий ожиданиям пользователей? Идеальным решением станет написание одноразовой специализированной обработки флагов.

Решение

Эту часто встречающуюся проблему можно решить несколькими способами. Некоторые приложения, такие как система управления контейнерами программного обеспечения Docker, имеют субпакеты, содержащие код для обработки флагов в стиле Linux. В некоторых случаях применяются разновидности пакета `flag` языка Go с дополнительными возможностями. Но реализация извлечения аргументов, отдельная для каждого проекта, требует достаточно большого дублирования усилий для многократного решения одной и той же проблемы.

Лучше всего для этих целей использовать одну из существующих библиотек. Можно импортировать в приложение несколько автономных пакетов. Многие из них основаны на пакете `flag` из стандартной библиотеки и имеют совместимые или аналогичные интерфейсы. Импортировать и использовать один из этих пакетов гораздо проще, чем вносить изменения в пакет `flag` и обслуживать его в дальнейшем.

Обсуждение

В следующих примерах вы познакомитесь с двумя такими пакетами, реализующими немного отличающиеся подходы. Первый из них является попыткой сохранить совместимость с интерфейсом пакета `flag`, в то время как второй отказывается от нее.

Пакет GNUFLAG

Пакет `launchpad.net/gnuflag`, похожий на `flag`, реализует обработку флагов в стиле GNU (Linux). Но не позволяйте ввести себя в заблуждение. Его лицензия отличается от лицензии GPL, на основе которой обычно распространяются библиотеки и программы, основанные на GNU, и больше напоминает лицензию в стиле BSD или лицензию Go.

Пакет `gnuflag` поддерживает несколько видов флагов, включая следующие:

- `-f` – односимвольные, или короткие, флаги;
- `-fg` – группы односимвольных флагов;
- `--flag` – многосимвольные, или длинные, флаги;
- `--flag x` – длинные флаги с аргументами;
- `-f x` или `-fx` – короткие влаги с аргументами.

Пакет `gnuflag` имеет практически тот же интерфейс, что и пакет `flag`, с одним заметным отличием. Его функция `Parse` имеет дополнительный аргумент. В остальном их интерфейсы полностью идентичны.

Пакет flag обрабатывает флаги между именем команды и первым аргументом, не являющимся флагом. Например, фрагмент в листинге 2.4 извлекает флаги -s и -n, последний из которых имеет аргумент. Когда пакет flag обнаруживает аргумент foo, он прекращает дальнейший парсинг командной строки. Это означает, что флаг -bar не будет обработан. Пакет gnuflag позволяет продолжить анализ и извлечь флаги, следующие за аргументом.

Листинг 2.4 ❖ Извлечение флагов пакетом flag

```
$ flag_based_cli -s -n Buttercup foo -bar
```

Чтобы перейти на использование пакета gnuflag, достаточно изменить в приложении две вещи. Во-первых, изменить инструкцию импорта. И во-вторых, добавить в функцию Parse дополнительный первый аргумент со значением true или false. Если передать значение true, Parse выполнит поиск флагов по всей командной строке, а если передать значение false, пакет gnuflag будет действовать в точности, как пакет flag. Это все.

> **ПРИМЕЧАНИЕ** Сайт lanuchpad.net использует систему управления версиями Bazaar. Вам придется установить ее, чтобы Go смог извлечь пакет. Более подробную информацию можно найти на странице: http://bazaar.canonical.com/.

Пакет GO-FLAGS

Некоторые пакеты обработки флагов, разработанные в сообществе, нарушают соглашения, принятые в пакете flag из стандартной библиотеки. Одним из таких пакетов является пакет github.com/jessevdk/go-flags. Он поддерживает обработку флагов в стиле Linux и BSD, обеспечивая еще больше возможностей, чем в пакете gnuflag, но использует совершенно другой интерфейс и стиль. Он поддерживает следующие возможности:

- короткие флаги, такие как -f, и группы коротких флагов, такие как -fg;
- многосимвольные, или длинные, флаги, такие как --flag;
- группы флагов;
- автоматическое создание справочного описания в удобной форме;
- поддержка значений флагов в нескольких форматах, например: -p/usr/local, -p /usr/local и -p=/usr/local;
- один и тот же параметр может извлекаться многократно и разбиваться на фрагменты.

Следующий листинг демонстрирует использование этого пакета. Он содержит код консольного приложения «Hello World», адаптированный под использование пакета `go-flags`.

Листинг 2.5 ❖ Использование пакета `go-flags`

```
package main import (

    "fmt"
    flags "github.com/jessevdk/go-flags"  ← Импорт пакета go-flags с присвоением
)                                           псевдонима flags

var opts struct {                                          ❶ Структура
    Name string `short:"n" long:"name" default:"World"      с определениями
    ➥description:"A name to say hello to."`                 флагов
    Spanish bool `short:"s" long:"spanish"
    ➥description:"Use Spanish Language"`
}

func main() {
    flags.Parse(&opts)  ← ❷ Извлечение значений флагов в структуру

    if opts.Spanish == true {                              ❸ Использование значений флагов
        fmt.Printf("Hola %s!\n", opts.Name)                  через свойства структуры
    } else {
        fmt.Printf("Hello %s!\n", opts.Name)
    }
}
```

Первое существенное отличие от предыдущих методов заключается в определении флагов. В данном случае флаги определяются как свойства структуры ❶. Имена свойств обеспечивают доступ к значениям флагов. Сведения о флаге, такие как краткое или длинное имя, значение по умолчанию и описание, определяются парами ключ/значение, следующими за именем свойства структуры. Парсинг этих пар реализован с применением пакета `reflect` из стандартной библиотеки. Например, свойству `opts.Name` соответствуют: флаг с коротким именем `-n`, и длинным именем `--name`, значение по умолчанию `World` и текст описания `"A name to say hello to."`.

Чтобы сделать значения доступными через структуру `opts`, необходимо вызвать функцию `Parse` и передать ей структуру `opts` ❷. После этого значения флагов (или их значения по умолчанию) будут доступны через свойства структуры ❸.

2.1.2. Фреймворки командной строки

Обработка флагов – не единственный важный аспект создания приложений командной строки. Go-разработчики конечно же знают, как реализовать функцию `main` и писать простые программы, но при этом им часто приходится снова и снова писать один и тот же набор возможностей. К счастью, существуют инструменты, предоставляющие удобную основу для создания программ командной строки, и в этом разделе мы познакомимся с одним из них.

РЕЦЕПТ 2 Как избежать написания повторяющегося кода в CLI-приложениях

Представьте, что вы получили задание написать быстрый инструмент командной строки для несложной обработки. Вы начинаете создавать его с нуля, добавляя уже существующий стереотипный код. И он работает. К сожалению, он работает настолько хорошо, что возникает желание написать более продвинутую версию, принимающую новые параметры. Но, чтобы добавить новые возможности, требуется модернизировать существующий стереотипный код и включить в него поддержку новых требований. Затем цикл повторяется.

Проблема

Приходится неоднократно писать однотипные программы командной строки. Сначала они задумываются как одноразовые, но некоторые из них в конечном итоге разрастаются далеко за пределы ожидаемых изначально возможностей. Зачем повторять один и тот же стереотипный код с последующей реорганизацией верхнего уровня программы? Существуют ли инструменты, реализующие простой шаблон многократного применения для быстрого создания CLI-программ?

Решение

Если вам требуется удобный и полный набор возможностей для создания консольного приложения, обратите внимание на фреймворки, которые в дополнение к обработке флагов обеспечивают маршрутизацию команд, вывод справочного текста, поддержку подкоманд и функцию автозавершения в командной строке. Одним из самых популярных фреймворков для создания консольных приложений является `cli` (https://github.com/urfave/cli). Он используется в таких проектах с открытым исходным кодом, как Cloud Foundry, система управления контейнерами Docker, система непрерывной интеграции и развертывания Drone.

Обсуждение

Сочетающий в себе поддержку маршрутизации, парсинг флагов, извлечение параметров и возможность подготовки справочной документации, фреймворк `cli.go` обеспечивает простейший способ разработки консольных приложений.

Простое консольное приложение

Перед знакомством с консольным приложением, поддерживающим несколько команд, имеет смысл остановится на консольном приложении, осуществляющем одно действие. Вы сможете использовать его как фундамент для расширения возможностей и поддержки нескольких команд и подкоманд. На рис. 2.1 изображена структура приложения командной строки с одним действием.

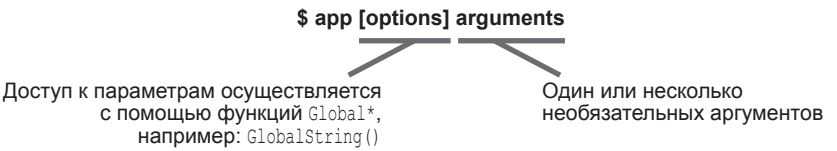


Рис. 2.1 ❖ *Структура запуска простого приложения*

В следующем листинге приводится сеанс работы с простым консольным приложением, которое выводит `Hello World!` или `Hello` с именем, указанным в командной строке.

Листинг 2.6 ❖ CLI-приложение Hello World: `hello_cli.go`

```
$ hello_cli
Hello World!
$ hello_cli --name Inigo
Hello Inigo!
$ hello_cli -n Inigo
Hello Inigo!
$ hello_cli -help
NAME:
   hello_cli - Print hello world

USAGE:
   hello_cli [global options] command [command options] [arguments...]

VERSION:
   0.0.0

COMMANDS:
```

```
   help, h Shows a list of commands or help for one command

GLOBAL OPTIONS:
   --name, -n 'World'     Who to say hello to.
   --help, -h             show help
   --version, -v          print the version
```

При использовании фреймворка `cli.go` это приложение легко можно уместить в 26 строк, как показано в следующем листинге. Остальную часть работы и поддержку пользовательского интерфейса берет на себя фреймворк `cli.go`.

Листинг 2.7 ❖ CLI-приложение Hello World: `hello_cli.go`

```
package main

import (
    "fmt"
    "os"

    "gopkg.in/urfave/cli.v1"  ← Подключение пакета cli.go
 )

func main() {
    app := cli.NewApp()                          ❶ Создание нового экземпляра приложения
    app.Name = "hello_cli"
    app.Usage = "Print hello world"
    app.Flags = []cli.Flag{                      ❷ Настройка флагов
           cli.StringFlag{
                  Name: "name, n",
                  Value: "World",
                  Usage: "Who to say hello to.",
           },
    }
    app.Action = func(c *cli.Context) error {    ❸ Определение выполняемого действия
          name := c.GlobalString("name")
          fmt.Printf("Hello %s!\n", name)
          return nil
    }
    app.Run(os.Args)  ← ❹ Запуск приложения
}
```

После импорта пакета `cli.go`, вызовом метода `cli.NewApp` создается новый экземпляр приложения. Возвращаемая при этом переменная является основой приложения. Настройка свойств `Name` и `Usage` необходима для хранения справочной информации ❶.

> **СОВЕТ** Если не определить значение свойства `Name`, фреймворк `cli.go` присвоит ему имя приложения. Такое поведение по умолчанию может пригодиться, если позднее приложение может быть переименовано.

Свойство `Flags` является частью свойства `cli.Flag`, содержащего глобальные флаги ❷. Каждый флаг может иметь длинное имя, короткое имя или оба сразу. Некоторые флаги, такие как `cli.StringFlag`, имеют свойство `Value` для значения по умолчанию. Флаг `cli.BoolFlag` является примером флага, не имеющего значения по умолчанию. При его наличии в командной строке считается, что он имеет значение `true`, а в отсутствие – значение `false`. Свойство `Usage` используется для хранения справочной информации, которая выводится по запросу.

Фреймворк `cli.go` позволяет определить действие по умолчанию, выполняемое при запуске приложения, команды и подкоманды. Команды и подкоманды будут рассмотрены в следующем разделе. Свойство `Action` экземпляра приложения определяет действие по умолчанию ❸. Ему присваивается функция, получающая `*cli.Context` в качестве аргумента и возвращающая ошибку. Эта функция содержит код, выполняемый приложением. В данном случае она извлекает имя для вставки в приветствие, вызывая метод `c.GlobalString("name")`. Если потребуется вернуть ошибку, чтобы завершить приложения с ненулевым кодом завершения, функция должна будет вернуть объект `cli.ExitError`, который можно создать вызовом `cli.NewExitError`.

Последний шаг – запуск только что созданного приложения. При этом аргументы приложения, хранящиеся в `os.Args`, передаются методу `Run` приложения ❹.

> **СОВЕТ** В теле функции `Action` также имеется доступ к аргументам, список которых можно получить вызовом `c.Args`. Например, первый аргумент доступен как `c.Args()[0]`.

Команды и подкоманды

Приложения, использующие фреймворк `cli.go`, часто поддерживают команды и подкоманды, как показано на рис. 2.2. Вспомните приложение Git, с помощью которого можно выполнять такие команды, как `git add`, `git commit` и `git push`.

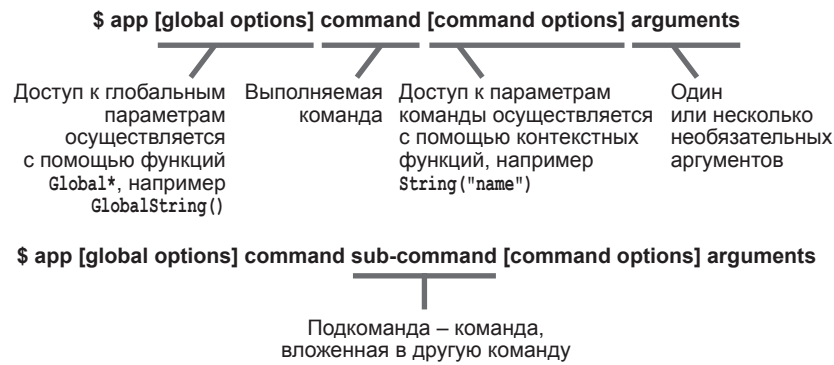


Рис. 2.2 ❖ *Структура командной строки для приложения с командами и подкомандами*

Для демонстрации использования команд в следующем листинге представлен код простого приложения, поддерживающего команды прямого и обратного отсчета.

Листинг 2.8 ❖ Прямой и обратный отсчет: `count_cli.go`

```
package main

import (
     "fmt"
     "os"
     " gopkg.in/urfave/cli.v1"
)

func main() {
     app := cli.NewApp()
     app.Usage = "Count up or down."
     app.Commands = []cli.Command{   ← ❶ Определение одной или нескольких команд
           {
                 Name: "up", ShortName: "u",
                 Usage: "Count Up",
                 Flags: []cli.Flag{
                        cli.IntFlag{
                              Name: "stop, s",
                              Usage: "Value to count up to",
                              Value: 10,
                        },
                 },
```

Определение ❷ параметров команды

```
                Action: func(c *cli.Context) error {
                        start := c.Int("stop")   ← ❸ Получение параметра
                        if start <= 0 {                команды
                                fmt.Println("Stop cannot be negative.")
                        }
                        for i := 1; i <= start; i++ {
                                fmt.Println(i)
                        }
                        return nil
                },
        },
        {
                Name: "down", ShortName: "d",
                Usage: "Count Down",
                Flags: []cli.Flag{
                        cli.IntFlag{
                                Name: "start, s",
                                Usage: "Start counting down from",
                                Value: 10,
                        },
                },
                Action: func(c *cli.Context) error {
                        start := c.Int("start")
                        if start < 0 {
                                fmt.Println("Start cannot be negative.")
                        }
                        for i := start; i >= 0; i-- {
                                fmt.Println(i)
                        }
                        return nil
                },
        },
    }

    app.Run(os.Args)
}
```

В этом приложении вместо свойства `Action` определяется свойство `Commands`, содержащее подробные сведения о каждой из команд ❶. Свойству `Action` приложения фреймворк присвоит функцию по умолчанию, которая просто выводит на экран справочную информацию. То есть если запустить приложение без любой из поддерживаемых команд, оно выведет справку. Каждая команда имеет собственные свойства `Name` и `Action`, а также необязательные свойства

`Flags`, `Usage` и `ShortName`. Подобно тому, как основное приложение может иметь свойство `Commands` со списком команд, каждая команда также может иметь свойство `Commands` с перечнем подкоманд этой команды.

Свойство `Flags` команды действует подобно одноименному свойству приложения. Свойства `Name` и `Usage` определяются подобно соответствующим свойствам приложения, как и свойство `Value` в некоторых случаях. Глобальные параметры, как показано на рис. 2.2, доступны всем командам и определяются до команд. Параметры каждой конкретной команды определяются внутри этой команды ❷. Кроме того, они извлекаются из `cli.Context` с помощью таких функций, как `String`, `Int` или `Bool` ❸. Они напоминают функции извлечения глобальных флагов, отличаясь лишь отсутствием префикса `Global`.

Этот раздел начался с демонстрации отличий Go-программ командной строки от обычных консольных программ, распространенных в UNIX-системах. Сначала мы познакомились с флагами командной строки. Затем с библиотеками для обработки флагов в стиле UNIX. Потом обсудили проблему более высокого уровня: как быстро создавать приложения командной строки, без необходимости повторно писать один и тот же стереотипный код, и познакомились с великолепной библиотекой, помогающей решить ее. Теперь можно перейти к теме, затрагивающей практически все программы, – к теме конфигурирования.

2.2. Обработка конфигурационной информации

Второй основной областью, дополняющей обработку флагов, является сохранение настроек приложения. Примерами могут служить файлы в каталоге `etc`, в некоторых дистрибутивах Linux, и пользовательские конфигурационные файлы, такие как `.gitconfig` или `.bashrc`.

Передача конфигурационной информации в приложение является насущной необходимостью. Практически все приложения, включая консольные и серверы, используют возможность сохранения настроек. Инструменты управления конфигурациями, такие как Ansible, Chef и Puppet, обладают довольно обширными возможностями управления настройками на уровне дистрибутива. Но как передавать и использовать конфигурационную информацию в приложениях?

Ansible, Chef и Puppet

Ansible, Chef и Puppet – популярные инструменты управления компьютерами и их конфигурациями. Они обычно используются для управления программным обеспечением кластеров и настройками работающих в них приложений. Например, возьмем приложение, выполняющее подключение к базе данных. База данных может находиться на том же сервере, на другом сервере или в кластере серверов. С помощью инструментов, перечисленных выше, можно управлять установкой и настройками этого приложения, определяя правильные параметры подключения к базе данных.

В предыдущем разделе речь шла о передаче настроек через параметры командной строки, а здесь рассматривается их передача через конфигурационные файлы или, в случае 12-факторых приложений[1], с помощью переменных окружения. В этом разделе мы добавим поддержку популярных форматов хранения конфигурационной информации, включая JSON, YAML и INI. Затем обсудим подход передачи настроек через переменные окружения, соответствующий шаблону 12 факторов.

РЕЦЕПТ 3 Использование конфигурационных файлов

Аргументы командной строки, как было показано в предыдущем разделе, хорошо подходят для многих случаев. Но они не годятся для однократной настройки программы, которая должна работать в конкретном окружении. Наиболее распространенным решением в этом случае является хранение конфигурационных данных в файле, загружаемом программой в момент запуска.

В следующих нескольких разделах рассматриваются три самых популярных формата конфигурационных файлов и порядок работы с ними на языке Go.

Проблема

Кроме нескольких аргументов командной строки, программе требуются дополнительные настройки. Для их хранения предпочтительнее использовать файл в одном из стандартных форматов.

Решение

Одним из самых популярных форматов конфигурационных файлов в настоящее время является формат JavaScript Object Notation

[1] https://12factor.net/ru/. – *Прим. ред.*

(JSON). Стандартная библиотека Go включает средства анализа и обработки данных в этом формате. И это неудивительно, потому что файлы в формате JSON используются очень широко.

Обсуждение

Рассмотрим JSON-файл `config.json` со следующим содержимым:

```
{
    "enabled": true,
    "path": "/usr/local"
}
```

В следующем листинге представлен код, выполняющий анализ этого файла.

Листинг 2.9 ❖ Анализ конфигурационного JSON-файла: `json_config.go`

```
package main

import (
    "encoding/json"
    "fmt"
    "os"
)

type configuration struct {
    Enabled bool
    Path string
}

func main() {
    file, _ := os.Open("conf.json")
    defer file.Close()

    decoder := json.NewDecoder(file)
    conf := configuration{}
    err := decoder.Decode(&conf)
    if err != nil {
        fmt.Println("Error:", err)
    }
    fmt.Println(conf.Path)
}
```

❶ Определение структуры, соответствующей значениям в JSON-файле

❷ Открытие конфигурационного файла

❸ Извлечение JSON-значений в переменные

Здесь можно выделить несколько этапов. Во-первых, необходимо определить тип или коллекцию типов, соответствующую структуре JSON-файла ❶. Имена значений и их структура должны соответствовать структуре JSON-файла. Функция `main` открывает конфигурационный файл ❷ и декодирует его содержимое в экземпляр струк-

туры `configuration` ❸. Если при этом не возникнет ошибок, значения из JSON-файла становятся доступными через переменную `conf`, и их можно будет использовать в приложении.

> **ПРИМЕЧАНИЕ** Парсинг и обработка данных в формате JSON имеет много особенностей и нюансов. В главе 6, посвященной работе с прикладным интерфейсом JSON, более подробно описываются возможности, связанные с форматом JSON.

Хранить конфигурацию в формате JSON удобно, если он вам хорошо знаком, вы пользуетесь одним из распределенных хранилищ конфигураций, таких как etcd, или хотите ограничиться только применением стандартной библиотеки Go. Файлы в формате JSON не могут содержать комментариев, которые часто используются для хранения дополнительных сведений или примеров. При использовании следующих двух технологий такие комментарии доступны.

Хранилище конфигураций etcd

Подобно Apache ZooKeeper и Doozer, etcd реализует распределенное хранилище конфигураций для совместного использования и обнаружения служб. Это высокодоступное хранилище использует алгоритм консенсуса Рафта (Raft) для управления репликациями и написано на языке Go. Более подробную информацию о нем можно найти на странице: https://github.com/coreos/etcd.

Решение

Рекурсивный акроним YAML расшифровывается как *YAML Ain't Markup Language* (YAML – не язык разметки) и обозначает удобочитаемый формат сериализации данных. Формат YAML легко читается, может содержать комментарии, и с ним достаточно просто работать. Формат YAML широко используется для хранения конфигураций приложений, и мы, как авторы, рекомендуем его. В языке Go отсутствует встроенный YAML-процессор, но существует несколько доступных сторонних библиотек, и одна из них рассматривается далее.

Обсуждение

Рассмотрим простой конфигурационный файл в формате YAML:

```
# Строка комментария
enabled: true
path: /usr/local
```

Следующий листинг демонстрирует пример извлечения и вывода конфигурационных данных из YAML-файла.

Листинг 2.10 ❖ Анализ конфигурационного YAML-файла: `yaml_config.go`

```
package main

import (
    "fmt"
    "github.com/kylelemons/go-gypsy/yaml"  ← ❶ Импорт пакета YAML сторонних
)                                              производителей

func main() {
    config, err := yaml.ReadFile("conf.yaml")  ← ❷ Чтение и анализ YAML-файла
if err != nil {
        fmt.Println(err)
    }
    fmt.Println(config.Get("path"))            ❸ Вывод значений из YAML-файла
    fmt.Println(config.GetBool("enabled"))
}
```

Для работы с YAML-файлами представленный в листинге код импортирует пакет `github.com/kylelemons/go-gypsy/yaml` ❶. Этот пакет, который мы сами используем и рекомендуем его вам, позволяет читать данные в формате YAML из строки или из файла ❷, поддерживает различные типы данных и обеспечивает доступ к конфигурационным данным в формате YAML.

Функция `ReadFile` читает файл конфигурации и возвращает структуру `File`, объявленную в пакете `yaml`. Эта структура открывает доступ к данным из YAML-файла. С помощью метода `Get` можно получить строковое значение ❸. Значения других типов, например логические, можно получить с помощью таких методов, как `GetBool`. Для каждого типа данных имеется свой метод, что позволяет гарантировать надлежащую обработку и возврат значений нужных типов.

Решение

Файлы в формате INI широко используются уже несколько десятилетий. Это еще один формат, который могут использовать приложения на языке Go. Хотя разработчики языка Go не включили его поддержку в состав языка, существуют сторонние библиотеки для удовлетворения этой потребности.

Обсуждение

Рассмотрим следующий INI-файл:

```
; Строка комментария
[Section]
enabled = true
path = /usr/local # еще один комментарий
```

Фрагмент в следующем листинге извлекает и выводит данные из этого файла.

Листинг 2.11 ❖ Извлечение данных из конфигурационного INI-файла: `ini_config.go`

```
package main

import (
    "fmt"
    "gopkg.in/gcfg.v1"   ← ❶ Импорт пакета для работы с INI-файлами
)

func main() {
    config := struct {
        Section struct {
            Enabled bool
            Path string
        }
    }{}

    err := gcfg.ReadFileInto(&config, "conf.ini")
    if err != nil {
         fmt.Println("Failed to parse config file: %s", err)
    }
    fmt.Println(config.Section.Enabled)
    fmt.Println(config.Section.Path)
}
```

❷ Создание структуры для хранения конфигурационных значений

❸ Извлечение данных из INI-файла в структуру с обработкой ошибок

❹ Использование значений из INI-файла

В данном случае извлечением данных из INI-файла и сохранением их в структуре занимается сторонний пакет `gopkg.in/gcfg.v1` ❶. Этот пакет включает средства анализа INI-файлов и действует подобно стандартному пакету для работы с форматом JSON.

Перед обработкой INI-файла необходимо создать переменную для сохранения значений из файла. Так же как в случае с форматом JSON, структура этой переменной должна соответствовать структуре INI-файла. В данном случае структура хотя и похожа на структуру из примера для формата JSON, но содержит вложенную структуру для раздела ❷. В этой структуре будут храниться извлеченные конфигурационные значения.

Функция `ReadFileInto` читает файл в созданную структуру ❸. Если при этом возникает ошибка, выводится соответствующее сообщение. После этого конфигурация из INI-файла доступна для использования ❹.

Пакет `gopkg.in/gcfg.v1` содержит несколько функций для чтения тегов, строк, файлов и прочих элементов, реализующих интерфейс `io.Reader`. Дополнительные сведения о них можно найти в документации с описанием пакета.

РЕЦЕПТ 4 Конфигурационные файлы или переменные среды

Конфигурационные файлы, безусловно, представляют собой отличное транспортное средство для передачи конфигурационных данных в программы. Но некоторые новые окружения не поддерживают предположений, которые мы делаем, когда используем традиционные конфигурационные файлы. Иногда приложение не имеет доступа к файловой системе на требуемом уровне. Некоторые системы полагают, что файлы конфигурации являются частью исходного кода (и считают их статической частью выполняемого файла). Это ограничивает возможности использования конфигурационных файлов.

Четче всего эта тенденция проявляется в новых системах PaaS облачных служб. Развертывание в этих системах обычно осуществляется путем помещения пакета исходного кода на управляющий сервер (подобно сохранению в Git). И единственным способом настройки приложений на таких серверах является получение данных из переменных окружения. Рассмотрим приемы работы с такими переменными.

Проблема

Многие системы PaaS не предусматривают доступа к конфигурационным файлам. Возможности настройки в них ограничиваются доступом к таким элементам, как переменные окружения.

Решение

12-факторные приложения, развертываемые в Heroku, Cloud Foundry, Deis и других платформах PaaS или системах управления контейнерами кластеров (рассматриваются в главе 11), становятся все более обычным явлением. И одним из двенадцати факторов явля-

ется хранение конфигурационных данных в переменных окружения. Это дает возможность определять отдельные конфигурации для каждого окружения, в котором выполняется приложение.

12-факторные приложения

Это популярная и широко используемая методика создания веб-приложений и служб. Подобные приложения создаются с учетом следующих 12 факторов:

1. Использование единой базы кода с контролем версий, которая может развертываться многократно.
2. Явное объявление зависимостей и изолированность от других приложений.
3. Хранение конфигурации приложения в переменных окружения.
4. Подключение поддерживающих служб.
5. Разделение этапов сборки и запуска.
6. Выполнение приложения как одного или нескольких процессов без состояния.
7. Экспорт служб через TCP-порт.
8. Горизонтальное масштабирование путем добавления процессов.
9. Повышение живучести приложений за счет быстрого запуска и штатного завершения.
10. Версии для разработки, тестирования и эксплуатации должны быть как можно более схожими.
11. Управление журналированием как потоками событий.
12. Запуск административных задач в отдельных процессах.

Более подробные сведения об этих факторах можно найти на странице http://12factor.net.

Рассмотрим, например, переменную окружения `PORT`, содержащую номер порта, через который веб-сервер должен принимать входящие соединения. Следующий фрагмент извлекает эту часть конфигурации и использует ее в момент запуска веб-сервера.

Листинг 2.12 ❖ Конфигурация в переменных окружения: `env_config.go`

```
package main

import (
    "fmt"
    "net/http"
    "os"
)
```

```
func main() {
    http.HandleFunc("/", homePage)
    http.ListenAndServe(":"+os.Getenv("PORT"), nil)  ← ❶ Извлечение значения переменной PORT из окружения
}

func homePage(res http.ResponseWriter, req *http.Request) {
    if req.URL.Path != "/" {
           http.NotFound(res, req)
           return
    }
    fmt.Fprint(res, "The homepage.")
}
```

В этом примере используется пакет `http` из стандартной библиотеки. Если вы помните, мы использовали его в простом веб-сервере «Hello World» (листинг 1.16). В следующем разделе мы подробнее остановимся на веб-серверах.

Процесс извлечения конфигурационных данных из окружения предельно прост. Функция `Getenv` из пакета `os` извлекает значение в виде строки ❶. Если переменная окружения не найдена, возвращается пустая строка. Чтобы преобразовать строку в другой тип, можно воспользоваться пакетом `strconv`. Например, если бы номер порта в этом примере должен был быть целым числом, мы могли бы использовать функцию `ParseInt`.

> **ПРЕДУПРЕЖДЕНИЕ** Проявляйте осторожность при работе с информацией из переменных окружения и с процессами, получающими эту информацию. Например, сторонний подчиненный процесс, запущенный из приложения, также может иметь доступ к переменным окружения.

2.3. Работа с действующими веб-серверами

Стандартная библиотека языка Go дает отличный фундамент для создания веб-серверов, но, как всегда, кое-что может потребоваться изменить и кое-что добавить. В данном разделе мы рассмотрим два важных вопроса: сопоставление путей URL с функциями обратного вызова и запуск/остановку серверов с особым упором на остановку в штатном режиме.

Создание веб-серверов является основной задачей пакета `http`. Этот пакет используется как основа для обработки TCP-соединений

из пакета net. Поскольку веб-серверы являются основной частью стандартной библиотеки и имеют широкое применение, создание простого веб-сервера было включено уже в главу 1. Этот раздел выходит за рамки создания простых веб-серверов и охватывает некоторые практические приемы создания приложений. Дополнительные сведения о пакете http можно найти на странице http://golang.org/pkg/net/http/.

2.3.1. Запуск и завершение работы сервера

Запуск сервера в языке Go выполняется довольно просто. С помощью пакета net или http можно создать сервер, подключенный к порту TCP, и начать отвечать на входящие соединения и запросы. Но что происходит при выключении сервера? Чем чревато завершение его работы в тот момент, когда к нему подключены пользователи, или до того, как были записаны на диск все данные (например, данные пользователей)?

Команды для запуска и остановки сервера в операционной системе должны выполняться демоном инициализации. Команду go run удобно использовать во время разработки, и ее можно задействовать в некоторых системах, опирающихся на двенадцать факторов, но это не стандартный и не рекомендуемый способ. Запуск приложений вручную прост, но он не предназначен для интеграции с функциональными инструментами и не может применяться для решения таких проблем, как неожиданная перезагрузка. Для таких ситуаций специально были созданы демоны инициализации, и они хорошо с ними справляются.

Большинство систем по умолчанию имеет набор инструментов для инициализации. Например, systemd (https://freedesktop.org/wiki/Software/systemd/) часто встречается в дистрибутивах Linux, таких как Debian и Ubuntu. В сценариях для systemd можно использовать следующие команды.

Листинг 2.13 ❖ Запуск и остановка приложений с помощью **upstart**

```
$ systemctl start myapp.service   ← Запуск приложения myapp
$ systemctl stop myapp.service    ← Остановка запущенного приложения myapp
```

Существует широкий диапазон демонов инициализации. Они разные в разных операционных системах, и многие из них предназначены для конкретных версий Linux. Вам могут быть знакомы некоторые из названий, такие как upstart, init и launchd. Поскольку разные си-

стемы поддерживают разные команды и конфигурационные сценарии, мы не будем рассматривать их здесь. Эти инструменты хорошо документированы, и для их изучения существует множество пособий и примеров.

> **ПРИМЕЧАНИЕ** Мы не рекомендуем писать приложения, выполняющиеся как демоны. Используйте лучше демоны инициализации для управления запуском/остановкой вашего приложения.

Типичный антишаблон: обработчик URL-адреса

Существует простой шаблон (точнее, антишаблон), часто используемый на этапе разработки, реализующий остановку сервера при переходе по определенному URL-адресу, например `/kill` или `/shutdown`. Следующий листинг демонстрирует простую версию этого метода.

Листинг 2.14 ❖ URL-адрес обратного вызова завершения работы: `callback_shutdown.go`

```
package main

import (
    "fmt"
    "net/http"
    "os"
)

func main() {
    http.HandleFunc("/shutdown", shutdown)  ← Регистрация специального пути
    http.HandleFunc("/", homePage)              для завершения работы сервера
    http.ListenAndServe(":8080", nil)
}

func shutdown(res http.ResponseWriter, req *http.Request) {
    os.Exit(0)  ← Немедленно завершает приложение
}

func homePage(res http.ResponseWriter, req *http.Request) {
    if req.URL.Path != "/" {
        http.NotFound(res, req)
        return
    }
    fmt.Fprint(res, "The homepage.")
}
```

Мы не рекомендуем этот метод и приводим только потому, что его легко найти в Интернете. Он прост, что определенно является его преимуществом, но имеет целый ряд недостатков, включая следующие:

- при передаче в промышленную эксплуатацию URL-адрес должен быть заблокирован или, что предпочтительнее, удален. Необходимость иметь разный код для разработки и эксплуатации чревата появлением ошибок. Если оставить этот URL-адрес в коде для промышленной эксплуатации, кто угодно легко сможет отключить службу;
- при переходе по URL-адресу обработчик запроса немедленно завершает работу сервера. При этом любые действия будут сразу же немедленно остановлены. Любые несохраненные данные будут потеряны, поскольку отсутствует возможность их сохранения перед выходом;
- использование URL-адреса происходит в обход обычных функциональных инструментов, таких как Ansible, Chef и Puppet, или других инструментов инициализации. Следует отдавать предпочтение более подходящим инструментам, управляющим обновлениями и запущенными приложениями.

Мы не рекомендуем использовать этот метод. Он удобен при разработке, когда процесс выполняется в фоновом режиме или действует как демон. Go-приложения, как правило, не должны быть демонами, и существуют более совершенные методы запуска и остановки сервера, даже в процессе разработки.

РЕЦЕПТ 5 Штатное завершение с помощью пакета manners

Перед выключением сервера обычно следует прекратить получение новых запросов, сохранить данные на диск и нормально завершить открытые подключения. Пакет `http` стандартной библиотеки завершает работу немедленно и не дает возможности выполнить ни одно из этих действий. В некоторых случаях это может привести к потере или повреждению данных.

Проблема

Чтобы избежать потери данных и непредусмотренного поведения, серверу может потребоваться выполнить некоторые действия перед завершением работы.

Решение

Для решения этой задачи необходимо реализовать собственную логику или использовать сторонний пакет, например `github.com/braintree/manners`.

Обсуждение

В Braintree, подразделении компании PayPal, создан пакет `manners`, позволяющий корректно завершить работу с использованием того же интерфейса, что предоставляет функция `ListenAndServe` из стандартного пакета `http`. Внутренне пакет использует сервер из стандартного пакета `http` и контролирует соединения с помощью функции `WaitGroup` из пакета `sync`. Метод `WaitGroup` предназначен для управления сопрограммами. Следующий листинг демонстрирует реализацию простого сервера, использующего пакет `manners`.

Листинг 2.15 ❖ Штатное завершение работы с помощью пакета manners: `manners_shutdown.go`

```
package main

import (
    "fmt"
    "net/http"
    "os"
    "os/signal"
    "github.com/braintree/manners"
)

func main() {
    handler := newHandler()  ← ❶ Получение экземпляра обработчика

    ch := make(chan os.Signal)                          ❷ Настройка мониторинга сигналов
    signal.Notify(ch, os.Interrupt, os.Kill)            операционной системы
    go listenForShutdown(ch)

    manners.ListenAndServe(":8080", handler)  ← ❸ Запуск веб-сервера
}

func newHandler() *handler {
    return &handler{}
}

type handler struct{}

func (h *handler) ServeHTTP(res http.ResponseWriter, req *http.Request) {
    query := req.URL.Query()                                    Обработчик, ❹
    name := query.Get("name")                                   отвечающий
    if name == "" {                                             на веб-запросы
        name = "Inigo Montoya"
    }
    fmt.Fprint(res, "Hello, my name is ", name)
}
```

```
func listenForShutdown(ch <-chan os.Signal) {
    <-ch
    manners.Close()
}
```

❺ Ожидание сигнала завершения работы

Функция `main` начинается с получения экземпляра функции `handler`, способного реагировать на веб-запросы ❶. Этот обработчик генерирует простой ответ Hello World ❹. Ниже в этой главе на его место будет помещен более сложный обработчик, поддерживающий правила маршрутизации, анализирующий пути и обрабатывающий регулярные выражения.

Для штатного завершения работы нужно знать, когда это можно сделать. Пакет `signal` предоставляет средства для получения сигналов от операционной системы, включая сигналы прерывания или завершения работы приложения. Следующий шаг – настройка канала для получения сигналов прерывания и завершения от операционной системы, чтобы получить возможность реагировать на них ❷. Функция `ListenAndServe`, подобно одноименной функции из пакета `http`, блокирует выполнение. Для мониторинга сигналов сопрограмма должна выполняться параллельно. Функция `ListenForShutdown` ожидает получения сигнала через канал ❺ и посылает сообщение `Shutdown` серверу. Оно указывает серверу прекратить прием новых подключений и отключиться, как только будут обработаны все текущие запросы.

Вызов функции `ListenAndServe` запускает сервер, так же как вызов одноименной функции из пакета `http` ❸.

> **СОВЕТ** Перед завершением работы сервер ждет только завершения обработки запросов. Если приложение запускает сопрограммы и должно дождаться их завершения перед выходом, вам придется реализовать свою версию `WaitGroup`.

Такой подход имеет несколько преимуществ, в том числе:

- позволяет завершить текущие HTTP-запросы, не прерывая их обработки;
- останавливает прием новых запросов, освобождая порт TCP. Это открывает возможность другому приложению выполнить привязку к порту и начать обслуживание запросов. При обновлении версии приложения одна может завершить обработку всех своих запросов, а другая – перейти в оперативный режим и начать обслуживание.

Однако имеется также пара недостатков, проявляющихся при определенных условиях:

- пакет `manners` работает только с HTTP-подключениями, а не с любыми TCP-соединениями. Если приложение не является веб-сервером, пакет `manners` работать в нем не будет;
- в некоторых случаях одной версии приложения может потребоваться перед завершением передать текущие сокеты другому экземпляру того же или другого приложения. Например, если между сервером и клиентскими приложениями имеются постоянные подключения, пакет `manners` может дождаться их завершения или закрыть их принудительно, но он не в состоянии передавать сокеты.

2.3.2. Маршрутизация веб-запросов

Одной из основных задач любого HTTP-сервера являются прием запроса и выбор внутренней функции, которая должна вернуть результат клиенту. Такая маршрутизация запросов имеет большое значение, позволяя создавать веб-службы, простые в обслуживании и обладающие достаточной гибкостью, чтобы соответствовать будущим потребностям. В этом разделе представлены различные сценарии маршрутизации и решения для каждого из них.

Начнем с простых сценариев и простых решений. Но здесь следует все предусмотреть заранее. Простое решение, которое будет рассмотрено первым, отлично подходит для прямого сопоставления, но не имеет той гибкости, необходимой современным веб-приложениям.

РЕЦЕПТ 6 Выбор пути по содержимому

Веб-приложения и серверы, поддерживающие интерфейс REST, обычно выполняют разные функции для разных путей. На рис. 2.3 показан фрагмент URL-адреса, представляющий путь. Пример «Hello World», представленный в листинге 1.16 (глава 1), использует одну функцию для обработки всех возможных путей. Для простого приложения в стиле «Hello World» этого вполне достаточно. Но одна функция не может нормально обрабатывать несколько путей и не обеспечивает масштабирования в реальных приложениях. В этом разделе рассматривается несколько методов обработки отличающихся путей, а в некоторых случаях и отличающихся HTTP-методов (иногда называемых *глаголами*).

Проблема

Для маршрутизации запросов веб-сервер должен быстро и эффективно анализировать фрагмент URL-адреса, представляющий путь.

Рис. 2.3 ❖ *Фрагмент пути в URL-адресе, используемой для маршрутизации запросов*

Решение: несколько обработчиков

Данное решение расширяет прием из листинга 1.16 и использует функции-обработчики для каждого из путей. Оно было представлено в руководстве «Writing Web Applications» (http://golang.org/doc/articles/wiki/) и реализует шаблон, подходящий для веб-приложений с несколькими простыми путями. Такой подход имеет определенные ограничения (о них вы узнаете чуть ниже), которые могут вынудить вас использовать другой прием, описываемый вслед за этим.

Обсуждение

Начнем с простой программы, представленной в следующем листинге и демонстрирующей использование нескольких обработчиков.

Листинг 2.16 ❖ Несколько функций-обработчиков: `multiple_handlers.go`

```
package main

import (
    "fmt"
    "net/http"
    "strings"
)

func main() {
    http.HandleFunc("/hello", hello)
    http.HandleFunc("/goodbye/", goodbye)
    http.HandleFunc("/", homePage)
    http.ListenAndServe(":8080", nil)
}

func hello(res http.ResponseWriter, req *http.Request) {
    query := req.URL.Query()
    name := query.Get("name")
    if name == "" {
        name = "Inigo Montoya"
    }
    fmt.Fprint(res, "Hello, my name is ", name)
}
```

❶ Регистрация обработчиков URL-адресов

← Запуск веб-сервера, подключенного к порту 8080

← ❷ Функция-обработчик для пути /hello

❸ Получение имени из строки запроса

```
func goodbye(res http.ResponseWriter, req *http.Request) {
    path := req.URL.Path
    parts := strings.Split(path, "/")
    name := parts[2]
    if name == "" {
        name = "Inigo Montoya"
    }
    fmt.Fprint(res, "Goodbye ", name)
}
```

❹ Функция-обработчик для пути /goodbye/

❺ Выборка имени из строки запроса

```
func homePage(res http.ResponseWriter, req *http.Request) {
    if req.URL.Path != "/" {
        http.NotFound(res, req)
        return
    }
    fmt.Fprint(res, "The homepage.")
}
```

❻ Функция-обработчик для домашней и неопознанной страницы

❼ Проверка соответствия пути домашней или неопознанной странице

ПРИМЕЧАНИЕ Данные, полученные от конечного пользователя, следует очистить и обезопасить перед использованием. То же касается данных, возвращаемых пользователю. Такую возможность предоставляет пакет из библиотеки Go для работы с шаблонами, который рассматривается в главе 5.

Здесь используются три функции для обслуживания трех путей ❶. Анализ пути производится в направлении от более конкретных маршрутов к менее конкретным. Таким наименее конкретным считается любой путь, которому не будет найдено соответствия до сопоставления с путем /.

Следует отметить, что пути, заканчивающиеся символом /, могут вызвать проблемы при перенаправлении. Например, при попытке перейти по пути `/goodbye` пользователь автоматически будет перенаправлен на `/goodbye/`. Параметры в строке запроса при этом могут быть потеряны. Так адрес `/goodbye?foo=bar` превратится `/goodbye/`.

Важно понимать, как действует перенаправление по умолчанию. Обработчик, зарегистрированный для пути `/hello`, будет использоваться исключительно для пути `/hello`. Обработчик, зарегистрированный для пути `/goodbye/`, будет вызываться для путей: `/goodbye` (с перенаправлением), `/goodbye/`, `/goodbye/foo`, `/goodbye/foo/bar` и так далее.

Пути `/hello` соответствует функция `hello` ❷. В качестве аргументов обработчик получает `http.ResponseWriter` и `http.Request`. При необходимости имя для вставки в текст приветствия можно извлечь из

строки запроса по ключу `name` ❸. Запрошенный URL хранится в свойстве `URL` аргумента `http.Request`. Его метод `Query` возвращает соответствующее ключу значение или пустую строку, если указанный ключ отсутствует в строке запроса. Если имя не указано пользователем, по умолчанию используется строка `Inigo Montoya`.

> **СОВЕТ** В пакете `net/url`, содержащем тип `URL`, имеется множество функций для работы с URL-адресами.

> **ПРИМЕЧАНИЕ** Определить использованный HTTP-метод можно с помощью свойства `http.Request.Method`. Оно содержит имя метода (например, `GET`, `POST` и так далее).

Функция `goodbye` обрабатывает путь `/goodbye/`, включая случаи присутствия дополнительного текста в конце ❹. В данном случае путь может содержать имя для текста приветствия. Например, из пути `/goodbye/Buttercup` будет извлечено имя `Buttercup` ❺. Для этого URL-адрес разбивается с помощью пакета `strings` для выделения фрагмента пути после `/goodbye/`.

Функция `homePage` обрабатывает корневой путь / и любые другие, нераспознаваемые пути ❻. Чтобы принять решение о возврате сообщения «404 Page Not Found» или содержимого домашней страницы, проверяется свойство `http.Request.Path` ❼. В пакете `http` имеется вспомогательная функция `NotFound`, которую можно использовать для установки кода HTTP-ответа 404 и отправки текста `404 page not found`.

> **СОВЕТ** В пакете `http` имеется функция `Error`, которую можно использовать для установки кода любой ошибки HTTP и возврата соответствующего сообщения. Функция `NotFound` предназначена только для возврата ошибки 404.

Использование нескольких функций-обработчиков является основным способом обслуживания разных путей. Этот способ обладает следующими преимуществами:

- поскольку этот способ поддерживается пакетом `http`, он хорошо документирован, проверен, и имеется масса примеров, демонстрирующих его использование;
- код отображения путей в функции легко читается и анализируется.

Но, помимо преимуществ, имеются также недостатки, которые заставляют многих разработчиков, включая авторов книги, пользоваться другими методами:

- нельзя использовать разные функции для разных HTTP-методов обращения к одному и тому же пути. При создании интерфейсов REST для каждого из глаголов (например, `GET`, `POST` или `DELETE`) может потребоваться реализовать совершенно разную функциональность;
- подстановки и именованные разделы пути, а также типовые возможности систем отображения не поддерживаются;
- практически все функции-обработчики должны проверять выход путей за пределы их компетенции и возвращать сообщение о том, что страница не найдена. Например, обработчику пути `/goodbye/`, представленному в листинге 2.16, будет передан любой путь, содержащий `/goodbye`. Все результаты, возвращаемые в ответ на любые запросы по этому пути, должны производиться только в нем, поэтому если для пути `/goodbye/foo/bar/baz` должно возвращаться сообщение о том, что страница не найдена, это следует делать только в этом обработчике.

Использование нескольких обработчиков удобно в простых случаях. Такое решение не требует дополнительных пакетов, кроме `http`, поэтому внешние зависимости сведены к минимуму. Но если планируется создать более или менее сложное приложение, лучше использовать один из методов, описанных ниже.

РЕЦЕПТ 7 Обработка сложных путей с подстановками

Выше был представлен простой метод, но он не способен обрабатывать шаблоны путей. При его применении приходится перечислять все поддерживаемые пути. Для больших приложений или приложений, реализующих интерфейс REST, необходимо более гибкое решение.

Проблема

Вместо определения конкретных путей для каждого обработчика приложению может понадобиться более гибкое решение, позволяющее связывать обработчики с простыми шаблонами путей.

Решение

В стандартной библиотеке Go имеется пакет `path` для обработки путей, где символы слеша используются как разделители. Этот пакет не имеет прямого отношения к путям URL и прекрасно поддерживает любые другие пути, но его на удивление удобно использовать в паре с HTTP-обработчиком.

Обсуждение

В следующем листинге демонстрируется маршрутизатор, отображающий URL-пути и HTTP-методы в функции-обработчики.

Листинг 2.17 ❖ Анализ URL-адресов с помощью пакета path: path_handlers.go

```
package main

import (
"fmt"
    "net/http"
    "path"      ← Импорт пакета path для обработки URL-путей
    "strings"
)

func main() {
    pr := newPathResolver()  ← ❶ Получение экземпляра маршрутизатора
    pr.Add("GET /hello", hello)      ❷ Отображение путей в функции
    pr.Add("* /goodbye/*", goodbye)
    http.ListenAndServe(":8080", pr)  ← ❸ Передача маршрутизатора HTTP-серверу
}

func newPathResolver() *pathResolver {                    Создание нового
    return &pathResolver{make(map[string]http.HandlerFunc)}  инициализированного
}                                                         объекта pathResolver

type pathResolver struct {
    handlers map[string]http.HandlerFunc
}

func (p *pathResolver) Add(path string, handler http.HandlerFunc) {
    p.handlers[path] = handler                    Добавление путей
}                                                 для внутреннего поиска

func (p *pathResolver) ServeHTTP(res http.ResponseWriter, req *http.Request){
    check := req.Method + " " + req.URL.Path ← Объединение метода и пути
                                              для проверки
    for pattern, handlerFunc := range p.handlers { ← ❹ Обход зарегистрированных путей
        if ok, err := path.Match(pattern, check); ok && err == nil { ← ❺
            handlerFunc(res, req) ← ❻ Вызов функции  Проверка соответствия текущего пути
            return                   для обработки пути  одному из зарегистрированных
        } else if err != nil {
            fmt.Fprint(res, err)
        }
    }
```

```
    http.NotFound(res, req)   ← ❼ Если для пути не нашлось соответствия,
}                                 вернуть сообщение о том, что страница не найдена

func hello(res http.ResponseWriter, req *http.Request) {
    query := req.URL.Query()
    name := query.Get("name")
    if name == "" {
        name = "Inigo Montoya"
    }
    fmt.Fprint(res, "Hello, my name is ", name)
}

func goodbye(res http.ResponseWriter, req *http.Request) {
    path := req.URL.Path
    parts := strings.Split(path, "/")
    name := parts[2]
    if name == "" {
        name = "Inigo Montoya"
    }
    fmt.Fprint(res, "Goodbye ", name)
}
```

Функция `main` начинается совсем иначе – с получения экземпляра `pathResolver` ❶. Он содержит основную логику сопоставления путей. После создания экземпляра `pathResolver` в него добавляются два пути ❷ в виде строк, состоящих из имени HTTP-метода и пути, разделенных пробелом. Вместо имени HTTP-метода или части пути можно использовать звездочку (`*`) для обозначения подстановки.

Экземпляр `pathResolver` передается встроенному HTTP-серверу в качестве обработчика ❸. Чтобы `pathResolver` мог действовать как обработчик, необходимо реализовать для него метод `ServeHTTP`, определяемый интерфейсом `HandlerFunc`, выполняющий анализ путей.

Когда в сервер поступает запрос, метод `ServeHTTP` просматривает зарегистрированные пути с помощью `pathResolver` ❹. Для каждого запроса проверяется HTTP-метод и путь и при обнаружении совпадения выбирается соответствующая функция ❺. Такая проверка необходима, потому что REST-серверам часто требуются разные функции для обработки разных HTTP-методов (например, запросов `DELETE` или `GET`). Если совпадение найдено, вызывается обрабатывающая функция ❻. Если не найдено ни одного подходящего пути, по умолчанию выводится ошибка 404 Page Not Found ❼.

Обработка путей с помощью пакета `path` имеет свои плюсы и минусы. Плюсы:

- простота, обусловленная простотой сопоставления путей;
- входит в состав стандартной библиотеки, и, как следствие, пакет `path` имеет хорошую документацию и тщательно проверен.

Минусы определяются назначением пакета `path` для работы с любыми путями, а не только с адресами URL:

- возможности использования подстановок весьма ограничены. Например, пути `foo/*` будет соответствовать путь `foo/bar`, но не путь `foo/bar/baz`. При использовании `*` в качестве знака подстановки сопоставление ведется до следующего символа /. Чтобы добиться соответствия пути `foo/bar/baz`, необходимо использовать шаблон `foo/*/*`;
- поскольку пакет предназначен для обработки любых путей, а не только URL-адресов, в нем отсутствуют некоторые полезные функции. Например, в листинге 2.17 регистрируется путь `/goodbye/*`. При переходе по адресу `/goodbye` в браузере появится сообщение об отсутствующей странице, тогда как переход по адресу `/goodbye/` будет выполнен без ошибок. Хотя чисто технически эти пути разные (отличаются наличием завершающего символа /), это распространенный в Интернете случай, который обрабатывается неожиданным образом. Такие случаи требуется выявлять и обрабатывать отдельно.

Этот метод удобно использовать для простых сценариев, и авторам удавалось с успехом использовать его.

РЕЦЕПТ 8 Сопоставление URL-адресов с шаблонами

Для большинства приложений в стиле REST простого сопоставления с регулярными выражениями более чем достаточно. Но как поступить, если требуется пойти дальше и предусмотреть какую-то особенную обработку URL-адресов? Пакет `path` для этого не подойдет, поскольку поддерживает только простое сопоставление в стиле POSIX.

Проблема

Простого сопоставления недостаточно для приложения, которому необходимо обрабатывать пути как текстовые строки, а не как пути к файлам. Это особенно важно при сопоставлении без учета разделителей путей (/).

Решение

Встроенный пакет `path` поддерживает простые схемы сопоставления путей, но иногда требуется сложное сопоставление или более

полный контроль над сопоставлением. В этих случаях можно использовать регулярные выражения. Объединение встроенной поддержки регулярных выражений в Go с HTTP-обработчиком позволит создать быстрое, но в то же время гибкое, средство сопоставления URL-путей.

Обсуждение

В следующем листинге представлен код, реализующий обход путей и их анализ, основанный на регулярных выражениях.

Листинг 2.18 ❖ Анализ URL с помощью регулярных выражений: regex_handlers.go

```
package main

import (
    "fmt"
    "net/http"
    "regexp"    ← Импорт пакета для работы с регулярными выражениями
    "strings"
)

func main() {
    rr := newPathResolver()
    rr.Add("GET /hello", hello)                                    ❶ Регистрация
    rr.Add("(GET|HEAD) /goodbye(/?[A-Za-z0-9]*)?", goodbye)        путей и функций
    http.ListenAndServe(":8080", rr)
}

func newPathResolver() *regexResolver {
    return &regexResolver{
        handlers: make(map[string]http.HandlerFunc),
        cache:    make(map[string]*regexp.Regexp),
    }
}

type regexResolver struct {
    handlers map[string]http.HandlerFunc
    cache    map[string]*regexp.Regexp  ← Сохранение скомпилированных регулярных
}                                          выражений для повторного использования

func (r *regexResolver) Add(regex string, handler http.HandlerFunc) {
    r.handlers[regex] = handler
    cache, _ := regexp.Compile(regex)
    r.cache[regex] = cache
}
```

```
func (r *regexResolver) ServeHTTP(res http.ResponseWriter, req *http.Request) {
    check := req.Method + " " + req.URL.Path
    for pattern, handlerFunc := range r.handlers {
        if r.cache[pattern].MatchString(check) == true {
            handlerFunc(res, req)
            return
        }
    }
    http.NotFound(res, req)
}

func hello(res http.ResponseWriter, req *http.Request) {
    query := req.URL.Query()
    name := query.Get("name")
    if name == "" {
        name = "Inigo Montoya"
    }
    fmt.Fprint(res, "Hello, my name is ", name)
}

func goodbye(res http.ResponseWriter, req *http.Request) {
    path := req.URL.Path
    parts := strings.Split(path, "/")
    name := ""
    if len(parts) > 2 {
        name = parts[2]
    }
    if name == "" {
        name = "Inigo Montoya"
    }
    fmt.Fprint(res, "Goodbye ", name)
}
```

❷ Поиск и вызов функции-обработчика

← Если соответствие не найдено, вернуть ошибку Page Not Found

Прием сопоставления путей с применением регулярных выражений (листинг 2.18) напоминает прием из предыдущего примера (листинг 2.17). Изменились лишь формат регистрации путей и обработчиков и реализация метода `ServeHTTP`.

Пути регистрируются как регулярные выражения ❶. Их структура ничем не отличается от структуры при использовании пакета `path`, то есть за именем HTTP-метода через пробел следует путь. Шаблон `GET /hello` выглядит достаточно просто, гораздо сложнее организован шаблон `(GET|HEAD) /goodbye(/?[A-Za-z0-9]*)?`. Этот более сложный шаблон соответствует HTTP-методам `GET` или `HEAD`. Регулярное выражение совпадает с путями `/goodbye`, `/goodbye/` (с символом / в конце) и `/goodbye/`, за которым следуют буквы и цифры.

В данном случае метод `ServeHTTP` перебирает регулярные выражения для поиска совпадения ❷. Обнаружив первое совпадение, он вызовет функцию, зарегистрированную для этого регулярного выражения. Если путь соответствует нескольким регулярным выражениям, будет использовано добавленное первым, которое первым и будет проверено.

> **ПРИМЕЧАНИЕ** При добавлении регулярных выражений они компилируются и кэшируются. Поиск совпадений выполняется с помощью функции `Match` из пакета `regexp`. Эта функция сначала пытается скомпилировать регулярное выражение. Поэтому предварительная компиляция и кэширование позволяют пропустить этап повторной компиляции при обработке запросов.

Использование регулярных выражений для проверки путей открывает широкие возможности, позволяя детально настраивать сопоставление путей. Такая гибкость вместе со сложностью регулярных выражений, в которых не всегда легко разобраться, определяет необходимость тестирования, чтобы удостовериться, что заданным регулярным выражениям соответствуют правильные пути.

РЕЦЕПТ 9 Быстрая маршрутизация (без листинга)

Многие критикуют встроенный пакет `http` за то, что его механизмы маршрутизации и мультиплексирования обеспечивают лишь базовые возможности. В предыдущих разделах было описано несколько простых приемов работы с пакетом `http`, но с их помощью не всегда можно достичь нужной гибкости и производительности или обеспечить все необходимые возможности встроенного HTTP-сервера. Кроме того, у вас может появиться желание избежать написания шаблонного кода маршрутизации.

Проблема

Встроенный пакет `http` недостаточно гибок, и его применение оправдано далеко не всегда.

Решение

Выбор функций-обработчиков по URL-адресам – типичная задача веб-приложений. Поэтому было создано множество пакетов маршрутизации. Многие просто импортируют существующий маршрутизатор запросов и используют его в приложении.

Ниже приводится список наиболее популярных решений:

- `github.com/julienschmidt/httprouter` – быстрый пакет маршрутизации с акцентом на использование минимального объема

памяти и как можно меньшего времени на обработку маршрутов. Поддерживает сопоставление путей без учета регистра символов, исключение фрагментов /../ из путей и обрабатывает необязательные завершающие символы /;

- `github.com/gorilla/mux` – часть комплекта веб-инструментов Gorilla. Это бесплатная коллекция пакетов компонентов, которые можно использовать в приложении. Пакет `mux` реализует универсальный набор критериев для сопоставления, включая хост, схемы, HTTP-заголовки и многое другое;
- `github.com/bmizerany/pat` – маршрутизатор, действующий подобно механизму маршрутизации из библиотеки Sinatra. Зарегистрированные пути удобно просматривать, и они могут содержать именованные параметры, например `/user/:name`. Имеются и другие пакеты с такими же возможностями, такие как `github.com/gorilla/pat`.

Библиотека для веб-приложений Sinatra

Sinatra – это фреймворк с открытым исходным кодом для разработки веб-приложений, написанный на языке Ruby. Используется множеством организаций и послужил прообразом для более чем 50 аналогичных фреймворков на разных языках, в том числе Go.

Каждый пакет имеет разный набор функциональных возможностей и свой программный интерфейс. Кроме того, существует масса других пакетов маршрутизации. Потратив немного времени на исследования, вы без труда найдете пакет, отвечающий вашим потребностям.

2.4. Итоги

После реализации фундаментальных элементов можно переходить к более специфичной части приложения, которая, по сути, и делает его полезным. В этой главе мы познакомились с несколькими базовыми элементами:

- удобная обработка параметров командной строки. Здесь был охвачен диапазон от легковесных решений до простых фреймворков для создания консольных приложений и утилит;
- различные способы извлечения конфигурационной информации в разных форматах из файлов и переменных окружения;

- запуск и остановка веб-сервера с помощью специализированных инструментов, обеспечивающих штатное его завершение и позволяющих избежать недоразумений с пользователями и потерь данных;
- несколько способов выбора функций для обработки URL-путей.

Следующая глава посвящена механизму параллельного выполнения в языке Go. Одновременность выполнения – важная особенность языка, она широко применяется в Go-приложениях, и мы будем учиться эффективно ее использовать.

Глава 3

Параллельные вычисления в Go

В этой главе рассматриваются следующие темы:

- *модель параллельных вычислений в языке Go;*
- *использование сопрограмм для параллельной обработки;*
- *блокировка и ожидание;*
- *использование каналов для связи между сопрограммами;*
- *важность правильного и своевременного закрытия каналов.*

В этой главе рассматривается модель параллельных вычислений, реализованная в Go. В отличие от множества современных процедурных и объектно-ориентированных языков, Go не использует модели потоков выполнения – ее заменяют сопрограммы (go-подпрограммы) и каналы. Модель, используемая в Go, имеет меньшие накладные расходы (за счет лучшего распоряжения ресурсами) и гораздо проще в управлении, чем пулы традиционных потоков. Это глава начинается с описания сопрограмм – функций, выполняющихся одновременно. Затем мы перейдем к каналам, обеспечивающим связь между сопрограммами.

3.1. Модель параллельных вычислений в Go

Грубо говоря, *параллельные вычисления* – это способность программы решать несколько задач одновременно. На практике, когда речь заходит о параллельных программах, имеются в виду программы, в течение примерно одного и того же промежутка времени решающие две или более задач, не зависящих друг от друга, но остающихся частями одной программы.

Многие популярные языки программирования, такие как Java и Python, реализуют параллельные вычисления с помощью потоков выполнения. В Go выбран другой подход. В нем используется модель *взаимодействующих последовательных процессов* (Communicating Sequential Processes, CSP), предложенная известным ученым Тони Хоаром (Tony Hoare). В этой главе затрагиваются лишь практические аспекты работы с моделью параллельных вычислений в Go, более полное изложение теории CSP можно найти на сайте golang.org.

Параллельные вычисления в Go основываются на двух важных понятиях:

- *сопрограммы (go-подпрограммы)*: сопрограммы – это функции, выполняющиеся независимо от вызвавшей их функции. Go-разработчики рассматривают go-подпрограммы как функции, выполняющиеся в собственном потоке;
- *каналы*: каналы – это конвейеры для отправки и получения данных. Их можно считать сокетами, действующими внутри программы. Каналы обеспечивают передачу структурированных данных из одной сопрограммы в другую.

Приемы, представленные в этой главе, основаны на использовании каналов и сопрограмм. Здесь вы не найдете теоретического обоснования системы сопрограмм и каналов, и основное внимание будет уделено практическому применению этих двух понятий.

Параллельные вычисления в языке Go реализованы просто и без особых затрат. Поэтому они часто применяются в библиотеках и инструментах. В этой книге они также часто будут использоваться. Эта глава посвящена вопросам, связанным с параллельной обработкой, с акцентом на отличия модели Go от моделей в других популярных языках. Здесь также содержатся примеры удачного ее применения на практике. Сопрограммы и каналы являются одной из немногих областей языка Go, способных вызвать утечки памяти. Этого можно избежать, следуя представленным в этой главе рекомендациям.

3.2. Работа с сопрограммами

С точки зрения синтаксиса, сопрограммой может быть любая функция, которая вызывается с помощью ключевого слова `go`. Теоретически любую функцию можно вызвать как сопрограмму, хотя часто в этом нет особого смысла. Чаще всего сопрограммы используются для запуска функций «в фоновом режиме», пока основная программа занимается чем-то другим. В качестве примера напишем короткую

программу, которая выводит любой введенный текст в течение 30 секунд. По истечении этого времени программа выводит собственную текстовую строку.

Листинг 3.1 ❖ Использование go-подпрограмм для выполнения заданий

```
package main

import (
    "fmt"
    "io"
    "os"
    "time"
)

func main() {
    go echo(os.Stdin, os.Stdout) ← Вызов функции echo как go-подпрограммы
    time.Sleep(30 * time.Second) ← 30-секундная пауза
    fmt.Println("Timed out.") ← Вывод сообщения о завершении ожидания
    os.Exit(0) ← Выход из программы. При этом сопрограмма будет остановлена.
}

func echo(in io.Reader, out io.Writer) { ← Функция echo является обычной функцией
    io.Copy(out, in) ← io.Copy скопирует данные из os.Reader в os.Writer
}
```

Эта программа использует сопрограмму для эхо-вывода в фоновом режиме, пока в основном режиме работает таймер. Если после запуска программы ввести некоторый текст, результат ее работы будет выглядеть следующим образом:

```
$ go run echoback.go
Hello.
Hello.
My name is Inigo Montoya
My name is Inigo Montoya
You killed my father
You killed my father
Prepare to die
Prepare to die
Timed out.
```

Вот как работает программа. Каждая строка выводится командной оболочкой в процессе ввода, а затем повторно выводится программой. И такой эхо-вывод выполняется, пока таймер не завершит работу. Как видно из примера, функция `echo` не содержит ничего особенного, но когда вызывается с ключевым словом `go`, выполняется как сопрограмма.

Функция main запускает сопрограмму и затем выжидает в течение 30 секунд. Когда функция main завершает работу, прекращается выполнение и сопрограммы, что позволяет корректно завершить работу всей программы.

РЕЦЕПТ 10 Использование замыканий сопрограмм

Любую функцию можно вызвать как сопрограмму. А поскольку язык Go поддерживает анонимные функции, можно организовать совместное использование переменных, объявив одну функцию в другой и получив замыкание разделяемых переменных.

Проблема

Нужно обеспечить применение одноразовой функции способом, который не блокирует вызывающую функцию, и убедиться, что это сработает. Такая необходимость часто возникает, когда требуется прочитать файл в фоновом режиме, отправить сообщение удаленному серверу или сохранить текущее состояние без приостановки программы.

Решение

Используем замыкание функции и дадим планировщику шанс сделать все остальное.

Обсуждение

В языке Go функции считаются самыми обычными значениями. Они могут встраиваться, передаваться в другие функции и присваиваться переменным. Можно даже объявить анонимную функцию и вызвать ее как сопрограмму, используя компактный синтаксис, как показано в следующем листинге.

Листинг 3.2 ❖ Анонимные go-подпрограммы

```
package main

import (
    "fmt"
    "runtime"
)

func main() {
    fmt.Println("Outside a goroutine.")
    go func() {                                  Объявление анонимной функции
        fmt.Println("Inside a goroutine")        и вызов ее как сопрограммы
    }()
```

```
    fmt.Println("Outside again.")
    runtime.Gosched()   ← Обращение к планировщику
}
```

Этот листинг демонстрирует, как создать анонимную функцию и сразу же вызвать ее как сопрограмму. Но при многократном выполнении этой программы ее результаты могут отличаться. Нередко можно наблюдать следующее:

```
$ go run ./simple.go
Outside a goroutine.
Outside again.
Inside a goroutine
```

Сопрограммы выполняются параллельно, но не обязательно одновременно. Вызывая сопрограмму с помощью оператора `go func`, вы сообщаете среде выполнения Go, что она должна выполнить эту функцию, как только появится такая возможность. Но это случится, скорее всего, не сразу. На самом деле если Go-программа может использовать только один процессор, практически всегда запуск сопрограммы будет отложен – планировщик продолжит выполнение охватывающей функции, пока новые обстоятельства не заставят его переключиться на другую задачу. Это ведет нас к другому аспекту данного примера.

Возможно, вы заметили в последней строке функции `main` вызов `runtime.Gosched()`. Таким способом среда выполнения Go уведомляется, что достигнута точка, где программа хочет сделать паузу и передать управление планировщику. Если в очереди планировщика имеются другие задачи (другие сопрограммы), он может выполнить одну или несколько из них, перед тем как вернуться к выполнению этой функции.

Если исключить эту строку и повторно запустить пример, результат, скорее всего, будет выглядеть так:

```
$ go run ./simple.go
Outside a goroutine.
Outside again.
```

Сопрограмма не выполняется. Почему? Функция `main` заканчивает выполнение (завершая программу) до того, как у планировщика появляется возможность запустить сопрограмму. Вызывая `runtime.Gosched`, вы даете среде выполнения шанс запустить сопрограмму до завершения программы.

Существуют и другие способы передачи управления планировщику. Наиболее распространенным их них является вызов функции `time.Sleep`. Но ни один из них не дает возможности явно указать планировщику, что делать после получения управления. В лучшем случае можно лишь уведомить планировщик, что сопрограмма достигла точки, где ее можно или нужно приостановить. Как правило, результат передачи управления планировщику является предсказуемым. Но имейте в виду, что и другие сопрограммы также могут достичь точки, где они делают паузу, и тогда планировщик может сразу же возобновить выполнение этой функции.

Например, если сопрограмма выполняет запрос к базе данных, простого вызова `runtime.Gosched` может быть недостаточно, чтобы обеспечить завершение запросов другими сопрограммами. Они могут находиться в состоянии ожидания ответа от базы данных, что вынудит планировщик продолжить выполнение текущей функции. То есть даже при том, что вызов планировщика дает ему возможность проверить состояние других сопрограмм, это не гарантирует их завершения.

Существует лучшее решение. Выход из этой сложной ситуации рассмотрен ниже.

РЕЦЕПТ 11 Ожидание завершения сопрограмм

Иногда требуется вызвать несколько сопрограмм и дождаться, когда все они завершат работу. Проще всего достичь нужного результата позволяет формирование *группы ожидания*.

Проблема

Сопрограмма должна запустить одну или несколько других сопрограмм и дождаться их завершения. В качестве практического примера рассмотрим более конкретную задачу: как можно быстрее сжать несколько файлов и вывести итоговый отчет о произведенных действиях.

Решение

Запустим отдельные задания как сопрограммы. Используем функцию `sync.WaitGroup`, чтобы известить внешний процесс, что все сопрограммы завершились и можно продолжить выполнение. На рис. 3.1 приведена обобщенная схема: запуск нескольких рабочих процессов и передача им заданий. Один из процессов передает задания рабочим процессам и ждет, пока они закончатся.

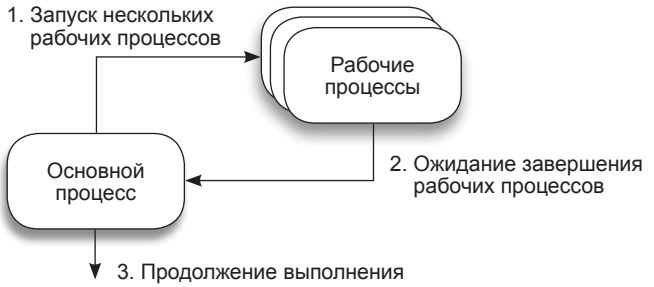


Рис. 3.1 ❖ *Запуск нескольких рабочих процессов и ожидание их завершения*

Обсуждение

Стандартная библиотека Go имеет несколько полезных инструментов синхронизации. Наиболее простым и удобным является функция `sync.WaitGroup`, позволяющая одной из сопрограмм дождаться завершения других сопрограмм.

Начнем с реализации простого инструмента сжатия произвольного количества отдельных файлов. Код в следующем листинге использует встроенную библиотеку Gzip (`compress/gzip`).

Листинг 3.3 ❖ Простой инструмент Gzip-сжатия

```
package main

import (
    "compress/gzip"
    "io"
    "os"
)

func main() {
    for _, file := range os.Args[1:] {
        compress(file)
    }
}

func compress(filename string) error {
    in, err := os.Open(filename)
    if err != nil {
        return err
    }
    defer in.Close()

    out, err := os.Create(filename + ".gz")
    if err != nil {
```

Создать список файлов, переданных в командной строке

← Открыть исходный файл для чтения

← Открыть файл архива с расширением .gz и именем исходного файла

```
        return err
    }
    defer out.Close()
                                          Сжать данные и записать
    gzout := gzip.NewWriter(out)    ← в соответствующий файл с помощью gzip.Writer
    _, err = io.Copy(gzout, in)     ← Функция io.Copy выполняет
    gzout.Close()                         необходимое копирование

    return err
}
```

Этот инструмент принимает список файлов из командной строки и сжимает по отдельности каждый из файлов, создавая файлы архивов с теми же именами, но с расширением .gz. Допустим, что имеется каталог `exampledata` со следующим содержимым:

```
$ ls -1 exampledata
example1.txt
example2.txt
example3.txt
```

В каталоге `exampledata` присутствуют три тестовых файла. Воспользовавшись вновь созданным инструментом, их можно сжать:

```
$ go run simple_gz.go exampledata/*
$ ls -1 exampledata
example1.txt
example1.txt.gz
example2.txt
example2.txt.gz
example3.txt
example3.txt.gz
```

После выполнения примера можно убедиться, что программа `simple_gz.go` создала сжатые версии файлов.

А теперь поговорим о производительности. Наша программа использует единственную сопрограмму (и, следовательно, нагружает только одно ядро процессора). Кроме того, маловероятно, что эта программа задействует всю пропускную способность дисковой подсистемы. Несмотря на то что этот код прекрасно справляется со своей задачей, он делает это не так быстро, как мог бы. И поскольку каждый файл может сжиматься по отдельности, мы легко могли бы разбить единственный поток выполнения на несколько, выполняющихся параллельно.

Эту программу можно переписать, организовав сжатие каждого файла в отдельной сопрограмме. Такое решение будет оптималь-

ным для сжатия тысяч файлов (и, вероятно, полностью использует возможности системы ввода-вывода), оно хорошо будет работать и в случае сжатия нескольких сотен файлов или даже меньшего их количества.

А теперь применим небольшой трюк: организуем параллельное сжатие нескольких файлов и заставим родительскую сопрограмму (`main`) ожидать завершения всех рабочих процессов. Этого легко можно добиться, создав группу ожидания. В листинге 3.4 представлен только измененный код (функция сжатия не претерпела никаких изменений). Такое решение считается наиболее оптимальным, поскольку не требует выполнять ожидание группы в рабочих функциях (`compress`), когда файлы должны сжиматься поочередно.

Листинг 3.4 ❖ Параллельное сжатие файлов с ожиданием завершения группы

```
package main

import (
    "compress/gzip"
    "fmt"
    "io"
    "os"
    "sync"
)

func main() {
    var wg sync.WaitGroup  ← Нет необходимости инициализировать WaitGroup
    var i int = -1         | Так как переменная i необходима за пределами цикла,
    var file string        | она объявляется именно здесь
    for i, file = range os.Args[1:] {
        wg.Add(1)  ← Для каждого файла сообщить группе,
                     что ожидается выполнение еще одной операции сжатия
        go func(filename string) {     | Эта функция вызывает функцию сжатия
            compress(filename)         | и уведомляет группу ожидания о ее завершении
            wg.Done()                  |
        }(file)  ← Поскольку вызов сопрограммы происходит в цикле for,
                   требуется небольшая хитрость, чтобы передать параметр
    }
    wg.Wait()                                    ← Внешняя сопрограмма (main) ожидает,
    fmt.Printf("Compressed %d files\n", i+1)       пока все сопрограммы, выполняющие
}                                                  сжатие, вызовут wg.Done

func compress(filename string) error {
    // Здесь без изменений
}
```

В этой измененной версии инструмента сжатия функция `main` претерпела значительные изменения. Во-первых, была добавлена группа ожидания.

Группа ожидания – это механизм передачи сообщений, оповещающих ожидающую сопрограмму, что она может продолжить работу. Чтобы воспользоваться им, нужно сообщить группе ожидания, что требуется дождаться завершения каких-то действий, а затем послать ей сигнал об их завершении. Группе ожидания не нужно знать ничего больше, кроме (а) количества ожидаемых действий и (б) момента их завершения. Количество ожидаемых действий увеличивается вызовом `wg.Add`, а момент завершения задания обозначается вызовом `wg.Done`. Функция `wg.Wait` блокирует выполнение, пока не будут завершены все задания, добавленные в группу. Процесс иллюстрирует рис. 3.2.

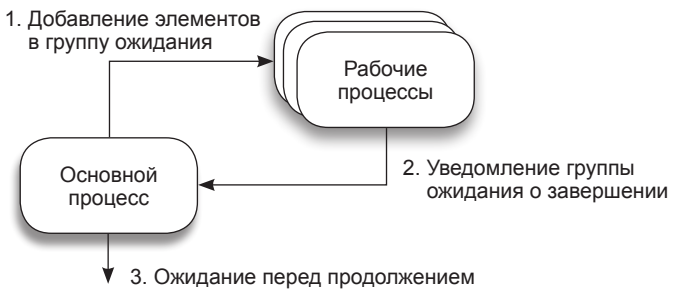


Рис. 3.2 ❖ *Группы ожидания в действии*

В этой программе вызов `wg.Done` производится внутри сопрограммы. Сопрограмма получает имя файла и вызывает для него функцию сжатия. Обратите внимание, что часть кода на первый взгляд кажется избыточным. Вместо запирания имени файла в замыкании оно передается в сопрограмму через параметр `filename`. Это сделано по причинам, связанным с планировщиком языка Go. Область видимости переменной `file` ограничена телом цикла `for`, то есть ее значение будет меняться в каждой итерации цикла. Но, как уже упоминалось выше в этой главе, объявление сопрограммы не приводит к ее немедленному выполнению. При пятикратном выполнении цикла планировщику будет передано пять сопрограмм, но весьма вероятно, что ни одна из них не будет выполнена. И в каждой из этих пяти итераций значение переменной `file` будет меняться. К моменту запуска сопро-

грамм все они могут получить одну и ту же (пятую) версию строкового значения переменной `file`. А это не то, что нужно. Нам нужно, чтобы в каждой итерации запланированное задание получало свое значение переменной `file`, поэтому оно передается как параметр функции, что гарантирует передачу нужного значения.

На первый взгляд эта проблема может показаться магической, но в ней нет ничего необычного. Всякий раз, запуская сопрограммы в цикле, проявляйте особую осторожность: используемые сопрограммой переменные не должны изменяться в цикле. Самый простой способ добиться этого заключается в создании копий переменных внутри цикла.

РЕЦЕПТ 12 Блокировка с помощью мьютексов

Всякий раз, когда две или более сопрограмм работают с одним и тем же фрагментом данных и эти данные могут изменяться, возникает вероятность появления *состояния гонки*. В состоянии гонки две сущности «состязаются» за использование одного и того же фрагмента информации. Проблемы возникают, когда они одновременно пытаются выполнять операции с данным. Одна из сопрограмм может успеть только частично изменить значение, когда другая попытается использовать его. Такая ситуация может иметь непредвиденные последствия.

Проблема

Нескольким сопрограммам необходим доступ на чтение и изменение одного и того же фрагмента данных.

Решение

Один из самых простых способов избежать состояния гонки – дать каждой сопрограмме возможность «заблокировать» ресурс перед использованием и разблокировать после выполнения операций. Все прочие сопрограммы, столкнувшиеся с блокировкой, будут вынуждены ждать снятия блокировки, перед тем как самим заблокировать ресурс. Для блокировки и разблокировки объекта используется функция `sync.Mutex`.

Обсуждение

Встроенный пакет `sync` определяет интерфейс `sync.Locker`, а также пару реализаций блокировок. Это обеспечивает все необходимое для работы с блокировками.

Не занимайтесь самостоятельной реализацией блокировок

Некоторое время назад прошел слух, что блокировки реализованы настолько плохо, что лучше реализовать собственные. Это подвигло многих разработчиков на создание собственных библиотеки блокировок. В лучшем случае они проделали никому не нужную работу. В худшем – получились ненадежные или медленные системы блокировок.
Не имея доказательств ненадежности или кода, чрезвычайно чувствительного к производительности, воздержитесь от разработки собственных блокировок. Встроенный пакет прост в использовании, прошел все испытания и соответствует требованиям к производительности большинства приложений.
Далее в главе будет описана работа с каналами. Кстати, код, использующий каналы, получает в их виде более эффективный механизм блокировки. Это отличное решение (и мы покажем, как его реализовать). Но, вообще говоря, нет никаких причин заниматься реализацией собственных библиотек блокировки.

Следующий листинг содержит пример программы, в которой возникает состояние гонки. Эта простая программа читает файлы, указанные в аргументах командной строки, и подсчитывает число вхождений каждого найденного в них слова. В завершение она выводит список слов, встречающихся более одного раза.

Листинг 3.5 ❖ Счетчик слов с состоянием гонки

```
package main

import (
    "bufio"
    "fmt"
    "os"
    "strings"
    "sync"
)

func main() {
    var wg sync.WaitGroup  ← И снова используется группа ожидания
    w := newWords()          для мониторинга группы сопрограмм
    for _, f := range os.Args[1:] {
        wg.Add(1)
        go func(file string) {
            if err := tallyWords(file, w); err != nil {
                fmt.Println(err.Error())
            }
```

Главный цикл использует прием из рецепта 12

```
            wg.Done()
        }(f)
    }
    wg.Wait()
    fmt.Println("Words that appear more than once:")
    for word, count := range w.found {
        if count > 1 {
            fmt.Printf("%s: %d\n", word, count)
        }
    }
}
```

Перед завершением программы вывести результаты поиска

```
type words struct {
    found map[string]int
}
```

← Извлекаемые слова помещаются в структуру. Можно также использовать тип данных map, но применение структуры облегчит последующую реорганизацию кода

```
func newWords() *words {
    return &words{found: map[string]int{}}
}
```

← Создание нового экземпляра слова

```
func (w *words) add(word string, n int) {
    count, ok := w.found[word]
    if !ok {
        w.found[word] = n
        return
    }
    w.found[word] = count + n
}
```

Фиксирует количество вхождений этого слова

Если слово еще не зафиксировано, добавим его. В противном случае увеличим счетчик

```
func tallyWords(filename string, dict *words) error {
    file, err := os.Open(filename)
    if err != nil {
        return err
    }
    defer file.Close()


    scanner := bufio.NewScanner(file)
    scanner.Split(bufio.ScanWords)
    for scanner.Scan() {
        word := strings.ToLower(scanner.Text())
        dict.add(word, 1)
    }
    return scanner.Err()
}
```

← Открытие файла, анализ содержимого и подсчет найденных в нем слов. Функция копирования делает все необходимое

Сканер – полезный инструмент для подобного анализа файлов

Функция main обходит все переданные ей в командной строке файлы и получает статистические данные для каждого из них. Как ожидается, эта программа будет читать текстовые файлы и выводить список найденных в них и повторяющихся слов. Проверим ее на одном файле:

```
$ go run race.go 1.txt
Words that appear more than once:
had: 2
down: 2
the: 5
have: 2
that: 3
would: 3
...
```

Собственно, это и ожидалось увидеть. Если передать программе несколько файлов, они будут обрабатываться в собственных сопрограммах. Давайте посмотрим, что из этого получится:

```
$ go run race.go *.txt
fatal error: concurrent map writes

goroutine 8 [running]:
runtime.throw(0x115890, 0xd)
    /usr/local/go/src/runtime/panic.go:527 +0x90 fp=0x82029cbf0
    sp=0x82029cbd8
runtime.evacuate(0xca600, 0x8202142d0, 0x16)
    /usr/local/go/src/runtime/hashmap.go:825 +0x3b0 fp=0x82029ccb0
sp=0x82029cbf0
runtime.growWork(0xca600, 0x8202142d0, 0x31)
    /usr/local/go/src/runtime/hashmap.go:795 +0x8a fp=0x82029ccd0
    sp=0x82029ccb0
runtime.mapassign1(0xca600, 0x8202142d0, 0x82029ce70, 0x82029cdb0)
    /usr/local/go/src/runtime/hashmap.go:433 +0x175 fp=0x82029cd78
    sp=0x82029ccd0
...
```

Через некоторое время после запуска возникает ошибка. Почему? Сообщение об ошибке содержит подсказку: concurrent map writes (одновременная запись в отображение). Если добавить в команду флаг --race, мы получим более подробное сообщение:

```
go run --race race.go *.txt
==================
WARNING: DATA RACE
```

```
Read by goroutine 8:
   runtime.mapaccess2_faststr()
       /tmp/workdir/go/src/runtime/hashmap_fast.go:281 +0x0
   main.tallyWords()
       /Users/mbutcher/Code/go-in-practice/chapter3/race/race.go:62 +0x3ed
   main.main.func1()
       /Users/mbutcher/Code/go-in-practice/chapter3/race/race.go:18 +0x66
Previous write by goroutine 6:
    runtime.mapassign1()
        /tmp/workdir/go/src/runtime/hashmap.go:411 +0x0
    main.tallyWords()
        /Users/mbutcher/Code/go-in-practice/chapter3/race/race.go:62 +0x48a
    main.main.func1()
        /Users/mbutcher/Code/go-in-practice/chapter3/race/race.go:18 +0x66
Goroutine 8 (running) created at:
    main.main()
        /Users/mbutcher/Code/go-in-practice/chapter3/race/race.go:22 +0x238
Goroutine 6 (running) created at:
    main.main()
        /Users/mbutcher/Code/go-in-practice/chapter3/race/race.go:22 +0x238
==================
```

Проблема возникла в функции `words.add`. Несколько сопрограмм одновременно обращаются к одному и тому же фрагменту памяти, объекту `words.found` типа `map` (обратите внимание на строки, выделенные жирным). В них указаны строки, вызывающие состояние гонки при изменении объекта `map`.

Go включает встроенные средства обнаружения состояния гонки

Многие инструменты Go, в том числе `go run` и `go test`, при передаче флага `--race` включают в работу механизм обнаружения состояния гонки. Этот механизм существенно замедляет выполнение, но может пригодиться для выявления состояний гонки в процессе разработки.

Если взглянуть еще раз на код программы, можно быстро выявить проблему. Когда метод `add` вызывается сразу несколькими сопрограммами, они одновременно могут выполнять операции с одним и тем же объектом `map`. Это гарантированно приводит к его порче.

Единственное простое решение – установка блокировки перед изменением объекта с последующим ее снятием. Для этого доста-

точно внести небольшие изменения в код, как показано в следующем листинге.

Листинг 3.6 ❖ Подсчет слов с блокировками

```
package main

import (
    // Те же пакеты, что и прежде...
    "sync"
)

func main() {
    var wg sync.WaitGroup
    w := newWords()
    for _, f := range os.Args[1:] {
        wg.Add(1)
        go func(file string) {
                if err := tallyWords(file, w); err != nil {
                       fmt.Println(err.Error())
                }
                wg.Done()
        }(f)
    }
    wg.Wait()
    fmt.Println("Words that appear more than once:")
    w.Lock()
    for word, count := range w.found {
        if count > 1 {
            fmt.Printf("%s: %d\n", word, count)
        }
    }
    w.Unlock()
}

type words struct {
    sync.Mutex  ← Структура words теперь наследует мьютекс
    found map[string]int
}

func newWords() *words {
    return &words{found: map[string]int{}}
}

func (w *words) add(word string, n int) {
    w.Lock()
    defer w.Unlock()
```

Установка и снятие блокировки доступа к объекту до и после итераций. Строго говоря, это делать не обязательно, поскольку этот фрагмент выполнится только после обработки всех файлов

Заблокировать объект, изменить и разблокировать

```
    count, ok := w.found[word]
    if !ok {
        w.found[word] = n return
    }
    w.found[word] = count + n
}

func tallyWords(filename string, dict *words) error {
    // Здесь без изменений
}
```

В этой версии структура `words` включает анонимное поле `sync.Mutex`, что дает возможность использовать методы `words.Lock` и `words.Unlock`. Это распространенный способ добавления блокировки в структуру. (Эти методы используются до и после обхода слов в конце функции `main`).

Теперь метод `add` блокирует объект `map`, изменяет его и затем разблокирует. Если он будет одновременно вызван сразу несколькими сопрограммами, первая из них установит блокировку, а другие будут ждать ее снятия. Это предотвратит одновременное изменение `map`-объекта несколькими сопрограммами.

Важно отметить, что блокировки оказываются действенным инструментом, только когда все попытки доступа к данным управляются одной и той же блокировкой. Если операции с одними данными осуществляются только с блокировкой, а с другими – нет, все еще возможно состояние гонки.

Иногда бывает желательно разрешить одновременное выполнение нескольких операций чтения фрагмента данных и запретить любые операции во время записи. Такую возможность дает функция `sync.RWLock`. Пакет `sync` содержит еще ряд полезных инструментов, упрощающих координацию сопрограмм.

А теперь перейдем к знакомству с другим основным понятием модели параллельного выполнения в Go – каналами.

3.3. Работа с каналами

Каналы позволяют передавать сообщения между сопрограммами. В этом разделе рассматривается несколько способов использования каналов для реализации распространенных задач и решения обычно возникающих при этом проблем.

Каналы можно сравнить с сетевыми сокетами. Два приложения могут связываться между собой посредством сетевых сокетов. В зависи-

мости от организации этих приложений сетевой трафик может быть однонаправленным или двунаправленным. Иногда сетевые соединения существуют недолго, а иногда продолжают работу в течение длительного времени. Сложные приложения могут использовать сразу несколько сетевых соединений для отправки и получения разных типов данных. Через сетевой сокет можно передавать любые данные, при условии что они будут преобразованы в простую последовательность байтов.

Каналы в языке Go действуют подобно сокетам, связывающим две сопрограммы в одном приложении. Подобно сетевым сокетам, они могут быть однонаправленными и двунаправленными. Каналы могут существовать недолго и длительный период времени. И в приложении может использоваться сразу несколько каналов для пересылки разных видов данных. Но, в отличие от сетевых соединений, каналы являются типизированными и способны передавать структурированные данные. Как правило, нет необходимости преобразовывать данные перед передачей в канал.

Для демонстрации каналов изменим предыдущий пример и задействуем в нем каналы.

Не злоупотребляйте каналами

Каналы – фантастический инструмент связи между сопрограммами. Они просты в использовании и значительно упрощают параллельное программирование, по сравнению с моделью потоков выполнения в других популярных языках.

Но их использованием не стоит увлекаться. Каналы вносят дополнительные затраты и могут негативно влиять на производительность. Они усложняют программу. И самое главное, каналы – единственный источник проблем, связанных с управлением памятью в программах на языке Go. Как любой другой инструмент, каналы следует использовать, только когда в них возникает необходимость, но не при каждом кажущимся удобным случае.

РЕЦЕПТ 13 Использование нескольких каналов

Go-разработчики любят подчеркнуть, что каналы являются средствами связи. Они обеспечивают передачу информации между сопрограммами. Иногда лучший способ решения проблем параллельной обработки данных в языке Go – увеличить объем передаваемой информации. И это часто приводит к использованию большего количества каналов.

Проблема

Использовать каналы для передачи данных между сопрограммами и получить возможность прерывать этот процесс при выходе.

Решение

Используем оператор `select` и несколько каналов. В языке Go каналы часто используются для передачи сигнала, когда завершилось выполнение какого-то задания или все готово к закрытию.

Обсуждение

Для продолжения знакомства с каналами вернемся к первому примеру в этой главе. Эта программа в течение 30 секунд осуществляет эхо-вывод всего, что вводит пользователь. Эхо-вывод в ней реализован в виде сопрограммы, а измерение 30-секундного интервала осуществляется вызовом метода `time.Sleep`. Перепишем эту программу, использовав каналы в дополнение к сопрограммам.

Мы не собираемся расширять возможности программы или както улучшать ее. Мы просто хотим показать другой подход к решению той же задачи. При этом вы увидите пример идиоматического использования каналов.

Прежде чем перейти к реализации в листинге 3.7, ознакомимся с некоторыми использованными в ней идеями:

- каналы создаются с помощью метода `make`, подобно объектам `map` и `slice`;
- оператор стрелки (`<-`) используется для обозначения направления канала (`out chan<-[]byte`) и для отправки или получения данных (`buf := <-echo`);
- инструкция `select` может следить сразу за несколькими каналами. Пока ничего не происходит, она ждет (или выполняет инструкцию `default`, если определена). Когда в канале возникает событие, `select` обрабатывает его. Более подробная информация о каналах будет представлена ниже в этой главе.

Листинг 3.7 ❖ Использование нескольких каналов

```
package main

import (
    "fmt"
    "os"
   "time"
)
```

```
func main() {
    done := time.After(30 * time.Second)
    echo := make(chan []byte)
    go readStdin(echo)
    for {
           select {
           case buf := <-echo:
                  os.Stdout.Write(buf)
           case <-done:
                  fmt.Println("Timed out")
                  os.Exit(0)
           }
     }
}
```

Создание канала, посылающего сообщение по истечении 30 секунд

Создание нового канала для передачи байтов из Stdin в Stdout. Поскольку размер не указан, этот канал может хранить одновременно только одно сообщение

Запуск сопрограммы для чтения данных из Stdin и передачи их в новый канал

Использование инструкции select для передачи данных из Stdin в Stdout, если имеются, и для завершения по событию таймера

```
func readStdin(out chan<- []byte) {
    for {
        data := make([]byte, 1024)
        l, _ := os.Stdin.Read(data)
        if l > 0 {
               out <- data
        }
    }
}
```

Принимает канал для записи (chan<-) и отправляет в канал введенные данные

Копирование данных из Stdin в объект data. Обратите внимание, что метод File.Read блокирует выполнение, пока не получит всех данных

Отправка буферизированных данных в канал

Этот пример выведет следующее:

```
$ go run echoredux.go
test 1
test 1
test 2
test 2
test 3
test 3
Timed out
```

Как видите, при вводе, например, текста `test 1`, программа вывела его повторно. Через 30 секунд программа самостоятельно завершается.

Переделка примера эхо-вывода помогла нам познакомиться с новыми понятиями, касающимися каналов. Первый канал в примере

выше создается с помощью пакета time. Функция time.After создает канал, в который по истечении заданного интервала времени будет отправлено сообщение (объект time.Time). Вызов time.After(30 * time.Second) возвращает канал <-chan time.Time (односторонний канал для передачи объектов time.Time), через который через 30 секунд придет сообщение. С практической точки зрения два способа организации паузы в следующем листинге функционально эквивалентны.

Листинг 3.8 ❖ Реализация паузы с помощью методов Sleep и After

```
package main

import (
    "time"
)

func main() {
    time.Sleep(5 * time.Second)   ← Приостановка на пять секунд
    sleep := time.After(5 * time.Second)   Создание канала для получения
    <-sleep                                уведомления через пять секунд
}                                          с последующей приостановкой
                                           до поступления уведомления
```

Некоторые функции (например, time.After) автоматически создают и инициализируют каналы. Но обычно для создания нового канала используется встроенная функция make.

По умолчанию каналы являются двунаправленными. Как было показано в предыдущем примере, каналу можно указать «направление» при передаче в функцию (или при выполнении любого другого назначения). Функция readStdin может только записывать данные в канал. Любая попытка чтения из него приведет к ошибке времени компиляции. Обычно считается хорошей привычкой указывать направление канала в сигнатуре функции.

Последним важным аспектом этой программы является оператор select. С синтаксической точки зрения оператор select похож на switch. Он может содержать любое количество операторов case, а также один оператор default.

Оператор select проверяет условия во всех операторах case, чтобы выяснить, какие из них содержат операцию отправки или получения, которую необходимо выполнить. Если только один из операторов case отправляет или получает данные, select выполняет его. Если таких операторов несколько, select выбирает один из них случайным образом. Если ни один из операторов case не отправляет или не получает данных, select выбирает оператор default (если присутствует).

Если оператор `default` отсутствует, `select` блокирует выполнение, пока один из операторов `case` не получит возможности отправить или получить данные.

В данном примере оператор `select` ожидает получения данных по двум каналам. Если сообщение поступит через канал `echo`, полученная строка будет сохранена в буфере `buf` (`buf := <-echo`), а затем записана в стандартный вывод. Этот пример демонстрирует, как присвоить переменной принятое значение.

Второй оператор `case` ожидает сообщения из канала `done`. Поскольку содержимое сообщения не представляет интереса, оно не присваивается переменной. После его получения надобность в канале отпадает, и оператор `select` отбрасывает полученное значение (`<-done`).

Оператор `select` не содержит инструкции `default`, поэтому он блокирует выполнение, пока не будет получено сообщение по каналу `<-echo` или `<-done`. Получив сообщение, `select` выполнит соответствующий блок `case` и вернет управление. Поскольку оператор `select` заключен в цикл `for`, он будет выполняться до тех пор, пока не будет получено сообщение по каналу `<-done`, что приведет к завершению программы.

Единственное, что мы пока не рассмотрели – это закрытие каналов по завершении их использования. В данном примере программа выполняется недолго, чтобы в этом возникала необходимость, поэтому здесь эта обязанность перекладывается на среду выполнения. Но в следующем рецепте вы узнаете, как правильно закрывать каналы.

РЕЦЕПТ 14 Закрытие каналов

Разработчики на языке Go перекладывают обязанность по уборке мусора на диспетчера памяти. Когда переменная выходит из области видимости, связанная с ней память автоматически освобождается. Но при работе с сопрограммами и каналами следует проявлять повышенную осторожность. Что произойдет, если отправителем и получателем данных являются сопрограммы и отправитель завершит отправку данных? Будут ли автоматически освобождены получатель и канал? Нет. Диспетчер памяти освобождает только значения, которые гарантированно не будут больше использоваться, к открытым каналам и сопрограммам это не относится.

Представьте на минуту, что этот код является частью более крупной программы и функция `main` является обычной функцией, многократно вызываемой на протяжении всего времени работы приложения. При каждом вызове она создает новый канал и новую сопрограмму.

Но ни один из каналов не закрывается и ни одна из сопрограмм не завершается. Такая программа создаст утечку каналов и сопрограмм.

Возникает вопрос: как правильно и безопасно реализовать очистку при использовании сопрограмм и каналов? Неправильная очистка может привести к утечке памяти или к утечке каналов/сопрограмм, когда ненужные сопрограммы и каналы потребляют системные ресурсы, ничего не делая при этом.

Проблема

Требуется исключить потребление ресурсов неиспользуемыми каналами и сопрограммами, предотвратив тем самым утечку памяти в приложении. Необходимо безопасно закрыть каналы и завершить сопрограммы.

Решение

Самый простой ответ на вопрос «как избежать утечки каналов и сопрограмм?»: «закрывать каналы и выходить из сопрограмм». Хотя этот ответ по сути правилен, но неполон. Неправильное закрытие каналов приведет программу к краху или утечке сопрограмм.

Основной способ предотвращения небезопасного закрытия каналов заключается в использовании дополнительных каналов для уведомления сопрограмм о возможности безопасно закрыть канал.

Обсуждение

Для безопасного закрытия каналов можно использовать несколько идиоматических методов.

Начнем с отрицательного примера, приведенного в следующем листинге. Основываясь на идее в листинге 3.7, напишем программу, которая неправильно управляет своим каналом.

Листинг 3.9 ❖ Неправильное закрытие канала

```
package main

import (
    "fmt"
    "time"
)

func main() {
    msg := make(chan string)
    until := time.After(5 * time.Second)
    go send(msg)   ← Вызов сопрограммы send с передачей канала для отправки
```

```
    for {            ← Цикл вокруг select, ожидающего сообщения от send
          select {       и об истечении заданного интервала времени
          case m := <-msg:   ← Если получено сообщение от send, вывести его
                fmt.Println(m)
          case <-until:                                    Выход по истечении
                close(msg)                                 заданного интервала времени.
                time.Sleep(500 * time.Millisecond)         Пауза позволит вам увидеть
                return                                     ошибку, возникающую перед
          }                                                выходом из сопрограммы
    }                                                      main
}

func send(ch chan string) {  ← Отправляет сообщение «hello» каждые полсекунды
    for {
          ch <- "hello"
          time.Sleep(500 * time.Millisecond)
    }
}
```

Цель этого надуманного примера – показать проблему, возникающую в программе с длительным временем выполнения. Программа выводит примерно 10 строк `hello`, а затем завершает работу. Но, запустив ее, вы получите следующий результат:

```
$ go run bad.go
hello
hello
hello
hello
hello
hello
hello
hello
hello
hello
panic: send on closed channel

goroutine 20 [running]: main.send(0x82024c060)
     /Users/mbutcher/Code/go-in-practice/chapter3/closing/bad.go:28   +0x4c
created by main.main
     /Users/mbutcher/Code/go-in-practice/chapter3/closing/bad.go:12   +0x90
goroutine 1 [sleep]:
time.Sleep(0x1dcd6500)
     /usr/local/go/src/runtime/time.go:59 +0xf9
main.main()
     /Users/mbutcher/Code/go-in-practice/chapter3/closing/bad.go:20   +0x24f
exit status 2
```

Перед завершением программы возникает авария, потому что функция `main` закрывает канал `msg`, в то время как сопрограмма `send` продолжает посылать в него сообщения. Отправка сообщения в закрытый канал вызывает аварию. Функция `close` должна вызываться только отправителем, причем с соблюдением некоторых защитных мер.

Что произойдет, если канал закроет отправитель? Авария не случится, но произойдет нечто интересное. Взгляните на короткий пример в следующем листинге.

Листинг 3.10 ❖ Закрытие отправителем

```
package main

import "time"

func main() {
    ch := make(chan bool)
    timeout := time.After(600 * time.Millisecond)
    go send(ch)
    for {
            select {
            case <-ch:
                    println("Got message.")
            case <-timeout:
                    println("Time out")
                    return
            default:
                    println("*yawn*")
                    time.Sleep(100 * time.Millisecond)
            }
    }
}

func send(ch chan bool) {
    time.Sleep(120 * time.Millisecond)
    ch <- true
    close(ch)
    println("Sent and closed")
}
```

Цикл вокруг select, проверяющего два канала и имеющего ветвь default

Получив сообщение по основному каналу, выведем что-нибудь определенное

По истечении заданного интервала времени завершим программу

По умолчанию делаем небольшую паузу. Это облегчает работу с примером

Отправка единственного сообщения в канал с последующим его закрытием

На основании анализа этого кода можно предположить, что он несколько раз выполнит действие по умолчанию, затем получит сообщение от `send`, а потом, после выполнения еще нескольких действий по умолчанию, истечет установленный интервал времени и программа завершит работу.

Но вместо этого вы увидите следующее:

```
$ go run sendclose.go
*yawn*
*yawn*
*yawn*
*yawn*
Got message.
Got message.
Sent and closed
*yawn*
Sent and closed
*yawn*
Got message.
Got message.
Got message.
Got message.
... #и еще тысячи раз
Time out
```

Такое происходит потому, что закрытый канал всегда возвращает значение `nil`. Итак, сопрограмма `send` пошлет одно значение `true`, а затем закроет канал. При каждой проверке канала `ch` после его закрытия оператор `select` будет получать значение `false` (значение `nil` для канала с типом данных `bool`).

Эту проблему можно обойти. Например, можно прервать цикл `for/select` после получения значения `false` из канала `ch`. Иногда так и следует поступать. Но лучше явно сообщить, что работа с каналом завершена, а затем закрыть его.

Этот способ демонстрируется в листинге 3.9, где используется дополнительный канал для оповещения о завершении работы с каналом. Это дает обеим сторонам возможность аккуратно обработать закрытие канала.

Листинг 3.11 ❖ Использование завершающего канала

```
package main

import (
    "fmt"
    "time"
)

func main() {
    msg := make(chan string)
```

```
    done := make(chan bool)              ← Дополнительный канал с типом данных bool
    until := time.After(5 * time.Second)   для сообщения о завершении
    go send(msg, done)  ← Передача двух каналов в send
    for {
          select {
          case m := <-msg:
                fmt.Println(m)
          case <-until:              По истечении заданного интервала времени сообщить
                done <- true  ← сопрограмме send, что работа завершена
                time.Sleep(500 * time.Millisecond)
                return
          }
     }
}

func send(ch chan<-string, done <-chan bool) {  ← ch используется для отправки,
    for {                                           а done – для получения
        select {
        case <-done:              | Завершить работу после получения сообщения
                println("Done")   | из канала done
                close(ch)
                return
        default:
                ch <- "hello"
                time.Sleep(500 * time.Millisecond)
        }
    }
}
```

Этот пример демонстрирует реализацию шаблона «Наблюдатель», часто применяемого в Go: использование канала (как правило, носящего имя `done`) для передачи сигнала от одной сопрограммы к другой. Обычно этот шаблон применяется, если имеются одна сопрограмма, получающая сообщения, и другая, основной задачей которой является их отправка. Если получатель решает закончить работу, он должен оповестить об этом отправителя.

Функция `main` в листинге 3.11 знает, когда следует закончить обработку. Но она также является получателем. А как упоминалось выше, получатель никогда не должен закрывать канал, используемый для приема. Вместо этого он должен отправить сообщение в канал `done`, информирующее о том, что он завершил свою работу. Теперь, получив сообщение из канала `done`, функция `send` сможет закрыть канал и завершиться.

РЕЦЕПТ 15 Блокировка с помощью буферизированных каналов

До сих пор мы рассматривали каналы, способные хранить не более одного значения и создаваемые с помощью `make(chan TYPE)`. Они называются *небуферизованными каналами*. Если такой канал уже хранит значение и ему передать еще одно, прежде чем предыдущее будет прочитано, вторая операция отправки будет заблокирована. Отправитель также окажется заблокированным, пока получатель не прочитает предыдущее значение.

Иногда подобные блокировки нежелательны и их можно избежать, создавая буферизованные каналы.

Проблема

В особенно критичной части программы необходимо заблокировать определенные ресурсы. Учитывая, что для этой цели часто используются каналы, было бы желательно сделать это с помощью каналов вместо пакета `sync`.

Решение

Используем канал с размером буфера 1 и обеспечим совместное использование этого канала сопрограммами, которые требуется синхронизировать.

Обсуждение

В рецепте 12 мы познакомились с использованием `sync.Locker` и `sync.Mutex` для блокировки доступа к критичным фрагментам данных. Пакет `sync` является частью ядра Go и, как следствие, хорошо протестирован. Но иногда (особенно в программах, уже использующих каналы) желательно реализовать блокировки с помощью каналов, а не с помощью мьютексов. Часто это определяется стилистическими причинами и разумным желанием сделать код максимально однородным.

При использовании каналов в роли блокировок часто реализуется следующий сценарий:

1. Функция устанавливает блокировку, отправляя сообщение в канал.
2. Продолжает выполнение критичных операций.
3. Освобождает блокировку, читая сообщение из канала.
4. Любая функция, попытавшаяся установить блокировку до ее снятия, будет приостановлена в при попытке выполнить запись в канал (заблокируется).

Такой сценарий невозможно реализовать с помощью небуферизованных каналов. Первый же шаг в этой последовательности заблокирует функцию-отправителя в момент отправки сообщения. Другими словами, отправитель вынужден ждать, пока кто-то другой не прочитает отправленное сообщение.

Но буферизованные каналы не блокируют функцию-отправителя, при условии что в буфере канала имеется свободное пространство. Отправитель может поместить сообщение в буфер и продолжить выполнение. Но если буфер уже заполнен, отправитель будет блокирован, пока в буфере не освободится место для нового сообщения.

Именно это и требуется для реализации блокировки. Мы создадим канал с пустым буфером единичного размера. Одна функция сможет записать сообщение, продолжить выполнение и позднее прочитать свое сообщение из буфера (разблокировав его). Следующий листинг содержит простую реализацию этого сценария.

Листинг 3.12 ❖ Простая блокировка посредством каналов

```
package main

import (
    "fmt"
    "time"
)

func main() {
    lock := make(chan bool, 1)
    for i := 1; i < 7; i++ {
        go worker(i, lock)
    }
    time.Sleep(10 * time.Second)
}

func worker(id int, lock chan bool) {
    fmt.Printf("%d wants the lock\n", id)
    lock <- true
    fmt.Printf("%d has the lock\n", id)
    time.Sleep(500 * time.Millisecond)
    fmt.Printf("%d is releasing the lock\n", id)
    <-lock
}
```

Создание буферизированного канала единичного объема

Вызов шести сопрограмм, совместно использующих блокирующий канал

Рабочий процесс устанавливает блокировку, отправляя сообщение. Первый рабочий процесс захватывает единичный объем, что делает его собственником блокировки. Остальные окажутся заблокированными

Фрагмент между lock <- true и <- lock выполняется под защитой блокировки

Снять блокировку, прочитав значение из канала. В результате в буфере освободится место, и следующая функция сможет установить блокировку

Схема достаточно проста: один шаг устанавливает блокировку, и еще один – снимает ее. Если выполнить эту программу, она выведет следующее:

```
$ go run lock.go
2 wants the lock
1 wants the lock
2 has the lock
5 wants the lock
6 wants the lock
4 wants the lock
3 wants the lock
2 is releasing the lock
1 has the lock
1 is releasing the lock
5 has the lock
5 is releasing the lock
6 has the lock
6 is releasing the lock
3 has the lock
3 is releasing the lock
4 has the lock
4 is releasing the lock
```

Как видите, все шесть сопрограмм последовательно получают возможность установить и снять блокировку. В течение первых нескольких миллисекунд после запуска программы все шесть попытались установить блокировку. Но только сопрограмме 2 это удалось. Через несколько сотен миллисекунд сопрограмма 2 сняла блокировку и сопрограмма 1 тут же установила ее. Процесс продолжается до тех пор, пока последняя сопрограмма (4) не установила и не сняла блокировку. (Обратите внимание, что в этом примере закрытие и удаление канала возлагается на диспетчера памяти. После удаления всех ссылок на канал он автоматически удалит его.)

Листинг 3.12 иллюстрирует одно из преимуществ использования буферизированных каналов для создания очередей: они не блокируют операции записи, если в очереди достаточно места для нового сообщения. Определяя длину очереди, можно управлять объемом памяти, которую занимает буфер. С помощью буферизованного канала можно реализовать одновременное выполнение, например, не более двух сопрограмм, установив длину буфера, равную 2. Буферизованные очереди также часто используются для создания очередей сообщений и конвейеров.

3.4. Итоги

Эта глава была посвящена системе параллельных вычислений в языке Go. Мы познакомились с сопрограммами и пакетами, обеспечивающими синхронизацию сопрограмм. Затем была описана мощная система каналов Go, позволяющая нескольким сопрограммам взаимодействовать посредством типизированных каналов. Попутно было представлено несколько важных идиом и шаблонов, включая:

- модель параллельных вычислений в языке Go, основанную на взаимодействующих последовательных процессах (Communicating Sequential Processes, CSP);
- параллельную обработку с помощью сопрограмм;
- использование пакета `sync` для организации ожидания и блокировок;
- обмен сообщениями между сопрограммами посредством каналов;
- корректное закрытие каналов.

В следующих главах мы рассмотрим применение сопрограмм и каналов на практике. Вы увидите, как веб-сервер на языке Go запускает новую сопрограмму для обработки каждого запроса, и познакомитесь с такими шаблонами, как распределение рабочей нагрузки между несколькими каналами. Если что-то и выделяет язык Go в ряду других языков, так это прежде всего его модель параллельных вычислений.

А теперь обратимся к обработке ошибок. Хотя это не самая гламурная тема, тем не менее обработка ошибок в языке Go является одной из его важных особенностей.

Часть II

НАДЕЖНЫЕ ПРИЛОЖЕНИЯ

Вторая часть посвящена решению проблем, когда что-то может пойти не так, как было запланировано. Что делать в случае аварии приложения? Какие имеются средства тестирования, позволяющие обнаружить проблемы, прежде чем они проявятся в процессе эксплуатации? Глава 4 посвящена ошибкам и авариям, точнее их обработке. Это особенно важно при обработке аварий и ошибок в сопрограммах. Глава 5 описывает порядок отладки и тестирования, включая журналирование проблем, чтобы можно было выполнить отладку в процессе производственной эксплуатации.

Глава 4

Обработка ошибок и аварий

В этой главе рассматриваются следующие темы:

- *идиомы обработки ошибок в языке Go;*
- *вывод осмысленных сведений об ошибках;*
- *определение пользовательских типов ошибок;*
- *обработка аварийных ситуаций;*
- *трансформация аварий в ошибки;*
- *обработка аварий в сопрограммах.*

Как лихо выразился в своей поэме «К мыши» Роберт Бернс (Robert Burns): «рушатся самые хитроумные планы мышей и людей». Даже самый лучший план не гарантирует успеха. Ни один из представителей других профессий не познал эту истину так глубоко, как разработчики программного обеспечения. Эта глава посвящена обработке ситуаций, когда все пошло наперекосяк.

Язык Go различает ошибки и аварии – два вида неприятностей, возникающих в ходе выполнения программы. *Ошибка* указывает, что конкретная задача не может быть успешно завершена. *Авария* соответствует попаданию в тяжелую ситуацию, вероятнее всего, возникшую в результате ошибки программиста. Эта глава подробно рассматривает оба вида.

Начнем с ошибок. После краткого знакомства с идиомами обработки ошибок в языке Go будут представлены практические приемы их обработки. Ошибки могут сообщать разработчикам, что пошло не так, и если их правильно обрабатывать, сведения о них помогут восстановить нормальное продолжение выполнения. Приемы обработки ошибок в языке Go отличаются от используемых в таких языках, как

Python, Java и Ruby. Но при правильном их использовании можно добиться надежной работы кода.

Система аварий в Go сигнализирует об аномальных ситуациях, несущих угрозу целостности программы. Поскольку сталкиваться с ними приходится нечасто, основное внимание в этой главе будет уделено такому аспекту, как восстановление после аварий. Вы узнаете, когда использовать аварии, как (и когда) следует проводить восстановительные мероприятия, чем механизмы аварий и ошибок в языке Go отличаются от подобных им механизмов в других языках.

Систему ошибок в языке Go иногда критикуют за ее избыточность, но, как будет показано в этой главе, такая система способствует созданию надежного программного обеспечения. Сосредоточивая внимание разработчиков на возможных ошибках, Go борется с их излишней самоуверенностью. Разработчики всегда склонны верить, что написанный ими код не содержит ошибок. Но повышенное внимание, уделяемое ошибкам в языке Go, заставляет принять оборонительные меры, независимо от того, насколько хорошим кажется код.

4.1. Обработка ошибок

Обработка ошибок – одна из идиом языка Go, которая часто сбивает с толку новичков. Многие популярные языки, в том числе Python, Java и Ruby, следуют теории обработки исключений, предусматривающей возбуждение и перехват специальных объектов исключений. Другие языки, такие как C, обычно используют для обработки ошибок возвращаемое значение, а управление изменением данных осуществляют с помощью указателей.

Создатели языка Go заменили обработчики исключений возможностью возвращать несколько значений. Наиболее часто для передачи информации об ошибках в языке Go используется последнее возвращаемое значение, как показано в следующем листинге.

Листинг 4.1 ❖ Возврат ошибки

```
package main

import (
    "errors"  | Полезные утилиты для работы с ошибками и строками
    "strings" |
)

func Concat(parts ...string) (string, error) { ← Функция Concat возвращает
    if len(parts) == 0 {                          строку и ошибку
```

```
        return "", errors.New("No strings supplied")
    }
    return strings.Join(parts, " "), nil
}
```

Вернуть ошибку, если ничего не было передано

← Вернуть новую строку и значение nil

Функция `Concat` принимает произвольное количество строк, объединяет их (разделяя пробелами) и возвращает вновь созданную строку. Но если не передается ни одной строки, она возвращает ошибку.

Списки аргументов переменной длины

Как демонстрирует функция `Concat`, язык Go поддерживает списки аргументов переменной длины. Префикс `...` перед типом данных указывает, что функция принимает произвольное количество аргументов этого типа. Язык Go свернет их в массив этого типа. В листинге 4.1 аргумент `parts` интерпретируется как `[]string`.

Объявление функции `Concat` демонстрирует типичный шаблон возврата ошибок. Во всех идиомах языка Go ошибка всегда является последним возвращаемым значением.

Поскольку ошибки всегда возвращаются в конце списка возвращаемых значений, обработка ошибок в Go следует определенному шаблону. Функция, возвращающая ошибку, обертывается оператором `if/else`, проверяющим неравенство ошибки значению `nil` и обеспечивающим обработку ошибки, если это условие выполняется. В следующем листинге демонстрируется простая программа, принимающая и объединяющая список аргументов из командной строки.

Листинг 4.2 ❖ Обработка ошибок

```
func main() {
    args := os.Args[1:]
    if result, err := Concat(args...); err != nil {
        fmt.Printf("Error: %s\n", err)
    } else {
        fmt.Printf("Concatenated string: '%s'\n", result)
    }
}
```

Использовать только аргументы, следующие за Args[0].

← В первом аргументе передается имя программы

← Обработка ошибок

← Вывод результатов в отсутствие ошибок

Если запустить эту программу с аргументами, она выведет следующее:

```
$ go run error_example.go hello world
Concatenated string: 'hello world'
```

А если запустить ее без аргументов, она выведет сообщение об ошибке:

```
$ go run error_example.go
Error: No strings supplied
```

Листинг 4.2 демонстрирует использование созданной выше функции `Concat` и иллюстрирует общепринятые идиомы языка Go. Как вы наверняка помните, оператор `if` в языке Go позволяет выполнить присваивание перед условным выражением. Его цель – подготовить данные для оценки, область видимости которых ограничивается областью оператора `if/else`. Эту строку можно прочитать так: `if ПОДГОТОВКА; ОЦЕНКА УСЛОВИЯ`.

Листинг 4.2 демонстрирует этот прием в действии. Сначала выполняется вызов функции `Concat(args...)`, в котором происходит развертывание массива `args`, как если бы был сделан вызов `Concat(arg[0], arg[1],...)`. Два возвращаемых значения присваиваются переменным `result` и `err`. Затем, в этой же строке, поверяется неравенство `err` значению `nil`. Если `err` содержит что-то другое, значит, произошла ошибка и следует вывести сообщение об ошибке.

Важно отметить, что область видимости присвоенных значений распространяется на операторы `else` и `else if`, то есть переменная `result` будет доступна в инструкции вывода.

Такая область видимости определяет полезность операторов `if` из двух частей. Его применение положительно сказывается на управлении памятью и вместе с тем исключает шаблон `if a = b`, превращающий отладку в кошмар.

В листинге 4.1 была представлена функция `Concat`, а в листинге 4.2 демонстрируется ее использование. Но существует еще один прием, использованный в этом примере, но не представленный явно.

РЕЦЕПТ 16 Минимизация использования значений nil

Значения `nil` вызывают раздражение по нескольким причинам. Они часто являются причиной ошибок, и разработчики, как правило, вынуждены выполнять проверку значений, чтобы защититься от значений `nil`.

В некоторых случаях значения `nil` используются, чтобы сообщить о чем-то конкретном. Как можно заметить в примере выше, когда возвращаемое значение ошибки равно `nil`, его можно интерпретировать вполне осмысленно, в частности: «при выполнении функции ошибок

не возникло». Но во многих других случаях смысл значения nil остается неясным. И больше всего раздражают случаи, когда значения nil рассматриваются как заполнитель там, где разработчик затрудняется с выбором возвращаемого значения. В таких ситуациях можно использовать описываемый далее прием.

Проблема

Возвращение в результате значения nil вместе с ошибкой не всегда желательно. Это добавляет работы другим разработчикам, не несет полезной информации и затрудняет восстановление после ошибки.

Решение

Когда это имеет смысл, следует воспользоваться мощной возможностью языка Go возвращать несколько значений и вернуть не только ошибку, но и полезное значение.

Обсуждение

Этот прием иллюстрируется на примере рассмотренной ранее функции Concat. Взглянем еще раз на строку, где возвращается ошибка.

Листинг 4.3 ❖ Возвращение полезных данных вместе с ошибкой

```
func Concat(parts ...string) (string, error) {
    if len(parts) == 0 {
        return "", errors.New("No strings supplied") ← Возврат пустой строки
    }                                                  и ошибки
    return strings.Join(parts, " "), nil
}
```

Обнаружив ошибку, функция возвращает пустую строку и сообщение об ошибке. Предыдущий код избавляет внимательного пользователя от необходимости писать много кода для явной обработки ошибок. В данном случае возврат пустой строки не лишен смысла. Если данные для объединения отсутствуют, а описание возвращаемого значения утверждает, что функция возвращает строку, пустая строка в данном случае является именно тем, что следовало бы ожидать.

> **СОВЕТ** В языке Go имеются две функции, которые удобно использовать для создания ошибок. Функция errors.New из пакета errors хорошо подходит для создания простых ошибок. Функция fmt.Errorf из пакета fmt дает возможность использовать строку формата при конструировании сообщения об ошибке. Go-разработчики часто используют эти две функции.

Такая переделка функции Concat пойдет на пользу использующим ее. Пользователь функции, не заботящийся об ошибках, сможет упростить свой код, как показано в следующем листинге.

Листинг 4.4 ❖ На основании хорошей обработки ошибок

```
func main() {
    args := os.Args[1:]
    result, _ := Concat(args...)  ← Передача пакета значений
    fmt.Printf("Concatenated string: '%s'\n", result)
}
```

Как и прежде, этот код извлекает аргументы командной строки и передает их в функцию Concat. Но теперь отпала необходимость обертывать вызов функции Concat оператором if для обработки ошибки. Функция Concat реализована так, что возвращает полезное значение даже в случае ошибки, поэтому наличие или отсутствие ошибки не влияет на решение основной задачи и можно избежать дополнительной проверки. Вместо проверки вызова в блоке if/else ошибка просто игнорируется, и результат обрабатывается как строка.

Этот прием также помогает, когда ситуация требует определять факт ошибки и реагировать на нее. В этом случае можно получить значение ошибки и выяснить, что пошло не так и почему, поскольку функция по-прежнему возвращает ошибку и полезное значение, что упрощает реализацию ее обработки.

Но порой нежелательно, а иногда и невозможно возвращать вместе с ошибкой значения, отличные от nil. Если в случае ошибки нельзя вернуть полезные данные, предпочтение следует отдать значению nil. Проще говоря, если функция может вернуть полезный результат в случае ошибки, она должна вернуть его, в противном случае следует возвращать значение nil.

Наконец, очень важно, чтобы поведение вашего кода было понятно другим разработчикам. Язык Go справедливо подчеркивает необходимость написания краткого, но полезного комментария перед каждой общей функцией. Документирование поведения функции Concat должно выглядеть, как показано в следующем листинге.

Листинг 4.5 ❖ Описание возвращаемого значения в случае ошибки

```
// Функция Concat объединяет строки, разделяя их пробелами.
// Она возвращает пустую строку и ошибку, если не получила ни одной строки.
func Concat(parts ...string) (string, error) {
    //...
}
```

Этот краткий комментарий соответствует соглашению языка Go и поясняет, что произойдет при нормальном выполнении функции, а также что получится в случае ошибки.

Разработчикам с опытом программирования на таких языках, как Java или Python, система обработки ошибок может показаться примитивной. В ней отсутствуют специальные блоки `try/catch`. Вместо этого предлагается использовать операторы `if/else`. Большинство возвращаемых ошибок является ошибками с типом `error`. Новичков в Go часто беспокоит кажущаяся неуклюжей обработка ошибок.

Эти проблемы исчезают после привыкания к особенностям языка Go. Выбор в пользу соглашений окупается простотой написания и чтения кода. Следует отметить одну особенность языка Go: в то время как разработчики, пишущие на языках, таких как Java и Python, отдают предпочтение специальным типам ошибок и исключений, Go-разработчики редко создают типы конкретных ошибок.

Это, несомненно, связано с тем, что большинство основных библиотек в Go использует готовые типы ошибок. С точки зрения Go-разработчиков, большинство ошибок не имеет специальных атрибутов, отличающих специальные типы ошибок. Как следствие возврат обобщенных ошибок упрощает их обработку. Возьмем, к примеру, функцию `Concat`. Создание типа `ConcatError` для этой функции не дает никаких преимуществ, поэтому вполне можно использовать встроенный пакет `errors`.

Такая упрощенная обработка ошибок часто оказывается самой удобной. Но иногда бывает полезно создать и использовать специальный тип ошибок.

РЕЦЕПТ 17 Специальные типы ошибок

Тип данных `error` в языке Go – это интерфейс, определение которого приводится в следующем листинге.

Листинг 4.6 ❖ Интерфейс `error`

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

Все, что содержит функцию `Error`, должно возвращать строку, как того требует этот интерфейс. Обычно для работы с ошибками Go-разработчикам бывает достаточно типа `error`. Но иногда может потребоваться передавать в ошибках дополнительные сведения, кроме

обычной строки. В таких случаях можно создать специализированный тип ошибок.

Проблема

Функция возвращает ошибку. Детальные сведения о ней играют важную роль и определяют порядок дальнейшей обработки.

Решение

Создадим тип данных, реализующий интерфейс `error` и предоставляющий дополнительные возможности.

Обсуждение

Предположим, что требуется написать парсер файлов. Когда парсер обнаруживает синтаксическую ошибку, он генерирует ошибку. Вместе с сообщением об ошибке обычно полезно иметь информацию о местоположении ошибки в файле. Следующий листинг показывает, как это можно реализовать.

Листинг 4.7 ❖ Тип данных ParseError

```
type ParseError struct {
    Message     string  ← Сообщение об ошибке без информации о ее местоположении
    Line, Char int  ← Информация о местоположении
}

func (p *ParseError) Error() string {
    format := "%s on Line %d, Char %d"
    return fmt.Sprintf(format, p.Message, p.Line, p.Char)
}
```

Реализация интерфейса Error

Новая структура `ParseError` имеет три свойства: `Message`, `Line` и `Char`. Реализация функции `Error` с помощью форматирования размещает все три части информации в одной строке. Но предположим, что требуется вернуться к источнику ошибки синтаксического анализа и вывести строку, выделив в ней символ, вызвавший сомнения. Структура `ParseError` позволяет это сделать.

Этот прием хорошо подходит для возврата дополнительной информации. А что делать, если требуется, чтобы одна и та же функция возвращала различные типы ошибок?

РЕЦЕПТ 18 Переменные ошибок

Иногда требуется написать функцию, решающую сложную задачу, в которой может встретиться несколько различных ошибок. Предыдущий рецепт демонстрирует решение, основанное на реализации

интерфейса `error`, но оно оказывается слишком громоздким, если не все ошибки нуждаются в дополнительной информации. Рассмотрим другое идиоматическое использование ошибок Go.

Проблема

Одна сложная функция может столкнуться с несколькими видами ошибок. И было бы хорошо дать пользователям функции возможность определять типы ошибок, чтобы соответствующим образом обрабатывать их. В функции могут возникать ошибки разных видов, но ни с одной из них не требуется передавать дополнительную информацию (как в рецепте 17).

Решение

В языке Go (в отличие от других языков программирования) хорошей практикой считается создавать переменные для хранения ошибок на уровне пакетов, которые можно возвращать при возникновении определенных ошибок. Отличным примером может служить пакет `io` из стандартной библиотеки языка Go, содержащий такие ошибки, как `io.EOF` и `io.ErrNoProgress`.

Обсуждение

Прежде чем углубиться в детали использования переменных ошибок, рассмотрим проблему и ее очевидное, но не самое лучшее решение.

Наша цель – дать возможность различать две ошибки. Рассмотрим небольшую программу, имитирующую отправку простого сообщения получателю.

Листинг 4.8 ❖ Обработка двух различных ошибок

```
package main

import (
    "errors"
    "fmt"
    "math/rand"
)

var ErrTimeout = errors.New("The request timed out")
var ErrRejected = errors.New("The request was rejected")

var random = rand.New(rand.NewSource(35))
func main() {
    response, err := SendRequest("Hello")
```

Создание экземпляра ошибки превышения времени ожидания

Создание экземпляра ошибки отказа

← Генерация случайных чисел с помощью фиксированного источника

← Вызов функции-заглушки SendRequest

```
    for err == ErrTimeout {                          Обработка превышения времени
        fmt.Println("Timeout. Retrying.")            ожидания повторением попытки
        response, err = SendRequest("Hello")
    }
    if err != nil {                                  Обработка любой другой ошибки как сбоя
        fmt.Println(err)
    } else {                                         Если нет ошибок, напечатать результат
        fmt.Println(response)
    }
}

func SendRequest(req string) (string, error) {   ← Определение функции-заглушки,
    switch random.Int() % 3 {                        имитирующей отправку сообщений
    case 0:
        return "Success", nil
    case 1:                                          Имитировать отправку
        return "", ErrRejected                       сообщений, случайно
    default:                                         выбирая разное
        return "", ErrTimeout                        поведение
    }
}
```

Этот листинг демонстрирует использование переменных в роли фиксированных ошибок. Код имитирует отправку сообщений. Функция-заглушка `SendRequest` генерирует случайный ответ. Ответ может быть признаком успешной отправки или одной из двух ошибок: `ErrTimeout` и `ErrRejected`.

Не совсем случайное

Одной из интересных деталей примера в листинге 4.7 является генератор случайных чисел. Поскольку генератор случайных чисел инициализируется фиксированным значением, он всегда будет возвращать одну и ту же последовательность «случайных» чисел. В данном случае это удобно, поскольку позволяет проиллюстрировать работу примера. Но в промышленных приложениях не следует использовать фиксированные целые числа для инициализации источника. Простой альтернативой является использование в этом качестве `time.Now`.

Если запустить эту программу, она выведет следующее:

```
$ go run two_errors.go
Timeout. Retrying.
The request was rejected
```

Первый вызов `SendRequest` вернул ошибку превышения время ожидания, а второй – отказ. Если бы второй вызов вернул превышение времени ожидания, программа продолжила бы работу, пока `SendRequest` не вернула бы признак успеха или отказ. Этот прием часто применяется в сетевых серверах.

Разработчики программного обеспечения, пишущие, например, на языке Java или Python, реализовали бы `ErrTimeout` и `ErrRejected` как классы и вернули новые экземпляры этих классов. Шаблон `try/catch` используется во многих языках программирования для работы со сведениями об ошибках, инкапсулированных в типы данных. Но, как отмечалось выше, в Go нет блоков `try/catch`. Для реализации тех же возможностей можно использовать сопоставление типов данных (обычно с помощью оператора `switch`). Но это не соответствует соглашениям, принятым в языке Go. Вместо этого в Go используется более эффективный и простой метод: создание переменных ошибок с областью видимости внутри пакета и ссылка на эти переменные.

Как показывает предыдущий пример, работа с переменными ошибок так же проста, как проверка на равенство. Если возникла ошибка превышения времени ожидания, происходит повторная отправка сообщения. Но когда возникает ошибка, связанная с отказом, обработка завершается. Возврат значения `nil`, как и прежде, указывает, что ошибок не возникло и выполнение можно продолжать, как обычно. В этом сценарии одни и те же переменные ошибок используются многократно. Это эффективно, поскольку ошибки создаются только один раз. Кроме того, это концептуально просто. Пока ошибкам не понадобятся специальные свойства (как демонстрировалось в рецепте 16), таких переменных вполне достаточно.

Еще один аспект обработки ошибок в Go, который следует рассмотреть: обработка аварий.

4.2. Система аварий

В дополнение к описанной выше обработке ошибок язык Go поддерживает еще один способ сообщить, что что-то пошло не так: систему аварий. Как следует из названия, аварии сообщают о серьезных неполадках. Их следует использовать осторожно и аккуратно. В этом разделе мы расскажем, как и когда следует использовать аварии, и попутно приведем их примеры.

4.2.1. Отличия аварий от ошибок

Первое, что следует знать об авариях, – чем они отличаются от ошибок. Ошибка сообщает, что произошло событие, не соответствующее нашим ожиданиям. Авария же говорит о наличии таких неполадок, из-за которых система (или конкретная подсистема) не может продолжать функционировать.

Среда выполнения Go предполагает, что ошибки обрабатываются программистом. Если он решит проигнорировать ошибку, среда выполнения никак ему не воспрепятствует. Иначе обстоит дело с авариями. В случае аварии среда выполнения Go просматривает стек, пытаясь найти для нее обработчика. Если обработчик не найден, среда Go остановит программу. Необработанные аварии прекращают работу приложения.

Следующий листинг иллюстрирует различия.

Листинг 4.9 ❖ Ошибки и аварии

```
package main

import (
    "errors"
    "fmt"
)

var ErrDivideByZero = errors.New("Can't divide by zero")

func main() {
    fmt.Println("Divide 1 by 0")
    _, err := precheckDivide(1, 0)
    if err != nil {
        fmt.Printf("Error: %s\n", err)
    }
    fmt.Println("Divide 2 by 0")
    divide(2, 0)
}

func precheckDivide(a, b int) (int, error) {
    if b == 0 {
        return 0, ErrDivideByZero
    }
    return divide(a, b), nil
}

func divide(a, b int) int {
    return a / b
}
```

Первое деление выполняет функция precheckDivide, возвращающая ошибку

Затем выполняется такое же деление, но с помощью функции divide

← Функция precheckDivide возвращает ошибку при делении на 0

← Обычная функция divide просто выполняет операцию деления без проверок

Здесь определены две функции. Функция `divide` выполняет операцию деления. Она не обрабатывает попытку деления на 0. Функция `precheckDivide`, напротив, явно проверяет делитель и возвращает ошибку, если он равен 0. Нас интересует поведение среды выполнения Go в двух этих ситуациях, поэтому сначала вызывается функция `precheckDivide`, а затем обычная функция `divide`.

Результат выполнения этой программы:

```
go run zero_divider.go
Divide 1 by 0
Error: Can't divide by zero
Divide 2 by 0
panic: runtime error: integer divide by zero
[signal 0x8 code=0x7 addr=0x22d8 pc=0x22d8]

goroutine 1 [running]:
main.main()
    /Users/mbutcher/Code/go-in-practice/chapter4/zero_divider.go:18   +0x2d8
```

Первое деление с помощью функции `precheckDivide` вернет ошибку, а второе приведет к аварии, поскольку в этом случае допустимость делителя не проверяется. Такое поведение объясняется следующими важными причинами:

- когда проверка делителя выполняется до деления, от программы не требуется ничего, что она не в состоянии сделать;
- при делении система оказалась в состоянии, с которым не смогла справиться. То есть произошла авария.

С практической точки зрения, ошибки – это некие проблемы, появление которых было предусмотрено разработчиками. Они уже задокументированы в коде. Определение функции `precheckDivide` содержит указание на ошибку, которую необходимо обработать при вызове функции. Хотя ошибки представляют собой нечто, выходящее за пределы нормы, но *неожиданными* их назвать нельзя.

Аварии, напротив, всегда происходят неожиданно. Они возникают, когда непредсказуемым образом нарушено правило или превышен предел. Что касается объявления аварий в коде: их принято использовать, только когда нет другого способа обработать возникшую ситуацию в данном контексте. Всегда, где только возможно, следует возвращать ошибки.

4.2.2. Работа с авариями

Go-разработчики имеют свои представления о том, как правильно возбудить аварию, хотя эти представления не всегда четко изложены.

Перед тем как погрузиться в вопросы правильной обработки аварий, рассмотрим прием правильного возбуждения аварии, который должны знать все Go-разработчики.

РЕЦЕПТ 19 Возбуждение аварий

Определение функции, возбуждающей аварию, можно выразить так: `panic(interface{})`. В вызов функции `panic` можно передать практически любой аргумент. При желании ее можно вызвать с аргументом `nil`, как это сделано в следующем листинге.

Листинг 4.10 ❖ Авария с nil

```
package main

func main() {
     panic(nil)   ← Авария ни с чем!
}
```

После запуска этой программы среда выполнения Go перехватит аварию:

```
$ go run proper_panic.go
panic: nil
goroutine 1 [running]:
main.main()
    /Users/mbutcher/Code/go-in-practice/chapter4/proper_panic.go:4   +0x32
```

Данная ошибка не несет никакой полезной нагрузки. Аварию можно вызвать со строкой: `panic("Oops, I did it again.")`. Тогда результат получится немного иным:

```
$ go run proper_panic.go
panic: Oops, I did it again.
goroutine 1 [running]:
main.main()
    /Users/mbutcher/Code/go-in-practice/chapter4/proper_panic.go:4   +0x64
```

Уже лучше. По крайней мере, имеется хоть какая-то полезная информация. Но правильно ли так делать?

Проблема

Что следует передавать функции `panic` в качестве аргумента? Существуют ли полезные или идиоматически общепринятые способы возбуждения аварий?

Решение

Лучше всего в функцию `panic` передавать ошибки. Используйте тот тип ошибок, который упростит восстановление (если предусмотрено).

Обсуждение

Сигнатура функции с аргументом типа `interface{}` не определяет явно, что передавать ей. Можно, например, передать объект, вызвавший аварию. Как вы уже видели, можно передать строку, значение `nil` или ошибку.

Лучше всего передать (по крайней мере, при нормальных обстоятельствах) то, что реализует интерфейс `error`. На то есть две веские причины. Первая – очевидность. Что вызывает аварии? Ошибки. Разумно предположить, что разработчики ожидают именно этого. Вторая причина заключается в том, что это облегчает обработку аварий. Вы убедитесь в этом, когда мы начнем рассматривать приемы восстановления после аварии, – вы узнаете, как перехватить аварию, устранить ее последствия, а затем воспользоваться содержимым аварии, как это делается при работе со старой доброй ошибкой.

Итак, следующий листинг демонстрирует идиоматический способ возбуждения аварии.

Листинг 4.11 ❖ Правильное возбуждение аварии

```
package main

import "errors"
func main() {
    panic(errors.New("Something bad happened."))  ← Вызов функции panic
}                                                    с передачей ошибки
```

С таким подходом все еще можно вывести форматированное сообщение: `fmt.Printf("Error: %s", thePanic)`, а также достаточно просто послать аварию через систему ошибок. Поэтому передача ошибок в аварии считается идиоматически правильной.

Размышления, выходящие за рамки аварий

Мы только что заявили, что лучше всего в аварии передавать ошибки. Но аварии не ограничиваются только типами ошибок, и это придает системе аварий дополнительную гибкость. В некотором смысле это напоминает возможность возбуждать исключения в Java с наследниками класса `Trowable`, которые не обязательно являются исключениями. Аварии в Go являются одним из способов избежать раскручивания стека.

Мы никогда не видели иные способы возбуждения аварий, кроме тех, что представлены здесь. Но это не значит, что не существует по-

лезных, но неидиоматичных способов передачи в аварии других типов, не являющихся ошибками.

Теперь, получив представление, как правильно возбуждать аварии, займемся другой стороной этого вопроса – восстановлением после них.

4.2.3. Восстановление после аварий

Любое обсуждение аварий будет неполным без упоминания восстановления после аварий. Восстановление после аварий в Go основывается на одной из особенностей языка, называемой *отложенными вызовами функций*. Среда выполнения Go имеет возможность гарантировать выполнение функции в момент возврата из родительской функции. Это происходит независимо от того, что является причиной возврата – выполнение оператора `return`, достижение конца функции или авария. Поясняет это следующий код.

Листинг 4.12 ❖ Простой отложенный вызов

```
package main

import "fmt"

func main() {
    defer goodbye()   ← Отложенный вызов функции goodbye
    fmt.Println("Hello world.")   ← Вывод строки. Выполняется
}                                   до вызова функции goodbye

func goodbye() {
    fmt.Println("Goodbye")
}
```

Без оператора `defer` эта программа вывела бы сначала `Goodbye`, а затем `Hello world`. Но оператор `defer` меняет порядок выполнения. Он откладывает выполнение функции `goodbye` до завершения всех прочих операторов в функции `main`. Сразу после завершения `main` выполняется отложенный вызов функции `goodbye`. В результате программа выведет следующее:

```
$ go run simple_defer.go
Hello world.
Goodbye
```

Оператор `defer` – отличное средство для закрытия файлов и сокетов перед завершением работы, освобождения ресурсов, таких как дескрипторы базы данных, или обработки аварий. В рецепте 19 был по-

казан способ возбуждения аварий. Теперь, вооружившись знаниями об операторе `defer`, сосредоточимся на восстановлении после аварий.

РЕЦЕПТ 20 Восстановление после аварий

Перехват аварии в отложенных функциях – стандартная практика в языке Go. Этот прием рассматривается здесь по двум причинам. Во-первых, его обсуждение подготавливает переход к другим рецептам. Во-вторых, он дает возможность сделать шаг от шаблона к механизму и увидеть, что происходит, вместо заучивания формулы восстановления после аварии.

Проблема

Авария в одной из функций приводит к краху всей программы.

Решение

Используем отложенную функцию и вызовем функцию `recover`, чтобы выяснить причину и обработать аварию.

Схема слева на рис. 4.1 иллюстрирует, как необработанная авария ведет к краху программы. Схема справа демонстрирует, как функция `recover` останавливает обход стека вызовов функций и позволяет программе продолжить выполнение.

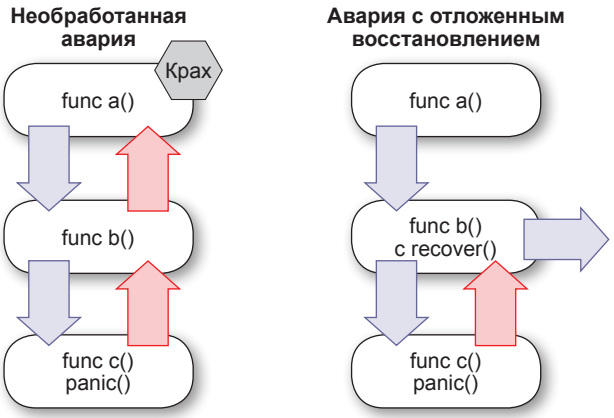


Рис. 4.1 ❖ *Восстановление после аварии*

Обсуждение

В языке Go имеется возможность получить информацию об аварии и остановить дальнейшее раскручивание стека вызовов. Данные извлекаются с помощью функции `recover`.

Рассмотрим следующий листинг, который демонстрирует, как возбудить и обработать аварию.

Листинг 4.13 ❖ Восстановление после аварии

```
package main

import (
    "errors"
    "fmt"
)

func main() {
    defer func() {
          if err := recover(); err != nil {
                fmt.Printf("Trapped panic: %s (%T)\n", err, err)
          }
    }()
    yikes()
}

func yikes() {
    panic(errors.New("Something bad happened."))
}
```

Подготовка отложенного вызова замыкания для восстановления после аварии

← Вызов функции, возбуждающей аварию

Возбуждение аварии ← с передачей ей ошибки

Эта программа демонстрирует самый распространенный сценарий восстановления после аварии. Для перехвата аварии, возбуждаемой функцией `yikes`, написана отложенная функция-замыкание, проверяющая наличие аварии и при необходимости выполняющая восстановление.

Чтобы создать отложенное замыкание, нужно определить функцию и отметить ее как предназначенную для отложенного вызова (в данном случае с пустым списком аргументов). С этой целью часто используется форма записи: `defer func(){ /*тело*/ }()`. Обратите внимание, что хотя она выглядит как определяемая и сразу же вызываемая функция, среда выполнения Go вызовет функцию, только когда наступит соответствующий момент. Чуть ниже вы увидите, в чем польза определения отложенной функции-замыкания.

Функция `recover` возвращает значение (`interface{}`), если авария имела место, и значение `nil` в противном случае. Возвращаемое ею значение – это значение в вызов `panic`. Если выполнить предыдущий пример, он выведет следующее:

```
$ go run recover_panic.go
Trapped panic: Something bad happened. (*errors.errorString)
```

Обратите внимание, что на месте спецификатора %T в строке форматирования будет выведена информация о типе err, который в данном случае является типом ошибки, созданной вызовом errors.New.

А теперь сделаем еще шаг и посмотрим, как использовать комбинацию замыкания и функции recover для восстановления после аварии. Замыкания наследуют область видимости своих родителей. Отложенные замыкания, как в предыдущем примере, наследуют все, что находилось в области видимости перед их объявлением. Например, следующий код:

Листинг 4.14 ❖ Область видимости для отложенных замыканий

```
package main

import "fmt"

func main() {
    var msg string  ← Определение переменной вне замыкания
    defer func() {
        fmt.Println(msg)  ← Вывод значения переменной в отложенном замыкании
    }()
    msg = "Hello world"  ← Присваивание значения переменной
}
```

выполнится без ошибок, потому что переменная msg объявлена до замыкания и на нее можно сослаться внутри замыкания. И, как ожидалось, сообщение отражает состояние переменной msg, имевшее место на момент выполнения отложенной функции. Предыдущий код выведет Hello world.

Но даже при том, что отложенная функция выполняется после всех прочих операторов функции, замыкание не имеет доступа к переменным, объявленным до него. Область видимости замыкания определяется во время компиляции, но выполняется оно, только когда функция завершится. По этой причине следующее объявление вызовет ошибку компиляции.

Листинг 4.15 ❖ Переменная msg вне области видимости

```
package main

import "fmt"

func main() {
    defer func() {
        fmt.Println(msg)  ← Вывод значения переменной
    }()
```

```
    msg := "Hello world"  ← Объявление переменной с присваиванием значения.
}                           Компиляция приведет к ошибке, поскольку это объявление
                            находится ниже отложенной функции
```

Поскольку переменная `msg` объявлена после отложенной функции, во время ее компиляции переменная `msg` еще не определена.

Объединив все вместе, можно сделать последний шаг. Рассмотрим более сложное использование отложенной функции. Она обрабатывает аварию, выполняет очистку перед возвратом и служит хорошим примером использования отложенных функций и функций восстановления.

Предположим, что требуется написать код для удаления пустых строк в начале CSV-файла. Для простоты мы напишем не полную реализацию функции `RemoveEmptyLines`, а только ее имитацию, которая всегда возбуждает аварию. Это даст возможность показать, как организовать восстановление после аварии, закрыть файл и вернуть ошибку.

Листинг 4.16 ❖ Очистка

```
package main

import (
    "errors"
    "fmt"
    "io"
    "os"
)

func main() {
    var file io.ReadCloser
    file, err := OpenCSV("data.csv")
    if err != nil {
        fmt.Printf("Error: %s", err) return
    }
    defer file.Close()

    // Что-то делается с файлом.
}

func OpenCSV(filename string) (file *os.File, err error) {
    defer func() {
        if r := recover(); r != nil {
            file.Close()
            err = r.(error)
        }
    }()
```

Вызов функции OpenCSV и обработка всех ошибок. Эта реализация всегда возвращает ошибку

Использовать отложенную функцию, чтобы гарантировать закрытие файла при любых обстоятельствах

← Обычно здесь помещается код для работы с файлом

Отложенная обработка возникшей ошибки с передачей ее в функцию main

```
    file, err = os.Open(filename)
    if err != nil {
        fmt.Printf("Failed to open file\n")
        return file, err
    }
    RemoveEmptyLines(file)
    return file, err
}

func RemoveEmptyLines(f *os.File) {
    panic(errors.New("Failed parse"))
}
```

Открытие файла с данными и обработка любых ошибок (таких как "файл не найден")

← Вызов функции RemoveEmptyLines, постоянно вызывающей аварии

← Вместо удаления пустых строк всегда возбуждает аварию

И снова проблема данного примера заключается в том, что функция `RemoveEmptyLines` всегда возбуждает аварию. В полной реализации функции мы могли бы проверить наличие пустых строк в начале файла и при необходимости пропустить их.

Пример в листинге 4.16 использует отложенные функции в двух местах. Отложенная функция в функции `main` гарантирует закрытие файла. Считается хорошей практикой в обязательном порядке закрывать файлы, сетевые подключения, дескрипторы баз данных и другие ресурсы, требующие закрытия, для предотвращения побочных эффектов и утечек. Вторая отложенная функция находится внутри `OpenCSV`. Эта отложенная функция преследует три цели:

- перехват всех аварий;
- гарантированное закрытие файла в случае аварии. Это также считается хорошей практикой, даже при том, что в данном контексте может показаться излишним;
- извлечение ошибки из аварии и передача ее обычному механизму обработки ошибок.

Стоит упомянуть еще одну особенность функции `OpenCSV`, а именно – определение возвращаемых значений в объявлении функции. Это позволяет ссылаться на переменные `file` и `err` внутри замыкания и гарантирует, что при присваивании переменной `err` ошибки из аварии функция `OpenCSV` вернет правильное значение.

Как уже было показано, оператор `defer` – мощное и удобное средство для обработки аварий, а также для устранения их последствий. В заключение описания этого приема дадим несколько полезных рекомендаций по работе с отложенными функциями:

- объявляйте отложенные функции как можно ближе к началу вмещающей функции;
- простые объявления, такие как `foo: = 1`, часто помещаются перед отложенными функциями;

- более сложные переменные объявляются перед отложенными функциями (`var myFile io.Reader`), но инициализируются после них;
- внутри функции допускается объявлять несколько отложенных функций, но такая практика не приветствуется;
- закрывайте файлы, сетевые подключения и другие подобные ресурсы в отложенных замыканиях. Это гарантирует освобождение системных ресурсов даже при возникновении ошибок и аварий.

В следующем рецепте мы сделаем еще шаг в обработке аварий и узнаем, как предотвратить остановку программы в случае аварии в сопрограмме.

4.2.4. Аварии и сопрограммы

Мы пока ничего не говорили об одной из самых мощных особенностей языка Go – сопрограммах. Сопрограммы запускаются с помощью ключевого слова `go`, то есть если имеется функция `run`, ее можно запустить как сопрограмму следующим образом: `go run`. В спецификации языка Go указывается, что оператор `go` «запускает указанную функцию в независимом, параллельном потоке выполнения, или *сопрограмме*, в том же адресном пространстве» (http://golang.org/ref/spec#Go_statements). Проще говоря, можно считать, что функция работает в собственном потоке выполнения, но не имеет доступа к этому потоку.

Внутреннее устройство сопрограмм

Реализация механизма сопрограмм несколько сложнее, чем просто запуск функции в отдельном потоке. На вики-странице (https://github.com/golang/go/wiki/LearnConcurrency), посвященной параллельным вычислениям в языке Go, приводится длинный список статей, подробно описывающих различные аспекты параллельных вычислений в языке Go, основанных на модели взаимодействующих последовательных процессов.

Для иллюстрации: предположим, что имеется простой сервер. Этот сервер функционирует следующим образом:

- функция `main` вызывает функцию `start`, выполняющую запуск нового сервера;
- функция `start` обрабатывает конфигурационные данные и вызывает функцию `listen`;

- функция `listen` открывает сетевой порт и приступает к приему запросов. Получив запрос, вместо его обработки она вызывает `go handle`, передавая функции `handle` необходимую информацию;
- функция `handle` обрабатывает запрос и вызывает функцию `response`;
- функция `response` отправляет данные клиенту и закрывает соединение.

Функция `listen` использует сопрограммы для одновременного обслуживания нескольких клиентов. Получив запрос, она распределяет рабочую нагрузку между несколькими функциями `handle`, каждая из которых работает в своем собственном адресном пространстве. Разные варианты этой мощной модели часто используются в серверных Go-приложениях. На рис. 4.2 показаны схема такого приложения и порядок использования сопрограмм в нем.

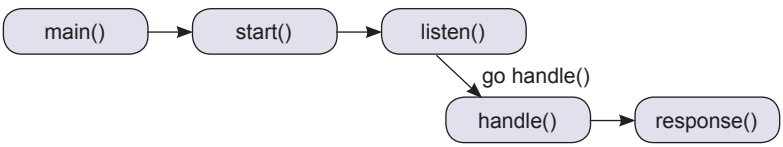


Рис. 4.2 ❖ *Простой сервер*

Каждый ряд в схеме представляет функциональный стек, и каждый вызов `go` создает новый функциональный стек. Получив запрос, функция `listen` создает новый функциональный стек для следующего экземпляра функции `handle`. И всякий раз, когда завершается функция `handle` (например, когда `response` возвращает управление), порожденная сопрограмма удаляется.

Сопрограммы – мощный и элегантный инструмент. Поскольку они просты в написании и дешевы в использовании (имеют небольшие накладные расходы), Go-разработчики активно их используют. Но в одной конкретной (и, к сожалению, часто встречающейся) ситуации сочетание сопрограмм и аварий может приводить к краху программы.

РЕЦЕПТ 21 Перехват аварий в сопрограммах

Когда возникает авария, среда выполнения Go раскручивает стек функции, пока не столкнется с вызовом `recover`. Но если обход завершится на вершине стека и функция `recover` нигде не будет вызвана,

программа прервет работу. Вспомним рис. 4.2, демонстрирующий получение каждой сопрограммой собственного стека вызовов функций. Что случится, если авария произойдет в этой сопрограмме? Взгляните на рис. 4.3. Предположим, что при обработке запроса функция `response` обнаруживает непредвиденную и неустранимую ошибку, приводящую к аварии. Разработчик сервера позаботился о логике обработки всех ошибок в функции `listen`. Но если не добавить еще чего-то в функцию `handle`, произойдет аварийное завершение программы. Почему? Потому что если аварию не обработать до достижения вершины стека функции, среда выполнения Go прекратит выполнение программы. Авария в сопрограмме не может выйти за границы стека функции, созданной сопрограммой. В этом примере отсутствует возможность передачи аварии из функции `handle` в функцию `listen`. Решению этой проблемы посвящен данный рецепт.

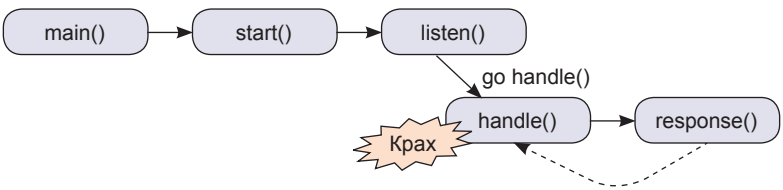


Рис. 4.3 ❖ *Крах сопрограммы*

Проблема

Если авария в сопрограмме остается необработанной, это приводит к краху всей программы. Иногда такое допустимо, но чаще желательно восстановить нормальную работу.

Решение

Самое простое решение – реализовать обработку аварий во всех сопрограммах, где они могут произойти. Наше решение поможет упростить разработку серверов, которые обрабатывают аварии, не полагаясь на их обработку в каждой отдельной функции `handle`.

Обсуждение

Интересный аспект этой конкретной проблемы заключается в том, что, несмотря на кажущуюся тривиальность решения, его реализация оказывается настолько сложной, что разработчик теряет над ней контроль. Начнем с базовой реализации, изображенной на рис. 4.3. Исследуем тривиальное решение и убедимся, что идиомы языка Go

делают такое решение излишне сложным. Затем рассмотрим более удобное решение.

В следующем листинге представлена реализация простого сервера, соответствующая схеме на рис. 4.3. Он функционирует как простой эхо-сервер. К нему можно подключиться и отправить в порт 1026 обычную строку текста. Удаленный сервер вернет переданный текст.

Листинг 4.17 ❖ Эхо-сервер

```
package main

import (
    "bufio"
    "fmt"
    "net"
)

func main() {
    listen()
}

func listen() {
    listener, err := net.Listen("tcp", ":1026")
    if err != nil {
        fmt.Println("Failed to open port on 1026")
        return
    }
    for {
        conn, err := listener.Accept()
        if err != nil {
            fmt.Println("Error accepting connection")
            continue
        }
        go handle(conn)
    }
}

func handle(conn net.Conn) {
    reader := bufio.NewReader(conn)
    data, err := reader.ReadBytes('\n')
    if err != nil {
        fmt.Println("Failed to read from socket.")
        conn.Close()
    }
    response(data, conn)
}
```

← Запуск нового сервера, обслуживающего порт 1026

Прием новых клиентских запросов и обработка ошибок подключения

← При появлении запроса передать его в функцию handle

Попытка чтения строки из подключения

В случае ошибки чтения строки вывести сообщение и закрыть подключение

← При получении строки передать ее в функцию response

```
func response(data []byte, conn net.Conn) {
    defer func() {
        conn.Close()
    }()
    conn.Write(data)
}
```

Записать данные в сокет для отправки клиенту и закрыть соединение

Этот код запускает сервер, с которым можно взаимодействовать, как показано ниже:

```
$ telnet localhost 1026
Trying ::1...
Connected to localhost.
Escape character is '^]'.
test
test
Connection closed by foreign host.
```

После ввода строки `test` (выделена жирным в примере выше) сервер вернул ее и закрыл соединение.

Этот простой сервер принимает запросы от клиентов, посылаемые в порт 1026. Получив новый запрос, сервер запускает сопрограмму, выполняющую функцию `handle`. Поскольку каждый запрос обрабатывается в отдельной сопрограмме, этот сервер может эффективно взаимодействовать со множеством клиентов.

Функция `handle` читает строку текста (необработанные байты), а затем передает ее и дескриптор подключения функции `response`. Функция `response` возвращает текст клиенту и закрывает соединение.

Это не идеальный сервер, но он хорошо иллюстрирует основные понятия. Он также демонстрирует некоторые ловушки. Предположим, что в функции `response` возникла авария. Чтобы смоделировать эту ситуацию, заменим ее предыдущую реализацию следующей.

Листинг 4.18 ❖ Авария в функции `response`

```
func response(data []byte, conn net.Conn) {
    panic(errors.New("Failure in response!"))
}
```

← Вместо выполнения каких-либо полезных действий возбудим аварию

Первое, что бросается в глаза, – эта реализация не закрывает соединение с клиентом в функции `response`, но в действительности ситуация еще хуже: вызов `panic` приведет к краху сервера. Серверы не должны быть настолько хрупкими, чтобы сбой в обработке одного конкретного запроса приводил их к краху. Добавление в `listen` функции, восстанавливающей работу, выглядит естественным решением

проблемы. Но это не поможет, поскольку сопрограммы выполняются, используя отдельный стек.

А сейчас сделаем первую попытку увеличить надежность сервера. Добавим обработку аварий в функцию `handle`. В следующем листинге приводится только код функций `handle` и `response`, поскольку прочий код не изменился.

Листинг 4.19 ❖ Обработка аварий в сопрограмме

```
func handle(conn net.Conn) {
    defer func() {
       if err := recover(); err != nil {
           fmt.Printf("Fatal error: %s", err)
       }
       conn.Close()
    }()
    reader := bufio.NewReader(conn)
    data, err := reader.ReadBytes('\n')
    if err != nil {
        fmt.Println("Failed to read from socket.")
    }
    response(data, conn)
}

func response(data []byte, conn net.Conn) {
    conn.Write(data)
    panic(errors.New("Pretend I'm a real error"))
}
```

Отложенная функция обрабатывает аварию и гарантирует закрытие соединения в любом случае

← И снова возбудить аварию для имитации сбоя

Новая функция `handle` теперь содержит отложенную функцию, которая вызывает `recover`, чтобы выяснить причину аварии. Это остановит распространение аварии вверх по стеку. Обратите внимание на немного улучшенное управление подключениями: в любом случае, независимо от того, что произойдет, соединение будет закрыто после завершения функции `handle`. В такой новой редакции сервер больше не останавливается при аварии в функции `response`.

Пока все в порядке. Но, взяв за основу этот пример, можно внести в функцию `handle` еще одно улучшение, воспользовавшись еще одной идиомой языка Go. В Go принято создавать серверные библиотеки, включающие гибкий метод обработки ответов. Отличным примером может служить библиотека `"net/http".Server`. Как уже было показано выше в этой книге, чтобы создать HTTP-сервер, достаточно реализовать функцию `handler` и запустить сервер, как показано в следующем листинге.

Листинг 4.20 ❖ Небольшой HTTP-сервер

```
package main

import (
    "errors"
    "net/http"
)

func main() {
    http.HandleFunc("/", handler)  ← Передача функции handler HTTP-серверу
    http.ListenAndServe(":8080", nil)  ← Запуск сервера
}
```

Вся логика запуска и управления сервером находится в пакете `net/http`. Но обязанность реализации обработки запросов пакет остается за разработчиком. В предыдущем примере требуется создать функцию обработки. Это может быть любая функция, удовлетворяющая следующему типу:

```
type HandlerFunc func(ResponseWriter, *Request)
```

Получив запрос, сервер вызовет ее, как это делалось в примере эхо-сервера, то есть запустит сопрограмму в отдельном потоке выполнения. Как вы думаете, что произойдет, если в обработчике случится авария?

Листинг 4.21 ❖ Авария в обработчике

```
func handler(res http.ResponseWriter, req *http.Request) {
    panic(errors.New("Fake panic!"))
}
```

Если запустить сервер вместе с этой функцией, он выведет сведения об аварии в консоль и продолжит работу:

```
2015/04/08 07:57:31 http: panic serving [::1]:51178: Fake panic!
goroutine 5 [running]:
net/http.func·011()
    /usr/local/Cellar/go/1.4.1/libexec/src/net/http/server.go:1130   +0xbb
main.handler(0x494fd0, 0xc208044000, 0xc208032410)
    /Users/mbutcher/Code/go-in-practice/chapter4/http_server.go:13   +0xdd
net/http.HandlerFunc.ServeHTTP(0x3191e0, 0x494fd0, 0xc208044000,
    0xc208032410)
...
```

Но функция обработки ничего не предпринимала, чтобы справиться с аварией! Безопасность сети обеспечивает сама библиотека.

Учитывая это, имеет смысл преобразовать эхо-службу в подобную библиотеку, изменив ее архитектуру так, чтобы аварии обрабатывались внутри библиотеки.

Начав серьезно работать с языком Go, мы написали небольшую тривиальную библиотеку (теперь являющуюся частью github.com/Masterminds/cookoo) для защиты от случайно необработанных аварий в сопрограммах. Упрощенная версия этой библиотеки приводится в следующем листинге.

Листинг 4.22 ❖ `safely.Go`

```
package safely

import (
    "log"
)

type GoDoer func()

func Go(todo GoDoer) {
    go func() {
        defer func() {
            if err := recover(); err != nil {
                log.Printf("Panic in safely.Go: %s", err)
            }
        }()
        todo()
    }()
}
```

safely.Go вызывает функцию как сопрограмму и обрабатывает все аварии

← GoDoer – простая функция без параметров

← Сначала запускается анонимная функция

Анонимная функция обрабатывает аварии по обычному сценарию восстановления

← Затем эта функция вызывает переданную в параметре функцию GoDoer

Эта простая библиотека самостоятельно обрабатывает аварии, поэтому разработчику не придется об этом заботиться. В следующем листинге приводится пример использования `safely.Go`.

Листинг 4.23 ❖ Использование `safely.Go` для перехвата аварий

```
package main

import (
    "github.com/Masterminds/cookoo/safely"
    "errors"
    "time"
)

func message() {
    println("Inside goroutine")
    panic(errors.New("Oops!"))
}
```

← Импорт пакета safely

← Определение функции обратного вызова, соответствующей типу GoDoer

```
func main() {
    safely.Go(message)   ← Заменяет go message
    println("Outside goroutine")
    time.Sleep(1000)   ← Гарантирует выполнение сопрограммы
}                          перед завершением работы программы
```

В этом примере определена простая функция, соответствующая типу `GoDoer` (не имеет параметров и возвращаемого значения). Затем вызывается метод `safely.Go(message)`, запускающий функцию `message` в новой сопрограмме и перехватывающий все аварии. Поскольку функция `message` возбуждает аварию, программа после запуска выведет следующее:

```
$ go run safely_example.go
Outside goroutine
Inside goroutine
2015/04/08 08:28:00 Panic in safely.Go: Oops!
```

Вместо аварийного прерывания программы `safely.Go` перехватывает аварию и выводит сообщение о ней.

Выгоды от использования замыканий

Вместо именованных функций, таких как `message`, здесь можно использовать замыкание. Замыкание позволяет получить доступ к переменным, находящимся в области видимости, и решить проблему отсутствия параметров в функции `GoDoer`. Но при этом следует позаботиться о состоянии гонки и других проблемах параллельной обработки!

Эта конкретная библиотека не в состоянии обеспечить все потребности, но она демонстрирует хорошее решение, заключающееся в создании библиотеки, которая предотвратит неожиданное или непреднамеренное завершение программы в случае аварии в сопрограмме.

Язык Go предоставляет элегантные системы обработки ошибок и аварий. Но, как было продемонстрировано в этом разделе, аварии имеют тенденцию к созданию непредвиденных ситуаций. Если не учитывать этого, можно завязнуть в плохо поддающихся отладке сбоях. Именно поэтому мы потратили много времени на изучение приемов восстановления и профилактики, таких как `safely.Go`, позволяющих не полагаться только на то, что разработчики сами сделают все правильно.

4.3. Итоги

Следует честно признать, что в обработке ошибок нет ничего гламурного. Но мы уверены, что по коду обработки ошибок можно отличить просто хорошего программиста от мастера. Мастера не забывают о системе защиты от сбоев и ошибок.

Именно поэтому целая глава была посвящена подробному рассмотрению системы обработки ошибок и аварий в языке Go и приемам решения возникающих при этом проблем. Мы показали наиболее эффективные способы извлечения значимых данных об ошибках и восстановления после аварий. И завершили главу рассмотрением сочетания аварий с сопрограммами.

В этой главе были представлены:

- общие сведения о сценариях обработки ошибок на языке Go;
- прием использования переменных ошибок;
- способ создания пользовательских типов ошибок;
- правильное использование и обработка аварий;
- приемы обработки ошибок в сопрограммах.

Следующая глава будет посвящена другим аспектам качества кода, таким как журналирование и тестирование. Освоив эти инструменты, вы станете успешным (и, может быть, даже *великим*) Go-разработчиком.

Глава 5

Отладка и тестирование

В этой главе рассматриваются следующие темы:

- *захват отладочной информации;*
- *регистрация сведений об ошибках и отладочной информации;*
- *работа с трассировкой стека;*
- *написание модульных тестов;*
- *создание приемочных тестов;*
- *определение состояния гонки;*
- *тестирование производительности.*

Одним из преимуществ работы с современными языками программирования являются предоставляемые ими инструменты. За годы, прошедшие с появления языка, разработчики создали фантастические инструменты, упрощающие процесс разработки. Язык Go проектировался как язык для системных разработчиков и поставляется с инструментами, упрощающими их работу. В этой главе основное внимание уделяется таким инструментам и стратегии создания надежного программного обеспечения. Мы рассмотрим журналирование, отладку и различные виды тестирования.

В предыдущей главе речь шла об ошибках и авариях, поэтому вполне уместно будет начать эту главу знакомством с приемами поиска причин сбоев, приводящих к неожиданным ошибкам и авариям. Начнем с отладки.

5.1. Определение мест возникновения ошибок

Иногда одного взгляда на ошибку достаточно, чтобы понять, что ее вызвало. Но обычно требуется некоторое время, чтобы выявить фрагмент кода, вызвавшего проблему. Иногда на выяснение причин уходит несколько часов или даже дней.

Эта, третья категория ошибок обычно требует использования специальных инструментов или методик, чтобы отследить проблему. В этом разделе рассматриваются некоторые из таких инструментов и методик.

5.1.1. Подождите, где мой отладчик?

Для отладки кода, написанного на языке Go, многие разработчики используют (сюрприз!) *отладчик*. Этот великолепный инструмент поддерживает пошаговое выполнение кода в любом темпе.

Прежде чем переходить к его описанию, рассмотрим один важный аспект. Несмотря на богатство функциональных возможностей, предназначенных для разработчиков, язык Go все еще не имеет полноценного отладчика. Основные разработчики языка заняты другими вопросами, поэтому почти официальным отладчиком языка Go является плагин для GNU Debugger (GDB). Почтенный старый GDB можно использовать для отладки, но он не настолько надежен, как того требуют многие разработчики.

> **СОВЕТ** Процесс настройки GDB для отладки кода, написанного на языке Go, подробно описан в инструкции на сайте golang (http://golang.org/doc/gdb).

Go-сообщество предложило свое решение этой проблемы и реализовало проект, на который стоит обратить внимание. Delve (https://github.com/derekparker/delve) – новый отладчик для Go и продолжает активно развиваться. На момент написания книги процесс установки Delve был достаточно сложным, особенно в Mac. Но если вы ищете полнофункциональный отладчик, который отлично справляется с трассировкой сопрограмм, попробуйте Delve. Мы полагаем, что Delve со временем потеснит GDB, поскольку этот отладчик выбрало Go-сообщество.

Еще одной альтернативой является менее известный инструмент Godebug (https://github.com/mailgun/godebug). Несмотря на то что

его применение требует определения точек останова в исходном коде, инструменты Godebug помогают получить полное представление о происходящем в программе.

Может быть, это связано с тем, что мы относимся к программистам старой школы, но мы не видим особой проблемы в сложившейся ситуации с отладчиком. В языке Go имеются отличные библиотеки и инструменты, которые выручали нас в самых сложных ситуациях. Сделав эту оговорку, можно погрузиться в методики улучшения качества кода, написанного на языке Go.

5.2. Журналирование

Давно стало общепринятой практикой записывать сведения о состоянии процессов в файлы журналов или подсистемы. Для всех популярных языков программирования существуют библиотеки функций журналирования, и язык Go – не исключение. Более того, разработчики языка Go решили включить журналирование в число основных библиотек.

Как правило, журналирование предназначено для захвата определенных сведений, которые разработчики, системные администраторы, другие программы смогут использовать для анализа выполнения приложения. Например, журнал веб-сервера должен содержать такие сведения, как время его запуска, когда и какие необычные ситуации возникли и как он обрабатывал запросы.

5.2.1. Журналирование в Go

Язык Go включает два встроенных пакета с функциями журналирования: `log` и `log/syslog`. Сначала рассмотрим первый из них, а затем, в разделе 5.2.2, перейдем к пакету `syslog`.

Пакет `log` реализует простейшую поддержку (главным образом в виде форматирования) записи сообщений в журнал. В простейшем случае он форматирует сообщения и отправляет их в поток стандартного вывода ошибок, как показано в следующем листинге.

Листинг 5.1 ❖ Простейшее использование пакета `log`

```
package main

import (
    "log"
)
```

```
func main() {
    log.Println("This is a regular message.") ← Запись сообщения в os.Stderr
    log.Fatalln("This is a fatal error.")
    log.Println("This is the end of the function.")
}
```

Запись сообщения в os.Stderr и выход с кодом ошибки

Эта инструкция никогда не будет выполнена

Этот код выведет следующее:

```
$ go run simple.go
2015/04/27 08:18:36 This is a regular message.
2015/04/27 08:18:36 This is a fatal error.
exit status 1
```

Хотелось бы дать несколько пояснений к этому примеру. Во-первых, все сведения об ошибках отправляются в поток стандартного вывода ошибок, независимо от того, действительно ли сообщение связано с ошибкой или оно чисто информационное. Документация к пакету `log` сообщает, что он не различает видов сообщений, но реализует свою систему различий, и этой особенности касается второе замечание.

При вызове `log.Fatalln` или любой другой «фатальной» функции библиотека выводит сообщение об ошибке, а затем выполняет `os.Exit(1)`, заставляя программу прервать выполнение. Кроме того, функция `log.Panic` выводит сообщение, а затем инициирует аварию.

Все функции журналирования поддерживают стиль `printf`, что позволяет вставлять информацию в строку сообщения: `log.Printf("The value of i is %s", i)`.

С практической точки зрения от простых функций журналирования не так много пользы. Хотя в мире Docker-контейнеров принято выполнять журналирование в потоки стандартного ввода и стандартного вывода ошибок, в более общем случае намного предпочтительнее выглядит отправка сообщений службам журналирования или в специальные файлы. С этой точки зрения тип `log.Logger` из того же пакета несет намного больше пользы.

РЕЦЕПТ 22 Журналирование в произвольный объект записи

Отправка сообщений в поток стандартного вывода ошибок полезна лишь в простых случаях. При создании серверов, приложений или системных служб требуется более надежное место для регистрационных данных. Позднее в этой же главе мы рассмотрим запись непосредственно в системный журнал, а сейчас познакомимся с исполь-

зованием пакета log для записи в произвольное место посредством объектов типа io.Writer.

Проблема

Требуется организовать запись регистрационных сообщений в файл или передачу их сетевой службе без создания собственной системы журналирования.

Решение

Инициализация нового регистратора log.Logger и отправка ему регистрационных сообщений.

Обсуждение

Регистратор – тип log.Logger – поддерживает функции передачи регистрационных данных любому объекту io.Writer, который способен работать с дескрипторами файлов и сетевыми подключениями (net.Conn). Следующий короткий пример демонстрирует создание файла журнала и запись сообщений в него.

Листинг 5.2 ❖ Журналирование в файл

```
package main

import (
    "log"
    "os"
)

func main() {
    logfile, _ := os.Create("./log.txt") ← Создание файла журнала
    defer    logfile.Close() ← Гарантировать закрытие

    logger := log.New(logfile, "example ", log.LstdFlags|log.Lshortfile) ← Создание регистратора

    logger.Println("This is a regular message.")  | Отсылка сообщений
    logger.Fatalln("This is a fatal error.")      |
    logger.Println("This is the end of the function.") ← Как и раньше, никогда не будет выполнена
}
```

В этом примере создается файл журнала, куда затем записываются сообщения.

> **ПРИМЕЧАНИЕ** При каждом запуске этот пример будет затирать старый файл журнала, потому что использует метод os.Create. Такое решение хорошо подходит для демонстрационного примера, но в действующих приложениях обычно требуется открывать существующий файл, а не затирать его.

При создании нового регистратора `log.Logger` можно передать три параметра. Первый – объект `io.Writer`, которому будут передаваться сообщения. Второй – префикс для сообщений, и третий – список флагов, определяющих формат сообщений. Чтобы прояснить назначение второго и третьего параметров, рассмотрим несколько примеров журналируемых данных, записанных программой выше в файл *log.txt*:

```
$ cat log.txt
example 2015/05/12 08:42:51 outfile.go:16: This is a regular message.
example 2015/05/12 08:42:51 outfile.go:17: This is a fatal error.
```

Как и раньше, выполняются только два из трех вызовов `logger.Log`, потому что второй генерирует фатальную ошибку. Но вернемся к содержимому файла, демонстрирующему порядок записи данных. Грубо говоря, каждое сообщение можно разбить на три части: префикс, автоматически добавляемая информация о сообщении и само сообщение, как показано на рис. 5.1.

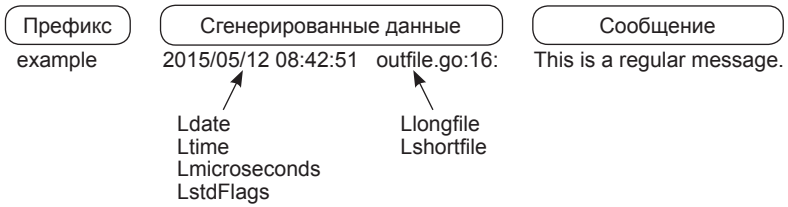


Рис. 5.1 ❖ *Элементы журнальных сообщений*

Содержимое префикса определяется вторым аргументом метода `log.New`. Обратите внимание, что префикс включает завершающий пробел (после слова `example`). Это сделано не случайно. По умолчанию регистратор не вставляет пробел между префиксом и генерируемыми данными.

Разработчик не имеет возможности напрямую управлять содержимым генерируемой информации, но может влиять на него косвенно. Например, у вас нет возможности отформатировать поля даты и времени по своему желанию, но вы можете указать флаги, определяющие состав сообщений. При создании регистратора `log.Logger` в третьем аргументе передается битовая маска флагов. В данном примере использованы флаги `log.LstdFlags | log.Lshortfile`. Первый определяет формат даты, а второй заставляет регистратор выводить имя файла и номер строки. (Флаги выделены жирным на рис. 5.1.)

Фактически регистратор автоматически генерирует время события и место, где оно произошло. Точность сведений о дате и времени можно регулировать:

- флаг `Ldate` отвечает за вывод даты;
- флаг `Ltime` выводит время;
- флаг `Lmicrosends` увеличивает точность до микросекунд. При наличии этого флага время будет выводиться автоматически, даже если не задан флаг `Ltime`;
- флаг `LstdFlags` заменяет флаги `Ldate` и `Ltime`.

И еще пара флагов, отвечающих за вывод информации о местоположении:

- флаг `Llongfile` выводит полный путь к файлу и номер строки: `/foo/bar/baz.go:123`;
- флаг `Lshortifle` выводит только имя файла и номер строки: `baz.go:123`.

Флаги можно комбинировать с помощью оператора логического «ИЛИ», но некоторые комбинации очевидно несовместимы (например, `Llongfile` и `Lshortfile`).

Запись в файлы обычно не вызывает сложностей, но журналирование в некоторые другие хранилища может оказаться непростой задачей. Начнем с самого сложного случая – передачи журналируемых сообщений сетевой службе, – а затем вернемся к более простым ситуациям, когда имеющихся инструментов журналирования достаточно для удовлетворения всех потребностей.

РЕЦЕПТ 23 Журналирование в сетевые ресурсы

Предыдущий рецепт продемонстрировал журналирование в универсальный интерфейс `io.Writer`. Он записывал сообщения в обычный файл. Но в наши дни приложения, особенно серверы, действуют в образах Docker в облаке, в виртуальных машинах или других ресурсах, имеющих лишь эфемерные хранилища. Кроме того, серверы часто работают в кластерах, где желательно собрать журналы всех серверов в единую службу журналирования.

Далее в этой же главе мы рассмотрим использование пакета `syslog` в качестве внешнего регистратора. А сейчас познакомимся с другим вариантом: журналированием в сетевой ресурс.

Многие популярные службы журналирования, такие как Logstash (http://logstash.net/) и Heka (http://hekad.readthedocs.org/en/v0.9.2/), обеспечивают комплексное журналирование. Эти службы обычно

предоставляют порт, к которому можно подключиться и направить в него поток сообщений. Такой стиль журналирования сделала популярным парадигма 12-факторных приложений (https://12factor.net/ru/), где одиннадцатый фактор формулируется как: «Рассматривайте журнал как поток событий». Несмотря на простоту формулировки, отправка сообщений в виде потока вызывает определенные трудности.

Проблема

Потоковая передача информации в сетевую службу чревата сбоями, поэтому ее нужно реализовать так, чтобы не потерять данные, если это возможно.

Решение

Каналы языка Go и буферизация могут значительно повысить надежность.

Обсуждение

Перед тем как погрузиться в код, подготовим сначала имитацию сервера журналирования. Несмотря на доступность существующих служб, таких как Logstash и Heka, воспользуемся простым инструментом UNIX, носящим название Netcat (`nc`). Netcat входит в состав большинства разновидностей UNIX и Linux, включая OS X. Кроме того, имеется версия для Windows.

Начнем с простого TCP-сервера, принимающего текстовые сообщения и выводящего их в консоль. Его роль будет играть простая команда Netcat:

```
nc -lk 1902
```

Теперь у нас есть приемник (`-l`), непрерывно (`-k`) прослушивающий порт 1902. (Некоторые версии Netcat могут потребовать также флага `-p`). Эта короткая команда прекрасно справляется с имитацией сервера журналирования.

А теперь рассмотрим следующий листинг, представляющий версию листинга 5.2, адаптированную для записи в сетевой сокет.

Листинг 5.3 ❖ Клиент сетевого регистратора

```
package main

import (
  "log"
  "net"
)
```

```
func main() {
    conn, err := net.Dial("tcp", "localhost:1902")  ← Подключение к серверу
    if err != nil {                                    журналирования
        panic("Failed to connect to localhost:1902")
    }
    defer conn.Close()  ← Гарантировать закрытие соединения даже в случае аварии
    f := log.Ldate | log.Lshortfile                 Оправка регистрационных сообщений
    logger := log.New(conn, "example ", f)  ← в сетевое соединение
    logger.Println("This is a regular message.")
    logger.Panicln("This is a panic.")  ← Вывести сообщение и инициировать
}                                          аварию, но не использовать Fatalln
```

Удивительно, как мало изменений потребовалось, чтобы заменить файл сетевым соединением. Сетевая библиотека в языке Go проста и удобна. Новое TCP-соединение создается с помощью метода `net.Dial`, он устанавливает соединение с портом, открытым с помощью Netcat. Всегда закрывайте сетевые соединения в блоке `defer`. Если случится авария (что будет реализовано в этом демонстрационном примере), сетевой буфер будет вытолкнут при закрытии, что уменьшит вероятность потери критически важного сообщения со сведениями о причинах аварии.

Для передачи сообщений удаленному серверу используется тот же пакет `log`. Такое решение имеет ряд преимуществ. Первое заключается в автоматическом добавлении отметки времени в записи – посылая информацию сетевому серверу, всегда указывайте время хоста-отправителя и не полагайтесь на запись отметки времени на сервере. Это поможет реконструировать последовательность событий, даже если сообщения задержатся на пути к серверу журналирования. Второе можно заметить, сравнив листинги 5.2 и 5.3: использование системы журналирования упрощает замену механизма сохранения сообщений. Это облегчает тестирование и выполнение в среде разработки.

Заметили, что в этом примере функцию `log.Fatalln` заменила функция `log.Panicln`? Эта замена объясняется просто: причина – все функции `log.Fatal*` имеют один неприятный побочный эффект: при их использовании не вызываются отложенные функции. Почему? Потому что функции `log.Fatal*` вызывают функцию `os.Exit`, немедленно завершающую программу без раскручивания стека функций. Этот вопрос рассматривался в предыдущей главе. Поскольку отложенные функции в этом случае не вызываются, сетевое соединение не будет правильно сброшено и закрыто. С другой стороны, аварии легче перехватывать. В действующем приложении, если это не простой

клиент командной строки, старайтесь избегать фатальных ошибок. Но, как было продемонстрировано в предыдущей главе, существуют конкретные случаи, когда необходимо инициировать аварию.

Итак, после запуска этого примера экземпляр `nc` должен получить следующие сообщения:

```
$ nc -lk 1902
example 2015/05/27 log_client.go:23: This is a regular message.
example 2015/05/27 log_client.go:24: This is a panic.
```

Эти сообщения преодолели путь от примера клиента до простого сервера, запущенного с помощью `nc`. Мы создали простую утилиту сетевого журналирования. Но при ее работе могут возникнуть проблемы, связанные с неприятностью, которую обычно называют *обратным давлением*.

РЕЦЕПТ 24 Обработка обратного давления при журналировании по сети

В предыдущем рецепте реализовано журналирование сообщений на сетевом сервере. Этот способ журналирования обладает бесспорными преимуществами:

- журналы из нескольких служб можно хранить в едином, централизованном хранилище;
- серверы в облаке, имеющие только эфемерные хранилища, получают возможность сохранять журналируемую информацию;
- улучшенная безопасность и больше возможностей для аудита;
- серверы журналирования и серверы приложений можно настраивать по-разному.

Но он имеет также один большой недостаток: его работоспособность зависит от надежности сети. В этом рецепте вы увидите, как решать проблемы сохранения журнальных записей, связанные с сетью.

Проблема

При работе с сетевыми службами журналирования могут возникать проблемы, связанные с ненадежностью сетевых соединений и ограниченной пропускной способностью. Они могут приводить к потере сообщений, а иногда даже к сбоям служб.

Решение

Разработка более надежного регистратора, использующего буферизацию данных.

Обсуждение

Вам наверняка приходилось сталкиваться с двумя основными проблемами сетей:

- сетевое соединение разрывается (когда сеть прекращает работу или происходит сбой в удаленном регистраторе);
- падает скорость соединения, через которое отправляются сообщения.

Первая проблема знакома всем, и пути ее решения особых сомнений не вызывают. Решение второй менее очевидно.

Листинг 5.3 представляет примерную последовательность событий. В общих чертах она изображена на рис. 5.2 (механика работы сети несколько сложнее, но для понимания ситуации это не так важно).

Приложение сначала открывает TCP-соединение, потом отправляет сообщения и закрывает соединение. Но в коде не видно, как обрабатываются ответы регистратора. Это связано с тем, что все взаимодействия происходят на уровне протокола TCP и не достигают прикладного уровня.

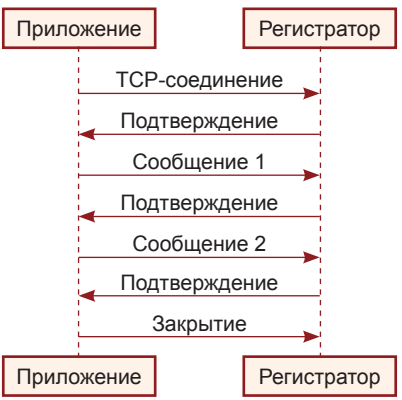


Рис. 5.2 ❖ *Отправка сообщений по протоколу TCP*

В частности, когда сообщение отправляется как пакет TCP/IP, получатель обязан отправить подтверждение (Acknowledging, ACK) о получении сообщения. Возможно (и даже вероятнее всего), что одно сообщение будет отправлено в виде нескольких сетевых пакетов. Предположим, что сообщение разделено на два пакета. Удаленный регистратор получил его первую часть и отправил ACK. Затем клиент отправил вторую половину сообщения, и регистратор также

вернул ACK. Такая схема работы позволяет клиенту убедиться, что посланные им данные действительно получены удаленным узлом.

Все это хорошо работает, пока удаленный узел не замедлит свою работу. Представьте, что сервер одновременно получает несколько тысяч сообщений от множества клиентов. Такой объем поступающих данных может замедлить его работу. Но при том, что скорость работы уменьшилась, количество поступающих регистрационных сообщений остается прежним. И при использовании протокола TCP сервер должен возвращать подтверждения для каждого поступившего сообщения.

Если он задержит отправку подтверждения, клиент приостановится в ожидании. То есть клиент замедлит свою работу, будучи вынужденным дожидаться подтверждений для уже отправленных сообщений. Такую ситуацию называют *обратным давлением*.

Одним из решений проблемы обратного давления является переход от протокола TCP к протоколу UDP. Это уменьшает объем накладных расходов на уровне протокола. И, самое главное, приложению не нужно ждать подтверждений от сервера. На рис. 5.3 показано, как работает это решение.

Рис. 5.3 ❖ *Отправка сообщений по протоколу UDP*

Протокол UDP не требует поддержания сетевого соединения. Клиент просто отправляет информацию на адрес сервера по готовности. Чтобы изменить протокол, достаточно внести небольшие изменения в листинг 5.3, как показано ниже.

Листинг 5.4 ❖ Журналирование по протоколу UDP

```
package main

import (
    "log"
    "net"
    "time"
)
```

```
func main() {
    timeout := 30 * time.Second  ← Явное добавление задержки
    conn, err := net.DialTimeout("udp", "localhost:1902", timeout)
    if err != nil {                                          Замена TCP-соединения
          panic("Failed to connect to localhost:1902")       на UDP-соединение
    }
    defer     conn.Close()

    f := log.Ldate | log.Lshortfile
    logger := log.New(conn, "example ", f)

    logger.Println("This is a regular message.")
    logger.Panicln("This is a panic.")
}
```

Код изменился незначительно. Вместо обычного метода `net.Dial` использован метод `net.DialTimeout`, который позволяет определить время ожидания установки соединения. Здесь задается ожидание не более 30 секунд. При использовании протокола TCP время ожидания включает время на отправку сообщения и получение подтверждения в ответ. Но для протокола UDP время включает только время разрешения адреса и отправки сообщения приложением. Установка времени ожидания помогает обезопасить приложение на случай, когда сеть не функционирует надлежащим образом.

Для взаимодействия с предыдущим кодом необходимо перезапустить сервер `nc` командой `nc -luk 1902`.

Использование протокола UDP для журналирования дает следующие явные преимущества:

- приложение устойчиво к обратному давлению и отключениям сервера. В случае сбоя сервера могут быть потеряны некоторые из UDP-пакетов, но на работу клиента это никак не повлияет;
- отправка журналируемой информации происходит быстрее, даже в отсутствие обратного давления;
- код реализации упрощается.

Но этот подход имеет серьезные недостатки. При определенных требованиях эти недостатки могут сделать его неприемлемым:

- сообщения легко могут теряться. Протокол UDP не предусматривает возможности убедиться в получении сообщения;
- сообщения могут быть получены в произвольном порядке. Большие сообщения разбиваются на несколько пакетов, и процесс их получения растягивается. Добавление в сообщения временных меток (что и было сделано) смягчает последствия, но не устраняет проблему полностью;

- массовая отправка UDP-сообщений может привести к захлебыванию удаленного сервера, поскольку он лишен возможности управлять подключениями и замедлять отправку данных. При том, что приложению не грозит обратное давление, это может осложнить работу сервера.

Как показывает наш собственный опыт, журналирование по протоколу UDP, безусловно, полезно в определенных случаях. Оно выполняется быстро и эффективно. Если имеется возможность с относительной точностью предсказать объем работы сервера, этот метод станет простым и удобным средством для сетевого журналирования.

Но в некоторых случаях использование протокола UDP неприемлемо. Нельзя использовать протокол UDP для журналирования, если невозможно точно предсказать, сколько данных будет передаваться серверу журналирования, или потеря каких-либо сообщений недопустима.

Журналирование по протоколу TCP более чувствительно к пропускной способности (обратному давлению), а журналирование по протоколу UDP не гарантирует сохранности данных. Эта головоломка сродни кодированию изображений: что выбрать – высокое качество изображения за счет увеличения размера файла (GIF, PNG) или компактность за счет с потери части данных (JPEG)? При реализации журналирования может потребоваться сделать подобный выбор. Это не означает, что ничего нельзя здесь улучшить. Например, эффект обратного давления можно ослабить увеличением размера буфера, куда временно будут помещаться данные при перенасыщении сети.

5.2.2. Работа с системными регистраторами

В предыдущих разделах мы узнали, как написать свою систему журналирования. Но часто удобнее бывает воспользоваться готовыми решениями. В этом разделе вы познакомитесь с такими системами.

> **ПРИМЕЧАНИЕ** Существуют мощные пакеты журналирования сторонних производителей, которые можно использовать в программах на языке Go. Двумя популярными примерами являются Logrus (https://github.com/Sirupsen/logrus) и Glog (https://github.com/golang/ glog).

Идеи, лежащие в основе журналирования, формировались десятилетиями, и одна из них, широко используемая в настоящее время, заключается в разделении сообщений на уровни. Уровень определяет значимость информации в сообщении и в то же время определяет его

вид. Обычно список уровней включает сообщения для трассировки, отладки, информирования, предупреждения, извещения об ошибках и критических ошибках, хотя иногда также используются уведомления, предупреждения об опасности и предупреждения о чрезвычайных ситуациях.

Приложения могут определять свои уровни журналируемых сообщений, тем не менее ниже приводится перечень наиболее общеупотребительных видов сообщений, соответствующих названиям уровней журналирования:

- *информационные сообщения* – как правило, содержат сведения о текущем состоянии приложения, времени перехода в это состояние или выхода из него. Результаты измерений обычно журналируются как информационные сообщения. Такого рода сообщениям соответствует уровень информирования;
- *сообщения об ошибках* – когда приложение сталкивается с ошибкой, оно записывает в журнал сведения об этой ошибке (включая вспомогательную информацию). Как правило, они делятся на три уровня по тяжести последствий: предупреждение, ошибка и критическая ошибка;
- *отладочная информация* – используется разработчиками для журналирования сведений, необходимых для отладки. Эта информация предназначена исключительно для программистов (операторы и системные администраторы редко пользуются ею). Ей назначается отладочный уровень журналирования;
- *дамп стека или подробная информация* – предназначена для сохранения подробных сведений о программе. Для особенно сложного участка кода создается дамп трассировки стека или сведения о сопрограммах. Соответствует уровню трассировки.

Многие системы, включая современные UNIX-подобные системы, поддерживают эти уровни в своих системных регистраторах. В стандартной библиотеке Go имеется даже собственный пакет для работы с системным журналом. Займемся его рассмотрением.

РЕЦЕПТ 25 Сохранение сообщений в системном журнале

Журналирование – хорошо известная задача, и за несколько десятилетий созданы стандартные средства, охватывающие все аспекты журналирования. Неудивительно, что язык Go изначально включает их поддержку.

Использование системных журналов имеет определенные преимущества перед созданием своего инструмента. Во-первых, это зрелые и стабильные продукты, оптимизированные для противодействия задержкам, избыточности сообщений и архивированию. Большинство современных систем журналирования обеспечивает периодическую ротацию журналов, сжатие и исключение повторяющихся записей. Они упрощают реализацию. Кроме того, системные администраторы имеют большой опыт анализа таких файлов, и для работы с ними имеется масса инструментов и утилит. Это веские причины, чтобы использовать стандартную систему журналирования вместо создания своей собственной.

Проблема

Требуется организовать запись сообщений в системный журнал.

Решение

Настроим пакет `syslog` из стандартной библиотеки Go и воспользуемся им.

Обсуждение

Пакет журналирования в стандартной библиотеке Go включает поддержку системного журнала. Именно для этого предназначен пакет `log/syslog`. Он обеспечивает два способа работы с системным журналом. Во-первых, его можно использовать как приемник информации, подобно тому, как это было сделано в примерах выше. Это позволяет реорганизовать имеющийся код, направив вывод в системный журнал.

Во-вторых, имеется возможность использовать все уровни журналирования и возможности системного журнала. Второй путь не так универсален, но он ближе к общепринятому использованию пакета `syslog`.

Далее основное внимание будет уделено второму способу, тем не менее код в следующем листинге демонстрирует пример реализации журналирования на языке Go с выводом через пакет `syslog`.

Листинг 5.5 ❖ Запись сообщений в системный журнал

```
package main

import (
    "fmt"
    "log"
    "log/syslog"
```

```
)
                                                    Определить порядок работы
                                                    с системным журналом
func main() {
    priority := syslog.LOG_LOCAL3 | syslog.LOG_NOTICE ←
    flags := log.Ldate | log.Lshortfile  ← Те же флаги, что уже использовались выше
    logger, err := syslog.NewLogger(priority, flags)  ← Создание нового
    if err != nil {                                     системного регистратора
        fmt.Printf("Can't attach to syslog: %s", err)
        return
    }
    logger.Println("This is a test log message.")  ← Отправка простого сообщения
}
```

После запуска этого кода в системном журнале должно появиться сообщение:

```
Jun 30 08:34:03 technosophos syslog_logger[76564]: 2015/06/30
     syslog_logger.go:18: This is a test log message.
```

Форматирование этого сообщения далеко от идеального, но оно содержит все необходимые сведения. Рассмотрим основные особенности кода.

Когда приложение на языке Go выполняет запись сообщений в системный журнал, необходимо сделать некоторые упрощения. Богатство видов и приоритетов сообщений в системном журнале недостаточно полно отражено в реализации регистратора на языке Go, поэтому ему следует сообщить вид сообщений и их серьезность. К сожалению, это можно сделать только один раз, во время создания. В этом примере определяются вид сообщений (`LOG_LOCAL3`) и их важность. Поскольку все это можно настроить только единожды, важность устанавливается равной `LOG_NOTICE` – этого достаточно, чтобы сообщение попало в журнал с настройками по умолчанию, но недостаточно, чтобы рассматривать его как тревожный сигнал.

В системном журнале UNIX-подобных систем (в том числе Linux и OS X) местоположение файлов определяется в файлах конфигурации. Например, в Mac сообщения записываются в файл */var/log/system.log*, а в некоторых версиях Linux – в */var/log/messages*.

При инициализации системы журналирования нужно определить не только приоритет (вид и важность), но и флаги форматирования, подобные тем, что используют другие инструменты журналирования в языке Go. Для простоты здесь использовались те же флаги, что и в предыдущих примерах.

Теперь все сообщения, отправленные в журнал, будут записаны в файл как уведомления, независимо от их истинной серьезности.

Регистратор из стандартной библиотеки Go удобен, но возможность динамического определения вида и уровня серьезности сделала бы его более полезным. Этого можно достичь непосредственным использованием функций из log/syslog, как показано в следующем листинге.

Листинг 5.6 ❖ Запись сообщений в системный журнал

```
package main

import (
   "log/syslog"
)

func main() {
    logger, err := syslog.New(syslog.LOG_LOCAL3, "narwhal")  ← Создание нового клиента системного регистратора
    if err != nil {
          panic("Cannot attach to syslog")
    }
    defer logger.Close()

    logger.Debug("Debug message.")
    logger.Notice("Notice message.")      Вывод различных сообщений
    logger.Warning("Warning message.")
    logger.Alert("Alert message.")
}
```

Этот код настраивает объект регистратора для системного журнала, а затем отправляет сообщения с разными уровнями серьезности. Настройка регистратора довольно проста. Ему передаются два параметра: вид сообщений (LOG_LOCAL3) и префикс, добавляемый в каждое из них (narwhal). Как правило, в качестве префикса выбирается имя службы или приложения.

Запись в удаленный системный журнал

В Go имеется функция syslog.Dial, обеспечивающая подключение к удаленному демону системного журнала. Этот прием часто используется, чтобы собрать журналы с нескольких серверов в одном месте. Многие локальные средства журналирования поддерживают возможность подключения к удаленным серверам через прокси. Но функция syslog.Dial предназначена для непосредственного подключения к удаленному серверу журналирования.

Библиотека syslog включает различные функции журналирования, позволяющие определять уровень серьезности сообщений. При

каждом вызове таких функций происходит отправка сообщения в системный журнал, который затем принимает решение (основываясь на собственных правилах), как поступить с этим сообщением.

Если выполнить предыдущий пример и затем заглянуть в файл системного журнала, то в нем должны появиться следующие строки:

```
Jun 30 08:52:06 technosophos narwhal[76635]: Notice message.
Jun 30 08:52:06 technosophos narwhal[76635]: Warning message.
Jun 30 08:52:06 technosophos narwhal[76635]: Alert message.
```

Первые два поля содержат время и имя хоста. Они генерируются системным журналом. Затем идет метка `narwhal`. За ней следует идентификатор процесса (Process ID, PID). И наконец, само сообщение. Порядок и формат полей определяются конфигурацией системного журнала.

В примере выше мы послали четыре сообщения, но в файл попали только три из них. Сообщение, отправленное вызовом функции `syslog.Debug`, пропало. Причина в том, что системный журнал в данном примере настроен так, чтобы не сохранять отладочные сообщения в файле. Если вам понадобится записывать отладочных сообщений, придется изменить конфигурацию системного журнала. Преимущество такого решения в том, что разработчик не должен решать, что отображать и при каких обстоятельствах. Это будут выбирать те, кто использует приложение.

В этом разделе мы рассмотрели большую часть видов сообщений, но остается еще один важный для программистов инструмент отладки, позволяющий сохранить в журнале трассировку стека.

5.3. Доступ к трассировке стека

Многие языки предоставляют доступ к стеку вызовов. *Трассировка стека* (или *дамп стека*) – это список функций, вызванных в момент захвата стека. Предположим, например, что имеется программа, в которой функция `main` вызывает функцию `foo`, а та, в свою очередь, вызывает функцию `bar`. Функция `bar` выполняет трассировку стека. Трассировка будет иметь глубину в три вызова и покажет, что текущая функция `bar` вызвана функцией `foo`, которая, в свою очередь, была вызвана функцией `main`.

РЕЦЕПТ 26 Захват трассировки стека

Трассировка стека помогает разработчику получить более полное представление о происходящем в системе. Сохранение трассировки

в журнал может пригодиться для отладки. Язык Go позволяет получить доступ к трассировке стека в любой момент выполнения программы.

Проблема

Требуется получить трассировку стека в критической точке выполнения приложения.

Решение

Используем пакет `runtime`, содержащий несколько инструментов.

Обсуждение

Генерация дампов стека в коде на языке Go не является особенно трудной задачей, если вам известно, как это сделать. Но похоже, что вопрос, как это сделать, является одним из самых распространенных. Если требуется лишь получить трассировку для отладки, ее можно отправить в стандартный вывод с помощью функции `PrintStack` из пакета `runtime/debug`, как показано в следующем листинге.

Листинг 5.7 ❖ Вывод трасиировки стека в поток стандартного вывода

```
package main

import (
    "runtime/debug"
)

func main() {
    foo()
}

func foo() {
    bar()
}

func bar() {
    debug.PrintStack()
}
```

Определение нескольких функций для трассировки

← Вывод трассировки

Этот код выведет следующий дамп стека:

```
$ go run trace.go
/Users/mbutcher/Code/go-in-practice/chapter5/stack/trace.go:20 (0x205b)
     bar: debug.PrintStack()
/Users/mbutcher/Code/go-in-practice/chapter5/stack/trace.go:13 (0x203b)
     foo: bar()
```

```
/Users/mbutcher/Code/go-in-practice/chapter5/stack/trace.go:9 (0x201b)
     main: foo()
/usr/local/Cellar/go/1.4.2/libexec/src/runtime/proc.go:63 (0x12983)
     main: main_main()
/usr/local/Cellar/go/1.4.2/libexec/src/runtime/asm_amd64.s:2232 (0x37711)
     goexit:
```

Этот прием можно использовать для отладки в простых случаях. Но если потребуется захватить трассировку с последующей передачей куда-то еще, сделать это будет несколько сложнее. Для этого можно использовать функцию `Stack` из пакета `runtime`, как показано в следующем листинге.

Листинг 5.8 ❖ Использование функции `Stack`

```
package main

import (
    "fmt"
    "runtime"
)

func main() {
    foo()
}

func foo() {
    bar()
}

func bar() {
    buf := make([]byte, 1024)  ← Создание буфера
    runtime.Stack(buf, false)  ← Запись стека в буфер
    fmt.Printf("Trace:\n %s\n", buf)  ← Вывод результата
}
```

В этом примере дамп стека отправляется в стандартный вывод, но его также можно вывести в журнал или в другом месте. Этот код выведет следующее:

```
$ go run trace.go
Trace:
goroutine 1 [running]:
main.bar()
/Users/mbutcher/Code/go-in-practice/chapter5/stack/trace.go:18   +0x7a
     main.foo()
/Users/mbutcher/Code/go-in-practice/chapter5/stack/trace.go:13   +0x1b
     main.main()
```

Как видите, данная версия дает более краткий вывод. Функция `Stack` исключила информацию о низкоуровневых вызовах. Коротко остановимся на этом коде.

Во-первых, функция `Stack` требует передать ей буфер достаточного размера. Причем не существует удобного способа определить объем буфера, достаточный для размещения всех данных. (В некоторых случаях объем вывода слишком велик, и он не нужен весь целиком.) Приходится заранее принимать решение о размере выделяемого пространства.

Во-вторых, функция `Stack` принимает два аргумента. Во втором аргументе передается логический флаг, в этом примере – `false`. Если передать в нем значение `true`, `Stack` выведет дампы стека для всех запущенных сопрограмм. Это может пригодиться при отладке параллельных вычислений, но при этом существенно увеличивается объем вывода. Трассировка для предыдущего кода, например, займет всю печатную страницу.

Если всего этого будет недостаточно, можно воспользоваться функциями `Caller` и `Callers` из пакета `runtime`, чтобы получить доступ к деталям стека вызовов. Извлечение и форматирование данных при этом потребуют определенных усилий, но эти функции дадут возможность подробно проанализировать стек конкретного вызова. Кроме того, пакеты `runtime` и `runtime/debug` содержат множество других функций для анализа использования программой памяти, сопрограмм, потоков выполнения и прочих ресурсов.

И в заключительной части этой главы перейдем от отладки к тестированию.

5.4. Тестирование

Тестирование и отладка требуют детального анализа сведений о программе. Если отладка носит реактивный характер, то тестирование – активный. В этом разделе рассматривается несколько подходов к работе с большинством инструментов тестирования в языке Go.

Код тестов сопровождает исходный код

Для программистов, обучающихся тестированию программ на языке Go, характерны две распространенные ошибки. Первая заключается в попытке поместить файлы тестов в отдельный каталог. Популярные средства тестирования других языков программирования позволяют это, но при тестировании кода на языке Go этого делать нельзя. Фай-

лы тестов должны находиться в том же каталоге, где находятся файлы с действующим кодом, которые они тестируют.

Вторая ошибка состоит в помещении тестов в другой пакет (`package hello_test` или что-то подобное). Тесты должны находиться в том же пакете, что и тестируемый ими код. Это позволяет тестировать не только экспортируемый API, но и внутренний код.

5.4.1. Модульное тестирование

Написание тестов вместе с кодом стало стандартным правилом разработки программного обеспечения. Некоторые методики разработки (например, разработка через написание тестов) даже ставят разработку тестов во главу угла.

Большинство введений в язык Go объясняет, как писать тесты с применением встроенных средств. Язык Go разрабатывался с учетом тестирования и включает инструменты для запуска тестов в проектах. Любой файл исходного кода на языке Go, имя которого заканчивается на *_test.go*, считается тестовым файлом. Для запуска этих тестов используется инструмент `go test`.

Файлы, имена которых заканчиваются на *_test.go*, могут содержать функции с именами, начинающимися со слова `Test` и требующими передачи единственного параметра типа `*testing.T`. Каждая такая функция будет выполняться как модульный тест. Предположим, имеется файл с исходным кодом *hello.go*. Этот файл содержит единственную функцию `Hello`, которая возвращает строку `hello`, как показано в следующем листинге.

Листинг 5.9 ❖ Просто возврат `hello`

```
package hello

func Hello() string {
     return "hello"
}
```

Чтобы написать тест для этой простой функции, создадим файл *hello_test.go* и поместим в него тесты, представленные в следующем листинге.

Листинг 5.10 ❖ Тест для `hello`

```
package hello  ← Тест всегда помещается в тот же пакет, что и тестируемый им код

import "testing"  ← Пакет testing содержит встроенные инструменты тестирования языка Go
```

```
func TestHello(t *testing.T) {          ← Функция TestHello соответствует шаблону
    if v := Hello(); v != "hello" {        тестовой функции
        t.Errorf("Expected 'hello', but got '%s'", v)  ← Отчет об ошибках
    }                                                     через объект
}                                                         *testing.T
```

В этом примере нашли отражение типичные черты Go-тестов. Язык Go не дает так много инструментов проверки утверждений, как другие фреймворки тестирования (хотя для этого имеются дополнительные библиотеки). Но объект `testing.T` предусматривает функции создания отчетов о непредвиденных ситуациях. Ниже перечислены наиболее часто используемые функции объекта `testing.T`:

- функции `T.Error(args …interface{})` и `T.Errorf(msg string, args interface{})` выводят сообщение и отмечают тест как потерпевший неудачу. Вторая функция позволяет форматировать строки, как показано в листинге 5.10;
- функции `T.Fatal(args …interface{})` и `T.Fatalf(msg string, args interface{})` выводят сообщение и отмечают тест как потерпевший неудачу с остановкой тестирования. Их следует использовать всяких раз, когда неудача одного теста препятствует удачному выполнению других тестов.

Существуют и другие функции, позволяющие пропускать тесты, останавливать тестирование и так далее. Учитывая все это, рассмотрим несколько методов тестирования.

РЕЦЕПТ 27 Использование интерфейсов для создания макетов и заглушек

Система типов данных в языке Go ориентирована скорее на композицию, а не наследование. Вместо создания больших деревьев объектов Go-разработчики используют интерфейсы, описывающие требуемое поведение. Все, что соответствует типу данных интерфейса, можно считать объектом этого типа. Например, на практике часто используется интерфейс `io.Writer`. Его определение представлено в следующем листинге.

Листинг 5.11 ❖ Интерфейс вывода

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

Интерфейс `io.Writer` применяется ко всему, что способно выводить последовательность байтов, согласно приведенной сигнатуре.

Типы `os.File` и `net.Conn` реализуют интерфейс `io.Writer`, как и многие другие типы. Одной из самых удачных особенностей системы типов в языке Go является отсутствие необходимости явно объявлять, какие интерфейсы реализует тот или иной тип (хотя для документирования это полезно). Можно даже объявить интерфейс, который соответствует некоторым свойствам существующего типа, и тем самым создать полезную абстракцию. Нигде это не приносит такой пользы, как при тестировании.

Проблема

Требуется написать код, зависящий от типов, объявляемых во внешних библиотеках, и тест, проверяющий правильность использования этих библиотек.

Решение

Создадим интерфейсы для описания типов, которые требуется протестировать. Используем эти интерфейсы в коде, а затем напишем заглушки с фиктивными реализациями для тестов.

Обсуждение

Предположим, что нужно написать программу, использующую сторонние библиотеки, как показано в следующем листинге.

Листинг 5.12 ❖ Структура сообщения

```
type Message struct {
     // ...
}

func (m *Message) Send(email, subject string, body []byte) error {
     // ...
     return nil
}
```

Это – описание системы отправки сообщений определенного вида. В вашей программе эта библиотека используется для отправки сообщений. Перед нами стоит цель – написать тесты, помогающие убедиться, что код, отправляющий сообщение, действительно вызывается, но сообщения при этом не должны отправляться. Для решения поставленной задачи можно написать собственный интерфейс, описывающий метод, представленный в листинге 5.12, и использовать этот интерфейс в своих объявлениях вместо непосредственного обращения к типу данных `Message`, как показано в следующем листинге.

Листинг 5.13 ❖ Использование интерфейса

```
type Messager interface {
     Send(email, subject string, body []byte) error
}

func Alert(m Messager, problem []byte) error {
     return m.Send("noc@example.com", "Critical Error", problem)
}
```

Определение интерфейса, описывающего метод, используемый типом Message

Передача интерфейса вместо типа Message

Поскольку интерфейс `Messager` представляет собой абстракцию типа `Message`, вы легко сможете написать заглушку и использовать ее для тестирования, как показано в следующем листинге.

Листинг 5.14 ❖ Тестирование с помощью заглушки

```
package msg

import (
     "testing"
)

type MockMessage struct {
     email, subject string
     body           []byte
}

func (m *MockMessage) Send(email, subject string, body []byte) error {
     m.email = email
     m.subject = subject
     m.body = body
     return nil
}

func TestAlert(t *testing.T) {
     msgr := new(MockMessage)
     body := []byte("Critical Error")

     Alert(msgr, body)

     if msgr.subject != "Critical Error" {
           t.Errorf("Expected 'Critical Error', Got '%s'", msgr.subject)
     }
     // ...
}
```

Тип MockMessage реализует интерфейс Messager

← Создание нового MockMessage

← Вызов метода Alert заглушки

Обращение к свойству MockMessage для проверки результата

Тип `MockMessage` реализует интерфейс `Messager`. Этот тип предоставляет ту же функцию, которую использует рабочий код, только вместо отправки сообщения она сохраняет данные. Благодаря этому вы мо-

жете проверить, что в `Messager` действительно была передана ожидаемая информация.

Это простой, но мощный прием тестирования. Кроме того, абстрагирование с помощью интерфейсов упрощает изменение реализации в будущем. Такой подход способствует модульному программированию.

РЕЦЕПТ 28 Проверка интерфейсов с помощью канареечных тестов

Предыдущий пример иллюстрирует прием, основанный на использовании интерфейсов для описания существующего набора функций. Но иногда источником неполадок при выполнении являются тонкие ошибки в определениях интерфейсов. Это особенно характерно для случаев, когда приходится полагаться на проверки типов или использовать внешние библиотеки, сигнатуры функций в которых часто меняются. Таких неполадок можно избежать с помощью достаточно тривиального приема тестирования.

Интерфейсы редко изменяются

В идеале, после того как интерфейс экспортируется и делается общедоступным, он не должен изменяться. Но в мире разработки программного обеспечения это ожидание оправдывается далеко не всегда. Авторы библиотек время от времени изменяют интерфейсы, внося исправления или добавляя новые функции. Такое изменение интерфейсов считается нормой, если сопровождается увеличением главного номера версии программы (например, номер версии 1.2.3 меняется на 2.0.0). Однако нужно иметь в виду, что многие проекты, включая довольно крупные, не следуют этой рекомендации.

Проблема

Необходимо удостовериться, что используемые интерфейсы соответствуют их описаниям. Это может потребоваться в следующих четырех случаях:

- при экспортировании типов, реализующих внешние интерфейсы;
- при создании интерфейсов, описывающих внешние типы;
- при использовании внешних интерфейсов, которые изменились (хотя предполагается, что этого не должно было случиться);
- когда использование интерфейсов ограничивается операциями проверки типов (это будет проиллюстрировано примером).

Решение

Напишем «канареечные» тесты проверки типов, неудача которых позволит быстро выявить ошибки, допущенные при определении интерфейса.

Обсуждение

При написании или реализации интерфейсов, особенно когда сведения о типах выясняются во время выполнения, имеет смысл создавать канареечные тесты с простыми проверками типов, которые позволят выявлять ошибки уже на этапе компиляции.

Предположим, что нужно создать инструмент записи, реализующий интерфейс `io.Writer`. Этот инструмент будет экспортироваться библиотекой, чтобы им можно было воспользоваться из другого кода. Реализация этого инструмента приводится в следующем листинге.

Листинг 5.15 ❖ Тип MyWriter

```
type MyWriter struct{
    // …
}

func (m *MyWriter) Write([]byte) error {
    // Запись данных куда-либо…
    return nil
}
```

Этот код выглядит просто, и на первый взгляд кажется, что он реализует интерфейс `io.Writer`. А теперь представьте, что он используется с утверждением о типе, как в следующем листинге.

Листинг 5.16 ❖ Доступ к средству записи

```
func main() {
     m := map[string]interface{}{
            "w": &MyWriter{},
     }
}

func doSomething(m map[string]interface{}) {
     w := m["w"].(io.Writer)   ← Этот оператор вызовет исключение во время выполнения
}
```

Этот код успешно компилируется и даже может успешно проходить тестирование. Тем не менее он содержит ошибку.

Чтобы выявить ее, достаточно написать простой канареечный тест. Канареечный тест (основанием для такого названия послужило ис-

пользование канареек в угольных шахтах для раннего предупреждения об опасности) – это тест, предназначенный для выявления ошибок в объявлениях. Следующий канареечный тест определяет, реализует ли тип `MyWriter` интерфейс `io.Writer`.

Листинг 5.17 ❖ Канареечный тест для MyWriter

```
func TestWriter(t *testing.T) {
     var _ io.Writer = &MyWriter{}  ← Компилятор проверит утверждение о типе
}
```

Это простой тест. Вам даже не придется запускать тест, чтобы выявить ошибку, – компилятор просто не сможет выполнить сборку:

```
$ go test
# _/Users/mbutcher/Code/go-in-practice/chapter5/tests/canary
./canary_test.go:15: cannot use MyWriter literal (type *MyWriter)
     as type io.Writer in assignment:
     *MyWriter does not implement io.Writer (wrong type for Write method)
            have Write([]byte) error
            want Write([]byte) (int, error)
FAIL _/Users/mbutcher/Code/go-in-practice/chapter5/tests/canary
[build failed]
```

Тест потерпел неудачу потому, что метод `Write` не соответствует сигнатуре `Write([]byte) (int, error)` интерфейса `io.Writer`. Ошибка компиляции точно сообщает, что следует изменить в методе `Write`, чтобы он соответствовал интерфейсу. Иногда применяются более сложные интерфейсы, чем `io.Writer`, и при их использовании написание канареечных тестов приносит еще более ощутимую пользу.

Последний прием демонстрирует, как создавать интерфейсы для описания существующих типов. Эта стратегия хорошо подходит для обобщающего тестирования. Но создание интерфейсов, соответствующих существующему типу, является еще одним примером ситуации, где могут пригодиться канареечные тесты. Канареечное тестирование интерфейсов позволит выявить ошибки, связанные с изменением сигнатур библиотечных функций.

5.4.2. Порождающее тестирование

Порождающее тестирование – большая и сложная тема. В общем случае под порождающим тестированием понимается стратегия автоматического создания тестовых данных для расширения тестовой информации и исключения субъективных предпочтений при выборе тестовых данных.

Проблема

Необходимо получить надежный код, справляющийся с самыми неожиданными ситуациями.

Решение

Используем пакет `testing/quick` из стандартной библиотеки Go для генерации тестовых данных.

Обсуждение

В языке Go имеется пакет `testing`, о котором часто забывают. Пакет `testing/quick` включает несколько вспомогательных функций для быстрого создания исчерпывающих тестов. Их можно применить не во всех случаях, но иногда они помогают добиться большей надежности тестирования.

Предположим, имеется простая функция, дополняющая переданную строку до заданной длины (или усекающая, если длина строки превышает заданное значение). Код функции приводится в следующем листинге.

Листинг 5.18 ❖ Функция дополнения

```
func Pad(s string, max uint) string {
     log.Printf("Testing Len: %d, Str: %s\n", max, s)
     ln := uint(len(s))
     if ln > max {
         return s[:max-1]
     }
     s += strings.Repeat(" ", int(max-ln))
     return s
}
```

Журналирование данных для удобства

Если строка длиннее, чем нужно, – обрезать

← Дополнить строку до нужной длины

Обычно для проверки подобных функций пишутся простые тесты, один из которых представлен в следующем листинге.

Листинг 5.19 ❖ Модульный тест для функции pad

```
func TestPad(t *testing.T) {
    if r := Pad("test", 6); len(r) != 6 {
        t.Errorf("Expected 6, got %d", len(r))
    }
}
```

Неудивительно, что этот тест будет успешно пройден. Однако именно эта функция отлично подходит для тестирования с помощью генератора. Вне зависимости от переданной строки она всегда долж-

на возвращать строку заданной длины. С помощью функции `Check()` из пакета `testing/quick` можно выполнить тестирование на широком диапазоне строк (включая строки, содержащие самые неожиданные символы), как показано в следующем листинге.

Листинг 5.20 ❖ Порождающий тест для функции дополнения

```
func TestPadGenerative(t *testing.T) {
     fn := func(s string, max uint8) bool {
            p := Pad(s, uint(max))
            return len(p) == int(max)
     }

     if err := quick.Check(fn, &quick.Config{MaxCount: 200}); err != nil {
            t.Error(err)
      }
}
```

Функция fn принимает строку и значение с типом данных uint8, вызывает Pad() и проверяет длину возвращаемой строки

С помощью пакета testing/quick функция fn вызывается с 200 строками, сгенерированными случайным образом

Отчет об ошибках выводится с помощью обычных средств из пакета testing package

Функция `Check` из пакета `testing/quick` является сердцем теста. Она принимает определяемую вами функцию и необязательные конфигурационные данные, а затем конструирует многочисленные тесты. На основании анализа параметров она генерирует случайные тестовые данные соответствующих типов. Например, для тестирования более длинных строк в функции `fn` можно заменить тип данных `uint8` на `uint16`.

Если запустить этот тест, он выявит ошибку:

```
$ go test
2015/07/21 09:20:15 Testing Len: 6, Str: test
2015/07/21 09:06:09 Testing Len: 32, Str: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
--- FAIL: TestPadGenerative (0.00s)
     generative_test.go:39: #1: failed on input
     "\U000305ea\U000664e9\U000cbd92\U00091bbf\U0010b40d\U000fd581…",   0x20
FAIL
exit status 1
FAIL _/Users/mbutcher/Code/go-in-practice/chapter5/tests/generative
0.005s
```

Что же случилось? Одна из сгенерированных строк оказалась длиннее заданного максимума, что привело к выполнению кода усечения (листинг 5.17), не охваченного предыдущим тестом. Так случилось, что усечение выполняется неправильно, – выражение `s[:max-1]` нужно заменить на `s[:max]`, потому что здесь задается длина, а не ин-

декс. После внесения исправлений и повторного запуска тест должен провести множество успешных испытаний с использованием случайно сгенерированных значений.

Генератор случайных чисел в Go

Если значения генерируются случайно, почему при повторном выполнении теста генерируются те же самые строки? При каждом его выполнении среда выполнения Go не передает методу `"math/rand".Rand` новое начальное значение. Если требуется в каждом запуске получить разные последовательности случайных чисел, нужно организовать инициализацию генератора случайных чисел разными начальными значениями вызовом метода `"testing/quick".Config`. Это хороший способ расширить диапазон тестовых данных, но за это придется заплатить отсутствием повторяемости. Столкнувшись с неудачей, вам придется заметить данные, которые вызвали проблему, потому что они могут не повториться еще довольно долго.

Пакет `testing/quick` включает множество других утилит. Большинство из них предназначено для быстрого создания тестовых данных, как это было сделано в предыдущем примере. В Go реализован хороший генератор случайных значений, способный формировать не только данные простых типов, такие как целые числа и строки, но и случайные экземпляры структур.

5.5. Тестирование производительности и хронометраж

Какой механизм сопоставления путей действует быстрее – из стандартной библиотеки или пользовательский, основанный на регулярных выражениях? Как этот шаблон ветвления действует на практике? Почему HTTP-сервер работает так медленно? Опытные программисты часто ставят перед собой задачу выявления и устранения проблем, связанных с производительностью. И для этого в языке Go имеются полезные инструменты.

Пакет `testing` включает несколько функций тестирования производительности, многократно выполняющих фрагменты кода, с последующим выводом данных об их эффективности.

РЕЦЕПТ 29 Хронометраж кода на языке Go

В этом рецепте вы узнаете, как пользоваться инструментом хронометража `testing.B` для оценки эффективности фрагментов кода.

Проблема

Существует несколько способов решения поставленной задачи, и требуется определить, какой из них работает быстрее. Что будет выполняться быстрее, форматирование с помощью пакета `text/template` или `fmt`?

Решение

Используем функцию хронометража `testing.B` для сравнения эффективности.

Обсуждение

Хронометраж выполняется аналогично тестированию. Для его проведения создаются файлы *_test.go* с тестами, которые запускаются командой `go test`. Но они имеют другую структуру.

В следующем листинге представлен написанный с нуля код для оценки среднего времени, необходимого для компиляции и применения простого текстового шаблона.

Листинг 5.21 ❖ Хронометраж компиляции и применения шаблона

```
package main

import (
    "bytes"
    "testing"
    "text/template"
)
                                                    Функция BenchmarkTemplates получает
func BenchmarkTemplates(b *testing.B) {  ← значение с типом данных *testing.B
     b.Logf("b.N is %d\n", b.N)  ← Вывод значения b.N
     tpl := "Hello {{.Name}}"
     data := &map[string]string{
            "Name": "World",
     }
     var buf bytes.Buffer
     for i := 0; i < b.N; i++ {  ← Выполнения ядра теста b.N раз
            t, _ := template.New("test").Parse(tpl)  ← Анализ шаблона
            t.Execute(&buf, data)  ← Выполнение шаблона
            buf.Reset()  ← Очистка буфера для исключения проблем с памятью
     }
}
```

Подобно тому, как имена тестов начинаются с префикса `Test`, имена функций хронометража начинаются с префикса `Benchmark`. И вместо значения типа `*testing.T` принимают значение типа `*testing.B`. Экземпляр `*testing.B` не только имеет большинство методов, которыми обладает экземпляр `*testing.T`, он еще имеет несколько свойств, специально предназначенных для хронометража. Наиболее важным из них является структура `N`. В предыдущем листинге она используется как верхняя граница цикла. Это ключ к хронометражу. Любой тест измерения производительности должен выполниться `*b.N` раз. Инструмент хронометража повторно запускает один и тот же тест и пытается получить значимое представление о производительности кода, изменяя количество итераций тестирования.

Если запустить этот пример, он должен вывести следующее:

```
$ go test -bench .
testing: warning: no tests to run
PASS
BenchmarkTemplates 100000 10102 ns/op
--- BENCH: BenchmarkTemplates
     bench_test.go:10: b.N is 1
     bench_test.go:10: b.N is 100
     bench_test.go:10: b.N is 10000
     bench_test.go:10: b.N is 100000
ok    /Users/mbutcher/Code/go-in-practice/chapter5/tests/bench    1.145s
```

Для запуска хронометража используется инструмент `go test`, но на этот раз ему передается параметр `-bench PATTERN`, где `PATTERN` – регулярное выражение, соответствующее именам функций, которые необходимо выполнить. Точка (`.`) указывает, что должны быть выполнены все тесты.

Предыдущий пример вывода сообщает, что тест выполнен 100 000 раз и для выполнения каждой итерации в цикле потребовалось в среднем 10 102 наносекунды. Благодаря включению метода `Printf` мы получили дополнительную информацию о том, как производится хронометраж. Сначала в `b.N` устанавливается самое низкое значение: 1. Затем значение `b.N` увеличивается (не всегда экспоненциально), пока алгоритмы хронометража не выведут усредненных значений.

Код в листинге 5.21 тестирует только один сценарий, который вы легко могли бы оптимизировать. Давайте расширим пример и добавим еще один тест, как показано в следующем листинге.

Листинг 5.22 ❖ Хронометраж двух шаблонов

```
func BenchmarkTemplates(b *testing.B) {
     b.Logf("b.N is %d\n", b.N)
     tpl := "Hello {{.Name}}"
     data := &map[string]string{
            "Name": "World",
     }
     var buf bytes.Buffer
     for i := 0; i < b.N; i++ {
            t, _ := template.New("test").Parse(tpl)
            t.Execute(&buf, data)
            buf.Reset()
     }
}

func BenchmarkCompiledTemplates(b *testing.B) {
     b.Logf("b.N is %d\n", b.N)
     tpl := "Hello {{.Name}}"
     t, _ := template.New("test").Parse(tpl)   ← Вынести компиляцию шаблона
     data := &map[string]string{                  за пределы цикла
            "Name": "World",
     }
     var buf bytes.Buffer
     for i := 0; i < b.N; i++ {
            t.Execute(&buf, data)
            buf.Reset()
     }
}
```

Второй тест `BenchmarkCompiledTemplates` компилирует шаблон один раз и затем многократно его использует. Нетрудно догадаться, что такая оптимизация сокращает время выполнения, но на сколько?

```
$ go test -bench .
testing: warning: no tests to run
PASS
BenchmarkTemplates 20000010167 ns/op
--- BENCH: BenchmarkTemplates
     bench_test.go:10: b.N is 1
     bench_test.go:10: b.N is 100
     bench_test.go:10: b.N is 10000
     bench_test.go:10: b.N is 200000
BenchmarkCompiledTemplates 1000000 1318 ns/op
--- BENCH: BenchmarkCompiledTemplates
     bench_test.go:23: b.N is 1
```

```
    bench_test.go:23: b.N is 100
    bench_test.go:23: b.N is 10000
    bench_test.go:23: b.N is 1000000
ok    _/Users/mbutcher/Code/go-in-practice/chapter5/tests/bench    3.483s
```

Эти результаты говорят о том, что повторное использование скомпилированного шаблона сокращает среднее время выполнения одной итерации почти на 9000 наносекунд. Применение скомпилированного шаблона занимает лишь одну десятую исходного времени! Пакет хронометража включает также другие функции, способные помочь в улучшении производительности. Далее мы рассмотрим еще один способ, предназначенный для хронометража параллельных вычислений.

РЕЦЕПТ 30 Хронометраж параллельных вычислений

Одной из сильных сторон языка Go является его модель параллельных вычислений, основанная на сопрограммах. Но как оценить производительность некоторого фрагмента кода, если он будет выполняться в нескольких сопрограммах? В этом поможет тот же инструмент хронометража.

Проблема

Требуется оценить производительность фрагмента кода, выполняющегося в сопрограммах. В идеале такую проверку желательно проводить при наличии разного количества процессоров.

Решение

Специально для этих целей в экземпляре `*testing.B` имеется метод `RunParallel`. В сочетании с флагами командной строки он позволяет проверить эффективность параллельных вычислений в сопрограммах.

Обсуждение

Код в листинге 5.21 можно использовать как шаблон теста, оценивающего производительность параллельных вычислений. Вместо цикла фреймворк несколько раз вызовет функцию как сопрограмму.

Листинг 5.23 ❖ Оценка производительности параллельных вычислений

```
func BenchmarkParallelTemplates(b *testing.B) {
    tpl := "Hello {{.Name}}"
    t, _ := template.New("test").Parse(tpl)
    data := &map[string]string{
          "Name": "World",
    }
```

```
    b.RunParallel(func(pb *testing.PB) {  ← Вместо цикла for выполняется
          var buf bytes.Buffer                передача функции в RunParallel
          for pb.Next() {
                t.Execute(&buf, data) buf.Reset()
          }
    })
}
```

Большая часть кода теста осталась без изменений. Но вместо выполнения вызовов `t.Execute()` в цикле здесь производится вызов метода `RunParallel`, запускающего функцию в нескольких сопрограммах. Каждая из них получает признак необходимости продолжения итераций через функцию `pb.Next()`. (И снова возникает необходимость выполнения в цикле.) Данный пример практически в точности повторяет пример, включенный в документацию с описанием языка Go.

Теперь попробуем запустить эту программу на одном процессоре (этот режим устанавливается по умолчанию) и сравним ее результаты с результатами, полученными в двух других примерах (хотя в нем отсутствует функция `Logf()`):

```
$ go test -bench .
testing: warning: no tests to run PASS
BenchmarkTemplates 20000 10695 ns/op
BenchmarkCompiledTemplates 1000000 1406 ns/op
BenchmarkParallelTemplates 1000000 1404 ns/op
ok   _/Users/mbutcher/Code/go-in-practice/chapter5/tests/bench   5.097s
```

Параллельная версия не превосходит в скорости обычную версию. Почему? Потому что сопрограммы выполняются только на одном процессоре. Теперь укажем инструменту тестирования, что он должен запустить код несколько раз с разным количеством процессоров:

```
$ go test -bench . -cpu=1,2,4
testing: warning: no tests to run
PASS
BenchmarkTemplates                200000      10019 ns/op
BenchmarkTemplates-2              100000      14033 ns/op
BenchmarkTemplates-4              100000      14971 ns/op
BenchmarkCompiledTemplates        1000000      1217 ns/op
BenchmarkCompiledTemplates-2      1000000      1137 ns/op
BenchmarkCompiledTemplates-4      1000000      1307 ns/op
BenchmarkParallelTemplates        1000000      1249 ns/op
BenchmarkParallelTemplates-2      2000000       784 ns/op
BenchmarkParallelTemplates-4      2000000       829 ns/op
ok   _/Users/mbutcher/Code/go-in-practice/chapter5/tests/bench   14.993s
```

Флаг `-cpu=1,2,4` сообщает команде `go test`, что она должна измерить производительность с одним, двумя, а затем и с четырьмя процессорами соответственно. Она проведет тестирование, как указано, и выведет результаты, добавляя *-N*, чтобы отразить факт использования нескольких процессоров.

Неудивительно, что скорость вычислений в единственном потоке осталась прежней, даже при использовании нескольких центральных процессоров. При наличии единственной главной сопрограммы никакая оптимизация невозможна. Заметное снижение производительности объясняется увеличением накладных расходов, связанных с переключением сопрограмм, но в результатах `BenchmarkParallelTemplates` наблюдается существенное увеличение производительности. Время выполнения уменьшается примерно на треть при распределении нагрузки между несколькими ядрами процессора. Несколько большее время выполнения при использовании четырех ядер, по сравнению с двумя, может указывать на замедление за счет блокировок.

А что, если при попытке достичь высокой скорости мы допустили тривиальную ошибку?

РЕЦЕПТ 31 Выявление состояний гонки

Когда обработка данных распределяется по нескольким сопрограммам и процессорам, возникает риск доступа к сущностям в неправильном порядке. Когда две или более сопрограмм пытаются одновременно изменить один и тот же фрагмент информации, возникает *состояние гонки*. Если порядок действий отличается от задуманного, можно получить обескураживающий результат. Такие состояния обычно трудно диагностировать. Но в языке Go имеется средство обнаружения состояния гонки, и предыдущий рецепт оценки производительности параллельных вычислений дает нам возможность опробовать его.

Проблема

В программах с несколькими сопрограммами может возникнуть состояние гонки. Было бы желательно иметь возможность выявлять ее.

Решение

Используем флаг `-race` (иногда называемый *go race* или *grace*).

Обсуждение

Начнем с примера в листинге 5.22 и осуществим непродуманную оптимизацию производительности. Вместо выделения нового буфе-

ра для каждой сопрограммы создадим общий буфер. Это позволит уменьшить затраты памяти.

Листинг 5.24 ❖ Оценка производительности и выявление состояния гонки

```
func BenchmarkParallelOops(b *testing.B) {
     tpl := "Hello {{.Name}}"
     t, _ := template.New("test").Parse(tpl)
     data := &map[string]string{
            "Name": "World",
     }
     var buf bytes.Buffer  ← Вынесен за пределы замыкания
     b.RunParallel(func(pb *testing.PB) {
            for pb.Next() {
                   t.Execute(&buf, data)
                   buf.Reset()
            }
     })
}
```

Теперь запустим этот пример. Скорее всего, эта попытка завершится сообщением об ошибке:

```
$ go test -bench Oops -cpu=1,2,4
testing: warning: no tests to run
PASS
BenchmarkParallelOops     1000000    1371 ns/op
BenchmarkParallelOops-2panic: runtime error: slice bounds out of
     range [recovered]
     panic: runtime error: slice bounds out of range

goroutine 26 [running]:
text/template.errRecover(0x208355f40)
     /usr/local/Cellar/go/1.4.2/libexec/src/text/template/exec.go:100   +0xbc
bytes.(*Buffer).Write(0x2082e2070…, 0x0)
…
```

Трассировка стека содержит несколько подсказок о произошедшем, но основная причина остается неясной. Если запустить тестирование с флагом `-race`, инструмент выявления состояния гонки предоставит более полезную информацию:

```
$ go test -bench Oops -race -cpu=1,2,4
testing: warning: no tests to run
PASS
BenchmarkParallelOops     200000   5675 ns/op
BenchmarkParallelOops-2   ==================
WARNING: DATA RACE
```

```
Write by goroutine 20:
  bytes.(*Buffer).Write()
      /usr/local/Cellar/go/1.4.2/libexec/src/bytes/buffer.go:126 +0x53
      text/template.(*state).walk()
      /usr/local/Cellar/go/1.4.2/libexec/src/text/template/exec.go:182
+0x401
      text/template.(*state).walk()
…
```

Теперь причина стала яснее: несколько сопрограмм пыталось одновременно использовать `bytes.Buffer`. Трассировка стека позволяет определить, где возникло состояние гонки. Несколько сопрограмм выполнило запись одновременно. Решить эту проблему можно двумя способами: вернуться к старому методу или использовать функцию `sync.Mutex` для блокировки и разблокировки доступа к буферу.

Эта иллюстрация является хорошим примером ошибки, возникающей из-за состояния гонки. Ее легко воспроизвести. Но многие другие состояния гонки менее предсказуемы. Гонка может негативно проявляться лишь в некоторых случаях, что затрудняет ее обнаружение и отладку. Здесь поможет флаг `-race`. Его можно использовать не только при оценке производительности (что тоже удобно), но также с командой `go run` и любыми командами `go test`.

5.6. Итоги

Эта глава была посвящена приемам отладки и тестирования Go-программ. Мы рассмотрели инструменты журналирования, трассировки стека, модульного тестирования и оценки производительности. Все они являются фундаментальными инструментами для разработки надежных программ на языке Go.

В этой главе были охвачены следующие вопросы:

- журналирование по сети;
- работа с пакетом `log` из стандартной библиотеки Go;
- захват трассировки стека;
- использование идиом языка Go для написания модульных тестов;
- оценка производительности с помощью инструментов тестирования языка Go;
- простое порождающее тестирование;
- выявление состояний гонки.

В следующих главах вы познакомитесь с рецептами, посвященными подходам к написанию кода.

Часть III

ИНТЕРФЕЙСЫ ПРИЛОЖЕНИЙ

Приложения взаимодействуют с внешним миром посредством прикладного интерфейса и интерфейса пользователя. Этому и будет посвящена часть 3.

Достаточно часто интерфейсы пользователя создаются с применением веб-технологий. Язык Go имеет множество встроенных средств, помогающих разрабатывать и поддерживать такие интерфейсы. Но это лишь базовые функции, не обеспечивающие развитых возможностей, доступных в других платформах. В главе 6 будут представлены приемы работы с шаблонами HTML и электронной почты, более мощные, чем это возможно с применением одной только стандартной библиотеки. В главе 7 будут представлены приемы работы с ресурсами и обработки форм, опирающиеся на рецепты из главы 6.

Архитектурный стиль REST описывает общие способы взаимодействий. Его реализация позволяет организовать взаимодействия с программами на JavaScript, мобильными и настольными приложениями. Кроме того, архитектурный стиль REST широко используется для реализации веб-служб. Приемы взаимодействия с веб-службами и их реализации описываются в главе 8.

Глава 6

Приемы работы с шаблонами HTML и электронной почты

В этой главе рассматриваются следующие темы:

- *добавление функциональных возможностей в шаблоны;*
- *вложенность шаблонов;*
- *наследование шаблонов;*
- *отображение объектов в HTML-разметке;*
- *шаблоны для работы с электронной почтой.*

Во многих программных окружениях для создания ответов в виде текста или разметки HTML требуется использовать специальные библиотеки. Язык Go решает эту проблему немного иначе. Стандартная библиотека Go поддерживает работу с обоими видами шаблонов, текстовыми и в формате HTML. Обработка разметки HTML реализована на основе механизма обработки текстовых шаблонов, за счет добавления возможностей, связанных с HTML.

Хотя стандартная библиотека позволяет работать с HTML-шаблонами, ее возможности не отличаются особой широтой. Вместо этого она закладывает фундамент, обеспечивающий возможности расширения и комбинирования шаблонов. Например, она позволяет вкладывать шаблоны друг в друга и наследовать одни шаблоны от других. Такая простая расширяемая схема позволяет использовать множество общепринятых приемов работы с шаблонами.

В этой главе вы узнаете, как расширить функциональные возможности HTML-шаблонов, и познакомитесь с приемами использования шаблонов в связке. Здесь же будут представлены советы по повыше-

нию производительности приложений. Например, вы узнаете, когда выполнять синтаксический анализ шаблонов, помогающий сэкономить время на их обработку. Также вы увидите, как использовать текстовые шаблоны для отправки электронных писем.

6.1. Работа с HTML-шаблонами

Пакеты `html` и `html/template` из стандартной библиотеки обеспечивают основные возможности работы с HTML-разметкой, включая применение переменных и функций в шаблонах. Пакет `html/template` опирается на пакет `text/template`, предназначенный для обработки текстовых шаблонов. Преимущество пакета `html/template` перед `text/template` заключается в наличии возможностей распознания HTML, значительно облегчающих жизнь разработчиков.

Несмотря на то что эти пакеты служат фундаментом для работы с HTML-разметкой, они не отличаются особой широтой возможностей. Структурирование HTML-шаблонов вменяется в обязанность авторам приложений. Следующие два рецепта будут посвящены приемам, позволяющим применять пакеты для работы с шаблонами в приложениях.

6.1.1. Обзор пакетов для работы с HTML-разметкой в стандартной библиотеке

Перед знакомством с шаблонами следует рассмотреть соответствующие пакеты из стандартной библиотеки. Пакет `html` предоставляет лишь несколько функций для обработки экранированных и неэкранированных HTML-строк, тогда как пакет `html/template` представляет неплохую основу для работы с шаблонами. Начнем с базового HTML-шаблона, представленного в следующем листинге.

Листинг 6.1 ❖ Простой HTML-шаблон: `simple.html`

```
<!DOCTYPE HTML>
<html>
  <head>
    <meta charset="utf-8">
    <title>{{.Title}}</title>  ← Заголовок, использующий свойство Title
  </head>
  <body>
    <h1>{{.Title}}</h1>
    <p>{{.Content}}</p>  | Отображение свойств Title и Content
  </body>
</html>
```

Листинг иллюстрирует основные приемы работы с шаблонами. За исключением присутствия директив, заключенных в двойные фигурные скобки, шаблон ничем не отличается от обычного HTML-файла. Директивы предназначены для вывода передаваемых в шаблон значений, например заголовка. Следующий этап – вызов этого шаблона из кода с передачей ему значений для заполнения `{{.Title}}` и `{{.Content}}`, как показано в следующем листинге.

Листинг 6.2 ❖ Использование простого HTML-шаблона: simple_template.go

```
package main

import (
     "html/template"  ← Использование пакета html вместо text/template
     "net/http"
)

type Page struct {
     Title, Content string
}

func displayPage(w http.ResponseWriter, r *http.Request) {
     p := &Page{
            Title: "An Example",
            Content: "Have fun stormin' da castle.",
     }
     t := template.Must(template.ParseFiles("templates/simple.html"))
     t.Execute(w, p)  ← Вывод HTTP-ответа с использованием шаблона и набора данных
}

func main() {
     http.HandleFunc("/", displayPage)
     http.ListenAndServe(":8080", nil)
}
```

Синтаксический разбор шаблона для дальнейшего использования

Объект с данными для передачи шаблону и вывода значений его свойств

Обслуживание вывода с помощью простого веб-сервера

Это простое приложение получает данные и отображает их с помощью простого веб-сервера, применяя шаблон из листинга 6.1. Здесь вместо пакета `text/template` используется пакет `html/template`, поскольку он лучше подходит для данного контекста и самостоятельно выполняет некоторые операции.

Под словами «лучше подходит для данного контекста» здесь понимается восприятие HTML-шаблонов. Пакет распознает, что происходит внутри шаблонов. Рассмотрим следующий фрагмент шаблона:

```
<a href="/user?id={{.Id}}">{{.Content}}</a>
```

Пакет html/template разумно подходит к подстановке. В зависимости от контекста он применяет соответствующее экранирование. Предыдущий фрагмент автоматически преобразуется в следующий:

```
<a href="/user?id={{.Id | urlquery}}">{{.Content | html}}</a>
```

Переменные (в данном случае .Id и .Content) обрабатываются комплексом функций, выполняющих экранирование содержимого перед включением в конечный результат. Экранированию подвергаются символы, которые иначе могут интерпретироваться как разметка и повлиять на структуру страницы и правильное ее отображение. При использовании пакета text/template подобное экранирование возлагается на разработчика.

Контекстно-зависимое экранирование соответствует безопасной модели, согласно которой шаблоны считаются надежными, а передаваемые пользовательские данные – не заслуживающими доверия и требующими экранирования. Например, если пользователь приложения введет строку <script>alert('busted pwned')</script>, которая должна выводиться системой обработки HTML-шаблонов, она будет экранирована и преобразована в строку <script>alert('busted pwned')</script&gt. Что обеспечит ее безопасный вывод и защиту от атак вида «межсайтовый скриптинг» (cross-site scripting, XSS).

Когда значение переменной должно выводиться как есть, без экранирования, можно воспользоваться типом HTML из пакета html/template. В рецепте 35 приводится пример использования типа HTML для подстановки данных в шаблон без экранирования.

Следующие четыре рецепта демонстрируют способы расширения встроенной системы шаблонов, позволяющие использовать общепринятые приемы обработки шаблонов.

6.1.2. Добавление функциональных возможностей в шаблоны

Шаблоны в языке Go могут включать функции, вызываемые непосредственно из них. Как вы уже видели выше, механизм HTML-шаблонов автоматически выполняет экранирование. Это позволяет добавлять в шаблоны сложную обработку. Пакеты поддержки шаблонов предоставляют чуть менее 20 функций, часть которых предназначена именно для этого.

Например, рассмотрим одну из встроенных функций printf, чья реализация обеспечивается функцией fmt.Sprintf. Ниже показано, как можно использовать ее в шаблоне:

```
{{"output" | printf "%q"}}
```

Этот фрагмент принимает строку `output` и передает ее функции `printf` вместе со строкой формата `%q`. В результате будет выведена строка `output` в кавычках.

РЕЦЕПТ 32 Расширение возможностей шаблонов с помощью функций

Несмотря на поддержку некоторых функций, часто возникает необходимость расширить возможности шаблонов. Расширение возможностей не является чем-то из ряда вон выходящим. Например, нередко требуется вывести дату и время в удобном для чтения формате. Эту полезную возможность можно легко реализовать, сделав ее частью системы шаблонов. Это всего лишь один из множества примеров расширения системы работы с шаблонами.

Проблема

Встроенные функции для использования в шаблонах не покрывают всех потребностей.

Решение

Подобно тому, как язык Go обеспечивает доступность функций в шаблонах (например, функции `fmt.Sprintf`, используемой в шаблонах как `printf`), реализуем поддержку собственной функции.

Обсуждение

Выводить информацию в шаблонах можно различными способами. При том, что логика подготовки данных должна находиться в приложениях, их форматирование и вывод имеет смысл поместить в шаблон. Хорошим примером может служить вывод даты и времени. В приложениях для хранения времени используется конкретный тип данных, например `time.Time`. Вывести эти сведения можно массой способов.

Директивы шаблонов, заключенные в двойные фигурные скобки, могут содержать команды, оперирующие данными. Команды можно объединять в конвейеры, разделяя их символом `|`. Здесь за основу принята идея конвейеров в интерфейсе командной строки UNIX. Язык Go предоставляет интерфейс для добавления команд в список доступных из шаблонов. Благодаря этому шаблоны не ограничены набором встроенных команд. В следующем листинге в шаблон добавляется возможность форматированного вывода дат.

Листинг 6.3 ❖ Добавление функций для шаблонов: date_command.go

```
package main

import (
    "html/template"
    "net/http" "time"
)

var tpl = `<!DOCTYPE HTML>  ← HTML-шаблон в виде строки
<html>
  <head>
    <meta charset="utf-8">
    <title>Date Example</title>
  </head>
  <body>
    <p>{{.Date | dateFormat "Jan 2, 2006"}}</p>  ← Применение команды
   </body>                                          dateFormat к Date
</html>`

var funcMap = template.FuncMap{
     "dateFormat": dateFormat,
}                                  Отображение функций Go в функции для шаблона

func dateFormat(layout string, d time.Time) string {
     return d.Format(layout)
}                                  Функция для преобразования времени в форматированную строку

func serveTemplate(res http.ResponseWriter, req *http.Request) {
     t := template.New("date")  ← Создание нового экземпляра template.Template
     t.Funcs(funcMap)  ← Передача карты с дополнительными функциями механизму шаблонов
     t.Parse(tpl)  ← Синтаксический разбор строки шаблона
     data := struct{ Date time.Time }{
             Date: time.Now(),
     }
     t.Execute(res, data)  ← Отправка шаблона с данными в ответ на запрос
}

func main() {
     http.HandleFunc("/", serveTemplate)
     http.ListenAndServe(":8080", nil)
}                                  Обслуживание шаблона и набора данных с помощью веб-сервера
```

Здесь HTML-шаблон хранится не во внешнем файле, а в строковой переменной. Внутри шаблона значение `Date` передается шаблонной функции `dateFormat` со строкой формата. Важно помнить, что механизм конвейера передает вывод одного элемента конвейера следующему в последнем аргументе.

Поскольку функция `dateFormat` не является одной из встроенных шаблонных функций, необходимо сделать ее доступной из шаблона. Для этого требуется выполнить два действия. Во-первых, необходимо создать карту, отображающую имена функций, используемых в шаблоне, в соответствующие функции на языке Go. Здесь функция с именем `dateFormat` отображается в функцию с тем же именем `dateFormat`. Несмотря на то что здесь имена функций совпадают, это не является обязательным. Имена могут быть разными.

При создании нового экземпляра `template.Template`, чтобы открыть доступ к новым функциям, необходимо вызвать его метод `Funcs` и передать ему карту функций (здесь она называется `funcMap`). После этого шаблон сможет использовать эти функции. Заключительный шаг перед использованием шаблона – синтаксический разбор шаблона в `template.Template`.

С этого момента экземпляр шаблона можно использовать как обычно. Структура данных в данном случае определяется как анонимная структура, содержащая пары ключ/значение. Эта структура данных передается методу `Execute` вместе с `io.Writer` для вывода шаблона. В данном случае там, где в шаблоне встретится команда `dateFormat`, ей будут переданы строка формата `Jan 2, 2006` и экземпляр `time.Time`. В результате значение типа `time.Time` будет преобразовано в строку соответствующего формата.

> **ПРИМЕЧАНИЕ** Дата и время в строке формата должны определяться, как это описано в документации с описанием пакета, на странице http://golang.org/pkg/time/#Time.Format.

Если одну и ту же функцию потребуется использовать в нескольких шаблонах, можно написать отдельную функцию, которая будет создавать шаблоны и добавлять в них новые функции:

```
func parseTemplateString(name, tpl string) *template.Template {
     t:= template.New(name)
     t.Funcs(funcMap)
     t = template.Must(t.Parse(tpl))
     return t
}
```

Эту функцию можно многократно использовать для создания новых объектов шаблонов из строки и включения в них пользовательских функций. Например, в листинге 6.3 ее можно вызвать внутри функции `serveTemplate` вместо синтаксического разбора шаблона

и добавления новых функций. Можно также написать аналогичную функцию для настройки шаблонов из файлов.

6.1.3. Сокращение затрат на синтаксический разбор шаблонов

Синтаксический разбор шаблонов, изначально хранящихся в текстовом виде, влечет определенные затраты. При синтаксическом разборе шаблон преобразуется из строки в объектную модель с узлами различных типов, которую использует среда выполнения Go. Синтаксический разбор выполняется функциями `Parse` и `ParseFiles` из пакета `text/template/parser`. Прямое использование низкоуровневых функций синтаксического разбора не рекомендуется, поскольку при этом легко пропустить вызовы нужных функций.

Приемы, описываемые в следующем рецепте, позволят избежать ненужных затрат, связанных с синтаксическим разбором шаблонов, и ускорить работу приложения.

РЕЦЕПТ 33 Кэширование разобранных шаблонов

Go-приложения, такие как серверы, отвечающие сразу на несколько запросов, могут генерировать множество ответов на запросы от различных клиентов. Если для каждого такого ответа выполнять синтаксический разбор шаблона, приложение будет многократно повторять одни и те же действия. Если можно исключить какую-то часть операций, связанных с формированием ответов, это улучшит производительность приложения.

Проблема

Необходимо избежать многократного синтаксического разбора шаблона во время выполнения приложения.

Решение

Проведем синтаксический разбор шаблона, сохраним его в переменной, в готовой к использованию форме, и будем многократно использовать один и тот же шаблон, когда потребуется отправить ответ.

Обсуждение

Вместо синтаксического разбора шаблона в функции обработки запроса, что означает синтаксический разбор шаблона при обработке каждого запроса, вынесем эту операцию из обработчика. Тогда появится возможность многократно использовать шаблон с различны-

ми наборами данных без повторного синтаксического разбора. Код в следующем листинге – это модифицированная версия кода из листинга 6.2, кэширующая готовый шаблон.

Листинг 6.4 ❖ Кэширование разобранного шаблона: cache_template.go

```
package main

import (
    "html/template"
    "net/http"
)

var t = template.Must(template.ParseFiles("templates/simple.html"))
type Page struct {
     Title, Content string
}

func diaplayPage(w http.ResponseWriter, r *http.Request) {
     p := &Page{
            Title: "An Example",
            Content: "Have fun stormin' da castle.",
     }
     t.Execute(w, p)
}

func main() {
     http.HandleFunc("/", diaplayPage)
     http.ListenAndServe(":8080", nil)
}
```

Синтаксический разбор при инициализации пакета

← Выполнение шаблона в обработчике

Вместо синтаксического разбора шаблона в обработчике запроса, как это делалось в листинге 6.2, теперь шаблон обрабатывается один раз при инициализации пакета. Функция-обработчик выполняет шаблон как обычно.

Как показывают примеры тестирования производительности из главы 5, приложение, выполняющее синтаксический разбор шаблона и многократно использующее готовый шаблон, выполняется быстрее, чем приложение, производящее синтаксический разбор при обработке каждого запроса. Это простой способ ускорить возврат ответов приложением.

6.1.4. Прерывание выполнения шаблона

Любое программное обеспечение может потерпеть неудачу. Обработка шаблонов не исключение. В случае неудачной попытки выполнить шаблон возвращается ошибка. Иногда шаблон успевает произвести

фрагмент ответа, и в результате конечному пользователю будет отправлено незаконченное отображение. Этого следует избегать.

РЕЦЕПТ 34 Обработка ошибок выполнения шаблонов

Во время обработки шаблона вывод результатов осуществляется по мере подготовки выходных данных. Если в середине возникнет ошибка, обработка шаблона прекратится, и программе вернется признак ошибки. Но часть шаблона, обработанная до ошибки, уже будет отправлена конечному пользователю.

Проблема

Ошибки, возникающие при обработке шаблона должны перехватываться до того, как что-либо будет отправлено пользователю. Неполную, испорченную страницу следует заменить чем-то более подходящим, например страницей с текстом сообщения об ошибке.

Решение

Поместим выходные данные обработки шаблона в буфер. Если ошибок не возникнет, отправим содержимое буфера конечному пользователю. В противном случае выполним обработку возникшей ошибки.

Обсуждение

Шаблоны должны быть максимально простыми. Их обработка заключается в выводе данных, форматируемых с помощью функций. Любые ошибки в данных должны обрабатываться до того, как они будут использованы в шаблоне, и функции, вызываемые в шаблоне, должны служить только целям вывода. Это позволит избежать распространения проблемы и ограничить последствия сбоев в обработке шаблонов, что особенно важно при потоковой передаче.

Потоковая передача ответов – полезная особенность. Когда отправка ответа производится по мере обработки шаблона, пользователь раньше начинает получать страницы. Буферизация ответов вызывает задержку их получения конечными пользователями. Пользователи ожидают от веб-приложений производительности, сравнимой с производительностью настольных приложений, и применение потоковой передачи позволяет достичь этого. Ее следует применять всегда, когда это возможно.

Но иногда, как известно, «лучшее – враг хорошего». Если в процессе обработки шаблонов могут возникать ошибки, выходные данные следует записывать в буфер. В этом случае появляется возможность

обрабатывать ошибки до того, как их последствия смогут увидеть конечные пользователи. Код в следующем листинге основан на коде из листинга 6.4 и реализует буферизацию вывода.

Листинг 6.5 ❖ Буферизация вывода шаблона: buffered_template.go

```
package main

import (
     "bytes"
     "fmt"
     "html/template"
     "io"
     "net/http"
)

var t *template.Template
func init() {
     t = template.Must(template.ParseFiles("./templates/simple.html"))
}

type Page struct {
     Title, Content string
}

func diaplayPage(w http.ResponseWriter, r *http.Request) {
     p := &Page{
            Title: "An Example",
            Content: "Have fun stormin' da castle.",
     }
     var b bytes.Buffer
     err := t.Execute(&b, p)
     if err != nil {
          fmt.Fprint(w, "A error occured.")
          return
     }
     b.WriteTo(w)
}

func main() {
     http.HandleFunc("/", diaplayPage)
     http.ListenAndServe(":8080", nil)
}
```

Создание буфера для сохранения результатов обработки шаблона

Обработка ошибок, возникших при обработке шаблона

← Вывод буферизированного ответа

При обработке шаблона запись производится в буфер, а не непосредственно в `http.ResponseWriter`. Если случится ошибка, она будет обработана перед копированием содержимого буфера в `http.ResponseWriter`.

6.1.5. Соединение шаблонов

Для генерации HTML-вывода используется пакет `html/template`. Он благополучно справляется с выводом разметки HTML. Но в документации с его описанием приводятся слишком простые примеры. В приложениях часто бывает необходимо соединять шаблоны, подготавливать их к повторному использованию и управлять ими, кэшировать вывод и многое другое. Далее описываются три способа работы с шаблонами, основанных на использовании стандартной библиотеки, которые позволяют выполнять более сложную обработку. Вот эти три способа: вложение шаблонов, расширение возможностей базового шаблона через наследование и отображение объекта данных в специальный шаблон (например, отображение объекта с информацией о пользователе в шаблон представления пользователя).

РЕЦЕПТ 35 Вложенные шаблоны

Совместное и повторное использование шаблонов, подобно повторному использованию кода, является общепризнанной необходимостью. Обычно в приложениях со множеством веб-страниц подавляющее большинство элементов используется всеми страницами, за исключением лишь некоторых специфичных элементов.

Проблема

Требуется предотвратить дублирование общих участков HTML-разметки в нескольких шаблонах и тем самым упростить обслуживание. Это даст те же преимущества, что и повторное использование программного кода.

Решение

Воспользуемся вложением шаблонов, чтобы организовать повторное использование общих участков HTML-разметки, как показано на рис. 6.1. Подшаблоны обеспечивают повторное использование фрагментов разметки, устраняя ее дублирование.

Обсуждение

Система шаблонов в языке Go проектировалась с учетом возможности обработки нескольких шаблонов и обеспечивает совместное их использование. Родительский шаблон может импортировать другие шаблоны. При обработке родительского шаблона для получения вывода выполняется и обработка подшаблонов. Следующий листинг демонстрирует это, затрагивая несколько важных нюансов.

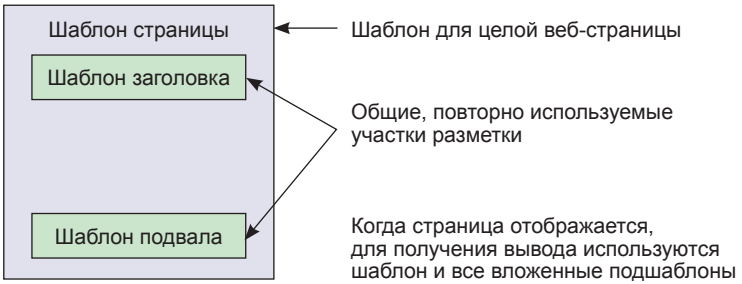


Рис. 6.1 ❖ *Использование вложенных подшаблонов для определения общих участков разметки*

Листинг 6.6 ❖ Шаблон главной страницы, подключающий шаблон заголовка: index.html

```
<!DOCTYPE HTML>
<html>
   {{template "head.html" .}}  ← Включение другого шаблона с передачей ему
   <body>                        полного набора данных
     <h1>{{.Title}}</h1>
     <p>{{.Content}}</p>
   </body>
</html>
```

В этом примере вложенный шаблон включается в начало файла *index.html*. Он похож на простой шаблон из листинга 6.1. Разница заключается в том, что раздел <head> заменила директива включения другого шаблона.

Директива `{{template "head.html" .}}` состоит из трех частей. Ключевое слово `template` требует от движка шаблонов включить другой шаблон, хранящийся в файле a `head.html`. И последний элемент – точка после `head.html` – требует передать вложенному шаблону полный набор данных. В данном случае передается набор данных родительского шаблона. Если одно из свойств набора данных содержит набор данных для подшаблона, можно было бы передать только его (например: `{{template "head.html" .Foo}}`, тогда в шаблоне `head.html` будет доступно лишь свойство `.Foo`). Рассмотрим следующий листинг.

Листинг 6.7 ❖ Шаблон заголовка, включаемый в главный шаблон: head.html

```
<head>
  <meta charset="utf-8">
  <title>{{.Title}}</title>  ← Свойство Title имеет то же значение, что и в index.html
</head>
```

Когда шаблон head.html, представленный в листинге 6.7, включается в шаблон index.html, ему передается полный набор данных. Свойство Title совпадает со свойством Title в шаблоне index.html, потому что шаблон head.html получает доступ к полному набору данных.

Пример в следующем листинге осуществляет объединение шаблонов.

Листинг 6.8 ❖ Использование вложенных шаблонов: nested_templates.go

```
package main

import (
     "html/template"
     "net/http"
)

var t *template.Template func init() {
     t = template.Must(template.ParseFiles("index.html", "head.html"))
}

type Page struct {
     Title, Content string
}

func diaplayPage(w http.ResponseWriter, r *http.Request) {
     p := &Page{
            Title: "An Example",
            Content: "Have fun stormin' da castle.",
     }
     t.ExecuteTemplate(w, "index.html", p)
}

func main() {
     http.HandleFunc("/", diaplayPage)
     http.ListenAndServe(":8080", nil)
}
```

Загрузка двух шаблонов в объект шаблона

← Обработка шаблона с передачей данных

Обслуживание страницы встроенным веб-сервером

Сначала этот пример выполняет синтаксический разбор двух шаблонов и помещает результат в общий объект шаблона. Это гарантирует доступность шаблона head.html при обработке шаблона index.html. Для обработки шаблона используется метод ExecuteTemplate, чтобы можно было указать имя основного шаблона. Если вызвать метод Execute, как в предыдущих листингах, он обработал бы первый шаблон из перечисленных в вызове функции ParseFiles. Метод ExecuteTemplate позволяет управлять выбором шаблона, когда задействовано несколько файлов шаблонов.

РЕЦЕПТ 36 Наследование шаблонов

Многие системы обработки шаблонов реализуют модель, в которой имеется базовый шаблон, а другие восполняют разделы, отсутствующие в базовом шаблоне. Они дополняют базовый шаблон. В этом заключается отличие этого рецепта от предыдущего, где подшаблоны совместно использовались несколькими шаблонами верхнего уровня. В данном случае роль общего играет шаблон верхнего уровня.

Проблема

Требуется создать базовый шаблон и другие шаблоны, дополняющие базовый. Шаблоны должны содержать несколько дополнительных разделов.

Решение

Давайте будем считать шаблонами не целые файлы, а их разделы. Базовый файл содержит общую разметку и ссылается на другие шаблоны, которые еще не определены, как показано на рис. 6.2. Шаблоны, расширяющие базовый файл, предоставляют недостающие подшаблоны или переопределяют их в базовом шаблоне. Соединив их вместе, получим полный рабочий шаблон с общей основой.

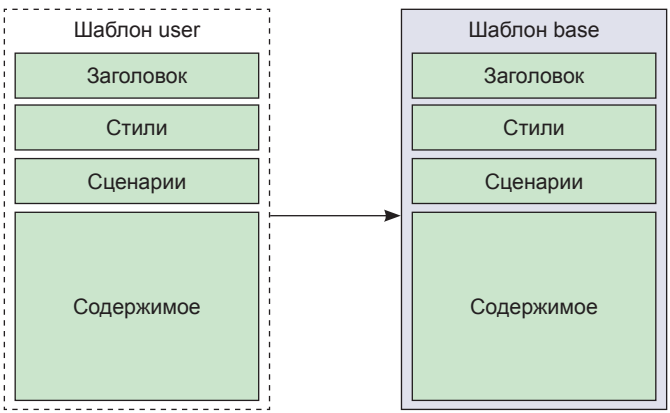


Рис. 6.2 ❖ *Совместно используемый базовый шаблон*

Обсуждение

Система шаблонов поддерживает механизм наследования. Она не предоставляет полного диапазона возможностей, доступных в других

системах обработки шаблонов, но некоторые приемы все же могут быть применены. В следующем листинге приводится базовый шаблон, который наследуется другими шаблонами.

Листинг 6.9 ❖ Шаблон base для наследования другими шаблонами: base.html

```
{{define "base"}}<!DOCTYPE HTML>
<html>
  <head>
    <meta charset="utf-8">
    <title>{{template "title" .}}</title>
    {{ block "styles" .
      }}<style> h1 {
        color: #400080
      }
    </style>{{ end }}
  </head>
  <body>
    <h1>{{template "title" .}}</h1>
    {{template "content" .}}
    {{block "scripts" .}}{{end}}
  </body>
</html>{{end}}
```

← Начало нового базового шаблона с его определением

Включение шаблона title, определяемого в другом месте

Определение и немедленное включение шаблона styles

Определение и включение шаблона scripts, который в настоящее время пуст. Содержимое этого шаблона может переопределить любой дочерний шаблон

← Конец основного шаблона

Вместо одного цельного шаблона файл содержит несколько шаблонов. Каждый шаблон начинается с директивы `define` или `block` и завершается директивой `end`. Директива `block` определяет шаблон и немедленно выполняет его. Этот файл начинается с определения шаблона `base`. Шаблон `base`, на который можно ссылаться по имени, подключает другие шаблоны, не требуя их определения. Расширяющие его шаблоны, такие как шаблон в листинге 6.10, должны определить отсутствующие шаблоны. Иногда в расширяющем шаблоне желательно переопределить содержимое по умолчанию. Некоторые разделы можно не заполнять и создать для них пустые шаблоны, используемые по умолчанию.

> **ПРИМЕЧАНИЕ** Директива `block` и возможность переопределения разделов шаблона появились в версии Go 1.6. До этого отсутствовала возможность переопределения шаблонов.

Листинг 6.10 ❖ Наследование разделов: user.html

```
{{define "title"}}User: {{.Username}}{{end}}
{{define "content"}}
<ul>
  <li>Userame: {{.Username}}</li>
  <li>Name: {{.Name}}</li>
</ul>
{{end}}
```

← Определение шаблона title

Определение шаблона content

Шаблоны, расширяющие базовый шаблон, должны гарантировать заполнение всех подшаблонов, не имеющих содержимого по умолчанию. Здесь должны быть определены разделы `title` и `content`, потому что они объявлены как обязательные. Как видите, необязательные разделы с пустым содержимым или содержимым по умолчанию можно не определять.

Следующий листинг демонстрирует добавление необязательного шаблона в дополнение к обязательным.

Листинг 6.11 ❖ Наследование с необязательным разделом: page.html

```
{{define "title"}}{{.Title}}{{end}}
{{define "content"}}
<p>
   {{.Content}}
</p>
{{end}}
{{define "styles"}}
<style>
h1 {
     color: #800080
}
</style>
{{end}}
```

Определение шаблона для заполнения необязательного раздела родителя

Здесь определяется шаблон `styles`. Он переопределяет значение по умолчанию в листинге 6.9.

В следующем листинге шаблоны объединяются в единое целое.

Листинг 6.12 ❖ Использование наследования шаблонов: inherit.go

```
package main

import (
     "html/template"
     "net/http"
)
```

```
var t map[string]*template.Template   ← Массив для хранения шаблонов с их именами
func init() {
     t = make(map[string]*template.Template)   ← Настройка карты шаблонов
     temp := template.Must(template.ParseFiles("base.html", "user.html"))
     t["user.html"] = temp
     temp = template.Must(template.ParseFiles("base.html", "page.html"))
     t["page.html"] = temp
}                                                  Загрузка шаблонов вместе
                                                     с основным шаблоном

type Page struct {
     Title, Content string
}                                  Передача объектов данных в шаблоны
type User struct {
     Username, Name string
}

func displayPage(w http.ResponseWriter, r *http.Request) {
     p := &Page{                                   Заполнение набора данных
            Title: "An Example",                   для страницы
            Content: "Have fun stormin' da castle.",
     }
     t["page.html"].ExecuteTemplate(w, "base", p)   ← Подключение шаблона
}                                                     к странице

func displayUser(w http.ResponseWriter, r *http.Request) {
     u := &User{
            Username: "swordsmith",
            Name: "Inigo Montoya",
     }
     t["user.html"].ExecuteTemplate(w, "base", u)
}

func main() {
     http.HandleFunc("/user", displayUser)
     http.HandleFunc("/", displayPage)             Обслуживание страниц с помощью
     http.ListenAndServe(":8080", nil)             встроенного сервера
}
```

Листинг начинается с создания ассоциативного массива для хранения шаблонов. Каждый шаблон хранится отдельно. Массив заполняется экземплярами шаблонов с использованием их имен в качестве ключей. Когда загружаются шаблоны `user.html` и `page.html`, вместе с ними загружается файл `base.html`. Это обеспечивает наследование.

Подготовка к отображению страницы выполняется так же, как это делалось для обычного шаблона. Определяется и заполняется набор

данных. Когда наступает момент вернуть ответ, из массива выбирается требуемый шаблон и затем вызывается базовый шаблон. Базовый шаблон является корневым шаблоном страницы, и обработка должна начинаться с него. Он вызовет подшаблоны, наследующие его.

РЕЦЕПТ 37 Отображение типов данных в шаблонах

В предыдущих двух рецептах все шаблоны выводились вместе. Требовалось подготовить для передачи полный набор данных, а сам шаблон должен был обрабатывать все возможные вариации в странице.

Альтернативный подход заключается в отображении страницы по частям, например экземпляра объекта пользователя, с последующей передачей содержимого в шаблон более высокого уровня. В этом случае шаблону верхнего уровня нет необходимости знать тип данных объекта и детали его отображения. Эту идею иллюстрирует рис. 6.3.

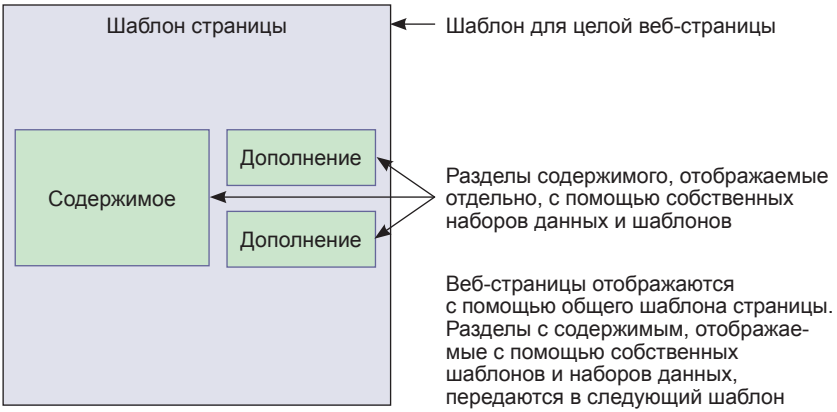


Рис. 6.3 ❖ *В HTML-разметке выводятся объекты, переданные в шаблон*

Проблема

Требуется преобразовать объект в HTML-разметку и передать ее в шаблон более высокого уровня, где он будет использован как часть вывода.

Решение

Используем шаблоны для вывода объектов в виде HTML-разметки. Сохраним HTML-разметку в переменной и будем передавать ее шаб-

лонам более высокого уровня в виде значения типа `template.HTML`, отмеченного как надежного и не требующего экранирования.

Обсуждение

Есть пара причин организовать вывод в несколько шагов. Во-первых, если определенная часть страницы требует существенных затрат на создание набора данных или HTML-разметки, имеет смысл не повторять эту работу при отображении всех страниц.

Предположим, что имеется список сведений о пользователе. Он содержит информацию о пользователе и его деятельности и может передаваться другим пользователям для просмотра. Чтобы получить такой набор данных, требуется выполнить поиск в нескольких источниках. Если сохранить эту информацию в кэше, можно исключить ее загрузку при каждом просмотре страницы.

Кэширование не отменяет вывода набора данных при каждом обращении к странице и требует места для его хранения. Вывод данных в таком случае заключается в их преобразовании пакетом поддержки шаблонов в надлежащий формат с гарантированным экранированием. Если кэшировать готовые фрагменты HTML-разметки для повторного использования, при каждом обращении к странице придется проделать гораздо меньший объем работы.

Во-вторых, предположим, что имеется приложение со сложной логикой и массой страниц. Оно может содержать много шаблонов с повторяющимися фрагментами разметки. Но если распределить обязанности между всеми шаблонами, поручив каждому что-то одно, будь то основное содержимое, боковая панель или оболочка страницы, это упростит управление шаблонами.

В следующем листинге показано, как отобразить объект из шаблона, сохранить HTML-разметку, а затем внедрить ее в другой шаблон.

Листинг 6.13 ❖ Шаблон объекта `Quote`: quote.html

```
<blockquote>
&ldquo;{{.Quote}}&rdquo;
&mdash; {{.Person}}
</blockquote>
```

Свойства объекта Quote для вывода

Шаблон *quote.html* связан с объектом `Quote`. Шаблон используется для отображения объекта `Quote` в виде HTML-разметки и получает поля объекта `Quote` для вывода. Как видите, здесь отсутствуют прочие элементы страницы – они помещены в шаблон *index.html*, приведенный в следующем листинге.

Листинг 6.14 ❖ Общая обертка страницы: index.html

```
<!DOCTYPE HTML>
<html>
  <head>
    <meta charset="utf-8">
    <title>{{.Title}}</title>
  </head>
  <body>
    <h1>{{.Title}}</h1>
    <p>{{.Content}}</p>
  </body>
</html>
```

Свойства для вывода в любую страницу

Файл *index.html* содержит шаблон обертки страницы. Он включает переменные, имеющие отношение ко всей странице. Эти переменные не имеют отношения к выбранному пользователю или чему-то конкретному. Код в следующем листинге объединяет все шаблоны.

Листинг 6.15 ❖ Объединение шаблонов: object_templates.go

```
package main

import (
     "bytes"
     "html/template"
     "net/http"
)

var t *template.Template
var qc template.HTML

func init() {
     t = template.Must(template.ParseFiles("index.html", "quote.html"))
}

type Page struct {
     Title string
     Content template.HTML
}

type Quote struct {
     Quote, Name string
}

func main() {
     q := &Quote{
            Quote: `You keep using that word. I do not think
                   it means what you think it means.`,
            Person: "Inigo Montoya",
```

Загрузка двух файлов шаблонов для дальнейшего использования

Переменные для хранения данных, совместно используемых запросами

Типы для хранения значений общих и специфичных свойств

Заполнение набора данных для передачи в шаблон

```
    }
    var b bytes.Buffer   ← Запись шаблона и данных
    t.ExecuteTemplate(&b, "quote.html", q)   Сохранение цитаты в виде
    qc = template.HTML(b.String())   ← HTML-разметки в глобальной переменной
    http.HandleFunc("/", diaplayPage)  | Обслуживание вывода с помощью
    http.ListenAndServe(":8080", nil)  | простого веб-сервера
}

func diaplayPage(w http.ResponseWriter, r *http.Request) {
    p := &Page{                        |
          Title: "A User",             | Создание страницы с использованием
          Content: qc,                 | HTML-разметки для цитаты
    }                                  |
    t.ExecuteTemplate(w, "index.html", p)   ← Запись цитаты и страницы
}                                               в вывод веб-сервера
```

Начало выглядит вполне обычно: выполняется синтаксический разбор двух шаблонов, *quote.html* и *index.html*, с сохранением результата в переменной. В данном случае используются две структуры данных. Первая предназначена для вывода веб-страницы. Вторая – `Quote` – хранит цитату для преобразования в HTML-разметку.

Чтобы создать фрагмент содержимого отдельно от основной страницы, в функции `main` создается цитата. Затем она передается в метод `ExecuteTemplate` вместе с шаблоном *quote.html* для ее вывода в виде HTML-разметки. Вместо записи шаблона в вывод он помещается в буфер. Далее буфер преобразуется в строку и передается в `template.HTML`. Пакет `html/template` обычно экранирует данные. Исключением является тип `template.HTML`, который содержит безопасную HTML-разметку. Поскольку содержимое сгенерировано из экранированного шаблона, результат обработки шаблона *quote.html* в дальнейшем можно считать безопасной HTML-разметкой.

Свойство `Content` структуры `Page` имеет тип `template.HTML`. Когда создается набор данных для страницы, HTML-разметка, полученная из объекта `Quote`, помещается в свойство `Content`. При подключении набора данных к шаблону *index.html* система шаблонов знает, что данные типа `template.HTML` не требуют экранирования. HTML-разметка цитаты используется без экранирования. Таким способом обеспечивается простой способ хранения и передачи безопасной HTML-разметки.

> **ПРЕДУПРЕЖДЕНИЕ** Информацию, введенную пользователем, никогда не следует считать безопасной. Такую информацию, например полученную из полей формы, всегда экранируйте перед отображением.

6.2. Использование шаблонов при работе с электронной почтой

Электронная почта является одним из современных средств коммуникации. Она часто используется для служебных уведомлений, проверки регистрации и многого другого. Даже службы, для которых работа с электронной почтой не является основной задачей, в той или иной степени используют ее.

Стандартная библиотека Go не имеет специального пакета для работы с шаблонами электронной почты, подобного пакету для обработки HTML-разметки. Но совместное использование шаблонов из пакетов `text` и `html` позволяет отправлять электронные письма с простым текстом и разметкой HTML.

РЕЦЕПТ 38 Генерация электронных писем из шаблонов

Электронная почта является одной из областей, где можно использовать шаблоны. Электронные письма могут формироваться как в виде текста, так и в виде HTML-разметки. При этом можно пользоваться пакетами поддержки шаблонов из стандартной библиотеки.

Проблема

Требуется использовать шаблоны при создании и отправке электронных писем.

Решение

Используем пакеты поддержки шаблонов для создания электронного письма в буфере. Затем реализуем отправку созданных электронных писем, например с помощью пакета `smtp`.

Обсуждение

Шаблоны можно использовать для различных целей, и электронные письма являются прекрасной областью их применения. Для иллюстрации в следующем листинге приводится код, создающий электронные письма из шаблона и отсылающий их с помощью пакета `net/smtp`.

Листинг 6.16 ❖ Отправка электронных писем, созданных из шаблона: email.go

```
package main

import (
    "bytes"
    "net/smtp"
```

```
    "strconv"
    "text/template"  ← Использовать поддержку текстовых шаблонов
)                      для отправки обычных текстовых писем

type EmailMessage struct {
    From, Subject, Body string
    To                  []string
}
```

Структура данных для представления электронного письма

```
type EmailCredentials struct {
    Username, Password, Server string
    Port                       int
}

const emailTemplate = `From: {{.From}}
To: {{.To}}
Subject {{.Subject}}

{{.Body}}
`
```

Шаблон письма в виде строки

```
var t *template.Template

func init() {
    t = template.New("email")
    t.Parse(emailTemplate)
}

func main() {
    message := &EmailMessage{
        From: "me@example.com",
        To: []string{"you@example.com"},
        Subject: "A test",
        Body: "Just saying hi",
    }
```

Заполнение набора данных для шаблона и клиента электронной почты

```
    var body bytes.Buffer
    t.Execute(&body, message)
```

Создание электронного письма по шаблону и запись его в буфер

```
    authCreds := &EmailCredentials{
        Username: "myUsername",
        Password: "myPass",
        Server: "smtp.example.com",
        Port: 25,
    }
    auth := smtp.PlainAuth("",
        authCreds.Username,
        authCreds.Password,
        authCreds.Server,
    )
```

Настройка SMTP-клиента

```
smtp.SendMail(authCreds.Server+":"+strconv.Itoa(authCreds.Port),
        auth,
        message.From,
        message.To,
        body.Bytes())
}
```

Отправка электронного письма

← Содержимое буфера передается методу отправки в виде последовательности байтов

Этот код отправляет простое электронное письмо, созданное из шаблона. Как видите, в данном примере используется пакет `text/template`, а не пакет `html/template`, использовавшийся в предыдущих примерах в этой главе. Пакет `html/template` создан на основе пакета `text/template`. Он добавляет к возможностям пакета `text/template` специфические для HTML-разметки функциональные возможности, такие как экранирование.

Использование пакета `text/template` означает, что внедряемые свойства (например, свойство `.Body`) не будут экранироваться. Если понадобится экранировать что-либо, внедряемое в шаблон, можно использовать функции экранирования из пакета `text/template`.

Результат обработки шаблона с соответствующим набором данных сохраняется в буфере. Буфер служит источником данных для почтового клиента, отправляющего сообщение.

Эту идею можно распространить на отправку разных электронных писем разными способами. Например, для отправки писем в формате HTML можно использовать пакет `html/template`. Или совместить ее с другими приемами для создания сложных шаблонов.

6.3. Итоги

Использование и расширение шаблонов для создания HTML-разметки или электронных писем позволяет справиться со сложностями, труднопреодолимыми другими способами. Это становится особенно заметно с увеличением сложности приложения. В этой главе мы рассмотрели сценарии решения следующих задач:

- расширение функциональных возможностей шаблонов путем добавления нестандартных команд;
- кэширование и буферизация шаблонов;
- определение разделов, совместно используемых несколькими шаблонами. В HTML-шаблонах к таким разделам относятся, например, верхний или нижний колонтитул;
- создание базового, или мастер-шаблона, расширяемого другими шаблонами;

- отображение объектов в шаблоны, например создание шаблона для вывода информации о пользователе и развертывание шаблонов на уровне страницы;
- создание электронных писем с помощью шаблонов.

В следующей главе мы рассмотрим, как обрабатываются статическое содержимое и ввод пользователей из HTML-форм, включая обслуживание файлов, таких как файлы сценариев на JavaScript, таблиц стилей и изображений. Будут показаны приемы обработки HTML-форм с элементами ввода, в том числе для работы с файлами.

Глава 7

Обслуживание и получение ресурсов и форм

В этой главе рассматриваются следующие темы:

- *обслуживание статических файлов, необходимых для предоставления страниц веб-сайта;*
- *обработка HTML-форм, включая выгрузку файлов;*
- *работа с необработанными составными сообщениями.*

Первоначально веб-службы предназначались для обслуживания файлов. Именно такие службы были созданы впервые, в далеком 1991 году, когда зарождался Интернет. Взаимодействий в сегодняшнем их понимании тогда не существовало. Взаимодействия осуществлялись через веб-формы. Эти конструкции, созданные несколько десятилетий назад, по-прежнему широко применяются в современном Интернете и являются основополагающими элементами современных веб-приложений.

Эта глава начинается со знакомства с методами обслуживания статических файлов в Go-приложениях. Поскольку среда выполнения Go может является веб-сервером, а не выполняется на веб-сервере, таком как Apache или Nginx, ее необходимо настроить для обслуживания файлов, таких как каскадные таблицы стилей (CSS), сценарии на JavaScript, изображения и т. д. Далее мы рассмотрим несколько способов хранения и обслуживания файлов, применяемых в различных приложениях.

Начнем с обработки форм. Обработка форм в языке Go может показаться достаточно простым делом, но это верно только для неслож-

ных случаев. В более сложных ситуациях, когда требуется, например, обрабатывать составные данные форм, может понадобиться владение достаточно сложными аспектами языка Go. Это особенно касается работы с большими файлами.

Обслуживание файлов и обработка форм в сочетании с поддержкой шаблонов, рассматривавшейся в предыдущей главе, станут основой для создания веб-приложений на языке Go. Описываемые здесь рецепты позволяют создавать веб-приложения с богатыми возможностями.

7.1. Обслуживание статического содержимого

Веб-сайту или веб-приложению на языке Go не нужен веб-сервер. Для обслуживания любого содержимого, страниц и статических файлов, таких как CSS, JavaScript и изображения, применяется собственный веб-сервер. На рис. 7.1 изображена схема, иллюстрирующая отличия Go-приложения от приложения, выполняющегося под управлением веб-сервера.

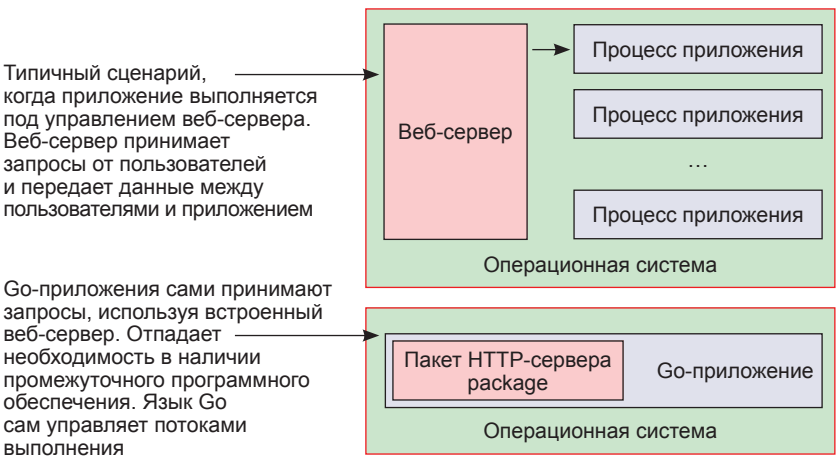


Рис. 7.1 ❖ *Сравнение Go-приложения, взаимодействующего по протоколу HTTP, с типичной моделью веб-сервера*

Интерфейс общего шлюза

Несмотря на то что обычно Go-приложения играют роль веб-сервера, обслуживающего все необходимое содержимое, они могут использоваться совместно с серверами, поддерживающими общий интерфейс шлюза (Common Gateway Interface, CGI) или FastCGI. Пакет `net/http/cgi` поддерживает работу с интерфейсом CGI, а пакет `net/http/fastcgi` – с интерфейсом FastCGI. В таком окружении обслуживание статического содержимого может быть поручено другому веб-серверу. Эти пакеты предназначены для обеспечения совместимости с существующими системами. Интерфейс CGI запускает новый процесс в ответ на каждый запрос, что менее эффективно, чем типичное Go-обслуживание запросов. Авторы не рекомендуют использовать этот подход.

Для обработки статических файлов в пакете `http` из стандартной библиотеки имеется ряд функций. Прежде чем переходить к знакомству с приемами обслуживания файлов в приложениях, желательно чуть больше узнать об этих функциях. Для этого рассмотрим пример, представленный в следующем листинге.

Листинг 7.1 ❖ Обслуживание файлов с помощью пакета `http`: file_serving.go

```
package main

import (
     "net/http"
)

func main() {
      dir := http.Dir("./files")  ← Каталог в файловой системе
      http.ListenAndServe(":8080", http.FileServer(dir))  ← Запустить обслуживание каталога
}
```

Обработчик `FileServer` из пакета `http` – это достаточно продвинутый файл-сервер. Он способен обслуживать файлы, хранящиеся в указанном каталоге в локальной файловой системе, учитывая соответствующие разрешения. Обработчик `FileServer` распознает HTTP-заголовок `If-Modified-Since` и возвращает ответ «304 Not Modified», если версия файла, имеющаяся у пользователя, совпадает с версией, обслуживаемой в настоящий момент.

Если вам понадобится написать собственный обработчик для обслуживания файлов, вам пригодится функция `ServeFile` из пакета `http`, как показано в следующем листинге.

Листинг 7.2 ❖ Обслуживание файлов с помощью пользовательского обработчика: servefile.go

```
package main

import (
     "net/http"
)

func main() {
     http.HandleFunc("/", readme)  ← Регистрация обработчика для всех путей
     http.ListenAndServe(":8080", nil)
}

func readme(res http.ResponseWriter, req *http.Request) {
     http.ServeFile(res, req, "./files/readme.txt")  ← Обслуживать содержимое
}                                                      файла readme.txt
```

В этом примере предпринят другой подход к обслуживанию файла. Основной веб-сервер использует один обработчик для обслуживания всех путей. Обработчик `readme` обслуживает содержимое файла `./files/readme.txt` с помощью функции `ServeFile`. Функция `ServeFile` принимает в третьем аргументе обслуживаемый файл или каталог. Подобно обработчику `FileServer`, функция `ServeFile` также распознает HTTP-заголовок `If-Modified-Since` и возвращает ответ «304 Not Modified», если это возможно.

Такая функциональность позволяет организовать обслуживание содержимого с применением самых разных приемов.

РЕЦЕПТ 39 Обслуживание подкаталогов

Обычной практикой, применяемой во многих фреймворках и приложениях, является обслуживание файлов в локальной файловой системе, где располагается само приложение. Это позволяет другим приложениям монтировать внешние файловые системы, как если бы они были локальными.

Проблема

Требуется организовать обслуживание каталога в файловой системе и его подкаталогов в рамках веб-приложения.

Решение

Используем встроенный файл-сервер или обработчики для обслуживания файлов в локальной файловой системе. Чтобы получить полный контроль над отправкой страниц с сообщениями об ошибках,

включая ошибку отсутствия файла, реализуем собственный файл-сервер.

Обсуждение

Рассмотрим простой пример обслуживания файлов. Возьмем каталог `example_app/files/` и реализуем его обслуживание, отобразив в путь `example.com/static/`. Эта задача может показаться очень простой, что верно в некоторых случаях, но если понадобится получить полный контроль над обслуживанием, вам придется отказаться от встроенной реализации обслуживания файлов в пользу собственной.

Листинг 7.3 ❖ Обслуживание подкаталога

```
func main() {
    dir := http.Dir("./files/")
    handler := http.StripPrefix("/static/", http.FileServer(dir))
    http.Handle("/static/", handler)

    http.HandleFunc("/", homePage)
    http.ListenAndServe(":8080", nil)
}
```

Выбор каталога в файловой системе для обслуживания

Путь /static/ обслуживает каталог и должен удаляться перед определением пути к файлу

← Обслуживание главной страницы, которая может подключать файлы из каталога static

Здесь встроенный веб-сервер обслуживает с помощью файл-сервера из пакета `http` каталог `./files/`, ассоциированный с путем `/static/`. Каталог в файловой системе может быть любым каталогом и не обязательно должен располагаться внутри каталога приложения. Метод `StripPrefix` удаляет префикс из URL-адреса перед передачей пути файл-серверу. Обслуживание вложенных путей требует от приложения искать файлы с их учетом.

Этот подход обладает двумя недостатками. Первый имеет отношение к созданию страниц с сообщениями об ошибках, включая известную ошибку «404 Not Found». Желательно иметь возможность настройки вида этих страниц на веб-сайте или в приложении. К ним обычно применяются собственные стили, или даже создаются специальные страницы ошибок, более полезные конечным пользователям. Обработчик `FileServer` и функция `ServeFile` возвращают базовое сообщение об ошибке в виде текста. Это не веб-страница, а просто текст на английском языке, отображаемый браузерами на белом фоне. При использовании обработчика `FileServer` и функции `ServeFile` нет возможности изменить их вид или язык сообщений. Рецепт 40 демонстрирует, как можно внести такие изменения.

Второй недостаток обойти проще. Для обслуживания каталога и его подкаталогов необходим метод разрешения путей, поддерживающий работу с подкаталогами. Например, если для получения путей использовать пакет `path`, обслуживание будет ограничено базовым каталогом. Если приложение обслуживает файлы и генерирует другое содержимое, разрешению путей следует уделять особое внимание. Для иллюстрации рассмотрим использование экземпляра `pathResolver` из листинга 2.17 в функции `main` из листинга 7.4.

> **СОВЕТ** Маршрутизация с шаблонными символами рассматривалась в главе 2. Листинг 2.17 демонстрирует простой пример, поясняющий описываемую проблему.

Листинг 7.4 ❖ Использование пакета path для определения путей

```
func main() {
     pr := newPathResolver()
     pr.Add("GET /hello", hello)

     dir := http.Dir("./files")
     handler := http.StripPrefix("/static/", http.FileServer(dir))
     pr.Add("GET /static/*", handler.ServeHTTP)

     http.ListenAndServe(":8080", pr)
}
```

Этот код реализует обслуживание содержимого и файлов. Файлы в каталоге *files* будут обслуживаться, но файлы в подкаталогах окажутся недоступными. Это объясняется тем, что при использовании шаблонного символа `*` пакет `path` остается на уровне текущего каталога. Решение заключается в использовании другого метода разрешения путей, например с использованием регулярных выражений, как описывалось в главе 2.

РЕЦЕПТ 40 Файл-сервер с пользовательскими страницами ошибок

Встроенный файл-сервер из стандартной библиотеки Go самостоятельно генерирует страницы с сообщениями об ошибках, включая широко известную страницу «404 Not Found». Это самый обычный текст на английском языке, а не веб-страница, и нет никакой возможности изменить его.

А что, если с приложением будут работать пользователи, не знающие английского языка? И как поступить, если потребуется создать

страницы, помогающие найти нужное содержимое, если при попытке получить его была допущена ошибка? Это вполне обычные требования.

Проблема

Как в приложении, обслуживающем файлы, определить собственные страницы ошибок, в том числе страницу с сообщением, что файл не найден?

Решение

Используем нестандартный файл-сервер, позволяющий определять обработчиков для страниц ошибок. Пакет `github.com/Masterminds/go-fileserver` дает возможность расширять встроенный файл-сервер пользовательской функцией обработки ошибок.

Обсуждение

Обработчик `FileServer` и функция `ServeFile` используют функцию `ServeContent` из пакета `http`. Эта функция вызывает внутренние функции пакета, которые, в свою очередь, генерируют ответы с помощью функций `Error` и `NotFound`. Обработка ошибок жестко зафиксирована на низком уровне. Для ее замены потребуется создать собственный файл-сервер. Это может быть что-то совершенно новое или надстройка над файл-сервером из стандартной библиотеки.

Пакет `github.com/Masterminds/go-fileserver` является надстройкой над файл-сервером из стандартной библиотеки. Он обеспечивает возможность использования пользовательских обработчиков ошибок, включая ошибку «404 Not Found». Этот пакет применяется вместе со стандартным пакетом `http`, но обеспечивает предоставление собственных возможностей, связанных с обслуживанием файлов. Для демонстрации использования этого файл-сервера рассмотрим следующий листинг.

Листинг 7.5 ❖ Страницы ошибок для пользовательского файл-сервера: file_not_found.go

```
package main

import (
     "fmt"
     fs "github.com/Masterminds/go-fileserver"   ← Импорт пакета файл-сервера
     "net/http"
)

func main() {
```

```
    fs.NotFoundHandler = func(w http.ResponseWriter, req *http.Request) {
        w.Header().Set("Content-Type", "text/plain; charset=utf-8")
        fmt.Fprintln(w, "The requested page could not be found.")
    }
    dir := http.Dir("./files")
    http.ListenAndServe(":8080", fs.FileServer(dir))
}
```

Определение функции, вызываемой при неудачном поиске

← Каталог с обслуживаемыми файлами

Использование встроенного веб-сервера и пользовательского файл-сервера

Этот пример похож на пример с использованием файл-сервера из стандартной библиотеки, за исключением нескольких нюансов. Во-первых, для обработки ситуации отсутствия файла определяется своя функция. В этом случае ответ запишет указанная функция. Хотя здесь это не предусмотрено, также можно определить свою функцию для обработки всех ошибок. Настройка обслуживания каталога выполняется так же, как и для файл-сервера из стандартной библиотеки: создается экземпляр `http.Dir` для каталога, где находятся обслуживаемые файлы. Второе отличие связано с обслуживанием файлов. Вместо `http.FileServer` используется функция `fs.FileServer`. Она обеспечивает вызов нужных обработчиков ошибок.

> **ПРИМЕЧАНИЕ** Пакет `github.com/Masterminds/go-fileserver` был специально разработан для этой книги. Из-за большого объема кода, занявшего бы не одну страницу, и реальной полезности такого файл-сервера он оформлен в виде пакета, готового к использованию в приложениях.

РЕЦЕПТ 41 Кэширование в файл-сервере

Иногда чтение файла из файловой системы или другого источника занимает недопустимо много времени. Этот процесс можно ускорить, применив кэширование, и возвращать файлы из памяти, сократив тем самым количество обращений к диску.

Ускорение обслуживания файлов может значительно улучшить быстродействие веб-сайтов с большим трафиком. Средства веб-кэширования, такие как Varnish, набирают популярность и широко используются во многих популярных веб-сайтах и приложениях. Вместе с Go-приложениями также можно использовать средства веб-кэширования для обслуживания из памяти таких файлов, как изображения, CSS и JavaScript. В некоторых случаях альтернативой применению внешнего приложения может стать организация хранения файлов в памяти самим Go-приложением.

Проблема

Вместо извлечения статических файлов из файловой системы при каждом обращении к ним имеет смысл кэшировать их в памяти для ускорения обслуживания запросов.

Решение

Сохраним файлы в памяти при первом обращении к ним и в дальнейшем будем использовать метод `ServeContent` вместо обращений к файл-серверу.

Обсуждение

Как правило, для кэширования и ускорения обслуживания файлов используется реверсный прокси-сервер, например популярный проект с открытым исходным кодом Varnish. Этот прием подразумевает кэширование в памяти часто используемых файлов. Код в следующем листинге демонстрирует, как загрузить файл с диска и обслуживать его из памяти.

Листинг 7.6 ❖ Загрузка статических файлов в память и их обслуживание: cache_serving.go

```
package main

import (
     "bytes"
     "io"
     "net/http"
     "os"
     "sync"
     "time"
)

type cacheFile struct {
     content io.ReadSeeker
     modTime time.Time
}
```

Структура данных для хранения файла в памяти

```
var cache map[string]*cacheFile
var mutex = new(sync.RWMutex)
```

← Мьютекс для устранения состояния гонки при параллельном внесении изменений в кэш

← Карта файлов, хранящихся в памяти

```
func main() {
     cache = make(map[string]*cacheFile)
     http.HandleFunc("/", serveFiles)
     http.ListenAndServe(":8080", nil)
}
```

← Создание карты файлов

```
func serveFiles(res http.ResponseWriter, req *http.Request) {
     mutex.RLock()
     v, found := cache[req.URL.Path]
     mutex.RUnlock()
     if !found {
          mutex.Lock()
          defer mutex.Unlock()
          fileName := "./files" + req.URL.Path
          f, err := os.Open(fileName)
          defer f.Close()
          if err != nil {
               http.NotFound(res, req)
               return
          }
          var b bytes.Buffer
          _, err = io.Copy(&b, f)
          if err != nil {
               http.NotFound(res, req)
               return
          }
          r := bytes.NewReader(b.Bytes())

          info, _ := f.Stat()
          v := &cacheFile{
               content: r,
               modTime: info.ModTime(),
          }
          cache[req.URL.Path] = v
     }
     http.ServeContent(res, req, req.URL.Path, v.modTime, v.content)
}
```

Загрузка файла из кэша, если он уже там

← Если файл отсутствует в кэше, загрузить его

Нельзя записывать в карту сразу несколько файлов или читать их во время записи. Использование мьютекса предотвращает такие попытки

Открыть файл для кэширования и отложить его закрытие

Обработка ошибок открытия файла

Копирование файла в буфер памяти

Обработать ошибки копирования файла в память

Поместить байты в Reader ← для дальнейшего использования

Заполнить объект и сохранить в кэше для дальнейшего использования

Вернуть файл из кэша

Этот пример начинается с определения структуры данных для хранения содержимого файлов в памяти. При обслуживании файлов большое значение имеют дата, время и содержимое. Время передается браузерам и используется как часть HTTP-заголовка `If-Modified-Since`. Хотя здесь это не предусмотрено, значение времени можно использовать для обхода кэша и удаления устаревших элементов. Мониторинг использования памяти кэшем и удаление устаревших элементов являются полезными возможностями.

Прежде всего функция обслуживания файлов пытается получить файл из кэша в памяти. Способность языка Go возвращать несколько значений позволяет одновременно выяснить, есть ли элемент в кэше,

и получить его значение из кэша. Поиск окружен вызовами методов `mutex.RLock` и `mutex.RUnlock`, чтобы предотвратить возникновение состояния гонки, когда два одновременно обрабатываемых запроса вызывают изменение кэша. Эти методы принадлежат объекту `RWMutex` из пакета `sync`. Объект `RWMutex` обеспечивает доступ для произвольного числа читающих процессов или лишь для одного пишущего процесса. Методы `RLock` и `RUnlock` предназначены для использования читающими процессами. Доступ для пишущих процессов будет пояснен чуть ниже.

Если файл отсутствует в кэше, процесс загружает его и сохраняет в кэш. Поскольку при изменении кэша требуется запретить выполнение параллельных операций с ним, вызывается метод `mutex.Lock`. Это заставит все процессы, желающие произвести чтение или запись, ждать снятия блокировки. Освобождение блокировки, чтобы другие процессы смогли получить доступ к кэшу, реализуется отложенным вызовом метода `mutex.UnLock`.

После установки блокировки предпринимается попытка загрузить файл из файловой системы с отложенным закрытием файла при выходе из функции. Если файл не найден или возникла другая ошибка, выводится сообщение «404 Not Found». Это тот момент, когда можно выполнить необязательное кэширование ответа «File Not Found» и записать сообщения об ошибках в журнал.

Содержимое файла копируется в `Buffer` и затем передается экземпляру `Reader`. Это связано с тем, что в экземпляр `Reader` можно записать только тип `Buffer`. Для этого содержимое файла копируется в него. Тип `Reader` реализует интерфейсы `io.Reader` и `io.Seeker`, необходимые функции `ServeContent`. Чтобы дать возможность использовать преимущества HTTP-заголовка `If-Modified-Since`, из файла извлекается время последнего изменения и сохраняется в кэше вместе с содержимым.

Наконец, кэшированный файл возвращается клиенту с помощью функции `ServeContent`. Эта функция выполняет множество действий. Просматривая имя файла, она пытается выяснить MIME-тип и установить соответствующие заголовки. Анализирует время последнего изменения и устанавливает заголовки для HTTP-ответов «304 Not Modified», если это необходимо. При обслуживании содержимого она определяет его размер и устанавливает соответствующие заголовки.

> **СОВЕТ** Использование службы кэширования в памяти, такой как groupcache (https://github.com/golang/groupcache), дает возможность организовать общий кэш для нескольких серверов. Это полезно, когда хранилище файлов перестает быть локальным, что является обычным явлением при масштабировании.

К обслуживанию файлов из памяти следует подходить взвешенно. Загрузка множества файлов без мониторинга и своевременной очистки памяти может вызвать проблемы. За этим процессом обязательно нужно следить и управлять им, оптимизировав под конкретные условия использования. Серверы и приложения не должны произвольно и бесконтрольно наращивать объем используемой памяти.

РЕЦЕПТ 42 Внедрение файлов в исполняемый файл

Иногда бывает необходимо поместить ресурсы непосредственно в исполняемый файл приложения. В этом случае отпадает необходимость их поиска в файловой системе, поскольку они включены в само приложение. Это может пригодиться при распространении приложения. Распространять нужно будет только исполняемый файл, а не вместе с набором сопровождающих его файлов.

Проблема

Необходимо внедрить статические ресурсы, такие как изображения и таблицы стилей, в исполняемый Go-файл.

Решение

Будем хранить ресурсы как данные, присвоенные переменным приложения. Извлекаться эти файлы будут из переменных, а не из файловой системы. Из-за сложности преобразования файлов в наборы байтов и использования их в коде используем пакет `github.com/GeertJohan/go.rice` и приложение командной строки для автоматизации этого процесса.

Обсуждение

Идея проста. Преобразуем файл в последовательность байтов, сохраним ее и сопутствующую информацию в переменной и используем эту переменную для возврата файла с помощью функции `ServeContent` из пакета `http`. Реализация процесса преобразования с учетом изменения состояния этих файлов во время разработки, тестирования и сборки не является здесь главной целью. Именно поэтому авторы рекомендуют использовать соответствующий пакет, например `go.rice`.

Пакет `go.rice` позволяет работать с файлами из файловой системы во время разработки (например, с помощью команды `go run`), использовать файлы, встроенные в исполняемый файл, и создавать сборки с внедрением файлов в исполняемые файлы. Следующий пример демонстрирует простоту использования файлов с помощью пакета `go.rice`.

Листинг 7.7 ❖ Внедрение файлов в исполняемый файл с помощью `go.rice`: embedded_files.go

```
package main

import (
     "github.com/GeertJohan/go.rice"  ← Импорт пакета go.rice для обработки файлов
     "net/http"
)

func main() {
     box := rice.MustFindBox("../files/")  ← Определить местоположение в файловой системе
     httpbox := box.HTTPBox()  ← Метод HTTPBox открывает доступ к файлам, реализуя интерфейс http.FileSystem
     http.ListenAndServe(":8080", http.FileServer(httpbox))  ← Запустить обслуживание файлов
}
```

Обслуживание файлов с пакетом `go.rice` напоминает обслуживание из файловой системы. Вместо `http.DIR` для определения каталога применяется функция `rice.MustFindBox`, которой передается местоположение в файловой системе. Обслуживание файлов осуществляется с помощью встроенного обработчика `FileServer` из пакета `http`. Но вместо объекта `http.Dir` ему передается объект `HTTPBox`. Объект `HTTPBox` реализует интерфейс `http.FileSystem`, требуемый обработчиком `FileServer`.

Если выполнить этот код командой `go run`, он будет читать файлы из файловой системы. Сборка исполняемого файла с внедрением файлов требует дополнительных действий. Чтобы выполнить их, нам понадобится инструмент `rice`, который можно установить из командной строки:

```
$ go get github.com/GeertJohan/go.rice/rice
```

После установки инструмента можно выполнить сборку с помощью следующих двух команд:

```
$ rice embed-go
$ go build
```

Первая команда, `rice embed-go`, преобразует элементы реальной файловой системы в виртуальную файловую систему, помещаемую внутрь Go-файла. Это касается и содержимого файлов. Важно отметить, что данная команда использует функцию `os.Walk`, которая не поддерживает символических ссылок. Команда `go build` создаст обычный исполняемый файл. Этот файл будет включать подготовленные с помощью пакета `rice` Go-файлы, составляющие виртуальную файловую систему.

> **СОВЕТ** Применение методов минификации к встраиваемым файлам, например удаление избыточных пробелов из CSS- и JavaScript-файлов, позволит уменьшить размер исполняемого файла.

Пакет `go.rice` можно использовать с шаблонами. Следующий пример демонстрирует загрузку шаблона из `box`.

Листинг 7.8 ❖ Шаблоны в виде внедренных файлов

```
box := rice.MustFindBox("templates")
templateString, err := box.String("example.html")
if err != nil {
     log.Fatal(err)
}
```

← Определить ссылку на каталог с шаблонами

← Извлечь шаблон в строковую переменную для передачи функции синтаксического разбора

Пакет `go.rice` содержит ряд других вспомогательных функций для работы со встроенными файлами. С его дополнительными возможностями можно ознакомиться на странице https://github.com/GeertJohan/go.rice.

РЕЦЕПТ 43 Обслуживание из альтернативных источников

Иногда требуется хранить и обслуживать файлы отдельно от приложения. Широко распространенным примером является обслуживание файлов JavaScript, CSS и прочих ресурсов из сети доставки содержимого (Content Delivery Network, CDN).

Проблема

Вместо хранения файлов на сервере, где находится приложение, требуется организовать их обслуживание из альтернативного источника. Альтернативные источники должны поддерживать работу с разными окружениями, такими как окружения для эксплуатации, тестирования и разработки.

Решение

В окружении для эксплуатации будем обслуживать файлы из альтернативных источников, таких как CDN. Местоположение файлов для каждого окружения будет определяться настройками в конфигурации. Определим местоположение в файлах шаблонов, чтобы сообщить браузерам, откуда загружать файлы.

Обсуждение

В каждом из окружений нужно иметь свою копию ресурсов приложения. Эти файлы могут обслуживаться отдельно от страниц приложения, но совсем не обязательно использовать один и тот же ис-

точник для всех окружений. Такое явное разделение, как показано на рис. 7.2, позволяет в окружении для тестирования осуществлять полноценное тестирование, разработчикам проводить творческие эксперименты в окружении для разработки и безопасно вести разработку и тестирование, не оказывая влияния на эксплуатационное окружение.

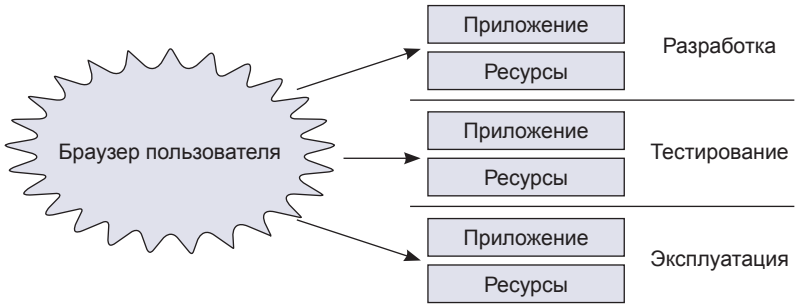


Рис. 7.2 ❖ *Браузер извлекает разные приложения и наборы ресурсов для каждого окружения*

Когда файлы для разных окружений хранятся в разных местах, их местоположение необходимо передавать в приложение через конфигурацию. Это можно осуществить через общую службу настройки, такую как etcd, конфигурационные файлы, с помощью аргументов, передаваемых в приложение в момент запуска и прочих средств передачи конфигурационной информации. Следующий листинг содержит базовый пример передачи местоположения через аргумент командной строки.

Листинг 7.9 ❖ Передача URL-адреса в шаблон

```
var t *template.Template
var l = flag.String("location", "http://localhost:8080", "A location.")

var tpl = `<!DOCTYPE HTML>
<html>
  <head>
    <meta charset="utf-8">
    <title>A Demo</title>
    <link rel="stylesheet" href="{{.Location}}/styles.css">
  </head>
  <body>
    <p>A demo.</p>
```

Получить местоположение статических файлов из аргументов приложения

Путь к CSS-файлу привязан к местоположению

```
  </body>
</html>`

func servePage(res http.ResponseWriter, req *http.Request) {
     data := struct{ Location *string }{
          Location: l,
     }
     t.Execute(res, data)
}
```

HTTP-обработчик передает местоположение в шаблон

В этом элементарном примере местоположение извлекается из аргумента командной строки, а затем используется в шаблоне. Если оно не указано, используется значение по умолчанию.

Этот пример предназначен только для иллюстрации идеи. В реальном программном обеспечении обычно применяются более сложные способы:

1. Передача местоположения в конфигурации. В главе 2 описано несколько способов реализации этого приема, включая конфигурационные файлы, etcd и аргументы командной строки.
2. Если значение не было передано, этот факт следует зафиксировать в журнале и, возможно, вызвать аварию. Отсутствие конфигурационного значения неприемлемо при эксплуатации, и следует предусмотреть проверку на допустимость URL-адреса содержимого.
3. Местоположение может быть включено в глобальный объект с настройками и использоваться при создании ответов приложением.

Если файл-сервер просто возвращает файлы клиентам, его следует оптимизировать для обслуживания статических файлов. Например, при использовании веб-сервера Apache можно отключить ненужные модули.

Последняя версия HTTP-спецификации HTTP/2 определяет функции, предполагающие обслуживание файлов вместе со страницами приложения. Например, если браузер запрашивает страницу, поддерживающий спецификацию HTTP/2 сервер может вернуть вместе со страницей все связанные файлы. Связанные файлы могут быть отправлены браузеру до того, как он их запросит, и все файлы могут быть отправлены через оригинальное соединение, запросившее страницу. В этом случае сервер должен вернуть не только приложение, но и файлы.

> **ПРИМЕЧАНИЕ** Спецификация HTTP/2 описана в RFC 7540 группой Internet Engineering Task Force. С ней можно ознакомиться на странице https://tools.ietf.org/html/ rfc7540[1].

Обслуживание содержимого является лишь частью обязанностей современных интерактивных приложений. Кроме этого, сервер должен обрабатывать взаимодействия с пользователями.

7.2. Обработка форм

Работа с HTML-формами и запросами `POST` и `PUT` является обычной задачей веб-приложений и веб-сайтов. Большинство необходимых для этого функций находится в пакете `http`. Несмотря на то что такая функциональность является встроенной, не всегда очевидно, как ее использовать. Приведенные ниже примеры поясняют методы работы с данными независимо от формы их представления или вида составного запроса `POST` или `PUT`.

7.2.1. Введение в формы

Когда на сервер поступает запрос с данными формы, он по умолчанию не преобразуется в структуру, пригодную для обработки. Большинству Go-программ, обрабатывающих запросы, не приходится иметь дела с формами, поэтому данную функцию требуется определять вручную. Это не сложно. Следующий пример демонстрирует простой способ разбора данных формы и получения доступа к ним:

```
func exampleHandler(w http.ResponseWriter, r *http.Request) {
     name := r.FormValue("name")
}
```

За вызовом метода `FormValue` кроется масса работы. Сначала метод `FormValue` преобразует данные формы в структуру данных. В данном случае предполагается извлекать из формы текстовые и составные данные, такие как файлы. После разбора данных выполняется поиск ключа (имени поля формы) и возвращается первое значение для ключа, если имеется. Если с этим ключом ничего не связано, возвращается пустая строка.

Несмотря на кажущуюся простоту, этот метод выполняет массу работы, порой не нужной в конкретном случае, но не позволяет полу-

---

[1] Перевод на русский язык можно найти по адресу https://bagder.gitbooks.io/http2-explained/content/ru/. – *Прим. ред.*

чить доступ к некоторым возможностям. Например, как поступить, если нужно пропустить поиск и извлечение составных данных, если точно известно, что их нет? Или что делать, если поле формы имеет несколько значений и нужно получить их все?

Первым шагом при работе с формами является их парсинг (как показано в листинге 7.10). Внутри обработчика запросов доступны два метода объекта `Request`, которые могут преобразовать форму в структуру данных. Метод `ParseForm` извлекает текстовые поля. Для работы с файлами и двоичными данными необходимо использовать метод `ParseMultipartForm`. Этот метод работает с составными данными форм (формами, содержащими данные различных MIME-типов). В предыдущем примере метод `ParseMultipartForm` вызывается методом `FormValue` автоматически, потому что разбор формы не был выполнен перед этим.

Данные формы разбиваются на две части:

- свойство `Form` объекта `Request` содержит значения из URL-адреса запроса вместе со значениями, отправленными в теле запроса `POST` или `PUT`. Каждому ключу в `Form` соответствует массив значений. Метод `FormValue` объекта `Request` извлекает первое значение, соответствующее указанному ключу. Это значение с индексом 0 в массиве `Form`;
- если требуется извлечь из тела запроса `POST` или `PUT`, игнорируя значения из строки запроса в URL, можно использовать свойство `PostForm` объекта `Request`. Подобно методу `FormValue`, метод `PostFormValue` извлекает первое значение ключа из свойства `PostForm`.

Листинг 7.10 ❖ Разбор простой формы из запроса

```
func exampleHandler(w http.ResponseWriter, r *http.Request) {
    err := r.ParseForm()  ← Разбор простой формы, содержащей только тестовые поля
    if err != nil {
        fmt.Println(err)  | Обработка ошибок, которые могут
    }                     | возникнуть при разборе формы
    name := r.FormValue("name")  ← Извлечение первого значения из поля name формы
}
```

Код в этом листинге осуществляет обработку простой формы. Этот простой пример демонстрирует обработку форм, содержащих только текстовые поля. Если в форме имеется поле файла, оно не будет извлечено и останется недоступным. Данный код открывает доступ только к одному значению в каждом поле формы. Но поля в HTML-

формах могут содержать несколько значений. Эти два случая рассматриваются в следующих рецептах.

РЕЦЕПТ 44 Доступ к нескольким значениям полей формы

Поля формы могут иметь несколько значений, соответствующих одному полю. Распространенным примером являются флажки. В форме можно создать группу флажков, которым в разметке HTML соответствует одно и то же имя.

Проблема

Методы `FormValue` и `PostFormValue` возвращают только первое значение поля формы. Как при наличии нескольких значений получить доступ ко всем?

Решение

Вместо использования методов `FormValue` и `PostFormValue` для извлечения значений выполним поиск поля в свойстве `Form` или `PostForm` объекта `Request`. Затем выполним обход в цикле всех значений.

Обсуждение

Если поле формы имеет несколько значений, доступ к ним потребует дополнительных усилий. Следующий листинг демонстрирует разбор формы и извлечение нескольких значений из поля.

Листинг 7.11 ❖ Разбор формы с несколькими значениями в поле

```
func exampleHandler(w http.ResponseWriter, r *http.Request) {
     maxMemory := 16 << 20
     err := r.ParseMultipartForm(maxMemory)
     if err != nil {
          fmt.Println(err)
     }
     for k, v := range r.PostForm["names"] {
          fmt.Println(v)
     }
}
```

Максимальный объем памяти для хранения части файла, остальная часть будет помещена на диск

← Разбор составной формы

Обработка всех ошибок, которые могут возникнуть при разборе формы

Обход в цикле всех значений поля names формы из тела запроса POST

Функция обработчика начинается с определения максимального объема памяти для разбора составной формы. В данном случае он составляет 16 мегабайт. В вызов метода `ParseMultipartForm` передается максимальный объем памяти для хранения части файла в памя-

ти. Остальная его часть будет сохранена на диск. Объем по умолчанию, используемый для `FormValue` и `PostFormValue` при вызове метода `ParseMultipartForm`, составляет 32 мегабайта.

Вместо метода `FormValue` или `PostFormValue`, возвращающего только первое значение поля формы, здесь выполняется цикл, выполняющий обход всех значений поля `names`. Поле `names` свойства `PostForm` используется, чтобы ограничиться только значениями, переданными в теле запроса `POST` или `PUT`.

> **СОВЕТ** При передаче форм пользователям и их обработке следует использовать элементы безопасности, такие как токен для защиты от межсайтовой подделки запросов (Cross-Site Request Forgery, CSRF). Более подробные сведения можно найти на странице https://ru.wikipedia.org/wiki/Межсайтовая_подделка_запроса.

7.2.2. Работа с файлами и предоставление составных данных

После перехода от обработки текста к обработке файлов и составных форм, включающих несколько видов содержимого, неизбежно меняется подход к обработке. Простейшим примером является выгрузка файла через форму. Файл хранит содержимое определенного типа, например изображение, кроме того, в форме присутствуют другие текстовые поля. Это как минимум два типа содержимого, которые должны обрабатываться по-разному.

В этом разделе мы рассмотрим обработку составных форм, часто воспринимаемую как обработку файлов. Формы могут выгружать простые и короткие файлы или большие файлы, требующие специальной обработки.

РЕЦЕПТ 45 Выгрузка одного файла

Работа с файлами отличается от работы с текстовыми полями ввода. Каждый файл – это двоичный файл с сопутствующими метаданными.

Проблема

Как обработать и сохранить файл, выгруженный с помощью формы?

Решение

Форму выгрузки файла будем обрабатывать как составную, с помощью метода `ProcessMultipartForm` объекта `Request`. Затем используем метод `FormFile` объекта `Request` для доступа к полю `file` и выгрузим

один файл. Для каждого файла доступны метаданные и объект файла, похожий на объект `File` из пакета `os`.

Обсуждение

Обработка файла реализуется почти так же просто, как обработка текстовых данных. Разница заключается в двоичном файле и сопутствующих ему метаданных, таких как имя файла. Следующий листинг демонстрирует простую форму для выгрузки файла.

Листинг 7.12 ❖ Форма с одним полем для выгрузки файла

```
<!doctype html>
<html>
  <head>
    <title>File Upload</title>
  </head>
  <body>
    <form action="/" method="POST" enctype="multipart/form-data">
      <label for="file">File:</label>
      <input type="file" name="file" id="file">
      <br>
      <button type="submit" name="submit">Submit</button>
    </form>
  </body>
</html>
```

Форма с полем для выгрузки файла должна быть отмечена как составная

Поле с именем "file" для выгрузки файла

Кнопка для отправки формы

Обратите внимание на несколько важных моментов. В этой форме указываются метод отправки `POST` и тип кодирования `multipart/form-data`. Такой тип кодирования применяется, когда форма содержит поле для выгрузки файлов, то есть форма делится на текстовую часть, которая обрабатывается как текст, и файл, который обрабатывается отдельно. Поле ввода с типом `file` сообщает браузеру, что тот должен использовать инструмент выбора файлов и выгрузить содержимое файла на сервер. Эта форма обслуживается и обрабатывается функцией обработчика из пакета `http`, как показано в следующем листинге.

Листинг 7.13 ❖ Обработка выгрузки одного файла

```
func fileForm(w http.ResponseWriter, r *http.Request) {
    if r.Method == "GET" {
        t, _ := template.ParseFiles("file.html")
        t.Execute(w, nil)
```

Обработчик из пакета http отображает и обрабатывает форму из файла file.html

Когда поступает запрос GET, клиенту отправляется HTML-страница с формой

```
    } else {
        f, h, err := r.FormFile("file")
        if err != nil {
              panic(err)
        }
        defer f.Close()
        filename := "/tmp/" + h.Filename
        out, err := os.Create(filename)
        if err != nil {
             panic(err)
        }
        defer out.Close()

        io.Copy(out, f)
        fmt.Fprint(w, "Upload complete")
    }
}
```

Получение указателя на файл, заголовочную информацию и ошибку для поля формы с заданным именем

Обработка любых ошибок, возникших при получении файла

← Гарантировать закрытие файла по завершении функции

← Создать локальный файл

Гарантировать закрытие локального файла перед выходом из функции

← Копировать загруженный файл в локальный

Создать локальный файл с именем загружаемого файла. В данном случае во временном каталоге, но в действующем приложении должно быть определено место для сохранения файлов

Этот обработчик, предназначенный для использования с вебсервером из пакета `http`, посылает форму пользователю и обрабатывает полученную в ней информацию. Он начинается с определения метода запроса. Если получен запрос `GET`, клиенту посылается страница с формой из листинга 7.12. Если получен запрос HTTP-методом `POST` или `PUT`, запускается обработка полученной формы.

Первый шаг в обработке поля файла – извлечение с помощью метода `FormFile` объекта `Request`. Если форма на этот момент еще не была разобрана, метод `FormFile` вызовет метод `ParseMultipartForm`. Метод `FormFile` возвращает объект `multipart.File`, объект `*multipart.FileHeader` и ошибку, если она возникла. Объект `*multipart.FileHeader` содержит свойство `Filename`, значение которого здесь используется как имя файла, сохраняемого в локальной файловой системе. Чтобы сохранить файл, в локальной файловой системе создается новый файл и в него копируется содержимого выгруженного файла.

Это решение хорошо подходит для поля с одним файлом. Но поля HTML-форм могут иметь несколько значений, и в данном случае будет извлечен только первый файл. Следующий рецепт демонстрирует выгрузку нескольких файлов.

РЕЦЕПТ 46 Выгрузка нескольких файлов

Поля формы для выгрузки файлов при необходимости можно отметить атрибутом `multiple`. С этим атрибутом элемент ввода сможет

выполнить выгрузку любого количества файлов. В этом случае нет смысла использовать метод `FormFile` для обработки формы. Этот метод предполагает наличие только одного файла у каждого поля ввода и для каждого вернет только первый файл.

Проблема

Как обрабатывать файлы, когда с помощью одного поля ввода выгружается несколько файлов?

Решение

Вместо использования метода `FormFile`, извлекающего единственный файл, выполним разбор формы и извлечем файлы по ключу `files` из свойства `MultipartForm` объекта `Request`. Затем обойдем в цикле все файлы и обработаем их.

Обсуждение

Поле ввода, позволяющее выгрузить сразу несколько файлов, должно иметь атрибут `multiple`. Например, следующий листинг отличается от листинга 7.12 с формой для выгрузки одного файла только дополнительным атрибутом `multiple`.

Листинг 7.14 ❖ Форма с полем для выгрузки файлов, отмеченным атрибутом multiple

```
<!doctype html>
<html>
  <head>
    <title>File Upload</title>
  </head>
  <body>
    <form action="/" method="POST" enctype="multipart/form-data">
      <label for="files">File:</label>
      <input type="file" name="files" id="files" multiple>
      <br>
      <button type="submit" name="submit">Submit</button>
    </form>
  </body>
</html>
```

Для выгрузки файлов форма должна быть определена как составная

Кнопка отправки формы

Поле ввода с несколькими значениями, с именем "files" и атрибутом multiple

Эта форма с типом кодирования `multipart/form-data` содержит поле ввода для обработки нескольких файлов. Атрибут `multiple` преобразует поле для ввода одного файла в поле для ввода нескольких файлов. Код в следующем листинге обрабатывает эту форму, выполняя выгрузку нескольких файлов.

Листинг 7.15 ❖ Выгрузка нескольких файлов с помощью формы

```
func fileForm(w http.ResponseWriter, r *http.Request) {
    if r.Method == "GET" {
        t, _ := template.ParseFiles("file_multiple.html")
        t.Execute(w, nil)
    } else {
        err := r.ParseMultipartForm(16 << 20)
        if err != nil {
            fmt.Fprint(w, err)
            return
        }
        data := r.MultipartForm
        files := data.File["files"]
        for _, fh := range files {
            f, err := fh.Open()
            defer f.Close()
            if err != nil {
                fmt.Fprint(w, err)
                return
            }
            out, err := os.Create("/tmp/" + fh.Filename)
            defer out.Close()
            if err != nil {
                fmt.Fprint(w, err)
                return
            }
            _, err = io.Copy(out, f)
            if err != nil {
                fmt.Fprintln(w, err)
                return
            }
        }
        fmt.Fprint(w, "Upload complete")
    }
}
```

Обработчик для пакета http, отображает и обрабатывает форму из файла file.html

Когда поступает запрос GET, клиенту отправляется HTML-страница с формой

Разбор формы и обработка любых ошибок

Получение из свойства MultipartForm данных о файлах по ключу, совпадающему с именем поля ввода

Обход выгруженных файлов

Открыть для обработки один из выгруженных файлов

Гарантировать закрытие файла с обработкой всех ошибок

Создать локальный файл для сохранения содержимого выгруженного файла

Гарантировать закрытие локального файла с обработкой всех ошибок при его создании

Копирование выгруженного файла в локальный файл

Обработка всех ошибок копирования выгруженного файла

В этом листинге определяется функция-обработчик для передачи веб-серверу из пакета `http`. Она начинается с отправки формы при поступлении запроса `GET`. Если поступает другой запрос, отличный от `GET`, выполняется обработка полученной формы.

Прежде чем работать с полями, форму нужно подготовить. Для этого вызывается метод `ParseMultipartForm` объекта `Request`. В предыдущих рецептах этот метод вызывался неявно, внутри таких методов, как `FormFile`. Передаваемый методу `ParseMultipartForm` аргумент указывает, что для хранения данных формы в памяти может использоваться не более 16 Мбайт. Не поместившиеся в памяти файлы будут размещены на диске в виде временных файлов.

После разбора формы ее поля становятся доступными через свойство `MultipartForm`. Выгрузка файлов, определенных в поле ввода `files`, выполняется с использованием имен, доступных по ключу `"files"` в виде массива в свойстве `File` объекта `MultipartForm`. Каждое значение в этом массиве является объектом `*multipart.FileHeader`.

Далее выполняются обход файлов и их обработка. Вызов метода `Open` с объектом `*multipart.FileHeader` возвращает экземпляр `File` – обработчик для файла. Чтобы сохранить файл на диск, нужно создать новый файл и скопировать в него содержимое выгруженного файла. Имя файла доступно через свойства `Filename` объекта `*multipart.FileHeader`. После создания локального файла в него, с помощью метода `io.Copy`, копируется выгруженный файл.

Такое решение требует опуститься на нижний уровень программного интерфейса пакета. Зато вы получаете больше возможностей при реализации обработки по своему усмотрению.

РЕЦЕПТ 47 Проверка типов выгруженных файлов

Выгружаемый файл может оказаться файлом любого типа. Но поле ввода может быть предназначено для выгрузки изображений, документов или чего-то еще. Как определить, что было выгружено? Как обработать выгрузку файла неправильного типа?

Иногда для этой цели реализуют определение типа файла на стороне клиента. Например, полю ввода с типом `file` можно придать свойство `accept` со списком расширений или MIME-типов, также называемых *типами содержимого*. К сожалению, свойство `accept` поддерживают не все браузеры. Даже в браузерах, где оно поддерживается, простота изменения его значения делает этот способ весьма ненадежным. Проверка типов должна выполняться в приложении.

Проблема

Как определить тип загруженного файла в приложении?

Решение

Для получения MIME-типа файла можно использовать несколько способов, обеспечивающих различную степень достоверности полученного ответа:

- когда выполняется выгрузка файла, заголовки запросов содержат поле `Content-Type`, значение которого определяет тип содержимого, например `image/png` или универсальный тип `application/octet-stream`;
- расширение файла связано с типом MIME, и по нему можно выяснить тип загруженного файла;
- можно произвести разбор файла и определить тип на основании содержимого.

Обсуждение

Все три решения имеют разную степень достоверности. Значение поля `Content-Type` устанавливается приложением после выгрузки, а расширение файла определяется пользователем, выгружающим файл. Эти два метода полагаются на аккуратность и достоверность определения типа другой стороной. Третье решение требует анализа файла и умения определять тип по содержимому. Это наиболее сложный метод, потребляющий больше системных ресурсов, но он самый надежный. Чтобы разобраться в работе этих методов, рассмотрим каждый из них в отдельности.

Когда происходит выгрузка файлов, как мы видели в рецептах 45 и 46, становится доступным объект `*multipart.FileHeader`. Это второе значение, возвращаемое методом `FormFile` объекта `Request`. Объект `*multipart.FileHeader` имеет свойство `Header`, содержащее все поля загруженных заголовков, включая поле `Content-Type`. Например:

```
file, header, err := r.FormFile("file")
contentType := header.Header["Content-Type"][0]
```

Здесь метод `FormFile` вызывается для поля с именем `file`. Поля заголовка могут содержать несколько значений. В данном случае необходимо получить первое из них, даже при том, что имеется только одно значение. Тип содержимого может быть конкретным MIME-типом, например `image/png`, или универсальным `application/octet-stream`, когда тип неизвестен.

Альтернативой значению в заголовке служит расширение файла, также определяющее его тип. Пакет `mime` содержит функцию `TypeByExtension`, которая пытается вернуть MIME-тип по расширению файла. Например:

```
file, header, err := r.FormFile("file")
extension := filepath.Ext(header.Filename)
type := mime.TypeByExtension(extension)
```

Определение типа по расширению файла дает лишь приблизительную точность. Расширения файлов могут меняться. Стандартная библиотека содержит лишь ограниченное число соответствий расширений MIME-типам, но она способна обратиться к операционной системе для получения более широкого списка.

Другой вариант – анализ файла и определение его типа по содержимому. Это можно реализовать двумя способами. Пакет `http` содержит функцию `DetectContentType`, способную определять типы ограниченного числа видов файлов. К ним относятся файлы HTML, текстовые, XML, PDF, PostScript, популярные графические форматы, сжатые файлы, такие как RAR, Zip и GZip, аудиофайлы в формате wave и видеофайлы в формате WebM.

Следующий пример демонстрирует использование функции `DetectContentType`:

```
file, header, err := r.FormFile("file")
buffer := make([]byte, 512)
_, err = file.Read(buffer)
filetype := http.DetectContentType(buffer)
```

Буфер имеет размер всего 512 байт, потому что функция `DetectContentType` анализирует только первые 512 байт содержимого файла. Если попытка определить тип не увенчается успехом, она вернет `application/octet-stream`.

Ограниченность списка типов содержимого, распознаваемых функцией `DetectContentType`, означает, что необходим другой метод для определения некоторых распространенных типов файлов, таких как документы Microsoft Word, файлы MP4 и многих других распространенных форматов. Самый простой способ определения этих форматов – интеграция с внешней библиотекой, например libmagic. На момент написания книги имелось несколько пакетов с поддержкой библиотеки libmagic, упрощающих ее использование в программах на языке Go.

> **ПРИМЕЧАНИЕ** Описание порядка определения MIME-типов можно найти на странице http://mimesniff.spec.whatwg.org/.

7.2.3. Работа с необработанными составными данными

Предыдущие рецепты обработки файлов хорошо справляются с небольшими файлами и с файлами в целом, но не позволяют работать с файлами в процессе их выгрузки. Например, если вы пишете прокси-сервер и желаете сразу передать файл в другое место, использование этих рецептов приведет к кэшированию больших файлов на прокси-сервере.

Стандартная библиотека Go предоставляет не только высокоуровневые функции для обычных случаев обработки файлов, но и низкоуровневые, для случаев, когда требуется реализовать собственную обработку.

Функция обработки запросов запускается в начале получения запроса, а не после его завершения. Многие запросы выполняются очень быстро, а вспомогательные функции рассчитаны на определенную задержку. Например, при работе с большими файлами можно выполнять некоторые действия, пока продолжается выгрузка файла.

Вместо использования метода `ParseMultipartForm` объекта `Request` внутри функции-обработчика для пакета `http` можно обращаться к необработанному потоку запроса через объект `*multipart.Reader`. Доступ к этому объекту можно получить вызовом метода `MultipartReader` объекта `Request`.

Следующий рецепт демонстрирует использование низкоуровневых функций для обработки составных данных. Этот прием можно использовать как дополнение к обычной обработке.

РЕЦЕПТ 48 Пошаговое сохранение файла

Предположим, что разрабатываемая система предназначена для обработки множества больших файлов. Файлы не сохраняются на сервере, а передаются специализированной службе. Метод `ParseMultipartForm` сохраняет выгружаемые файлы во временном каталоге на сервере в ходе их выгрузки. Для поддержки выгрузки больших файлов с помощью метода `ParseMultipartForm` серверу потребуются много места на диске для хранения файлов и тщательная обработка, предотвращающая переполнение диска при параллельной выгрузке файлов.

Проблема

Требуется сохранять файл по мере его выгрузки в заданном месте. Этим местом может быть сервер, общий диск или что-то еще.

Решение

Вместо использования метода `ParseMultipartForm` будем читать составные данные из запроса по мере их поступления. Это можно осуществить с помощью метода `MultipartReader` объекта `Request`. По мере поступления файлов или других данных, блок за блоком, будем сохранять и обрабатывать их по частям, не дожидаясь завершения выгрузки.

Обсуждение

Использование сервера в качестве транзитного пункта на пути данных к конечной цели является общепринятой моделью. На практике часто приходится сталкиваться с данными, которые не являются файлами и хранятся в базе данных. Обработка больших файлов или параллельная обработка множества файлов становится проблемой при их кэшировании в локальных ресурсах на пути к конечной точке. Проще всего передать решение этой проблемы конечному пункту назначения, который должен быть в состоянии обеспечить хранение больших файлов. Не стоит их кэшировать в локальной системе, если на это нет особых причин.

Прием прямого доступа к потоку составных данных, который поддерживается методом `ParseMultipartForm`, заключается в использовании объекта `Reader`, возвращаемого методом `MultipartReader` объекта `Request`. Получив этот объект, можно организовать обход в цикле блоков данных и читать их по мере поступления.

При обработке составных форм часто требуется обрабатывать поля ввода файлов наряду с текстовыми полями. Следующий листинг содержит простую форму с текстовым полем ввода, полем ввода файла и кнопку отправки.

Листинг 7.16 ❖ HTML-форма с полем ввода файла и текстовым полем

```
<!doctype html>
<html>
  <head>
    <title>File Upload</title>
  </head>
  <body>
    <form action="/" method="POST" enctype="multipart/form-data">
      <label for="name">Name:</label>
      <input type="text" name="name" id="name">
      <br>
      <label for="file">File:</label>
      <input type="file" name="file" id="file">
```

Поле для ввода текста

Поле для ввода файла, присутствие которого требует, чтобы форма была составной

```
        <br>
        <button type="submit" name="submit">Submit</button>
    </form>
  </body>
</html>
```

Кнопка отправки, также доступная как поле

Следующий листинг содержит функцию-обработчик для пакета http, которая отображает и обрабатывает форму из листинга 7.16. В процессе обработки формы эта функция сохраняет файл по частям.

Листинг 7.17 ❖ Пошаговое сохранение выгружаемых файлов

Обработчик для пакета http, отображает и обрабатывает форму из файла file_plus.html

```
func fileForm(w http.ResponseWriter, r *http.Request) {
    if r.Method == "GET" {
        t, _ := template.ParseFiles("file_plus.html")
        t.Execute(w, nil)
    } else {
        mr, err := r.MultipartReader()
        if err != nil {
            panic("Failed to read multipart message")
        }
        values := make(map[string][]string)
        maxValueBytes := int64(10 << 20)
        for {
            part, err := mr.NextPart()
            if err == io.EOF {
                break
            }
            name := part.FormName()
            if name == "" {
                continue
            }
            filename := part.FileName()
            var b bytes.Buffer
            if filename == "" {
                n, err := io.CopyN(&b, part, maxValueBytes)
                if err != nil && err != io.EOF {
                    fmt.Fprint(w, "Error processing form")
                    return
                }
                maxValueBytes -= n
```

Когда поступает запрос GET, клиенту отправляется HTML-страница с формой

Карта для сохранения значений полей запроса, не связанных с файлами

Ограничить 10 мегабайтами объем используемой памяти для хранения полей, не связанных с файлами

Извлечение объекта для чтения составных данных, обеспечивающего доступ к выгружаемым файлам и обработку любых ошибок

Попытаться прочитать следующий блок и прервать цикл, если достигнут конец запроса

Получить имя поля формы, продолжить цикл при отсутствии имени

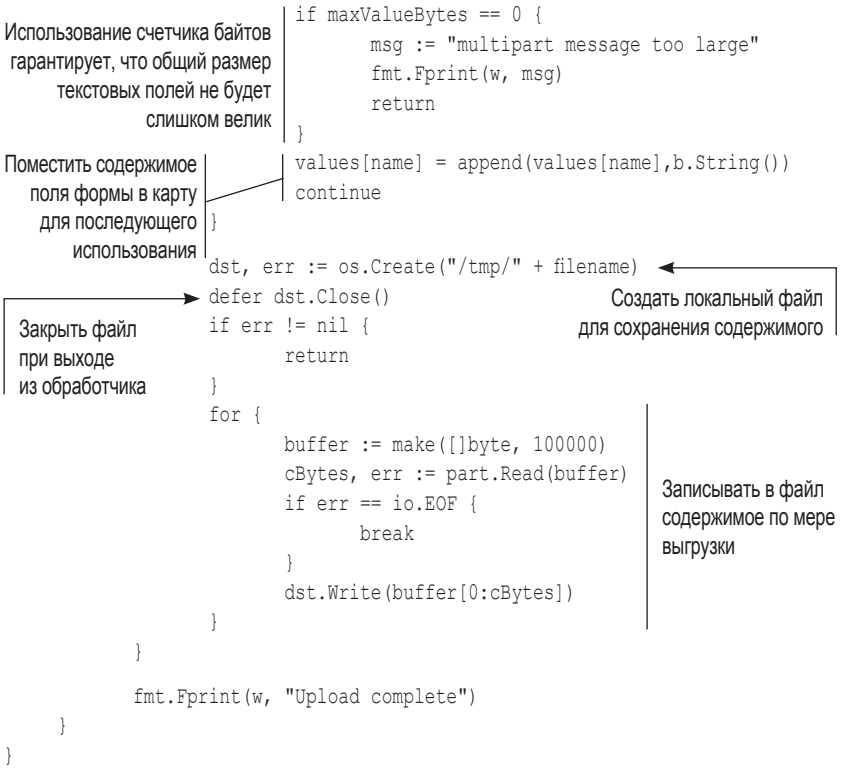
Если имя файла отсутствует, поле интерпретируется как текстовое

Получить имя файла, если имеется

Буфер для чтения значений текстовых полей

Если возникла ошибка при чтении части содержимого, обработать ее

Копировать часть содержимого в буфер




```
        if maxValueBytes == 0 {
                msg := "multipart message too large"
                fmt.Fprint(w, msg)
                return
        }
        values[name] = append(values[name],b.String())
        continue
    }
    dst, err := os.Create("/tmp/" + filename)
    defer dst.Close()
    if err != nil {
            return
    }
    for {
            buffer := make([]byte, 100000)
            cBytes, err := part.Read(buffer)
            if err == io.EOF {
                    break
            }
            dst.Write(buffer[0:cBytes])
    }
}

fmt.Fprint(w, "Upload complete")
}
}
```

Здесь определяется функция-обработчик для пакета `http`. Получив HTTP-запрос `GET`, она выводит HTML-форму. А если поступила форма – обрабатывает ее.

Поскольку функция-обработчик разбирает форму, не полагаясь на метод `ParseMultipartForm`, она может выполнить некоторые настройки до начала работы с формой. Для доступа к данным формы по мере их поступления необходим объект `Reader`. Метод `MultipartReader` объекта `Request` возвращает объект `*mime.Reader`, который можно использовать для итераций по телу составного запроса. Объект `Reader` просматривает данные по мере необходимости. Для полей формы, которые не обрабатываются как файлы, потребуется место для хранения значений. Здесь для этой цели создает ассоциативный массив.

После завершения настройки обработчик выполняет обход частей составного сообщения. Цикл начинается с извлечения следующей части составного сообщения. Если частей больше нет, возвращается

ошибка `io.EOF` и функция прерывает цикл извлечения. Ошибка *EOF* означает *конец файла* (End Of File).

Теперь цикл может начать обработку частей сообщения. Сначала, вызовом метода `FormName`, проверяется имя поля формы, и если оно отсутствует, цикл переходит к следующей итерации. Помимо имени поля, файлы должны иметь еще и имя файла. Его можно извлечь с помощью метода `FileName`. Наличие имени файла помогает отделить поля для выгрузки файлов от текстовых полей.

При отсутствии имени файла обработчик копирует значение поля в буфер и уменьшает значение счетчика объема, изначально равного 10 мегабайтам. При уменьшении значения счетчика до 0 разбор прекращается и возбуждается ошибка. Это – защитная мера, помогающая избежать исчерпания памяти вследствие попыток передать в текстовом поле слишком большой объем данных. Размер 10 Мбайт достаточно большой и используется как значение по умолчанию в методе `ParseMultipartForm`. При отсутствии ошибок содержимое текстового поля формы сохраняется в ранее созданной карте значений (ассоциативном массиве), и цикл разбора переходит к обработке следующей части.

Если цикл достиг этой точки, значит, поле формы является полем выгрузки файла. Чтобы сохранить содержимое этого файла, создается локальный файл. В данный момент можно произвести запись файла в другое место, например в облачное хранилище. Вместо локального файла можно создать подключение к другой системе хранения. После того как доступ к месту назначения будет открыт, обработчик в цикле обрабатывает содержимое по частям, последовательно читая их по мере поступления. Пока не получено уведомление об окончании части в форме ошибки `io.EOF`, поступающие байты последовательно переписываются в место назначения. Например, при выгрузке большого файла можно отслеживать запись данных в выходной файл. После завершения цикла все файлы буду сохранены на диске, а значения текстовых полей доступны через карту `values`.

7.3. Итоги

Без обслуживания файлов и работы с формами не может обойтись ни одно веб-приложение. Эти функции составляют основу Интернета и применяются уже несколько десятилетий. В этой главе были рассмотрены следующие методы, основанные на дополнительных возможностях языка Go:

- выгрузка файлов пользователей на Go-сервер различными способами, в зависимости от потребностей;
- использование вспомогательных функций Go для обработки форм;
- работа с основными элементами парсера форм и обработка результатов его работы;
- получение доступа к механизму обработки составных форм и его использование для разбора и обработки форм.

В следующей главе вы узнаете, как действует REST API. Вы увидите, как разрабатываются интерфейсы REST, как реализуется поддержка нескольких версий, и познакомитесь со множеством других особенностей, знание которых пригодится для создания стабильных и надежных библиотек, которые вы сможете использовать в своих приложениях и передавать другим.

Глава 8

Работа с веб-службами

В этой главе рассматриваются следующие темы:

- *создание REST-запросов;*
- *определение превышения времени ожидания и возобновление загрузки;*
- *передача ошибок по протоколу HTTP;*
- *парсинг данных в формате JSON, включая произвольные JSON-структуры;*
- *версионирование REST API.*

REST API – основа современного Интернета. Они обеспечивают облачные вычисления, стали фундаментом методологии разработки DevOps и автоматизации перемещений, широко используются в разработке клиентских веб-приложений. Они стали движущей силой Интернета.

Несмотря на большое количество руководств, посвященных созданию и применению простых программных интерфейсов, порой трудно найти ответы на возникающие вопросы. Интернет задумывался как отказоустойчивая среда. Серверы должны обрабатывать запросы безотказно.

Эта глава начинается со знакомства с основами REST API и быстро переходит к обсуждению ситуаций, когда что-то пошло не так, как запланировано. Вы увидите, как обнаруживать сбои, связанные с превышением времени ожидания, включая те, которые язык Go формально не причисляет к ошибкам превышения времени ожидания. Вы также узнаете, как возобновить передачу файлов после истечения времени ожидания и как передавать ошибки между конечной точкой API и запрашивающим клиентом.

Многие прикладные программные интерфейсы передают информацию в формате JSON. После краткого знакомства с парсингом

JSON-данных в языке Go вы научитесь обрабатывать JSON-данные с заранее неизвестной структурой. Это может пригодиться при работе со слабо документированными и недокументированными JSON-данными.

Функциональность приложений изменяется с течением времени, что часто приводит к изменениям в прикладном программном интерфейсе. А когда вносятся такие изменения, необходимо обеспечить управление версиями. Здесь вы узнаете, как осуществить такое управление.

Вы узнаете, как перейти от простых API обработки запросов интерфейсами к более надежным и устойчивым функциональным возможностям.

8.1. Использование REST API

Стандартная библиотека Go включает клиента HTTP, прекрасно подходящего для большинства ситуаций. Но при выходе за рамки типичного его использования регулярно возникают вопросы, остающиеся без четкого решения. Однако, прежде чем заняться их обсуждением, необходимо познакомиться с работой HTTP-клиента.

8.1.1. Использование HTTP-клиента

HTTP-клиент входит в стандартный пакет `net/http`. Он предоставляет дополнительные функции для выполнения запросов `GET`, `HEAD` и `POST`, способен выполнить практически любой HTTP-запрос и может настраиваться в самых широких пределах.

В числе вспомогательных функций клиента можно назвать `http.Get`, `http.Head`, `http.Post` и `http.PostForm`. Все функции, за исключением `http.PostForm`, предназначены для обработки HTTP-команд с соответствующими именами. Функция `http.PostForm` обрабатывает `POST`-запросы, когда данные посылаются в виде формы. Для иллюстрации особенностей работы с этими функциями в следующем листинге приводится простой пример использования функции `http.Get`.

Листинг 8.1 ❖ Пример использования функции `http.Get`

```
package main

import (
     "fmt"
     "io/ioutil"
     "net/http"
```

```
)

func main() {
     res, _ :=
     ➥http.Get("http://goinpracticebook.com")  ← Выполнение GET-запроса
     b, _ := ioutil.ReadAll(res.Body)  | Чтение тела ответа и закрытие объекта Body
     res.Body.Close()                  | после завершения чтения
     fmt.Printf("%s", b)  ← Вывод тела в поток стандартного вывода
}
```

Все вспомогательные функции поддерживаются клиентом HTTP по умолчанию, способным выполнить любые HTTP-запросы. Например, следующий листинг демонстрирует использование клиента по умолчанию для выполнения запроса `DELETE`.

Листинг 8.2 ❖ Выполнение запроса DELETE с помощью HTTP-клиента

```
package main

import (
     "fmt"
     "net/http"
)                                          Выполнение запроса с помощью
                                           клиента по умолчанию
func main() {
     req, _ := http.NewRequest("DELETE",   | Создание нового объекта
     ➥"http://example.com/foo/bar", nil)  | HTTP-запроса DELETE
     res, _ := http.DefaultClient.Do(req)  ←
     fmt.Printf("%s", res.Status)  ← Вывод кода состояния выполненного запроса
}
```

Выполнение запроса можно разделить на два этапа. На первом создается экземпляр `http.Request`. Он включает сведения о запросе. На втором этапе клиент отправляет запрос серверу. В этом примере используется клиент по умолчанию. Отделение запроса в виде объекта обеспечивает разделение задач. Это позволяет осуществить настройку обоих этапов. Вспомогательные функции объединяют создание экземпляра запроса и его отправку с помощью клиента.

Клиент по умолчанию при соответствующей настройке способен осуществить перенаправление HTTP-запросов, работу с cookies и определять момент тайм-аута. Он также имеет транспортный уровень по умолчанию, который можно настраивать.

Клиент поддерживает все необходимые настройки для приведения его в соответствие с практически любыми требованиями. Следующий листинг демонстрирует создание простого клиента со временем ожидания в одну секунду.

Листинг 8.3 ❖ Простой пользовательский HTTP-клиент

```
func main() {
     cc := &http.Client{Timeout: time.Second}
     res, err :=
     ➥cc.Get("http://goinpracticebook.com")
     if err != nil {
            fmt.Println(err)
            os.Exit(1)
     }
     b, _ := ioutil.ReadAll(res.Body)
     res.Body.Close()
     fmt.Printf("%s", b)
}
```

← Обработчики любых ошибок, включая ошибку превышения времени ожидания

← Выполнение GET-запроса с помощью пользовательского клиента

Создание пользовательского HTTP-клиента со временем ожидания в одну секунду

Пользовательские клиенты имеют множества параметров настройки, включая транспортный уровень, обработку cookies и перенаправление.

8.1.2. При возникновении сбоев

Интернет задумывался как отказоустойчивая среда. Многое может перестать работать или работать неправильно, и часто требуется предусмотреть различные варианты решения проблем наряду с их регистрацией. В эпоху облачных вычислений появилась возможность переноса приложения в другое место и возобновления его работы. При работе с HTTP-подключениями всегда желательно своевременно обнаруживать проблемы, сообщать о них и пытаться справиться с ними автоматически, если это возможно.

РЕЦЕПТ 49 Определение превышения времени ожидания

Превышение времени ожидания является часто возникающей проблемой, которую необходимо уметь обнаруживать. Например, если время ожидания истекло в середине процесса, возможно, имеет смысл повторить операцию. Но на момент повтора сервер, к которому было выполнено подключение, может быть недоступен, и нужно направить запрос другому работающему серверу.

Для выявления ошибки превышения времени ожидания в пакете `net` предусмотрен метод `Timeout()`, возвращающий значение `true`, если время ожидания истекло. Но в некоторых ситуациях время ожидания может пройти, а метод `Timeout()` не возвращает значения `true`, или используется другой пакет, такой как `url`, который не имеет метода `Timeout()`.

Пакет net выявляет превышение времени ожидания, только когда время задано явно, например как в листинге 8.3. Если время ожидания определено, запрос должен завершиться в течение этого времени. Время на чтение тела ответа включается во время ожидания. Но превышение времени ожидания может возникнуть также, когда оно не задано. В этом случае превышение времени ожидания возникает в сети и не контролируется активными средствами.

Проблема

Как надежно обнаружить истечение времени ожидания в сети?

Решение

Когда истекает время ожидания, возникает несколько ошибок. Проверим наличие этих ошибок, чтобы выявить проблему.

Обсуждение

Если ошибка возникает в процессе выполнения операции пакетом net или другим пакетом, использующим пакет net, таким как http, достаточно проверить наличие ошибки, сопровождающей превышение времени ожидания. Некоторые ошибки имеют прямую связь с превышением заданного интервала времени. Другие возникают, когда время ожидания не задано, но истекло время ожидания в сети.

Следующий листинг содержит функцию, которая проверяет различные ошибочные ситуации, чтобы выяснить, была ли ошибка вызвана превышением времени ожидания.

Листинг 8.4 ❖ Определение превышения времени ожидания в сети по ошибкам

Функция, возвращающая true, если ошибка вызвана превышением времени ожидания

```
func hasTimedOut(err error) bool {
      switch err := err.(type) {
      case *url.Error:
            if err, ok := err.Err.(net.Error); ok && err.Timeout() {
                  return true
            }
      case net.Error:
            if err.Timeout() {
                  return true
            }
      case *net.OpError:
            if err.Timeout() {
                  return true
```

← Оператор switch помогает определить тип ошибки

Ошибка url.Error может возникать из-за превышения времени ожидания в пакете net

Проверить, не обнаружил ли ошибку превышения времени ожидания пакет net

```
            }
        }
        errTxt := "use of closed network connection"
        if err != nil && strings.Contains(err.Error(), errTxt) {
            return true
        }
        return false
}
```

Превышение времени ожидания может быть выявлено дополнительной проверкой некоторых ошибок

Эта функция помогает выявить превышение времени ожидания в различных ситуациях. Ниже демонстрируется пример использования этой функции:

```
res, err := http.Get("http://example.com/test.zip")
if err != nil && hasTimedOut(err) {
    fmt.Println("A timeout error occured")
    return
}
```

Надежное обнаружение превышения времени ожидания является очень полезной возможностью, и его применение на практике демонстрирует следующий рецепт.

РЕЦЕПТ 50 Превышение времени ожидания и возобновление операции

Если при загрузке большого файла происходит превышение времени ожидания, повторение загрузки с самого начала выглядит не самым идеальным решением. С ростом размеров файлов эта проблема становится все более актуальной. Часто приходится иметь дело с файлами размером в несколько гигабайт и более. Неплохо было бы избежать дополнительной нагрузки на соединение и сэкономить время на повторной загрузке уже полученных ранее данных.

Проблема

Требуется возобновить загрузку файла, не загружая еще раз данных, полученных на момент истечения времени ожидания.

Решение

Повторим попытку загрузки, попытавшись использовать HTTP-заголовок `Range`, в котором указывается диапазон байтов для загрузки. Это позволит запросить часть файла, начиная с места, где загрузка прервалась.

Обсуждение

Серверы, подобные тому, что входит в стандартную библиотеку Go, поддерживают работу с фрагментами файлов. Эту возможность обеспечивают многие файл-серверы, а интерфейс определения диапазонов стал стандартным еще в 1999 году, когда вышла спецификация HTTP 1.1:

```
func main() {
     file, err := os.Create("file.zip")
     if err != nil {
            fmt.Println(err)
            return
     }
     defer file.Close()
     location := https://example.com/file.zip
     err = download(location, file, 100)
     if err != nil {
            fmt.Println(err)
            return
     }
     fi, err := file.Stat()
     if err != nil {
            fmt.Println(err)
            return
     }
     fmt.Printf("Got it with %v bytes downloaded", fi.Size())
}
```

Создание локального файла для сохранения загружаемых данных

Загрузка удаленного файла в локальный файл с повторением до 100 раз

Вывод размера файла после завершения загрузки

Этот фрагмент создает локальный файл, загружает удаленный файл, отображает число загруженных байтов и возобновляет загрузку не более 100 раз при превышении времени ожидания. Основную работу выполняет функция `download`, код которой приводится в следующем листинге.

Листинг 8.5 ❖ Загрузка с повторениями

```
func download(location string, file *os.File, retries int64) error {
     req, err := http.NewRequest("GET", location, nil)
     if err != nil {
            return err
     }
     fi, err := file.Stat()
     if err != nil {
            return err
     }
```

Создание нового GET-запроса для загрузки файла

Получение сведений о текущем состоянии локального файла

```
    current := fi.Size()  ← Получение размера локального файла
    if current > 0 {
          start := strconv.FormatInt(current, 10)
          req.Header.Set("Range", "bytes="+start+"-")
    }
```

Если локальный файл частично заполнен, указать в заголовке запроса количество загруженных байтов. Отсчет начинается с 0, то есть текущий размер служит индексом следующего загружаемого байта

```
    cc := &http.Client{Timeout: 5 * time.Minute}
    res, err := cc.Do(req)  ← Выполнение запроса для загрузки файла или части файла, если его часть уже сохранена
```

HTTP явно настроен на проверку превышения времени ожидания

```
    if err != nil && hasTimedOut(err) {
          if retries > 0 {
                return download(location, file, retries-1)
          }
          return err
    } else if err != nil {
          return err
    }
```

Если ошибка связана с превышением времени ожидания, повторить попытку загрузки

```
    if res.StatusCode < 200 || res.StatusCode >= 300 {
          errFmt := "Unsuccess HTTP request. Status: %s"
          return fmt.Errorf(errFmt, res.Status)
    }
```

Обработка ошибочных HTTP-кодов состояний

```
    if res.Header.Get("Accept-Ranges") != "bytes" {
          retries = 0
    }
```

Если сервер не поддерживает обработку файлов по частям, сбросить число попыток в 0

```
    _, err = io.Copy(file, res.Body)  ← Копирование данных ответа в локальный файл
    if err != nil && hasTimedOut(err) {
          if retries > 0 {
                return download(location, file, retries-1)
          }
          return err
    } else if err != nil {
          return err
    }
    return nil
}
```

Если ошибка превышения времени ожидания возникла при копировании файла, попытаться получить оставшуюся часть

Функция `download` успешно справляется с ошибками превышения времени ожидания, используя довольно простой алгоритм, но ее

можно было бы изменить, чтобы дать возможность настраивать под конкретные ситуации:

- время ожидания в функции составляет 5 минут. Его может понадобиться изменить для конкретного приложения. Уменьшение или увеличение продолжительности ожидания иногда может улучшить производительность. Например, если планируется загрузка файлов, время загрузки которых обычно превышает пять минут, увеличение времени ожидания до значения большего, чем время загрузки большинства файлов, позволит ограничить число HTTP-запросов, необходимых для загрузки;
- если доступен хэш файла, можно выполнить проверку соответствия хэша после завершения загрузки. Такая проверка целостности позволяет удостовериться в успешности загрузки после нескольких попыток загрузить файл.

Выявление ошибок и попытки обойти их могут привести к повышению отказоустойчивости приложения.

8.2. Передача и обработка ошибок по протоколу HTTP

Ошибки – обычная часть информации, передаваемой по протоколу HTTP. Двумя наиболее распространенными примерами являются ошибки «Not Found» и «Access Denied». Они возникают настолько часто, что изначально включены в спецификацию HTTP. Стандартная библиотека Go включает простое средство передачи ошибок. Например, код в следующем листинге демонстрирует простой пример передачи ошибки по протоколу HTTP.

Листинг 8.6 ❖ Передача ошибок по протоколу HTTP

```
package main

import "net/http"

func displayError(w http.ResponseWriter, r *http.Request) {
    http.Error(w, "An Error Occurred", http.StatusForbidden)
}

func main() {
    http.HandleFunc("/", displayError)
    http.ListenAndServe(":8080", nil)
}
```

Возврат кода HTTP-состояния 403 с сообщением

← Зарегистрировать displayError как обработчик всех запросов

Этот простой сервер всегда возвращает сообщение «An Error Occurred». Вместе с пользовательским сообщением в виде обычного текста возвращается код HTTP-состояния 403, сообщающий о запрете доступа.

Пакет `http` из стандартной библиотеки определяет несколько констант для различных кодов состояния. Подробные сведения о кодах состояния можно найти на странице https://ru.wikipedia.org/wiki/Список_кодов_состояния_HTTP.

Клиент может читать коды состояния, возвращаемые сервером, и определять, что произошло с запросом. В листинге 8.5 выполняется проверка значения `res.StatusCode` – коды в диапазоне от 200 до 299 говорят об успешном выполнении запроса. Следующий фрагмент демонстрирует простой пример вывода сведений о состоянии:

```
res, _ := http.Get("http://example.com")
fmt.Println(res.Status)
fmt.Println(res.StatusCode)
```

Свойство `res.Status` содержит текстовое сообщение о состоянии. Например, «200 OK» или «404 Not Found». Свойство `res.StatusCode` содержит код ошибки в виде целого числа.

Оба они, код и сообщение об ошибке, с успехом могут использоваться клиентами. Имея их, можно отображать сообщения об ошибках и автоматически обрабатывать возникшие ситуации.

8.2.1. Генерация пользовательских ошибок

Обычного текстового сообщения и кода HTTP-состояния часто бывает недостаточно. Например, нередко желательно выводить веб-страницы с описаниями ошибок, оформленные в том же стиле, что и весь сайт. Или для серверной библиотеки, возвращающей ответы в формате JSON, естественным было бы возвращать сообщения об ошибках также в формате JSON.

Первый этап работы по реализации нестандартных ответов с сообщениями об ошибках – это их создание.

РЕЦЕПТ 51 Передача пользовательских сообщений об ошибках по протоколу HTTP

Использование функции `Error` из пакета `http` не оставляет пространства для маневра. Ответ всегда имеет вид обычного текста, а заголо-

вок `X-Content-Type-Options` всегда имеет значение `nosniff`. Такой заголовок сообщает браузерам, таким как Microsoft Internet Explorer и Google Chrome, чтобы они не пытались определить другой тип содержимого. Это ограничивает возможность поддержки любых других пользовательских сообщений об ошибках, кроме обычной текстовой строки.

Проблема

Как реализовать нестандартное тело и тип содержимого ответа, сообщающего об ошибке?

Решение

Вместо встроенной функции `Error` используем пользовательскую функцию, которая посылает код HTTP-состояния и сообщение об ошибке, более подходящее для конкретной ситуации.

Обсуждение

Поддержка сообщений об ошибках, содержащих не только текстовое описание, способно облегчить жизнь пользователям приложения. Например, представьте, что некто, пытаясь перейти на страницу, получает ошибку «404 Not Found». Если страница с сообщением об ошибке выглядит как прочие страницы сайта и содержит информацию, помогающую пользователям найти то, что они ищут, она послужит рекомендацией, а не только озадачит тем, что страница не найдена.

Второй пример связан с сообщениями об ошибках REST API. Прикладные программные интерфейсы обычно используются инструментами разработчика (SDK) или приложениями. Например, если в ответ на запрос возвращается сообщение «409 Conflict message», наличие более подробной информации принесло бы пользователю гораздо больше пользы. Существуют ли специальные коды ошибок для приложений, которые может использовать SDK? Можно ли передавать пользователю дополнительные пояснения вдобавок к сообщению об ошибке?

Для иллюстрации рассмотрим ответ с сообщением об ошибке в формате JSON. Здесь используется тот же формат ответа, что и в других ответах REST API, содержащий код ошибки приложения в дополнение к ошибке HTTP. Этот пример ориентирован на ответы прикладного программного интерфейса, но тот же подход применим к веб-страницам.

Листинг 8.7 ❖ Ответ с сообщением об ошибке в формате JSON

```
type Error struct {
     HTTPCode int    `json:"-"`
     Code     int    `json:"code,omitempty"`
     Message  string `json:"message"`
}

func JSONError(w http.ResponseWriter, e Error) {
     data := struct {
         Err Error `json:"error"`
     }{e}

     b, err := json.Marshal(data)
     if err != nil {
            http.Error(w, "Internal Server Error", 500)
            return
     }

     w.Header().Set("Content-Type", "application/json")
     w.WriteHeader(e.HTTPCode)
     fmt.Fprint(w, string(b))
}

func displayError(w http.ResponseWriter, r *http.Request) {
     e := Error{
            HTTPCode: http.StatusForbidden,
            Code:     123,
            Message:  "An Error Occurred",
     }
     JSONError(w, e)
}

func main() {
     http.HandleFunc("/", displayError)
     http.ListenAndServe(":8080", nil)
}
```

Тип для хранения информации об ошибке, включая метаданные о структуре JSON

Обертывание структуры Error анонимной структурой со свойством error

Функция JSONError похожа на функцию http.Error, но тело ответа имеет формат JSON

Обертывание структуры Error анонимной структурой со свойством error

Установка MIME-типа ответа в application/json

Гарантирование правильной установки кода ошибки HTTP

Вывод тела в формате JSON

Создание экземпляра Error для использования в ответе с описанием ошибки

Возврат сообщения об ошибке в формате JSON при вызове обработчика HTTP

Этот листинг концептуально похож на листинг 8.6. Разница лишь в том, что код в листинге 8.6 возвращает строку с сообщением об ошибке, а код в листинге 8.7 возвращает следующий ответ в формате JSON:

```
{
     "error": {
          "code": 123,
          "message": "An Error Occurred"
     }
}
```

После передачи ошибки в формате JSON приложение может воспользоваться преимуществами этого структурированного формата передачи данных. Использование сведений об ошибке в формате JSON демонстрирует следующий рецепт.

8.2.2. Чтение и использование пользовательских сообщений об ошибках

Любой клиент может работать с кодами HTTP-состояния для обнаружения ошибки. Например, следующий фрагмент определяет различные типы ошибок:

```
res, err := http.Get("http://goinpracticebook.com/")
switch {
case 300 <= res.StatusCode && res.StatusCode < 400:
     fmt.Println("Redirect message")
case 400 <= res.StatusCode && res.StatusCode < 500:
     fmt.Println("Client error")
case 500 <= res.StatusCode && res.StatusCode < 600:
     fmt.Println("Server error")
}
```

Сообщения об ошибках с кодами в диапазоне от 300 до 399 связаны с переадресацией. Они редко возникают, поскольку настройки HTTP-клиента по умолчанию предусматривают до 10 переадресаций. Сообщения об ошибках с кодами в диапазоне от 400 до 499 представляют ошибки клиента. В этот диапазон входят ошибки «Access Denied», «Not Found» и прочие. Сообщения об ошибках с кодами в диапазоне от 500 до 599 представляют ошибки сервера, оповещающие, что на сервере что-то пошло не так.

Код состояния помогает понять суть происходящего. Например, код 401 сообщает, что для обработки запроса необходимо зарегистрироваться. В ответ на эту ошибку пользовательский интерфейс может дать возможность зарегистрироваться и повторить запрос или SDK может попытаться проверить и перепроверить подлинность и повторить запрос.

РЕЦЕПТ 52 Чтение пользовательских ошибок

Если приложение возвращает ответ с пользовательской ошибкой, например такой, как в рецепте 51, этот ответ будет иметь структуру, отличную от той, что ожидалась.

Проблема

Как определить, что в ответе возвращена пользовательская ошибка с определенной структурой, и как обработать ее надлежащим образом?

Решение

Получив ответ, проверим код HTTP-состояния и MIME-тип, чтобы определить факт ошибки. Если хотя бы один будет содержать неожиданное значение или сообщение об ошибке, преобразуем ответ в ошибку, вернем и обработаем ошибку.

Обсуждение

Язык Go реализует принцип явной обработки ошибок и использует стандартные коды HTTP. Когда в ответ на HTTP-запрос возвращается неожиданный код состояния, его можно обработать как прочие ошибки. Первый шаг – вернуть ошибку, если обработка HTTP-запроса прошла не так, как ожидалось, как показано в следующем листинге.

Листинг 8.8 ❖ Преобразование HTTP-ответа в ошибку

```
type Error struct {
     HTTPCode int    `json:"-"`
     Code     int    `json:"code,omitempty"`
     Message  string `json:"message"`
}
```

Структура для хранения данных об ошибке

```
func (e Error) Error() string {
     fs := "HTTP: %d, Code: %d, Message: %s"
     return fmt.Sprintf(fs, e.HTTPCode, e.Code, e.Message)
}
```

Метод Error реализует интерфейс error и опирается на структуру Error

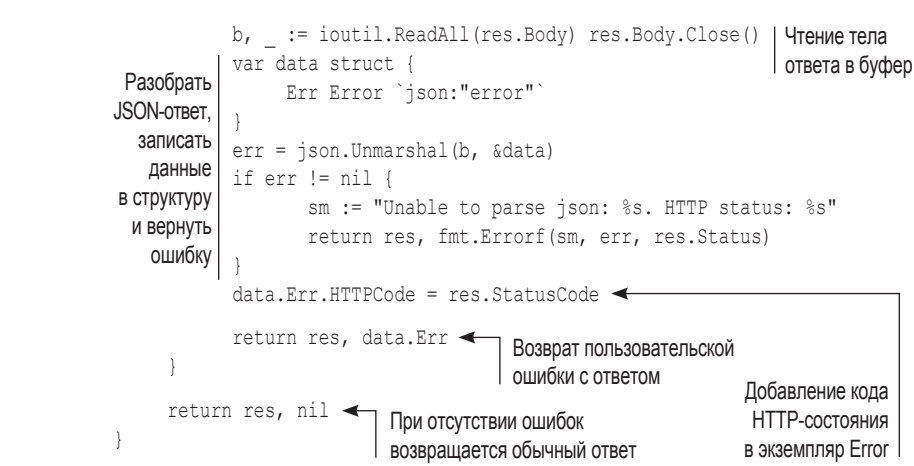
```
func get(u string) (*http.Response, error) {
     res, err := http.Get(u)
     if err != nil {
            return res, err
     }
     if res.StatusCode < 200 || res.StatusCode >= 300 {
          if res.Header.Get("Content-Type") != "application/json" {
               sm := "Unknown error. HTTP status: %s"
               return res, fmt.Errorf(sm, res.Status)
          }
```

Использование http.Get для получения ресурса и возврата любых ошибок http.Get errors.

Для выполнения запросов вместо функции http.Get должна использоваться функция get

Проверка, находится ли код ответа вне диапазона от 200 до 299, свидетельствующего об успехе

Проверка типа содержимого ответа и возврат ошибки, если он неправильный



Здесь определяется функция `get`, которая обрабатывает пользовательские ошибки и должна использоваться для выполнения запросов взамен `http.Get`. Структура `Error`, содержащая данные об ошибке, аналогична структуре из рецепта 51. Эта реализация обработки пользовательских ошибок предполагает работу с сервером, возвращающим ошибки, как показано в рецепте 51. Эти два рецепта могли бы использовать общий пакет с определениями ошибок.

Добавление метода `Error()` в тип `Error` обеспечивает реализацию интерфейса `error`. Это позволяет передавать экземпляры `Error` между функциями как экземпляры `error`, подобно любым другим ошибкам.

Функция `main` в следующем фрагменте демонстрирует использование функции `get` вместо функции `http.Get`. Получив ошибку, она выводит данные, полученные в формате JSON, и завершает приложение:

```
func main() {
     res, err := get("http://localhost:8080")
     if err != nil {
            fmt.Println(err)
            os.Exit(1)
     }
     b, _ := ioutil.ReadAll(res.Body)
     res.Body.Close()
     fmt.Printf("%s", b)
}
```

Такой способ получения и передачи HTTP-ошибок позволяет использовать все преимущества механизма обработки ошибок в языке Go. Например, с помощью оператора `switch` можно определить тип

ошибки и соответствующим образом среагировать на нее, как было показано в листинге 8.4.

8.3. Разбор и отображение данных в формате JSON

При взаимодействии через REST API для передачи информации чаще других используется формат JSON. Поэтому возможность простого и быстрого преобразования JSON-строк в структуры данных на языке Go является весьма полезной, и стандартная библиотека языка Go дает такую возможность в виде пакета `encoding/json`. Например, код в следующем листинге преобразует простую структуру данных в формате JSON в тип `struct`.

Листинг 8.9 ❖ Пример преобразования данных в формате JSON

```
package main

import (
     "encoding/json"
     "fmt"
)

type Person struct {
     Name string `json:"name"`
}

var JSON = `{
  "name": "Miracle Max"
}`

func main() {
     var p Person
     err := json.Unmarshal([]byte(JSON), &p)
     if err != nil {
            fmt.Println(err)
            return
     }
     fmt.Println(p)
}
```

Структура, представляющая информацию в формате JSON. Тег json отображает свойство Name в поле name в данных в формате JSON

Строка с данными в формате JSON

Экземпляр структуры Person для сохранения данных в формате JSON

Преобразование данных в формате JSON в экземпляр структуры Person

Обработка любых ошибок преобразования

Использование заполненного объекта Person, в данном случае его вывод

Хотя стандартная библиотека предоставляет все необходимое для разбора данных в формате JSON, однако вы можете столкнуться с достаточно распространенными ситуациями, не имеющими очевидного решения.

РЕЦЕПТ 53 Разбор данных в формате JSON с неизвестной структурой

Структуру данных в формате JSON обычно можно определить из описания в документации, из примеров или наблюдая пересылаемые данные на практике. Несмотря на существование схем для формата JSON, например JSON Schema, они редко применяются на практике. Но отсутствие схем данных в формате JSON не единственная проблема – разные ответы прикладного программного интерфейса могут иметь разную структуру данных, в результате иногда приходится иметь дело с данными в формате JSON, структура которых неизвестна.

В процессе разбора данных в формате JSON они преобразуются в структуру, объявленную в коде. Если структура данных в формате JSON неизвестна на момент объявления структуры в коде или может изменяться, это становится проблемой. На первый взгляд кажется, что определение структуры данных в формате JSON или обработка таких данных с изменяющейся структурой является сложной задачей. Но это не так.

Проблема

Как реализовать преобразование данные в формате JSON в структуру на языке Go, когда их структура заранее неизвестна?

Решение

Преобразуем данные в формате JSON в тип `interface{}`, а не в структуру. Поместив данные в формате JSON в интерфейс, вы сможете изучать их и использовать.

Обсуждение

Пакет `encoding/json` обладает малоизвестной возможностью преобразовывать данные в формате JSON с произвольной структурой в тип `interface{}`. Работа с данными, помещенными в тип `interface{}`, отличается от работы с данными в заранее известной структуре из-за системы типов языка Go. Следующий листинг содержит пример подобного преобразования.

Листинг 8.10 ❖ Преобразование данных в формате JSON в тип interface{}

```
package main

import (
    "encoding/json"
```

```
    "fmt"
    "os"
)

var ks = []byte(`{
"firstName": "Jean",
"lastName": "Bartik",
"age": 86,
"education": [
    {
        "institution": "Northwest Missouri State Teachers College",
        "degree": "Bachelor of Science in Mathematics"
    },
    {
        "institution": "University of Pennsylvania",
        "degree": "Masters in English"
    }
    ],
        "spouse": "William Bartik",
"children": [
    "Timothy John Bartik",
    "Jane Helen Bartik",
    "Mary Ruth Bartik"
]
}`)

func main() {
    var f interface{}
    err := json.Unmarshal(ks, &f)
    if err != nil {
        fmt.Println(err)
        os.Exit(1)
    }
    fmt.Println(f)
}
```

Документ в формате JSON для преобразования

Переменная с экземпляром типа interface{} для преобразования данных из формата JSON

Разбор данных в формате JSON и помещение их в переменную с типом interface{}

Обработка любых ошибок, например ошибки в JSON-данных

Д

Преобразуемые данные в формате JSON могут иметь произвольную структуру. Эта важная особенность делает работу с типом `interface{}` непохожей на обработку данных, помещенных в обычную структуру. Ниже описывается порядок работы с такими данными.

После преобразования данных из формата JSON в структуру, например как это было сделано в листинге 8.9, получить к ним доступ очень легко. В том конкретном случае имя лица после преобразования данных из формата JSON можно было получить как `p.Name`. Если

попытаться таким же способом обратиться к свойству firstName в экземпляре interface{}, это вызовет ошибку. Например:

```
fmt.Println(f.firstName)
```

Доступ к firstName как свойству вызовет ошибку:

```
f.firstName undefined (type interface {} is interface with no methods)
```

Перед работой с данными необходимо обеспечить доступ к ним через какой-либо тип, отличный от interface{}. В подобном случае JSON-данные представляют объект, поэтому можно воспользоваться типом map[string] interface{}. Он обеспечивает доступ к следующему уровню данных. Следующий фрагмент демонстрирует доступ к firstName:

```
m := f.(map[string]interface{})
fmt.Println(m["firstName"])
```

В этой точке становятся доступными все ключи верхнего уровня, что позволяет извлечь значение firstName по имени.

Чтобы программно обойти данные, полученные из формата JSON, важно знать, как язык Go выполняет преобразование. Данные в формате JSON преобразуются в следующие типы языка Go:

- логические значения JSON – в bool ;
- числа JSON – в float64;
- массивы JSON – в []interface{};
- объекты JSON – в map[string]interface{};
- значения null JSON – в nil;
- строки JSON – в string.

Зная это, можно реализовать обход структуры данных. Например, в следующем листинге представлена рекурсивная функция для перебора преобразованных JSON-данных, которая выводит имена ключей, типы и значения.

Листинг 8.11 ❖ Обход JSON-данных произвольной структуры

```
func printJSON(v interface{}) {
    switch vv := v.(type) {
    case string:
        fmt.Println("is string", vv)
    case float64:
        fmt.Println("is float64", vv)
    case []interface{}:
        fmt.Println("is an array:")
        for i, u := range vv {
```

← Обработка в зависимости от типа значения

Для каждого значения, полученного из формата JSON, выводится информация о типе и само значение. Для объектов и массивов JSON рекурсивно вызывается функция printJSON, которая выводит их внутреннее содержимое

```
                fmt.Print(i, " ")
                printJSON(u)
        }
    case map[string]interface{}:
        fmt.Println("is an object:")
        for i, u := range vv {
                fmt.Print(i, " ")
                printJSON(u)
        }
    default:
        fmt.Println("Unknown type")
    }
}
```

Такой способ преобразования и обработки может пригодиться не только при работе с данными в формате JSON с неизвестной структурой, но и при использовании заранее известных структур, которые могут изменяться в других версиях. В следующем разделе вы узнаете, как учитывать версии прикладных программных интерфейсов, включая изменения в структурах данных в формате JSON.

8.4. Версионирование REST API

Веб-службы развиваются и изменяются, что приводит к изменениям в прикладных программных интерфейсах, используемых для доступа или управления ими. Для стабильности взаимодействий потребителей с прикладными программными интерфейсами применяется прием версионирования. Для перехода на новые версии прикладных программных интерфейсов в программы требуется вносить изменения, а это занимает определенное время.

Версионирование прикладных программных интерфейсов обычно заключается в изменении старших номеров версий, таких как v1, v2 и v3. Такая смена номеров означает внесение критических изменений в прикладной программный интерфейс. Приложение, работающее с версией v2 прикладного программного интерфейса, не сможет использовать версию v3, поскольку они имеют коренные отличия.

А как насчет изменений, добавляющих функциональные возможности к существующим? Например, предположим, что в версию v1 прикладного программного интерфейса добавлены новые возможности. В этом случае увеличивается младший номер версии – число после точки. Например, после расширения возможностей версия может быть увеличена до 1.1. Это указывает разработчикам и приложениям о наличии дополнений.

Следующие два рецепта демонстрируют несколько способов учета версий прикладного программного интерфейса.

РЕЦЕПТ 54 Версии прикладных программных интерфейсов в URL-адресе

Изменение версии прикладного программного интерфейса должно легко обнаруживаться и обрабатываться. Разработчики должны иметь простой способ определения, с чем они имеют дело, чтобы в полной мере воспользоваться службами.

Не менее важную роль играет версионирование прикладных программных интерфейсов для работы с имеющимися инструментами. Например, возможность быстрого тестирования вызовов прикладных программных интерфейсов с помощью cURL, Postman или популярных расширений для Chrome облегчает разработку и тестирование прикладных программных интерфейсов.

Проблема

Как проще всего организовать версионирование прикладного программного интерфейса?

Решение

Передача номера версии прикладного программного интерфейса в URL-адресе. Например, вместо прямого вызова API, такого как https://example.com/api/todos, можно добавить в путь номер версии: https://example.com/api/v1/todos.

Обсуждение

Рисунок 8.1 иллюстрирует популярный метод версионирования прикладных программных интерфейсов через URL-адрес. Такой подход применяется, в частности, в Google, OpenStack, Salesforce, Twitter и Facebook.

Рис. 8.1 ❖ *Версионирование REST API в URL-адресе*

Как показано в следующем листинге, поддержка адресов URL с такой структурой реализуется посредством отображения путей в обработчики.

Листинг 8.12 ❖ Регистрация пути к API с включением версии

```
package main

import (
     "encoding/json"
     "fmt"
     "net/http"
)

type testMessage struct {
     Message string `json:"message"`
}

func displayTest(w http.ResponseWriter, r *http.Request) {
     data := testMessage{"A test message."}
     b, err := json.Marshal(data)
     if err != nil {
            http.Error(w, "Internal Server Error", 500)
            return
     }
     w.Header().Set("Content-Type", "application/json")
     fmt.Fprint(w, string(b))
}

func main() {
     http.HandleFunc("/api/v1/test", displayTest)
     http.ListenAndServe(":8080", nil)
}
```

Пример функции, возвращающей ответ в формате JSON

Включение версии API в путь, который отображается в функцию-обработчик

В этом примере путь напрямую отображается в функцию обработки, что не позволяет легко обрабатывать различные методы запроса, такие как `POST`, `PUT` и `DELETE`. Если конечная точка представляет ресурс, для всех видов запросов обычно используется один URL-адрес. Приемы обработки нескольких HTTP-методов с помощью одного и того же URL-адреса можно найти в главе 2.

Несмотря на простоту, этот метод передачи версии API имеет семантические недостатки. URL-адрес не представляет объект – он представляет путь к объекту с определенной версией прикладного программного интерфейса. Компромисс заключается в упрощении. Включение версии в URL-адрес – это самый простой для разработчиков способ воспользоваться прикладным программным интерфейсом.

РЕЦЕПТ 55 Версии прикладных программных интерфейсов в типе содержимого

Предыдущий рецепт описывает метод, облегчающий жизнь разработчикам, но он не лишен семантических недостатков. Согласно оригинальной теории REST, URL-адрес должен предоставлять что-то. Это может быть объект, список или что-то еще. В зависимости от деталей запроса, например типа запрашиваемого содержимого или HTTP-метода, этот объект будет отвечать или действовать по-разному.

Проблема

Как версионировать прикладной программный интерфейс более семантическим способом?

Решение

В запросе и ответе вместо ссылки на JSON используем нестандартный тип содержимого, включающий номер версии. Например, вместо `application/json` используем, например, тип `application/vnd.mytodo.v1.json` или `application/vnd.mytodo.json;version=1.0`. Такие нестандартные типы определяют схему данных.

Обсуждение

Для доступа к разным версиям API с использованием одного пути, помимо всего прочего, необходимо учитывать тип содержимого, как показано на рис. 8.2. Листинг 8.13 демонстрирует один из методов определения типа содержимого и использования его при конструировании ответа.

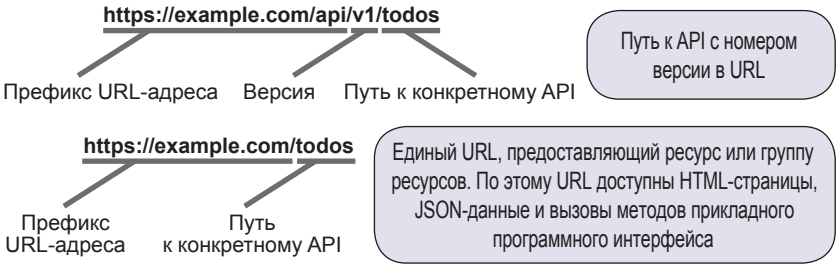


Рис. 8.2 ❖ *Различия между семантическими URL-адресами и URL-адресами с номерами версий API*

Листинг 8.13 ❖ Передача версии API в типе содержимого

```
func main() {
     http.HandleFunc("/test", displayTest)  ← Регистрация пути с несколькими
     http.ListenAndServe(":8080", nil)          типами содержимого
}

func displayTest(w http.ResponseWriter, r *http.Request) {
     t := r.Header.Get("Accept")  ← Определение зарегистрированного ранее
     var err error                    типа содержимого
     var b []byte
     var ct string
     switch t {
     case "application/vnd.mytodos.json; version=2.0":
            data := testMessageV2{"Version 2"}
            b, err = json.Marshal(data)
            ct = "application/vnd.mytodos.json; version=2.0"
     case "application/vnd.mytodos.json; version=1.0":
            fallthrough
     default:
            data := testMessageV1{"Version 1"}
            b, err = json.Marshal(data)
            ct = "application/vnd.mytodos.json; version=1.0"
     }
     if err != nil {
            http.Error(w, "Internal Server Error", 500)
            return
      }
      w.Header().Set("Content-Type", ct)
      fmt.Fprint(w, string(b))  ← Вернуть ответ клиенту
}

type testMessageV1 struct {
     Message string `json:"message"`
}

type testMessageV2 struct {
     Info string `json:"info"`
}
```

Конструирование разных ответов, в зависимости от запрошенного типа содержимого

При появлении ошибки во время преобразования данных в формат JSON вернуть ее

Установить тип содержимого в соответствии с запрошенным типом

Когда клиент выполняет запрос, он может не указывать тип содержимого, чтобы получить ответ по умолчанию. Но если ему потребуется использовать прикладной программный интерфейс версии 2, он должен отказаться от простого запроса `GET` и передать дополнительные сведения. Например, следующий фрагмент выполняет запрос к API версии 2 и выводит ответ:

Тип содержимого с номером версии прикладного программного интерфейса

Создание нового GET-запроса для сервера из листинга 8.13

```
ct := "application/vnd.mytodos.json; version=2.0"
req, _ := http.NewRequest("GET", "http://localhost:8080/test", nil)
req.Header.Set("Accept", ct)  ← Добавление нужного типа содержимого в запрос
res, _ := http.DefaultClient.Do(req)  ← Выполнение запроса

if res.Header.Get("Content-Type") != ct {
     fmt.Println("Unexpected content type returned")
     return
}
```

Проверка типа содержимого в ответе

```
b, _ := ioutil.ReadAll(res.Body)
res.Body.Close()
fmt.Printf("%s", b)
```

Вывод тела ответа

Этот метод позволяет работать с несколькими версиями прикладного программного интерфейса, но, используя его, следует учесть следующие нюансы:

- типы содержимого в `vnd.namespace` должны быть зарегистрированы в Internet Assigned Numbers Authority (IANA);
- при выполнении запроса к версии API не по умолчанию необходимо дополнительно указать тип содержимого для требуемой версии. Для этого приложению понадобится приложить дополнительные усилия.

8.5. Итоги

Эта глава началась со знакомства с основами использования веб-служб, такими как выполнение REST-запросов. Затем мы быстро перешли к обсуждению приемов, обеспечивающих надежные взаимодействия с веб-службами, в том числе:

- определение превышения времени ожидания в сети, даже если на уровне сети он формально не указан, и возобновление загрузки по истечении времени ожидания;
- передача ошибок между конечными точками API и клиентами с помощью и без помощи HTTP-заголовков;
- разбор JSON-данных неизвестной структуры;
- использование двух методов версионирования REST API и работа с прикладными программными интерфейсами разных версий.

Следующая глава посвящена работе с облачными службами. Эффективное выполнение приложения в облаке обеспечивает не только работа с прикладным программным интерфейсом, предоставляющим возможность их настройки. Глава 9 знакомит с методами, помогающими повысить эффективность работы Go-приложений в облаке.

Часть IV

РАЗМЕЩЕНИЕ ПРИЛОЖЕНИЙ В ОБЛАКЕ

Облачные вычисления меняют подход к разработке и использованию приложений. Язык Go идеально приспособлен для создания вычислительных систем и приложений, работающих в облаке. Часть 4 будет посвящена особенностям облачных вычислений.

Глава 9 открывает четвертую часть и знакомит с облачными вычислениями и соображениями, касающимися создания приложений, работающих в облаке. Шаблоны, представленные в этой главе, помогают упростить запуск и мониторинг приложений в облаке. Глава 10 продолжает облачную тему, демонстрируя совместную работу служб и приемы реализации высокопроизводительных взаимодействий. Они особенно эффективны при использовании архитектуры микрослужб.

Глава 11, завершающая четвертую часть и книгу, посвящена механизму рефлексии и метапрограммированию. Механизм рефлексии упрощает работу со статическими типами и широко используется в языке Go, поскольку не привносит особых затрат. Метапрограммирование помогает автоматически генерировать код и открывает новые пути в разработке некоторых приложений и библиотек.

Глава 9

Использование облака

В этой главе рассматриваются следующие темы:

- *введение в облачные вычисления;*
- *работа с несколькими провайдерами облачных услуг;*
- *сбор информации об облачном хосте;*
- *компиляция для различных операционных систем и архитектур;*
- *мониторинг среды выполнения Go в приложении.*

Облачные вычисления стали модным словосочетанием. Но что это, просто дань моде или нечто большее? Эта глава начинается с введения в облачные вычисления, она отвечает на данный вопрос и разъясняет практическую ценность облачных вычислений. Здесь вы увидите, как облачные вычисления соотносятся с традиционными моделями, основанными на использовании аппаратных серверов и виртуальных машин.

Облачные вычисления поддерживаются множеством различных провайдеров. При создании облачных приложений легко попасть в зависимость от одного конкретного провайдера. Чтобы этого не произошло, здесь будет рассмотрен подход, позволяющий избежать такой зависимости, а также упрощающий разработку и тестирование в локальной системе.

Когда все готово к запуску приложения в облаке, могут возникать проблемы, требующие решения, например получение сведений о хосте, где будет выполняться приложение, необходимость мониторинга среды выполнения Go из приложения и кросс-платформенная компиляция перед развертыванием. Здесь вы узнаете, как решаются эти проблемы и как избежать ловушек, способных застать врасплох.

Эта глава посвящена знакомству с ключевыми понятиями облачных вычислений. Прочитав ее и все предыдущие главы, вы будете иметь полное представление о том, что необходимо для создания и эксплуатации облачных приложений, написанных на языке Go.

9.1. Что такое облачные вычисления?

Это один из фундаментальных вопросов, требующих перечисления всех особенностей, связанных с разработкой и эксплуатацией приложений. Является ли слово «облако» маркетинговым термином? Да, это так. Подразумевается ли под ним нечто принципиально новое? Да, и это тоже верно. Учитывая, что этот термин применяется для обозначения всего и вся, от мобильных и веб-приложений до физических серверов и виртуальных машин, в его значении сложно разобраться тем, кому незнакомо понятие пространства. Во вступительном разделе вы узнаете об облачных вычислениях все, что потребуется для разработки и эксплуатации программного обеспечения.

9.1.1. Виды облачных вычислений

В простейшей форме облачные вычисления – это часть системы, которой управляет кто угодно, но не сам разработчик. Это может быть кто-то из вашей компании, внешний поставщик услуг, автоматизированная система или любая их комбинация. Если внешний поставщик услуг предоставляет часть функций, то какие именно функции предоставляются? На рис. 9.1 показаны три формы организации облачных вычислений то, и как они соотносятся со средой, которой полностью распоряжается разработчик.

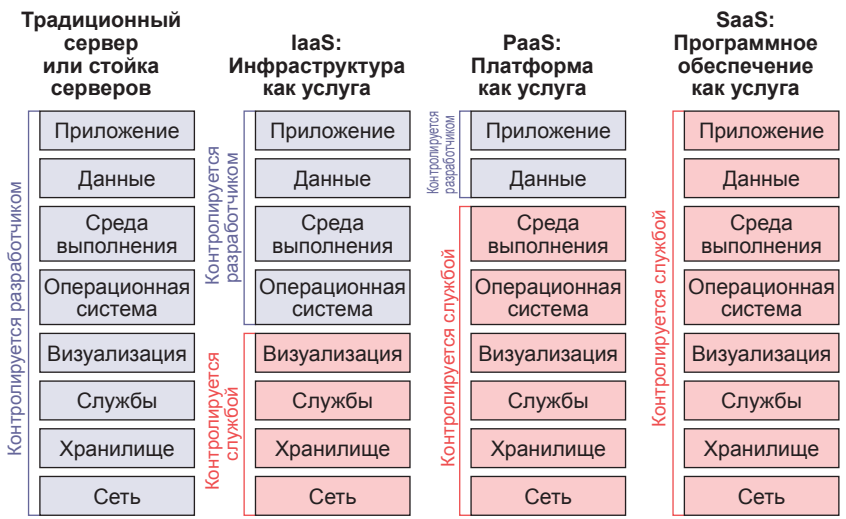


Рис. 9.1 ❖ *Виды облачных вычислений*

При использовании традиционного сервера или стойки серверов вы должны управлять всеми компонентами, вплоть до физического пространства для размещения аппаратуры. Когда что-то нужно изменить, приходится заказывать оборудование, ждать его получения, обеспечивать подключение и обслуживание. Это занимает определенное время.

Инфраструктура как услуга

Доступ к *инфраструктуре как к услуге* (Infrastructure as a Service, IaaS) отличается от работы с виртуальными машинами. Услуги предоставления виртуальных машин и совместного использования серверов предлагаются уже несколько лет. С появлением IaaS изменился лишь подход к использованию и обеспечению доступа к этим серверам.

До появления IaaS клиенту предоставлялся сервер, который он использовал в течение нескольких месяцев или даже лет. IaaS поставила все с ног на голову. В IaaS виртуальные серверы создаются и уничтожаются по мере необходимости. Когда клиенту требуется сервер, он создается. Как только он станет не нужен, его вернут в пул доступных ресурсов. Например, чтобы проверить идею, вы можете создать несколько серверов, оценить идею, а затем немедленно избавиться от этих серверов.

Создание ресурсов IaaS и работа с ними осуществляются через специализированный программный интерфейс. Обычно это REST API, позволяющий использовать инструменты командной строки и приложения для управления ресурсами.

Инфраструктура как служба – это больше, чем просто набор серверов. Хранилища, сети и другие компоненты инфраструктуры можно настраивать таким же образом. Как следует из названия, настройке поддаются все компоненты инфраструктуры. Операционная система, среда выполнения, приложение и данные контролируются пользователями облака и их программным обеспечением.

Примерами IaaS являются службы, предоставляемые Amazon Web Services, Microsoft Azure и Google Cloud.

Платформа как услуга

Платформа как услуга (Platform as a service, PaaS) имеет несколько существенных отличий от IaaS. Одно из них – порядок использования. Чтобы развернуть приложение на платформе, пользователь использует прикладной программный интерфейс для развертывания

кода приложения и поддерживающего окружения, такого как язык, на котором написано приложение. Платформа обрабатывает эту информацию, а затем создает и запускает приложение.

В этой модели платформа управляет операционной системой и средой выполнения. Пользователю не нужно запускать виртуальные машины, указывать параметры их ресурсов, выбирать операционную систему и устанавливать системное программное обеспечение. Решение этих задач передается платформе. Это экономит время пользователя и позволяет ему заняться другими задачами, такими как работа над его собственным приложением.

Масштабирование приложений в этом случае достигается созданием и запуском дополнительных экземпляров приложения, выполняющихся параллельно. Такое масштабирование называют горизонтальным. PaaS способна автоматически выполнять масштабирование, но вы можете сами определить количество экземпляров через прикладной программный интерфейс.

Тремя самыми популярными примерами PaaS являются Heroku, Cloud Foundry и Deis.

Программное обеспечение как услуга

Предположим, что приложению требуется база данных. Можно создать кластер виртуальных машин с помощью IaaS, установить программное обеспечение базы данных, настроить его, провести мониторинг базы данных, чтобы убедиться, что все работает правильно, и постоянно обновлять системное программное обеспечение для поддержания безопасности. Или же можно использовать базу данных как услугу и переложить хлопоты, связанные с эксплуатацией, масштабированием и обновлением, на поставщика услуг. С помощью прикладного программного интерфейса вы можете настроить базу данных и обращаться к ней. Схема на рис. 9.2 иллюстрирует применение программного обеспечения как услуги (SaaS).

SaaS предлагает широкий спектр программного обеспечения, от стандартных компонентов до целых приложений, необходимых приложениям пользователя. Использование SaaS для получения доступа к дополнительным компонентам, таким как базы данных и хранилища, позволяет командам сосредоточиться на проблемах собственного приложения.

Примеры SaaS весьма разнообразны, к ним относятся Salesforce, Microsoft Office 365 и платежный процессор Stripe.

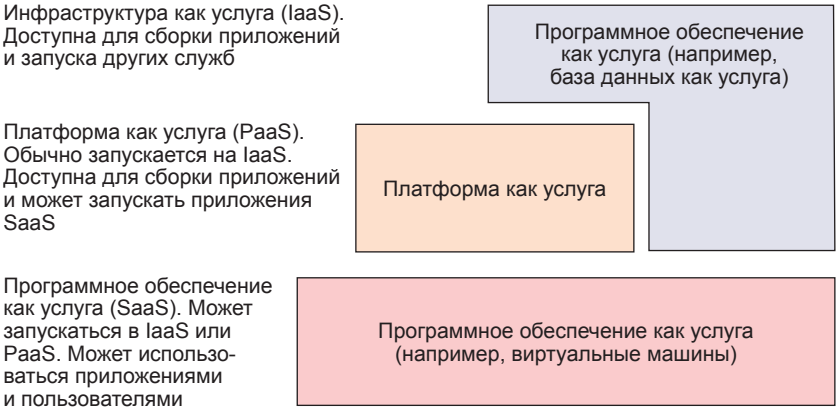


Рис. 9.2 ❖ *Разные слои облачных служб могут накладываться друг на друга*

9.1.2. Контейнеры и натуральные облачные приложения

С ростом популярности Docker – программного обеспечения для управлениями контейнерами – стали широко применяться технологии выполнения и распространения приложений с использованием контейнеров. Контейнеры отличаются от виртуальных машин и от традиционных серверов.

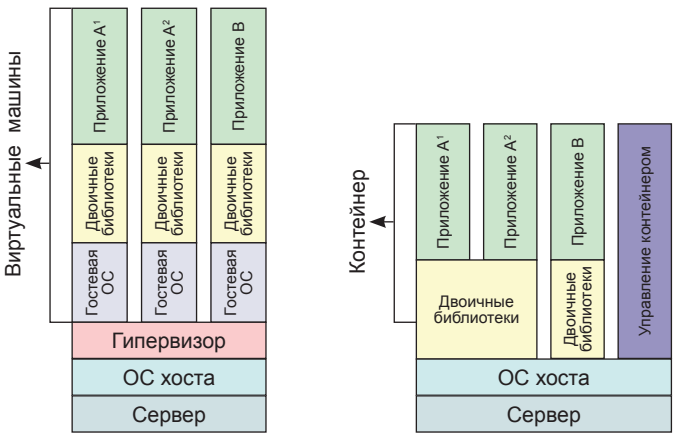


Рис. 9.3 ❖ *Сравнение контейнеров и виртуальных машин*

Схема на рис. 9.3 иллюстрирует сравнение виртуальных машин и контейнеров. Слева показано устройство системы вплоть до аппаратного сервера, в котором выполняются виртуальные машины. Справа – система, в которой выполняются отдельные контейнеры. В каждой из систем выполняются по два приложения, одно из которых (приложение А) масштабируется по горизонтали, за счет запуска второго экземпляра. В общей сложности получаются три действующих экземпляра.

После запуска виртуальных машин гипервизор предоставляет среду выполнения для операционной системы, имитирующую работу аппаратного обеспечения или использующую специализированное аппаратное обеспечение для виртуальных машин. Каждая виртуальная машина содержит гостевую операционную систему с ее собственным ядром. Внутри операционной системы размещаются приложения, исполняемые файлы и библиотеки операционной системы, а также настройки пользователей. Каждое приложение выполняется в собственном окружении. Когда требуется запустить второй экземпляр приложения, выполняющийся параллельно (именно так работает горизонтальное масштабирование), создается вторая копия гостевой операционной системы со всеми ее приложениями и библиотеками, а также самим приложением.

Использование виртуальных машин привносит следующие архитектурные особенности:

- при запуске виртуальной машины ее ядру и гостевой операционной системе требуется определенное время для загрузки, потому что компьютеры не могут загружаться мгновенно;
- гипервизоры и современное оборудование способны обеспечить разделение виртуальных машин;
- виртуальные машины предоставляют инкапсулированный сервер с назначенными ему ресурсами. Эти ресурсы могут использоваться только на этой машине или только для этой машины.

Контейнеры используют другую модель. На хост-сервере, будь то физический сервер или виртуальная машина, выполняется диспетчер контейнеров. Все контейнеры запускаются в операционной системе хоста. При использовании ядра хоста запуск происходит практически мгновенно. Контейнеры совместно используют ядро, а драйверы оборудования и операционная система поделены между контейнерами. Используемые исполняемые файлы и библиотеки, обычно связанные с операционной системой, могут быть разными. Например, на рис. 9.3 приложение А может использовать исполняемые файлы и библио-

теки Debian, а приложение B – исполняемые файлы и библиотеки CentOS. Приложения, работающие в контейнерах, будут определять свою среду выполнения как Debian или CentOS.

Понятие *натуральных облачных приложений* обычно применяется вместе с понятием контейнеров. Натуральные облачные приложения способны использовать на программном уровне преимущества облака для масштабирования созданием дополнительных экземпляров по требованию, устранения последствий множества сбоев в системе, не позволяя им доходить до конечных пользователей, увязывания вместе микрослужб для создания более крупных приложений и многого другого. Контейнеры, с их способностью запускаться практически мгновенно и более плотным размещением на серверах, чем виртуальные машины, создают идеальные условия для масштабирования, восстановления и применения микрослужб.

> **ОПРЕДЕЛЕНИЕ** *Восстановление* – это автоматическая коррекция проблем выполнения приложений. *Микрослужбы* – небольшие, независимые процессы, взаимодействующие с другими небольшими процессами посредством предопределенного прикладного программного интерфейса. Микрослужбы используются совместно для создания более крупных приложений. Микрослужбы более подробно рассматриваются в главе 10.

Это лишь введение в контейнеры, натуральные облачные вычисления и облачные вычисления в целом. Более подробную информацию по этой теме можно найти в многочисленных специализированных книгах, обучающих курсах и прочих материалах.

Одним из важных аспектов облачных служб, который следует рассмотреть более подробно, является подход к управлению службами. Интерфейс управления облачными службами позволяет Go-приложениям взаимодействовать с ними.

9.2. Управление облачными службами

Управление облачными службами, будь то IaaS, PaaS или SaaS, обычно осуществляется через прикладной программный интерфейс. Он включает инструменты командной строки, пользовательские интерфейсы, автономные приложения (боты) и другие средства управления службами. Облачные службы поддерживают программное управление.

Веб-приложения, такие как веб-консоль, предоставляемая всеми поставщиками облачных услуг, на первый взгляд кажутся простыми и эффективными средствами управления облачными службами.

В некоторых простых случаях их возможностей действительно достаточно. Однако истинная мощь облака заключается в возможности управления им программным способом. В число таких возможностей входят, например, автоматическое горизонтальное масштабирование, автоматическое восстановление после сбоев и одновременная работа с большим количеством облачных ресурсов.

Все поставщики облачных услуг предоставляют REST API, а большая их часть еще и пакет SDK, обеспечивающий возможность взаимодействия с прикладным программным интерфейсом. Пакет SDK или прикладной программный интерфейс можно использовать в приложениях для управления облачными службами.

9.2.1. Независимость от конкретного провайдера облачных услуг

Все облачные службы, такие как платформы, имеют собственный прикладной программный интерфейс. Они обеспечивают одинаковые или похожие возможности, но часто имеют совершенно разные интерфейсы REST API. Пакет SDK одного поставщика услуг не будет работать со службами его конкурента. Общей спецификации прикладного программного интерфейса не существует, и все поставщики реализуют его по-своему.

РЕЦЕПТ 56 Работа с несколькими облачными провайдерами

Существует множество поставщиков облачных услуг. Некоторые обслуживают только конкретные регионы, в соответствии с законами о суверенитете данных, другие являются глобальными компаниями. Некоторые поставщики доступны всем и используют базовую инфраструктуру совместно с другими поставщиками, другие закрыты и используют только собственное оборудование. Облачные провайдеры предлагают разные цены, возможности и обеспечивают разные потребности.

Учитывая динамичность рынка облачных услуг, имеет смысл обеспечить максимальную гибкость при работе с провайдерами. Код, написанный специально под конкретного облачного провайдера, заставит работать только с ним, даже если у пользователя появится желание переключиться на другого поставщика. Переключение потребует больших усилий и задержки в предоставлении новых функциональных возможностей. Хитрость заключается в том, чтобы избежать такой зависимости.

Проблема

Поставщики облачных услуг, как правило, предоставляют свои собственные программные интерфейсы, даже если они обеспечивают одинаковые или аналогичные функции, что и у других поставщиков. Написание приложения, ориентированного на работу с конкретным программным интерфейсом, приведет к зависимости от этого интерфейса и поставщика.

Решение

Разделим решение на две части. Сначала создадим интерфейс для описания облачных взаимодействий. Если вам необходимы такие операции, как сохранение файла или запуск экземпляра сервера, добавьте их описание в интерфейс. Когда потребуется использовать эти облачные функции, вы сможете обратиться за ними к интерфейсу.

Затем создадим отдельную реализацию интерфейса для каждого поставщика облачных услуг, с которым планируется работать. Благодаря этому переключение между поставщиками сведется к написанию реализации интерфейса.

Обсуждение

Это – старая модель, доказавшая свою эффективность. Представьте ситуацию, когда все компьютеры способны работать с принтерами только одного производителя. Чтобы этого избежать, операционные системы предоставляют общие интерфейсы и драйверы для конкретных принтеров. Та же идея применима к работе с облачными провайдерами.

Первым делом нужно определить и задействовать интерфейс для определенной функциональной возможности, а реализацию для конкретного поставщика мы напишем потом. Следующий пример демонстрирует интерфейс для работы с файлами.

Листинг 9.1 ❖ Интерфейс для доступа к облачным функциям

```
type File interface {   ← Интерфейс для работы с файлами
     Load(string) (io.ReadCloser, error)  | Обобщенные методы для работы
     Save(string, io.ReadSeeker) error    | с файлами. Имена не содержат
}                                         | подробностей базовой реализации
```

Рассмотрим работу с файлами подробнее, поскольку облачные поставщики предлагают несколько видов хранилищ файлов, предоставляют разные программные интерфейсы и операции с файлами часто используются на практике.

После определения интерфейса необходимо написать его первую реализацию. Для простоты, и чтобы дать возможность разработки и тестирования приложения в локальной системе, используем в качестве хранилища локальную файловую систему. Это позволит убедиться в работоспособности приложения до внедрения в него сетевых операций и взаимодействий с облачными поставщиками. Следующий листинг демонстрирует реализацию интерфейса `File`, загружающего локальные файлы и сохраняющего их в локальной файловой системе.

Листинг 9.2 ❖ Простая реализация облачного хранилища файлов

```
type LocalFile struct {
     Base string
}

func (l LocalFile) Load(path string) (io.ReadCloser, error) {
     p := filepath.Join(l.Base, path)
     return os.Open(p)
}

func (l LocalFile) Save(path string, body io.ReadSeeker) error {
     p := filepath.Join(l.Base, path)
     d := filepath.Dir(p)
     err := os.MkdirAll(d, os.ModeDir|os.ModePerm)
     if err != nil {
            return err
     }
     f, err := os.Create(p)
     if err != nil {
            return err
     }
     defer f.Close()
     _, err = io.Copy(f, body)
     return err
}
```

Открыть локальный файл или вернуть ошибку. Экземпляры os.File реализуют интерфейс io.ReadCloser, что позволяет вернуть результат os.Open.

Структура для основы реализации. В свойстве Base хранится базовый путь к хранилищу файлов.

Чтобы сохранить файл, необходимо гарантировать существование указанного каталога и скопировать содержимое в файл.

После создания базовой реализации можно написать использующий ее прикладной код. В следующем листинге приводится простой пример сохранения и загрузки файла.

Листинг 9.3 ❖ Пример использования интерфейса для облачного провайдера

```
func main() {
     content := `Lorem ipsum dolor sit amet, consectetur` +
            `adipiscing elit. Donec a diam lectus.Sed sit` +
            `amet ipsum mauris. Maecenascongue ligula ac` +
            `quam viverra nec consectetur ante hendrerit.`
     body := bytes.NewReader([]byte(content))
     store, err := fileStore()
     if err != nil {
            fmt.Println(err)
            os.Exit(1)
     }
     fmt.Println("Storing content...")
     err = store.Save("foo/bar", body)
     if err != nil {
            fmt.Println(err)
            os.Exit(1)
     }
     fmt.Println("Retrieving content...")
     c, err := store.Load("foo/bar")
     if err != nil {
            fmt.Println(err)
            os.Exit(1)
     }
     o, err := ioutil.ReadAll(c)
     if err != nil {
            fmt.Println(err)
            os.Exit(1)
     }
     fmt.Println(string(o))
}
```

Пример содержимого, помещаемого в экземпляр, реализующий интерфейс io.ReadSeeker

Получение реализации интерфейса File из листинга 9.1 и обработка ошибок

Сохранение примера содержимого в файл

Чтение и вывод содержимого из файла

Функция `fileStore` извлекает объект, реализующий интерфейс `File` из листинга 9.1. Это может быть реализация подключения к облачному поставщику, например к Amazon Web Services, Google Cloud или Microsoft Azure. Существует большое разнообразие видов облачных хранилищ файлов, включая хранилища объектов, блоков и файлов. Реализация может подключаться к любому из них. Поскольку реализацию можно менять, приложение не будет зависеть ни от одного из этих видов.

Используемое хранилище файлов можно определить в конфигурации, как рассказывалось в главе 2. В конфигурации могут также

храниться подробные сведения о типе хранилища и необходимые учетные данные. Учетным данным следует уделить особое внимание, поскольку при разработке и тестировании следует использовать учетную запись, отличную от той, что будет применяться во время эксплуатации.

В данном примере функция использует локальную файловую систему для хранения и извлечения файлов. В приложении путь к файлу почти наверняка будет фиксированным:

```
func fileStore() (File, error) {
     return &LocalFile{Base: "."}, nil
}
```

Та же идея применяется ко всем взаимодействиям с облачным поставщиком, будь то добавление вычислительных ресурсов (например, создание виртуальных машин), добавление пользователей в систему управления взаимосвязями (Customer Relationship Management, CRM) SaaS или что-нибудь еще.

9.2.2. Обработка накапливающихся ошибок

При взаимодействии с облачными провайдерами обработка ошибок так же важна, как и обработка успешных операций. Как свидетельствует опыт интернет-сообщества, для успешного взаимодействия с облаком необходимо тщательно обрабатывать ошибки.

РЕЦЕПТ 57 Аккуратная обработка ошибок облачных провайдеров

В рецепте 56 ошибки, возвращаемые методами `Load` и `Save`, передавались вызвавшему коду. При таком подходе приложению будут передаваться ошибки, различные для каждой реализации.

Если вы решите, что прикладной код, вызвавший методы `Load` и `Save`, должен выводить эти ошибки перед конечным пользователем или пытаться выяснить их смысл для целей восстановления, вы быстро столкнетесь с проблемой поддержки нескольких облачных поставщиков. Каждый поставщик, каждый SDK и каждая реализация могут иметь разные ошибки. И вместо того чтобы взаимодействовать с прозрачным интерфейсом, подобным интерфейсу `File` из листинга 9.1, приложение вынуждено будет учитывать особенности ошибок для каждой из реализаций.

Проблема

Как выводить сообщения или еще как-то реагировать на возникновение ошибок в методах взаимозаменяемых реализаций интерфейса?

Решение

В реализации интерфейса вместе с методами следует организовать экспорт ошибок. Вместо конкретных ошибок реализации будем возвращать собственные ошибки.

Обсуждение

Следующий листинг продолжает пример из рецепта 56 и определяет несколько распространенных ошибок, которые будут применяться вместе с интерфейсом.

Листинг 9.4 ❖ Ошибки, применяемые с интерфейсом для доступа к облачным функциям

```
var (
     ErrFileNotFound   = errors.New("File not found")
     ErrCannotLoadFile = errors.New("Unable to load file")
     ErrCannotSaveFile = errors.New("Unable to save file")
)
```

Экспорт трех ошибок для использования с интерфейсом из листинга 9.1

Когда ошибки определяются как экспортируемые переменные пакета, их можно использовать в операциях сравнения. Например:

```
if err == ErrFileNotFound {
       fmt.Println("Cannot find the file")
}
```

Для иллюстрации использования этих ошибок в реализациях интерфейса в следующем листинге приводится измененный код метода `Load` из листинга 9.2.

Листинг 9.5 ❖ Добавление ошибок в метод загрузки файла

```
func (l LocalFile) Load(path string) (io.ReadCloser, error) {
     p := filepath.Join(l.Base, path)
     var oerr error
     o, err := os.Open(p)
     if err != nil && os.IsNotExist(err) {
          log.Printf("Unable to find %s", path)
          oerr = ErrFileNotFound
     }
     return o, oerr
}
```

← Переменная для хранения возвращаемой ошибки. Значение по умолчанию nil

Если файл не найден, возвращается ошибка File Not Found

Регистрация ошибки, чтобы она не была потеряна

```
    } else if err != nil {
          log.Printf("Error loading file %s, err: %s", path, err)
          oerr = ErrCannotLoadFile
    }
    return o, oerr
}
```

← Возврат ошибки, если она имеется, вместе с файлом

Обработка ошибок, не связанных с отсутствием файла

Регистрация исходной ошибки имеет большое значение. Если при подключении к удаленной системе возникает проблема, она должна быть зафиксирована в журнале. Система мониторинга может перехватывать ошибки связи с внешними системами и выводить предупреждения, чтобы дать возможность устранения таких проблем.

Регистрируя исходную и возвращая обобщенную ошибку, вы сможете при необходимости перехватить ошибку реализации и обработать ее. Код, вызывающий этот метод, может знать, как реагировать на возвращаемую ошибку, не вдаваясь в детали реализации.

Регистрация имеет большое значение для наблюдения за приложениями в среде выполнения и отладки проблем, но это только основа работы в облаке. Определение параметров окружающей среды, мониторинг среды выполнения, извлечение информации по требованию и многие другие характеристики оказывают влияние на эффективность работы приложения и определяют возможности его настройки. Далее будут рассматриваться приложения, работающие в облаке.

9.3. Выполнение на облачных серверах

При разработке приложений для работы в облаке сведения о среде выполнения могут быть известны или неизвестны заранее. Создание приложений, толерантных по отношению к неизвестной среде, поможет в выявлении и решении проблем, которые могут возникнуть при их эксплуатации.

Иногда бывает необходимо писать приложение в одной операционной системе и архитектуре, а эксплуатировать в другой. Например, вам может понадобиться писать приложение в Windows или Mac OS X, которое будет эксплуатироваться в Linux.

В этом разделе вы познакомитесь с рекомендациями, как избежать ошибок, связанных с особенностями конкретной среды выполнения.

9.3.1. Получение сведений о среде выполнения

Как правило, информацию об окружении предпочтительнее извлекать непосредственно во время выполнения, а не закладывать пред-

полагаемые характеристики в код. Поскольку Go-приложения взаимодействуют с ядром, единственное, что требуется знать изначально, сводится к определению операционной системы: Linux, Windows или какая-то другая. Подробные сведения обо всем, что не касается ядра, для которого было скомпилировано Go-приложение, могут быть получены во время выполнения. Это позволяет Go-приложению выполняться в Red Hat Linux и в Ubuntu. Кроме того, Go-приложение способно предупреждать об отсутствии необходимых зависимостей, что значительно упрощает устранение подобных неполадок.

РЕЦЕПТ 58 Получение информации о хосте

Облачные приложения могут выполняться в разных окружениях, например в окружении для разработки, для тестирования и для эксплуатации. Они могут масштабироваться по горизонтали и иметь несколько выполняющихся экземпляров, запускаемых динамически. Кроме того, они могут одновременно выполняться в нескольких центрах обработки данных. Подобное разнообразие затрудняет предсказание характеристик среды выполнения и передачу этих сведений через конфигурацию приложения.

Вместо передачи через конфигурацию или использования предположений о хосте информацию об окружении можно получать в процессе выполнения.

Проблема

Как получить информацию о хосте внутри Go-приложения?

Решение

Пакет `os` позволяет получать сведения о базовой системе. Извлеченную с его помощью информацию можно объединить с информацией, полученной с помощью других пакетов, таких как `net`, или через вызовы внешних приложений.

Пакет `os` позволяет обнаруживать широкий спектр подробных сведений об окружении. Несколько примеров получения таких сведений перечислено ниже:

- метод `os.Hostname()` возвращает имя хоста;
- идентификатор процесса приложения можно получить, вызвав метод `os.Getpid()`;
- операционные системы используют различные разделители элементов путей и путей в списках. Использование свойства `os.PathSeparator` или `os.PathListSeparator` вместо жестко за-

данных символов позволит приложению выполняться в любой системе;

- для получения текущего рабочего каталога используется метод `os.Getwd()`.

Информацию из пакета `os` можно использовать в сочетании со сведениями, полученными из других источников. Например, попытка определить IP-адреса компьютера, на котором выполняется приложение, может завершиться получением длинного списка всех адресов, присвоенных всем сетевым интерфейсами на этом компьютере. Этот список будет включать локальный адрес, IPv4-адреса и IPv6-адреса, даже если с их помощью нельзя получить доступ к компьютеру. Для получения используемого IP-адреса приложению следует определить имя хоста, известное системе, и найти связанный с ним IP-адрес. Это решение демонстрируется в следующем листинге.

Листинг 9.6 ❖ Получение IP-адреса с помощью имени хоста

```
func main() {
     name, err := os.Hostname()
     if err != nil {
            fmt.Println(err)
            return
     }

     addrs, err := net.LookupHost(name)
     if err != nil {
            fmt.Println(err)
            return
     }

     for _, a := range addrs {
            fmt.Println(a)
     }
```

Получение имени хоста с обработкой ошибок

Поиск IP-адресов, связанных с этим именем хоста

Вывод IP-адресов в цикле, так как их может быть несколько

Системе известно имя собственного хоста, и поиск адресов для этого имени вернет также локальный адрес. Это решение может пригодиться в приложениях, способных выполняться в разных окружениях и масштабироваться по горизонтали. Имя хоста и его адрес могут меняться достаточно часто.

Go-приложения могут компилироваться для разных операционных систем и работать в разных окружениях. Приложения могут извлекать сведения об окружении, а не пользоваться предположениями. Это исключает возможность ошибок и других непредвиденных ситуаций.

РЕЦЕПТ 59 Выявление зависимостей

Помимо взаимодействия с ядром и базовой операционной системой, Go-приложения могут обращаться к другим приложениям. Обычно для этой цели используется пакет `os/exec` из стандартной библиотеки. Но что произойдет, если попытаться вызвать приложение, отсутствующее в системе? Предположение о существовании зависимостей может привести к непредвиденным ситуациям, а неспособность обнаруживать подобные проблемы усложнит диагностирование ошибок.

Проблема

Как проверить доступность внешнего приложения до его вызова?

Решение

Перед вызовом внешнего приложения удостоверимся, что оно установлено и готово к использованию. Если приложение отсутствует, запишем в журнал ошибку, что поможет в устранении неполадок.

Обсуждение

Выше уже говорилось, что Go-приложения могут выполняться в различных операционных системах. Например, приложение, скомпилированное для Linux, можно запустить в различных дистрибутивах с разными установленными приложениями. Если приложение полагается на другое приложение, последнее может присутствовать или отсутствовать в системе. Ситуация еще больше усложняется в облаке, где может использоваться большое количество дистрибутивов. Некоторые специализированные дистрибутивы Linux для облака имеют порой сильно ограниченный набор команд.

Всякий раз, когда облачное приложение полагается на другое приложение, оно должно проверить его присутствие и зафиксировать факт отсутствия требуемой зависимости. Это легко можно сделать с помощью пакета `os/exec`. Следующий листинг демонстрирует функцию, выполняющую такую проверку.

Листинг 9.7 ❖ Функция для проверки доступности приложения

```
func checkDep(name string) error {
     if _, err := exec.LookPath(name); err != nil {
          es := "Could not find '%s' in PATH: %s"
          return fmt.Errorf(es, name, err)
     }
     return nil
}
```

← Возврат значения nil при отсутствии ошибок

Возврат ошибки при отсутствии зависимости

Проверка наличия зависимости в одном из путей в PATH. При отсутствии генерируется ошибка.

Эту функцию можно использовать для проверки существования зависимости. Следующий фрагмент демонстрирует такую проверку и реакцию на ошибку:

```
err := checkDep("fortune")
if err != nil {
     log.Fatalln(err)
}
fmt.Println("Time to get your fortunte")
```

Этот пример регистрирует ошибку, если зависимость не обнаружена. Но не всегда следует ограничиваться регистрацией. Может существовать резервный метод доступа к отсутствующим зависимостям, или могут иметься альтернативные зависимости. Иногда отсутствие зависимости может оказаться фатальным для приложения, но бывают и случаи, когда можно пропустить определенное действие, если зависимость отсутствует. Когда выясняется, что чего-то не хватает, возникшую ситуацию можно обработать наиболее подходящим способом.

9.3.2. Сборка для облака

Не существует единой аппаратной архитектуры и операционной системы для облака. Можно написать приложение для архитектуры AMD64, работающее в Windows, а затем узнать, что его требуется запускать на ARM8 в Linux. Сборка для облака требует от разработчика гарантировать поддержку нескольких окружений, чего достаточно легко добиться с помощью стандартной библиотеки.

РЕЦЕПТ 60 Кросс-платформенная компиляция

Помимо различных облачных окружений, часто бывает так, что приложение разрабатывается в Microsoft Windows или Apple OS X, а эксплуатируется в Linux или распространяется через облако в версиях для Windows, OS X и Linux. Очень часто приложение разрабатывается в одной операционной системе, а эксплуатироваться должно в другой.

Проблема

Как скомпилировать приложение для архитектур и операционных систем, отличных от тех, на которых выполняется компиляция?

Решение

Набор инструментов `go` поддерживает кросс-платформенную компиляцию для других архитектур и операционных систем. Инстру-

мент gox, дополняющий набор go, позволяет выполнять параллельную кросс-платформенную компиляцию нескольких исполняемых файлов. Также можно использовать пакеты, например пакет filepath, для учета различий между операционными системами вместо жестко зафиксированных в коде значений, таких как разделитель путей / в стиле POSIX.

Обсуждение

Начиная с версии 1.5 компилятор языка Go, устанавливаемый вместе с набором инструментов go, поддерживает кросс-платформенную компиляцию, что называется, «из коробки». Для этого необходимо в переменных окружения GOARCH и GOOS указать целевую архитектуру и операционную систему. Переменная GOARCH определяет аппаратную архитектуру, например amd64, 386 или arm, в то время как переменная GOOS задает операционную систему, например windows, linux, darwin или freebsd.

Следующий пример служит краткой иллюстрацией этого решения:

```
$ GOOS=windows GOARCH=386 go build
```

Данная команда соберет двоичный файл для Windows и архитектуры 386. В частности, полученный исполняемый файл будет иметь тип «PE32 executable for MS Windows (console) Intel 80386 32-bit».

> **ПРЕДУПРЕЖДЕНИЕ** Если приложение использует пакет cgo для взаимодействия с библиотеками на языке C, могут возникнуть осложнения. Обязательно протестируйте такое приложение на всех используемых платформах.

Чтобы скомпилировать приложение сразу для нескольких операционных систем и архитектур, можно воспользоваться инструментом gox, который позволяет одновременно создавать несколько исполняемых файлов, как показано на рис. 9.4.

Установить инструмент gox можно следующей командой:

```
$ go get -u github.com/mitchellh/gox
```

После установки gox можно создавать двоичные файлы параллельно. Следующий листинг демонстрирует сборку приложения для OS X, Windows и Linux на архитектурах AMD64 и 386.

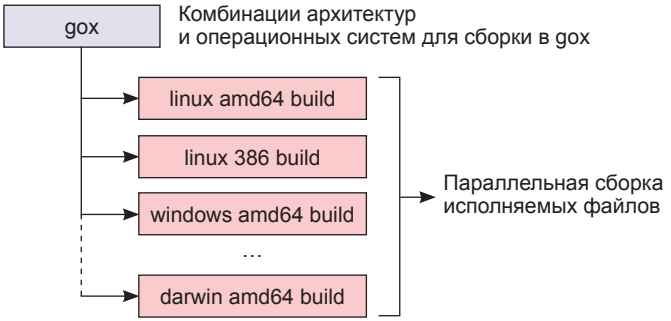


Рис. 9.4 ❖ *Инструмент gox собирает исполняемы файлы для различных операционных систем и архитектур параллельно*

Листинг 9.8 ❖ Кросс-платформенная компиляция приложений с помощью `gox`

```
$ gox \
  -os="linux darwin windows " \   ← Флаг os определяет список операционных систем
  -arch="amd64 386" \             ← Флаг arch определяет одну или несколько архитектур
  -output="dist/{{.OS}}-{{.Arch}}/{{.Dir}}" .
```

Местоположение исполняемых файлов определяется с помощью шаблона. В данном случае файлы с одинаковыми именами будут помещены в разные каталоги, в зависимости от операционной системы и архитектуры

При сборке исполняемых файлов в других операционных системах, особенно при их эксплуатации в облаке, рекомендуется протестировать результат перед развертыванием. Это позволит выявить все ошибки, связанные с окружением, до развертывания приложения.

Помимо компиляции для различных окружений, приложение должно учитывать различия операционных систем. Язык Go обеспечивает две возможности, помогающие обеспечить это.

Во-первых, пакеты предоставляют единый интерфейс, который внутренне обрабатывает различия. Например, одним из наиболее известных отличий являются разные разделители элементов путей и путей в списках. В Linux и в других POSIX-совместимых системах такими разделителями являются / и : соответственно. В Windows используются разделители \ и ;. Для учета этого нужно использовать пакет `path/filepath`, что гарантирует безопасную в этом смысле обработку путей. Этот пакет предлагает следующие возможности:

- `filepath.Separator` и `filepath.ListSeparator` – представляют значения разделителей элементов путей и путей в списках для опе-

рационной системы, под которую приложение компилируется. Их можно использовать везде, где требуется прямой доступ к разделителям;

- `filepath.ToSlash` – принимает строку, представляющую путь, и заменяет в ней разделители нужными символами;
- `filepath.Split` и `filepath.SplitList` – разбивают пути на элементы и списки путей на отдельные пути. Эти методы также используют правильные разделители;
- `filepath.Join` – объединяет элементы в полный путь, используя соответствующий разделитель для операционной системы.

Набор инструментов `go` поддерживает также теги сборки для фильтрации файлов с исходным кодом по типам операционных систем и архитектур, для которых выполняется компиляция. Тег сборки помещается в начало файла и выглядит следующим образом:

```
// +build !windows
```

Этот специальный комментарий требует от компилятора пропустить данный файл при сборке для Windows. Теги сборки могут содержать несколько значений. Следующий пример пропускает файл при сборке для Linux и OS X (`darwin`):

```
// +build !linux,!darwin
```

Их значения напрямую соответствуют значениям переменных окружения `GOOS` и `GOARCH`.

Язык Go также позволяет присваивать файлам такие имена, что они будут выбираться в различных окружениях. Например, файл `foo_windows.go` будет скомпилирован и использован в сборке для Windows, а файл `foo_386.go` будет использован при компиляции для аппаратной архитектуры 386 (иногда называемый x86).

Эти возможности позволяют писать приложения для нескольких платформ и учитывать их различия и уникальные особенности.

9.3.3. Мониторинг среды выполнения

Мониторинг – важный аспект эксплуатации приложений. Обычно мониторинг применяется для выявления моментов, когда нагрузка достигает уровней, требующих масштабирования вверх или вниз, или для понимания происходящего внутри приложения в целях ускорения его выполнения.

Самым простым способом мониторинга является журналирование проблем и прочих сведений. Подсистема журналирования может

записывать сведения на диск, а другое приложение – читать их, или подсистема журналирования может передавать данные в приложение, осуществляющее мониторинг.

РЕЦЕПТ 61 Мониторинг среды выполнения Go

Go-приложение включает в себя не только прикладной код и код библиотек, но также среду выполнения Go, обеспечивающую параллельные вычисления, сборку мусора и другие аспекты работы приложения.

Среда выполнения имеет доступ к массе информации. Это и число процессоров, доступных приложению, и текущее количество сопрограмм, и сведения о выделенной и использованной памяти, и подробные сведения о сборке мусора, и многое другое. Эта информация будет полезна для выявления проблем внутри приложения и запуска определенных действий, таких как горизонтальное масштабирование.

Проблема

Как организовать журналирование или как-то иначе осуществлять мониторинг среды выполнения Go?

Решение

Пакеты `runtime` и `runtime/debug` предоставляют доступ к сведениям о среде выполнения. Регулярно получая информацию с помощью этих пакетов, ее можно записывать в журнал или передавать службе мониторинга.

Обсуждение

Предположим, что при обновлении импортируемой библиотеки возникает серьезная ошибка, препятствующая уничтожению сопрограмм. Они начинают накапливаться, и среда выполнения вынуждена обрабатывать несколько миллионов сопрограмм, тогда как их должно быть несколько сотен. (Авторам не надо представлять эту ситуацию, поскольку они с ней уже столкнулись.) Мониторинг среды выполнения позволяет выявлять подобные случаи.

В начале работы приложение может запустить сопрограмму для мониторинга среды выполнения и записи сведений в журнал. Осуществление мониторинга в сопрограмме позволяет вести наблюдение и журналирование параллельно с основной работой приложения, как показано в следующем листинге.

Листинг 9.9 ❖ Мониторинг среды выполнения приложения

Функция для мониторинга среды выполнения

При запуске мониторинга вывести количество доступных процессоров

```
func monitorRuntime() {
     log.Println("Number of CPUs:", runtime.NumCPU())
     m := &runtime.MemStats{}
     for {
          r := runtime.NumGoroutine()
          log.Println("Number of goroutines", r)

          runtime.ReadMemStats(m)
          log.Println("Allocated memory", m.Alloc)
          time.Sleep(10 * time.Second)
     }
}
```

Запись количества сопрограмм и объема используемой памяти

Выполнение в цикле с задержкой в 10 секунд между итерациями

```
func main() {
     go monitorRuntime()
     i := 0
     for i < 40 {
          go func() {
          time.Sleep(15 * time.Second)
          }()
          i = i + 1
          time.Sleep(1 * time.Second)
     }
}
```

← В момент запуска приложения начать мониторинг

Создание примеров сопрограмм для имитации использования памяти в течение 40 секунд

Следует отметить, что вызовы функции `runtime.ReadMemStats` приостанавливают среду выполнения Go на короткое время, что может негативно повлиять на производительность приложения. Этого не стоит делать слишком часто, и выполнение операций, приостанавливающих среду выполнения Go, допустимо только в режиме отладки.

Подобная организация мониторинга позволяет заменить журналирование взаимодействием с внешней службой мониторинга. Например, если воспользоваться одной из служб мониторинга New Relic, можно реализовать отправку информации ей или предназначенной для этого библиотеке.

Пакет `runtime` обеспечивает доступ к массе информации:

- информация о сборке мусора, в том числе момент последней сборки, размер кучи, вызывающий сборку, продолжительность последней сборки мусора и многое другое;
- статистика кучи, например количество объектов в куче, размер кучи, используемый объем кучи и т. д.;

- количество сопрограмм, процессоров и вызовов cgo.

Мониторинг среды выполнения может помочь разобраться в самых неожиданных ситуациях и выявить причины сбоев. Он позволяет решить проблемы, связанные с сопрограммами, утечками памяти и другими аспектами.

9.4. Итоги

Облачные вычисления становятся все более популярными, и язык Go активно их поддерживает. Эта глава была посвящена теме использования языка Go в облаке. В то время как предыдущие главы лишь касались облачных вычислений, эта глава охватила все аспекты успешного использования Go-приложений в облаке, в том числе:

- работу с различными поставщиками облачных услуг и приемы, позволяющие избежать зависимости от конкретного поставщика;
- сбор информации о хосте вместо выдвижения предположений о нем;
- компиляцию приложений для различных операционных систем и приемы, позволяющие избежать зависимости от конкретной операционной системы;
- мониторинг среды выполнения Go для выявления причин возникающих проблем и получения подробных сведений о работе приложения.

Следующая глава будет посвящена взаимодействию облачных служб с применением технологий, не связанных с REST API.

Глава 10

Взаимодействие облачных служб

В этой главе рассматриваются следующие темы:

- *введение во взаимодействие между микрослужбами;*
- *повторное использование соединений между службами для повышения производительности;*
- *реализация быстрых операций маршалинга и демаршалинга данных в формате JSON;*
- *буферизация протоколов для ускорения передачи;*
- *взаимодействия через RPC.*

Интерфейс передачи репрезентативного состояния (Representational State Transfer, REST) – самый популярный способ организации связи между службами, а наиболее распространенным форматом данных, используемым для передачи информации, является JSON. REST API – это невероятно мощный способ взаимодействий с приложениями.

Для обмена данными между облачными службами и микрослужбами при широкомасштабном их применении также могут использоваться другие средства, а не только REST API. Некоторые из этих средств обеспечивают более высокую скорость обмена с меньшей нагрузкой на соединения. В архитектуре микрослужб применение сетевых коммуникаций способно заметно увеличить производительность и дать возможность оптимизировать некоторые задачи.

Эта глава начинается со знакомства с архитектурой микрослужб и обсуждения ситуаций, когда сеть может стать узким местом и вызвать ухудшение производительности взаимодействующих служб. Затем будут представлены рецепты, позволяющие ускорить обмен по

REST-соединениям с использованием, в частности, формата JSON. Далее будут рассмотрены приемы, не связанные с REST и JSON и демонстрирующие альтернативные подходы.

Эта глава позволит выйти за рамки приемов взаимодействия, основанных на REST, заменив их более быстрыми решениями, улучшающими производительность и масштабируемость микрослужб.

10.1. Микрослужбы и высокая доступность

Приложения, основанные на архитектуре микрослужб, создаются как коллекции служб, развертываемых независимо друг от друга. Увеличение сложности систем, требование независимого масштабирования отдельных частей приложений и необходимость повышения надежности и отказоустойчивости приложений привели к появлению микрослужб. Примерами микрослужб могут служить приложения управления конфигурациями, например etcd, или перекодировки медиафайлов из одного формата в другой. Обычно микрослужбы обладают следующими характеристиками:

- выполняют одно действие. Например, хранят конфигурацию или преобразуют файлы из одного формата в другой;
- обеспечивают гибкость и поддерживают горизонтальное масштабирование. При изменении нагрузки на микрослужбу ее можно масштабировать вверх или вниз;
- устойчивость к сбоям и ошибкам. Службу можно развернуть так, что она будет работать безотказно, даже когда происходят сбои в отдельных экземплярах.

Это вполне согласуется с философией UNIX: *делать что-то одно, но делать это хорошо*.

Предположим, что нужно разработать службу для преобразования медиафайлов из одного формата в другой. Пользователь выгружает медиафайл, он перекодируется, и медиафайл в новом формате предлагается для загрузки. Это можно реализовать как одно монолитное приложение, содержащее все необходимые компоненты, или в виде набора микрослужб с различными функциями, каждая из которых будет отдельным приложением.

Схема приложения перекодировки, построенного в виде комплекса микрослужб, показана на рис. 10.1.

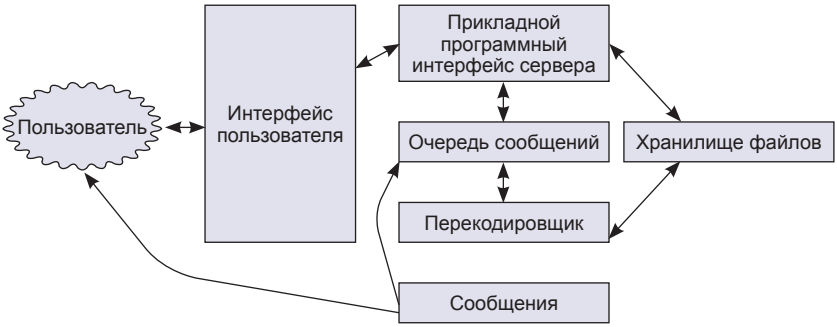


Рис. 10.1 ❖ *Пример приложения перекодировки, разбитого на микрослужбы*

В этом приложении медиафайлы передаются через пользовательский интерфейс серверу. Сервер помещает файл в хранилище и добавляет задание на его преобразование в очередь сообщений. Перекодировщик извлекает задание из очереди, преобразует медиафайл в новый формат, помещает новый файл в хранилище и добавляет в очередь задание на уведомление пользователя о завершении преобразования. Пользователь через пользовательский интерфейс может загрузить преобразованный файл. Пользовательский интерфейс связывается с сервером для получения файла из хранилища.

Каждая из микрослужб может быть написана на своем языке программирования, многократно использоваться разными приложениями и даже применяться в качестве службы. Например, хранилище файлов можно рассматривать как самостоятельную службу.

Масштабирование каждой из этих служб определяется потребностью в услуге. Например, масштабирование службы преобразования можно привязать к количеству медиафайлов, которые требуется перекодировать. К масштабированию прикладного программного интерфейса сервера и службы уведомлений нужно подходить не так, как к масштабированию службы преобразования, то есть в зависимости от соответствующего количества необходимых ресурсов.

Пользователям должна быть гарантирована безотказная работа служб. Выходные дни, когда службы не работают, ушли в прошлое. Случайные отказы ведут к подрыву доверия пользователей и уменьшению доходов. Одно из преимуществ микрослужб заключается в том, что для каждой из служб можно использовать свой подход обеспечения высокой доступности. Например, поддержание высокой

доступности прикладного программного интерфейса сервера коренным образом отличается от поддержания высокой доступности очереди сообщений.

10.2. Взаимодействия между службами

Одним из ключевых элементов архитектуры микрослужб являются взаимодействия между микрослужбами. Некачественная их реализация может повлиять на производительность всего приложения.

В примере приложения перекодировки, изображенного на рис. 10.1, четыре микрослужбы взаимодействуют друг с другом после выгрузки нового медиафайла для перекодировки. Если для взаимодействий, как обычно, используется REST API или протокол TLS, обмен информацией будет осуществляться достаточно медленно.

Значимость производительности взаимодействий возрастает с увеличением количества микрослужб. Компании, использующие микрослужбы, такие как Google, зашли так далеко, что создали новые, более быстрые способы взаимодействий микрослужб, превосходящие все, что уже имеется на рынке.

Вы также можете ускорить взаимодействия в своих приложениях. В этой главе будет продемонстрировано, что это не так уж сложно.

10.2.1. Ускорение REST API

Прикладной программный интерфейс REST является наиболее распространенной формой коммуникации в Интернете и широко применяется при работе с облачными службами. Передача репрезентативного состояния данных по протоколу HTTP получила широкое распространение, но этот протокол не является самым быстрым и эффективным. Кроме того, большинство его настроек по умолчанию является не самыми оптимальными. Их подгонка под конкретные условия часто дает возможность ускорить взаимодействия и улучшить производительность приложения.

РЕЦЕПТ 62 Повторное использование соединений

Нередко для каждого HTTP-запроса создается отдельное соединение. Подготовка соединения занимает определенное время, включающее еще и время на согласование по протоколу TLS для защищенных взаимодействий. Затем, когда выполняется передача сообщения, в работу включается механизм замедления ускорения в реализации

протокола TCP. Целью этого механизма является предотвращение перегрузки сети. При замедленном ускорении передача одного сообщения может потребовать нескольких циклов обмена между клиентом и сервером.

Проблема

При создании отдельного соединения для каждого запроса значительное время тратится на подготовку сетевого соединения. Как можно избежать таких потерь?

Решение

Повторно использовать открытые соединения. Через одно соединение можно передать несколько HTTP-запросов. Многократное использование соединения потребует согласования и замедления ускорения только один раз. После прохождения первого сообщения остальные передаются быстрее.

Обсуждение

Не важно, какую спецификацию использует приложение для взаимодействий – HTTP/2 (доступна начиная с версии Go 1.6) или HTTP/1 и HTTP/1.1, – в любом случае, имеется возможность повторного использования соединений. Среда выполнения Go по умолчанию повторно использует установленные соединения, поэтому в коде приложения ничего менять не придется.

Повторное использование соединений позволяет сократить время на их открытие и закрытие, как иллюстрирует схема на рис. 10.2. Кроме того, механизм замедления ускорения в протоколе TCP включается в работу только один раз, что также сокращает время на передачу следующих сообщений. Поэтому передача второго, третьего и четвертого сообщений при повторном использовании соединения занимает меньше времени.

Сервер, входящий в состав пакета `net/http`, поддерживает сохранение HTTP-соединения открытым. Кроме того, большинство систем изначально поддерживает сохранение TCP-соединений открытыми для повторного использования. Начиная с версии Go 1.6 пакет `net/http` по умолчанию поддерживает спецификацию HTTP/2, обладающую дополнительными преимуществами, позволяющими еще больше ускорить взаимодействия.

> **ПРИМЕЧАНИЕ** Сохранение открытыми HTTP- и TCP-соединений – это разные вещи. Сохранение HTTP-соединений является одной из функций протокола HTTP, реализуемой веб-сервером. Веб-сервер

должен периодически проверять наличие соединения для входящих HTTP-запросов через определенные интервалы. Если в течение этого промежутка времени не поступит ни одного HTTP-запроса, соединение закрывается. С другой стороны, сохранение TCP-соединений осуществляется операционной системой. Параметр `DisableKeepAlives` отключает сохранение обоих видов соединений.

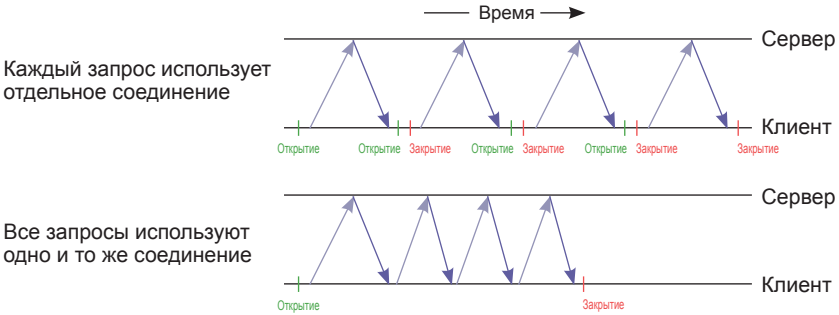


Рис. 10.2 ❖ *Передача сообщений без повторного и с повторным использованием соединений*

Большинство проблем, мешающих повторному использованию соединений, связано с особенностями клиентов, взаимодействующих с HTTP-серверами. Первой и, наверное, наиболее широко распространенной проблемой является применение пользовательских клиентов с отключенной поддержкой сохранения соединений.

Основные функции из пакета `net/http`, такие как `http.Get()` и `http.Post()`, используют клиента `http.DefaultClient`, настроенного так, что он сохраняет соединения, проверяя их с интервалом в 30 секунд. Когда приложение создает клиента, но не определяет транспорт, используется `http.DefaultTransport`. Транспорт `http.DefaultTransport` используется клиентом `http.DefaultClient`, поддерживающим сохранение соединений.

Передачу сообщений без сохранения соединений можно найти в приложениях с открытым исходным кодом, онлайн-примерах и даже в документации с описанием языка Go. Так, в документации с описанием языка Go можно найти такой пример:

```
tr := &http.Transport{
       TLSClientConfig: &tls.Config{RootCAs: pool},
       DisableCompression: true,
}
```

```
client := &http.Client{Transport: tr}
resp, err := client.Get("https://example.com")
```

В этом примере используется пользовательский экземпляр `Transport` с нестандартной проверкой подлинности сертификата и отключенным сжатием. В этом случае сохранение соединений отключено. Следующий листинг содержит аналогичный пример с включенным сохранением соединений.

```
tr := &http.Transport{
     TLSClientConfig: &tls.Config{RootCAs: pool},
     DisableCompression: true,
     Dial: (&net.Dialer{
            Timeout: 30 * time.Second,
            KeepAlive: 30 * time.Second,
     }).Dial,
}
client := &http.Client{Transport: tr}
resp, err := client.Get("https://example.com")
```

Функция Dial настроена на сохранение соединений с интервалом проверки 30 секунд. Прочие настройки совпадают с настройками http.DefaultTransport

При использовании `http.Transport` легко запутаться. Установка в его свойстве `DisableKeepAlives` значения `true` отключает повторное использование соединений. Но установка свойства `DisableKeepAlives` в значение `false` не приводит к автоматическому включению повторного использования соединений. Это лишь означает, что появляется возможность сохранять HTTP- или TCP-соединения.

Если нет особых причин отключать сохранение соединений, не делайте этого. При большом количестве HTTP-запросов сохранение соединений открытыми увеличивает производительность.

Одной из причин, препятствующих повторному использованию соединений, могут быть незакрытые ответы. До появления спецификации HTTP/2 конвейерная обработка практически никогда не использовалась. *Конвейерная обработка* обеспечивает параллельную обработку нескольких запросов и отправку ответов, как показано на рис. 10.3. До спецификации HTTP/2 необходимо было сначала отправить ответ на один запрос и только потом переходить к другому. Тело ответа должно было быть закрыто, чтобы следующий HTTP-запрос и ответ могли повторно использовать соединение.

Следующий листинг демонстрирует часто встречающейся случай, когда тело ответа не закрывается перед передачей следующего HTTP-запроса.

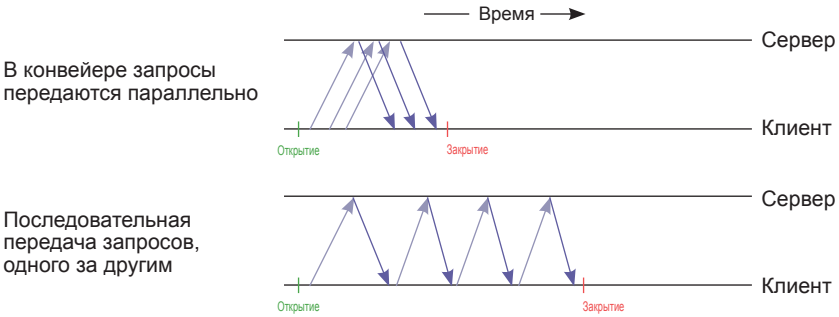


Рис. 10.3 ❖ *Сравнение HTTP-конвейера и последовательных запросов*

Листинг 10.2 ❖ Тело HTTP-ответа не закрывается

```
func main() {
r, err := http.Get("http://example.com")   ← Создание HTTP-запроса
if err != nil {                                и получение ответа
            …
}
defer r.Body.Close()   ← Отложенное закрытие тела при выходе из функции main()
o, err := ioutil.ReadAll(r.Body)
if err != nil {
            …
}
// Использование содержимого тела

r2, err := http.Get("http://example.com/foo")   ← Создание второго HTTP-запроса.
if err != nil {                                    Поскольку тело еще не закрыто,
          …                                        создается новое соединение
}
defer r2.Body.Close()
o, err = ioutil.ReadAll(r2.Body)
if err != nil {
         …
}
…
}
```

В этом случае отложенное закрытие является не лучшим решением. Тело следует закрывать сразу, как только оно становится ненужным. Следующий листинг демонстрирует тот же пример, в котором соединения используются повторно благодаря своевременному закрытию ответа.

Листинг 10.3 ❖ Закрытие HTTP-ответа сразу после использования

```
func main() {
      r, err := http.Get("http://example.com")   ← Создание HTTP-запроса
      if err != nil {                               и получение ответа
            …
      }
      o, err := ioutil.ReadAll(r.Body)
      if err != nil {                               Копирование тела ответа
            …                                       в другой экземпляр
      }                                             с последующим его закрытием
      r.Body.Close()
      // Использовать содержимое тела
      r2, err := http.Get("http://example.com/foo")  ← Выполнение другого
      if err != nil {                                  HTTP-запроса. Этот запрос
            …                                          повторно использует
      }                                                соединение, созданное
      o, err = ioutil.ReadAll(r2.Body)                 для предыдущего запроса
      if err != nil {
            …
      }
      r2.Body.Close()
      …
}
```

Небольшие изменения способны повлиять на работу сетевых соединений и значительно улучшить общую производительность приложения, особенно при масштабировании.

РЕЦЕПТ 63 Ускорение маршалинга и демаршалинга данных в формате JSON

Большинство взаимодействий с REST API сопряжено с передачей данных в формате JSON. Маршалинг и демаршалинг данных в формате JSON обеспечивает пакет `encoding/json`, использующий механизм рефлексии для определения значений и их типов. Пакету `reflect`, реализующему рефлексию, требуется время, чтобы определить типы и значения в каждом сообщении. Если они всегда имеют одну и ту же структуру, имеет смысл сэкономить время, затрачиваемое на рефлексию. Более подробно рефлексия будет рассмотрена в главе 11.

Проблема

Как добиться однократного определения типов данных в формате JSON, чтобы при следующих операциях маршалинга и демаршалинга уже не заниматься этим?

Решение

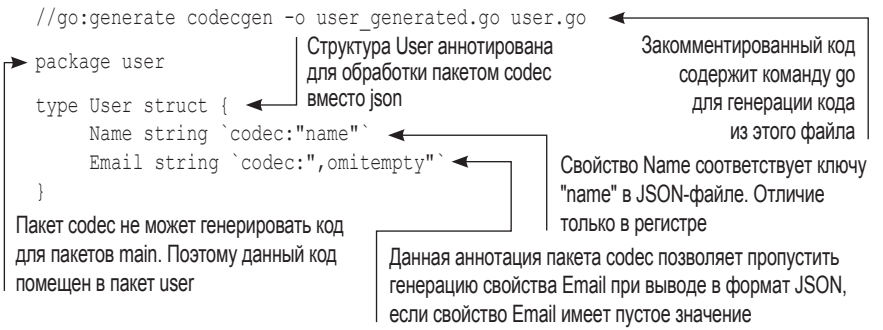
Используем пакет, способный генерировать код для маршалинга и демаршалинга данных в формате JSON. Вновь сгенерированный код пропускает этап анализа данных с применением механизма рефлексии, за счет чего выполняется быстрее и использует меньше памяти.

Обсуждение

Рефлексия в языке Go выполняется недостаточно быстро. Она требует выделения памяти, которую затем сборщик мусора должен утилизировать, и определенных вычислительных затрат. При использовании сгенерированного оптимизированного кода эти издержки могут быть снижены и достигнуто заметное улучшение производительности.

Специально для этого разработано несколько пакетов. Код в листинге 10.4 демонстрирует применение пакета `github.com/ugorji/go/codec`, предназначенного для работы с форматами Binc, MessagePack и Concise Binary Object Representation (CBOR), а также JSON. Binc, MessagePack и CBOR – это альтернативные форматы обмена, но ни один из них не получил такой популярности, как JSON.

Листинг 10.4 ❖ Аннотированная структура для пакета codec



Структура, размеченная для пакета `codec`, практически не отличается от структуры для пакета `json`. Разница лишь в присутствии имени `codec`.

Чтобы сгенерировать код, требуется установить команду `codecgen`, как показано ниже:

```
$ go get -u github.com/ugorji/go/codec/codecgen
```

После установки командой `codecgen` можно сгенерировать код на основе файла `user.go`, выполнив следующую команду:

```
$ codecgen -o user_generated.go user.go
```

Выходной файл будет иметь имя `user_generated.go`. В созданном файле для типа `User` будут добавлены два общедоступных метода `CodecEncodeSelf` и `CodecDecodeSelf`. Пакет `codec` будет использовать эти методы для кодирования или декодирования типа. Если они отсутствуют, пакет `codec` будет использовать обычный способ преобразования во время выполнения.

После установки команды `codecgen` ее можно использовать вместе с командой `go generate`. Команда `go generate` в первой закомментированной строке файла сформирована специально для него и предназначена для выполнения команды `codecgen`. Для использования `go generate` следует выполнить следующую команду:

```
$ go generate ./...
```

> **ПРИМЕЧАНИЕ** Более подробно генераторы и механизм рефлексии будут рассмотрены в следующей главе.

После подготовки типа `User` кодирование и декодирование можно добавлять в нужные места приложения, как показано в следующем листинге.

Листинг 10.5 ❖ Кодирование экземпляра в формат JSON с помощью пакета `codec`

```
jh := new(codec.JsonHandle)
u := &user.User{
     Name: "Inigo Montoya",
     Email: "inigo@montoya.example.com",
}

var out []byte
err := codec.NewEncoderBytes(&out, jh).Encode(&u)
if err != nil {
       …
}

fmt.Println(string(out))
```

Создание нового JSON-обработчика для выполнения кодирования. Пакет codec будет обрабатывать преобразование каждого типа

Создание экземпляра User с данными

Создание массива байтов для сохранения вывода. Это будут сгенерированные JSON-данные для экземпляра User

Декодирование экземпляра User в out с помощью JSON-обработчика. Пакет codec выполняет это в два этапа

Преобразует массив байтов в строку и выводит ее

Этот код выведет:

```
{"name":"Inigo Montoya","Email":"inigo@montoya.example.com"}
```

Обратите внимание, что ключ `name` указан в нижнем регистре, тогда как ключ `Email` начинается с прописной буквы. Имена свойств типа `User`, объявленного в листинге 10.4, начинающиеся с букв в верхнем регистре, должны непосредственно соответствовать ключам. Но для свойства `Name` указан ключ `name`, который и будет использоваться.

Массив байтов с данными в формате JSON, созданный кодом из листинга 10.5, можно декодировать в экземпляр `User`, как показано в следующем листинге.

Листинг 10.6 ❖ Декодирование данных в формате JSON в экземпляр типа

```
var u2 user.User   ← Переменная для хранения декодированных JSON-данных
err = codec.NewDecoderBytes(out, jh).Decode(&u2)
if err != nil {    Декодирование JSON-данных с помощью JSON-обработчика из листинга 10.5
        …          в новый экземпляр типа User. Декодирование выполняется в два этапа.
}                  Декодер имеет возможность повторно использовать JSON-обработчик
fmt.Println(u2)   ← Вывод экземпляра объекта User
```

Хотя интерфейс пакета `github.com/ugorji/go/codec` отличается от интерфейса пакета `encoding/json` из стандартной библиотеки, использовать его достаточно просто.

10.2.2. Выход за рамки прикладного программного интерфейса REST

Архитектурный стиль REST пользуется большой популярностью и с успехом может использоваться для связи с конечными пользователями, но для взаимодействий между микрослужбами существуют более быстрые и эффективные способы. С увеличением числа микрослужб увеличивается и количество взаимосвязей между ними, что делает передачу сообщений по сети узким местом, поэтому стоит рассмотреть другие варианты.

Проблемы передачи по сети стали столь значительными, что крупные компании, такие как Google и Facebook, разработали новые технологии, позволяющие ускорить взаимодействия между микрослужбами.

РЕЦЕПТ 64 Использование формата Protocol Buffers

Обычно для сериализации данных используются форматы JSON и XML. Эти форматы просты в использовании и легко читаются, но они не оптимизированы для передачи по сети и сериализации.

Проблема

Какие форматы, оптимизированные для сериализации и передачи данных по сети, можно использованы в Go-приложениях?

Решение

Некоторые из недавно появившихся форматов, включая Protocol Buffers (известный также как protobuf) от Google и Apache Thrift, первоначально разработанный в Facebook, были созданы с целью ускорить передачу по сети и сериализацию данных. Он поддерживается Go и многими другими языками программирования.

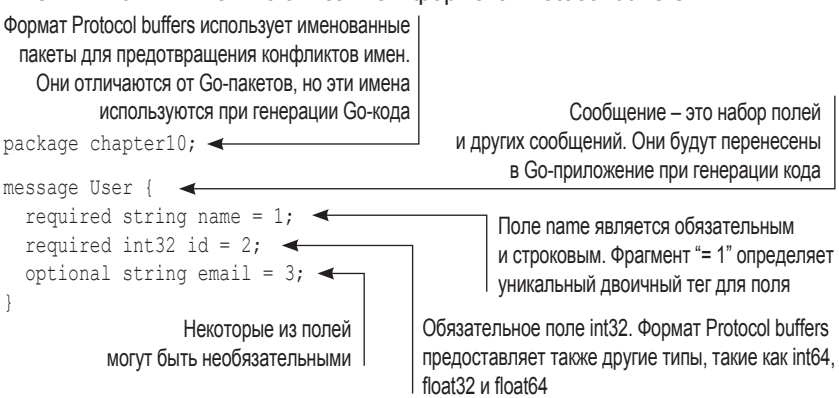
Обсуждение

Protocol Buffers от Google – популярный формат, обеспечивающий высокую скорость передачи. При его использовании по сети передается меньший объем данных, чем при использовании форматов XML и JSON, и преобразования выполняются быстрее, чем для форматов XML и JSON. Передача данных в формате Protocol Buffers не ограничена конкретным методом. Данные могут передаваться внутри файловой системы, с использованием RPC, по протоколу HTTP, через очередь сообщений и множеством других способов.

Google обеспечивает поддержку формата Protocol Buffers для языков C++, C#, Go, Java и Python. Для поддержки в других языках, таких как PHP, существуют сторонние библиотеки.

Формат протокола определяется в файлах. Они содержат структуру сообщения и могут использоваться для автоматического создания необходимого кода. Следующий листинг демонстрирует пример файла *user.proto*.

Листинг 10.7 ❖ Файл с описанием формата Protocol buffers




```
package chapter10;

message User {
  required string name = 1;
  required int32 id = 2;
  optional string email = 3;
}
```

> **СОВЕТ** Более подробную информацию о том, что можно передавать в сообщениях, в том числе о включении сообщений внутрь других сообщений, можно найти в документации, на странице https:// developers.google.com/protocol-buffers/docs/overview.

Поскольку формат Protocol Buffers используется для генерации Go-кода, для работы с ним рекомендуется использовать отдельный пакет. В этом Go-пакете могут размещаться файл Protocol Buffers и сгенерированный код. Как показано в листинге 10.8, в данном случае используется каталог `userpb`.

Чтобы скомпилировать файл Protocol Buffers в код, необходимо сначала установить компилятор:

1. Загрузите и установите компилятор. Его можно найти на странице https://developers.google.com/protocol-buffers/docs/downloads.html.
2. Установите дополнительный плагин Protocol Buffers для Go. Это можно сделать с помощью команды `go get`:

```
$ go get -u github.com/golang/protobuf/protoc-gen-go
```

Чтобы сгенерировать код, выполните следующую команду из каталога, где находится файл с расширением *.proto*:

```
$ protoc -I=. --go_out=. ./user.proto
```

Эта команда содержит следующие параметры:

- `-I` определяет каталог исходных данных;
- `--go_out` определяет, куда поместить сгенерированные Go-файлы;
- `./user.proto` – имя исходного файла для генерации кода.

После того как код будет сгенерирован, его можно использовать для передачи сообщений между клиентом и сервером. Следующий листинг содержит настройки для сервера, возвращающего ответы в формате Protocol Buffers.

Листинг 10.8 ❖ Настройки сервера для работы с Protocol Buffers

```
import (
    "net/http"
    pb "github.com/Masterminds/go-in-practice/chapter10/userpb"
    "github.com/golang/protobuf/proto"
)

func main() {
```

Импорт сгенерированного кода и пользовательских сообщений

Сообщения отправляются с помощью обычного http-сервера

Для работы с форматом Protocol Buffers необходимо импортировать пакет protobuf

```
    http.HandleFunc("/", handler)
    http.ListenAndServe(":8080", nil)
}
```

Простой сервер с единственным обработчиком сгенерированного кода

ПРЕДУПРЕЖДЕНИЕ В этом примере для простоты информация о пользователе передается по протоколу HTTP, но в реальных приложениях следует использовать соединения с шифрованием, чтобы обеспечить надлежащую безопасность.

Листинг 10.8 начинается так же, как все реализации веб-серверов в предыдущих главах. Здесь используется тот же шаблон. Основная работа выполняется в обработчике, представленном в следующем листинге.

Листинг 10.9 ❖ Обработчик формата Protocol Buffers на сервере

```
func handler(res http.ResponseWriter, req *http.Request) {
    u := &pb.User{
        Name:  proto.String("Inigo Montoya"),
        Id:    proto.Int32(1234),
        Email: proto.String("inigo@montoya.example.com"),
    }
    body, err := proto.Marshal(u)
    if err != nil {
        http.Error(res, err.Error(), http.StatusInternalServerError)
        return
    }
    res.Header().Set("Content-Type", "application/x-protobuf")
    res.Write(body)
}
```

Указатель на новый экземпляр сгенерированного типа User в Protocol Buffers

Значения обертываются в вызовы методов из пакета proto, возвращающие ссылки на значения

Преобразование экземпляра в сообщение

Определение типа содержимого и запись сообщения в ответ

Запись ответа в формате Protocol Buffers похожа на запись в формате JSON или XML. Единственное отличие: значения свойств в сообщении являются указателями на значения, а не самими значениями. Вызовы функций `proto.String`, `proto.Int32` и других возвращают указатели на значения.

Для получения и чтения сообщений используется клиент. В следующем листинге приводится код простого клиента.

Листинг 10.10 ❖ Клиент, использующий формат Protocol Buffers

```
res, err := http.Get("http://localhost:8080")
if err != nil {
```

Выполнение GET-запроса к серверу из листингов 10.8 и 10.9

```
        …
}
defer res.Body.Close()
b, err = ioutil.ReadAll(res.Body)
if err != nil {
        …
}
var u pb.User
err = proto.Unmarshal(b, &u)
if err != nil {
        …
}
fmt.Println(u.GetName())
fmt.Println(u.GetId())
fmt.Println(u.GetEmail())
```

← Чтение тела ответа для получения одного полного сообщения в формате Protocol Buffers

Создание переменной и запись в нее преобразованного сообщения. Пакет pb – все тот же пакет, что был импортирован в листинге 10.8

Свойства являются указателями на значения. Сгенерированный код содержит методы Get*, возвращающие значения свойств. Эти методы используются для извлечения значений

Формат Protocol Buffers идеально подходит для передачи сообщений между микрослужбами, если требуется максимально сократить время передачи сообщений.

РЕЦЕПТ 65 Взаимодействие через RPC с помощью Protocol Buffers

Взаимодействие – это нечто большее, чем просто обмен полезными данными. Взаимодействие включает в себя организацию передачи сообщений. REST API обладает определенной семантикой, которую определяют путь и HTTP-метод, и основывается на понятии ресурса. Иногда семантика REST нежелательна, например при вызове прикладного программного интерфейса для перезапуска сервера. Для этого действия больше подходит семантика операций.

Альтернативой является использование вызов удаленных процедур (Remote Procedure Call, RPC). Согласно представлениям RPC, программа выполняет процедуру, которая обычно размещена удаленно. Часто это вообще отдельная служба. Вызовы не привязаны к семантике среды и скорее связаны с выполнением функций в удаленной системе.

Одна из потенциальных проблем RPC заключается в том, что обе взаимодействующие стороны должны обладать подробными сведениями о выполняемой процедуре. Этим RPC отличается от REST, где достаточно информации, передаваемой в сообщениях. Необходимость иметь сведения о процедуре может затруднить работу с RPC, особенно когда службы пишутся на разных языках.

Проблема

Как организовать взаимодействия через RPC способом, не зависящим от используемых языков программирования?

Решение

Используйте gRPC и формат Protocol Buffers для определения интерфейсов, генерируйте обобщенный код и реализуйте RPC-взаимодействия.

Обсуждение

gRPC (*www.grpc.io*) – это высокопроизводительный RPC-фреймворк с открытым исходным кодом, способный использовать HTTP/2 на транспортном уровне. Он был разработан в Google и имеет поддержку для языков Go, Java, Python, Objective-C, C# и некоторых других. С учетом поддерживаемых языков его можно использовать в серверных и в мобильных приложениях. Фреймворк gRPC может использоваться для обмена данными между мобильными устройствами и службами поддержки.

Фреймворк gRPC генерирует код создания сообщений и обработки некоторых этапов взаимодействий. Он упрощает реализацию обмена данными между программами, написанными на разных языках, поскольку интерфейсы и сообщения генерируются для каждого из используемых языков. Для определения сообщений и RPC-вызовов фреймворк gRPC использует формат Protocol Buffers, как показано в следующем листинге.

Листинг 10.11 ❖ Определение сообщений и RPC-вызовов с помощью Protocol Buffers

Фреймворк gRPC требует версию 3 Protocol Buffers. Чтобы получить версию 3, ее необходимо указать в первой строке файла, отличной от комментария

Пакет имеет уникальное имя, которое нигде больше не используется в Protocol Buffers. Это имя не привязано к Go-пакетам, но Go учитывает его

```
syntax = "proto3";

package chapter10;

service Hello {
     rpc Say (HelloRequest) returns (HelloResponse) {}
}
```

При передаче сообщений через службу, подобную RPC, они определяются с помощью службы

RPC-служба вызывает процедуру Say, получающую HelloRequest и возвращающую HelloResponse

```
message HelloRequest {
     string name = 1;
}

message HelloResponse {
     string message = 1;
}
```

Сообщения передаются через RPC-службу Hello. Каждое содержит одно строковое свойство

СОВЕТ Более полную информацию о версии 3 Protocol Buffers можно найти в документации, на странице https://developers.google.com/protocol-buffers/docs/proto3.

Основную работу выполняет код, сгенерированный на основе этого файла Protocol Buffers. Команда генерации кода несколько отличается от примера в рецепте 64. Она показана ниже (выполняется из того же каталога, где находится файл *hello.proto*):

```
protoc -I=. --go_out=plugins=grpc:. ./hello.proto
```

Разница, как видите, в параметре `--go_out=plugins=gprc:.`, где дополнительный встраиваемый модуль gRPC указывается как часть вывода при генерации Go-кода. При отсутствии этого параметра код заглушки службы не будет сгенерирован. В результате выполнения этой команды будет сгенерирован код для сообщений и работы с пакетом gRPC на языке Go.

ПРЕДУПРЕЖДЕНИЕ Для использования версии 3 Protocol Buffers необходимо установить версию 3.0.0 приложения Protocol Buffers. Ее можно загрузить на странице https://github.com/google/protobuf/releases.

Следующий листинг демонстрирует простой сервер в стиле Hello World, принимающий RPC-сообщения и отвечающий на них. Это – пример микрослужбы, отвечающей через RPC.

Листинг 10.12 ❖ Отвечающий на запросы gRPC-сервер

```
package main

import (
     "log"
     "net"
     pb "github.com/Masterminds/go-in-practice/chapter10/hellopb"
     "golang.org/x/net/context"
     "google.golang.org/grpc"
)
```

Импорт кода, сгенерированного с помощью Protocol Buffers из листинга 10.11

Импорт пакетов context и gRPC, обеспечивающих работу со сгенерированным кодом. Эти пакеты необходимо установить

```
type server struct{}

func (s *server) Say(ctx context.Context, \
in *pb.HelloRequest) (*pb.HelloResponse, error) {
     msg := "Hello " + in.Name + "!"
     return &pb.HelloResponse{Message: msg}, nil
}

func main() {
     l, err := net.Listen("tcp", ":55555")
     if err != nil {
            ...
     }
     s := grpc.NewServer()
     pb.RegisterHelloServer(s, &server{})
     s.Serve(l)
}
```

Создание процедуры Say, определенной как служба в листинге 10.11. Она принимает и отвечает на сообщение из листинга 10.11

Запуск TCP-сервера, прослушивающего порт 55555 и обрабатывающего все ошибки. Его будет использовать gRPC-сервер

Создание gRPC-сервера и обработка запросов через TCP-соединение

Регистрация службы hello на RPC-сервере. Это гарантирует соответствие интерфейсу

Пакет `golang.org/x/net/context` является важной частью реализации взаимодействий, и фреймворк `gRPC` зависит от него. Этот пакет переносит пределы ожидания, сигналы отмены и другие значения через границы API. Например, пакет `context` может содержать время ожидания, которое должен знать вызываемый. Более подробно об этом рассказывается в обсуждении клиента.

> **ПРИМЕЧАНИЕ** Пакет `context` будет включен в стандартную библиотеку Go в версии 1.7 или позднее.

Для использования службы из листинга 10.12 нужен клиент. Клиент может быть написан на любом языке, а клиентский код можно сформировать с помощью Protocol Buffers. Следующий листинг содержит код Go-клиента, использующий сгенерированный ранее Go-код.

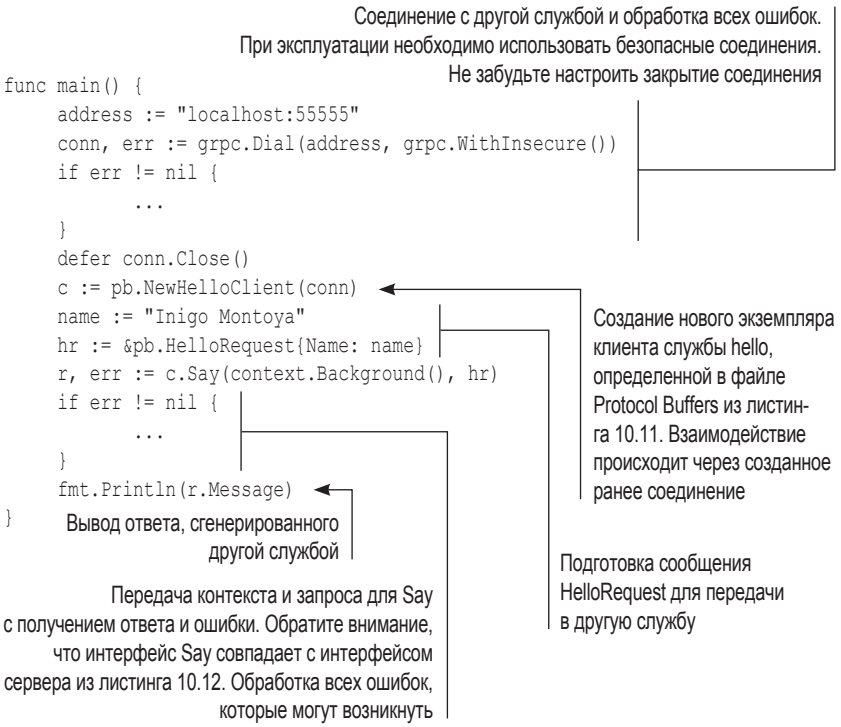
Листинг 10.13 ❖ Запрос к gRPC-серверу с помощью Protocol Buffers

```
package main

import (
     "fmt"
     "os"
     pb "github.com/Masterminds/go-in-practice/chapter10/hellopb"
     "golang.org/x/net/context"
     "google.golang.org/grpc"
)
```

Включение сгенерированного кода из Protocol Buffers для сообщения и службы

← Пакет gRPC нужен для взаимодействия с сервером



Контекст, передаваемый в функцию `Say`, имеет большое значение. В данном случае это пустой контекст, который не может быть отменен и не имеет значений. Это просто фоновый процесс. Пакет `context` содержит и другие контексты, информацию о которых можно найти в документации, на странице https://godoc.org/golang.org/x/net/context.

Одним из примеров может служить использование контекста с функцией отмены. Он полезен при отключении вызывающего (например, когда приложение завершается), когда об этом должен быть проинформирован вызванный объект. Клиент может создать контекст следующим образом:

```
ctx, cancel := context.WithCancel(context.Background())
defer cancel()
```

Завершая выполнение, этот код вызывает функцию `cancel()`. Она посылает контекст, требующий отменить уже проделанную работу. Он будет передан другой службе, даже если она выполняется в дру-

гой системе. На сервере этот контекст может быть получен из канала, связывающего сервер и клиента. Например, RPC-функция, такая как `Say`, может содержать следующий фрагмент кода:

```
select {
case <-ctx.Done():
     return nil, ctx.Err()
}
```

Когда сообщение передается через канал, доступный благодаря функции `Done()`, как это происходит при вызове функции `cancel()`, оно будет получено в этой точке. Функция `ctx.Err()` знает установленный тип отмены. В данном случае сервер вернет ошибку, сообщающую, что операция отменена.

Контексты – мощный инструмент для передачи информации через удаленные вызовы, особенно когда для выполнения удаленного вызова требуется значительный период времени.

Если клиент и сервер используют спецификацию HTTP/2, как это происходит по умолчанию, когда оба абонента используют gRPC, взаимодействия осуществляются с повторным использованием соединений и мультиплексированием, как описывалось выше в этой главе. Это позволяет пользоваться современными быстрыми средствами передачи двоичных сообщений в формате Protocol Buffers.

Перед применением gRPC следует оценить преимущества и недостатки RPC. К преимуществам можно отнести:

- использование формата Protocol Buffers, сокращение объема передаваемых данных, более высокую скорость маршалинга или демаршалинга, чем при работе с JSON и XML;
- контекст, позволяющий отменять операции, устанавливать предельное время ожидания или срок выполнения операции, а также передавать сопутствующую информацию в процессе удаленного вызова;
- взаимодействие осуществляется посредством вызова процедуры, а не отправки сообщения;
- взаимодействие через RPC не ограничено семантикой, например HTTP-методами.

К недостаткам:

- передаваемые данные не настолько удобочитаемы, как при использовании форматов JSON и XML;
- приложениям необходимо знать интерфейс и детали RPC-вызовов. Одной семантики сообщений недостаточно;

- более глубокая интеграция, чем при использовании REST-сообщений, поскольку происходит вызов удаленной процедуры. Предоставлять удаленный доступ непроверенным клиентам следует с большой осторожностью, с соблюдением всех мер безопасности.

В общем, использование RPC может послужить хорошей альтернативой для организации взаимодействий между микрослужбами, контролируемыми разработчиком, которые являются частью большой службы. Эта технология способна обеспечить быстрое и эффективное взаимодействие даже для служб, написанных на разных языках программирования. Обычно возможности использования RPC не должны предоставляться клиентам через средства, не контролируемые разработчиком, такие как общедоступные прикладные программные интерфейсы. Если необходимо просто передавать информацию таким клиентам, предпочтительнее использовать REST и JSON.

10.3. Итоги

В этой главе мы рассмотрели приемы организации взаимодействий через REST API и более быстрые альтернативы. Подобные альтернативы широко используются в Google, Facebook и в других компаниях, применяющих множество отдельных служб, взаимодействующих между собой. Здесь мы охватили следующие темы:

- взаимодействие микрослужб и почему оно может стать проблемой;
- повторное использование соединений для повышения производительности за счет исключения из рабочего процесса механизмов замедления ускорения передачи в протоколе TCP, управления нагрузкой и согласования соединений;
- ускорение маршалинга и демаршалинга JSON-данных за счет исключения затрат на рефлексию;
- использование формата Protocol Buffers вместо JSON при обмене сообщениями;
- взаимодействие через RPC с помощью gRPC.

Следующая глава знакомит с использованием механизма рефлексии и метапрограммирования в языке Go. В ней мы посмотрим, как генерировать код и использовать теги в структурах.

Глава 11

Рефлексия и генерация кода

В этой главе рассматриваются следующие темы:

- *использование значений, видов и типов системы рефлексии языка Go;*
- *синтаксический анализ пользовательских аннотаций структур;*
- *написание генераторов кода для инструмента* `go generate`.

В этой главе мы обратим свои взоры на самые интересные особенности языка Go. Сначала вашему вниманию будет представлена система рефлексии в языке Go. Под *рефлексией* в области разработки программного обеспечения понимается способность программы исследовать собственную структуру. Несмотря на то что система рефлексии в языке Go не настолько универсальна, как в языке Java, она предоставляет достаточно мощные возможности. Другой особенностью является аннотирование структур. Вы увидите, как писать собственные теги для полей структур. Эта особенность поддерживает те же возможности, что и система рефлексии, но при этом позволяет избежать выполнения сложных и затратных операций, заменив их кодом, генерирующим код. Генерация кода помогает добиться того же, что в других языках достигается с применением приемов обобщенного программирования. Такая методика называется *метапрограммированием*, и это последнее, что мы рассмотрим в данной главе.

11.1. Три области применения рефлексии

Разработчики программного обеспечения используют механизм рефлексии для анализа объектов во время выполнения. В строго типизированных языках, подобных языку Go, нужно уметь определять,

соответствует ли заданный объект интерфейсу, выяснять его базовый тип или обходить поля и изменять их значения.

Инструменты рефлексии в языке Go находятся в пакете `reflect`. Чтобы научиться применять эти инструменты, определим несколько понятий, разделив их на три основные области применения: значения, типы и виды.

Первое понятие, *значение*, подразумевает переменную. *Переменная* – это имя, указывающее на элемент данных, как показано на рис. 11.1 (отмечена надписью *Имя переменной*). Элемент данных, на который указывает переменная, называется *значением*. В зависимости от типа в качестве значения может использоваться `nil`. Значение может быть указателем, который, в свою очередь, указывает на другое значение, или непустым элементом данных. Например, после выполнения инструкции `x := 5` переменная `x` получит значение `5`. Код `var b bytes.Buffer` присвоит переменной `b` пустой буфер. Если записать `myFunc:= strings.Split`, значением переменной `myFunc` окажется функция. В пакете `reflect` тип `reflect.Value` соответствует значению.

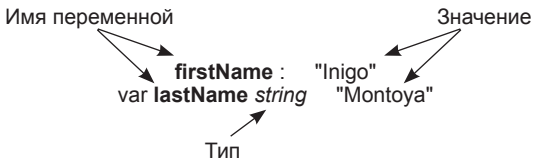


Рис. 11.1 ❖ *Переменные и значения*

Язык Go – это типизированный язык. С каждым значением в языке Go связан конкретный *тип*. Например, значение `var b bytes.Buffer` имеет тип `bytes.Buffer`. Для любого значения `reflect.Value` в языке Go можно получить его тип. Сведения о типе доступны через интерфейс `reflect.Type`.

И наконец, язык Go определяет множество элементарных *видов*, таких как `struct`, `ptr` (указатель), `int`, `float64`, `string`, `slice`, `func` (функция) и т. д. Пакет `reflect` перечисляет все возможные виды с помощью типа `reflect.Kind`. (Обратите внимание, что на рис. 11.1 значение типа `string` имеет также вид `string`.)

Во всех типовых задачах, решаемых посредством рефлексии, присутствуют все эти три понятия. Как правило, рефлексия начинается с получения значения, а затем оно анализируется для определения его содержимого, типа и вида.

РЕЦЕПТ 66 Ветвление на основе типа или вида

Чаще всего механизм рефлексии в языке Go применяется для определения типа и вида заданного значения. Для этого в языке Go имеются различные инструменты.

Проблема

Требуется написать функцию, которая принимает значения обобщенного типа (`interface{}`) и как-то обрабатывает их значения, определяя фактические типы.

Решение

Язык Go поддерживает различные способы анализа подобной информации, включая использование типов `reflect.Type` и `reflect.Kind`. Каждый из них имеет свои сильные и слабые стороны. Для начала рассмотрим, как реализовать ветвление на основе типа, а затем используем пакет `reflect` для реализации ветвления на основе вида.

Обсуждение

Допустим, нужно написать функцию с сигнатурой `sum(... interface{}) float64`. Эта функция должна принимать любое количество аргументов различных типов. Их следует преобразовать в значения с типом `float64` и просуммировать.

Наиболее удобным инструментом для этой цели является оператор `switch`. В этом специальном случае управляющая структура `switch` должна выполнять ветвление, основываясь на типе значения, а не на данных, содержащихся в значении. В стандартной библиотеке Go часто встречается такая конструкция, реализующая ветвление, основанное на типе (однако встречаются также конструкции ветвления, основанные на видах, которые будут рассмотрены далее в этом разделе). Начнем с простого (пусть и неполного) примера.

Листинг 11.1 ❖ Суммирование с применением приема ветвления, основанного на типе

```
package main

import (
     "fmt"
     "strconv"
)

func main() {
     var a uint8 = 2
     var b int = 37
```

```
    var c string = "3.2"
    res := sum(a, b, c)  ← Суммирование переменных с типами uint8, int и string
    fmt.Printf("Result: %f\n", res)
}

func sum(v ...interface{}) float64 {
    var res float64 = 0
    for _, val := range v {
        switch val.(type) {
        case int:
            res += float64(val.(int))
        case int64:
            res += float64(val.(int64))
        case uint8:
            res += float64(val.(uint8))
        case string:
            a, err := strconv.ParseFloat(val.(string), 64)
            if err != nil {
                panic(err)
            }
            res += a
        default:
            fmt.Printf("Unsupported type %T. Ignoring.\n", val)
        }
    }
    return res
}
```

Обход всех заданных значений с ветвлением на основе их типов

Значения всех поддерживаемых типов (int, int64, uint8, string) преобразуются в float64 и суммируются

Для преобразования string в float64 используется библиотека strconv

Если тип значения не является одним из четырех поддерживаемых, выводится сообщение об ошибке, и значение игнорируется

Если выполнить этот код, он выведет `Result: 42.200000`. Этот пример иллюстрирует простой прием ветвления на основе типа, а также одно из его ограничений, по сравнению с обычными операторами `switch`.

В стандартном операторе `switch` можно объединить несколько значений в одной фразе `case`, например `case 1, 2, 3: println("Less than four")`. Если объединить несколько типов во фразе `case`, это приводит к проблемам при присваивании значений, поэтому операторы `switch`, работающие с типами, обычно определяют свое предложение `case` для каждого типа. То есть для поддержки всех целочисленных типов (`int`, `int8`, `int16`, `int32`, `int64`, `uint`, `uint8`, `uint16`, `uint32`, `uint64`) потребуются 10 отдельных фраз `case`. Подобный подход к обработке 10 типов выглядит неудобным, но это не создает особых проблем. Однако важно помнить, что прием ветвления на основе типов ориентирован именно на *типы* (а не виды).

Добавим новый тип в предыдущий пример, как показано в следующем листинге.

Листинг 11.2 ❖ Ветвление на основе типов с дополнительным типом

```
package main

import (
     "fmt"
     "strconv"
)

type MyInt int64   ← MyInt является типом int64

func main() {
//…
     var d MyInt = 1   ← Создание нового экземпляра MyInt со значением 1
     res := sum(a, b, c, d)
     fmt.Printf("Result: %f\n", res)
}

func sum(v ...interface{}) float64 {
     var res float64 = 0
     for _, val := range v {
            switch val.(type) {
            case int:
                 res += float64(val.(int))
            case int64:                                  Это не соответствует типу MyInt
                 res += float64(val.(int64))
            case uint8:
                 res += float64(val.(uint8))
            case string:
                 a, err := strconv.ParseFloat(val.(string), 64)
                 if err != nil {
                        panic(err)
                 }
                 res += a
            default:   ← Здесь будет перехвачено значение с типом MyInt
                 fmt.Printf("Unsupported type %T. Ignoring.\n", val)
            }
       }
       return res
}
```

Эта программа выведет следующее:

```
$ go run typekind.go
Unsupported type main.MyInt. Ignoring.
Result: 42.200000
```

Переменная `var d MyInt` получает тип `MyInt`, а не `int64`. В операторе `switch` ее значение будет обработано фразой `default`, а не `case int64`. Иногда это именно то, что нужно. Но в данном случае было бы желательно, если бы функция `sum()` могла ориентироваться на базовый вид, а не на фактический тип.

Эту проблему можно решить, задействовав пакет `reflect` и работая с видами, а не типами. Следующий пример, демонстрирующий это решение, выглядит так же, как предыдущий, отличаясь только функцией `sum()`.

Листинг 11.3 ❖ Ветвление, основанное на виде

```
package main

import (
     "fmt"
     "reflect"
     "strconv"
)

type MyInt int64

func main() {
//…
     var a uint8 = 2
     var b int = 37
     var c string = "3.2"
     var d MyInt = 1
     res := sum(a, b, c, d)
     fmt.Printf("Result: %f\n", res)
}

func sum(v ...interface{}) float64 {
     var res float64 = 0
     for _, val := range v {
          ref := reflect.ValueOf(val)
          switch ref.Kind() {
          case reflect.Int, reflect.Int64:
               res += float64(ref.Int())
          case reflect.Uint8:
               res += float64(ref.Uint())
          case reflect.String:
               a, err := strconv.ParseFloat(ref.String(), 64)
               if err != nil {
                    panic(err)
               }
               res += a
```

reflect.Kind – это обычный тип, поэтому во фразах case можно использовать несколько значений

С помощью этого значения можно организовать ветвление по Kind()

← Получение reflect.Value элемента

Тип reflect.Value предоставляет функции для преобразования родственных подвидов в их более «широкие» версии (например, int, int8, int16…в int64)

```
            default:
                fmt.Printf("Unsupported type %T. Ignoring.\n", val)
            }
    }
    return res
}
```

В этой переработанной версии ветвление на основе типов заменено обычным ветвлением на основе значения и использованы возможности пакета `reflect` для извлечения значений `val interface{}` с определением описывающего типа `reflect.Value`. Кроме всего прочего, с помощью типа `reflect.Value` можно определить базовый вид значения.

Другой полезной особенностью типа `reflect.Value` является наличие ряда функций, позволяющих преобразовать родственный тип в более широкий тип. Значения `reflect.Value` с типами `uint8` и `uint16` можно без особых усилий преобразовать в более широкий тип целого числа без знака, вызвав метод `Uint()` типа `reflect.Value`.

С помощью этих функций ветвление на основе типов можно преобразовать в более сжатое ветвление на основе видов. Вместо 10 фраз `case` с целочисленными типами можно использовать лишь две фразы `case` (одна – для всех целых чисел со знаком, и одна – для всех целых чисел без знака).

Однако типы и виды – это разные вещи. Здесь приведены два примера, решающих практически одну и ту же задачу. Суммирование числовых значений легче реализовать с помощью определения видов. Но иногда требуется учесть все особенности. Как было продемонстрировано выше, ветвление на основе типов хорошо справляется с обработкой ошибок. Вы можете использовать этот подход для отбора ошибок разных типов, аналогично тому, как это реализовано в других языках с помощью нескольких операторов `catch` в блоке `try/catch`.

Далее в этой главе мы еще вернемся к исследованию типов. Там для получения сведений о структуре будет использоваться тип `reflect.Type` . Но прежде следует познакомиться с еще одной часто встречающейся задачей, заключающейся в выяснении, реализует ли заданный тип некоторый интерфейс.

РЕЦЕПТ 67 Проверка поддержки интерфейса

Система типов в языке Go отличается от подобных систем в традиционных объектно-ориентированных языках, опирающихся на наследование. Язык Go заменяет наследование композицией. Go-интерфейс

определяет коллекцию методов, которые тип должен иметь, чтобы его можно было считать реализующим этот интерфейс. Чтобы было понятнее, о чем речь, рассмотрим простой конкретный пример. Пакет `fmt` определяет интерфейс `Stringer`, который описывает сущность, способную представить себя в виде строки:

```
type Stringer interface {
     String() string
}
```

Любой тип, имеющий метод `String()` без аргументов и возвращающий строку, де-факто реализует интерфейс `fmt.Stringer`.

Проблема

Требуется выяснить, реализует ли заданный тип определенный интерфейс.

Решение

Эту задачу можно решить двумя способами. Один из них основан на проверке типа, а второй использует пакет `reflect`. Вы можете использовать тот, который лучше соответствует конкретным потребностям.

Обсуждение

Интерфейсы в языке Go интерпретируются иначе, чем в объектно-ориентированных языках, таких как Java. В Go отсутствует объявление о реализации интерфейса. Интерфейс – это простое описание, на соответствие которому проверяется тип. Сами интерфейсы также являются типами. Именно поэтому в определениях типов в языке Go не объявляется, каким интерфейсам они соответствуют. Более того, как было показано в главе 4, интерфейсы обычно пишут так, чтобы они соответствовали существующему коду.

Смысл сказанного проще уловить, если обратиться к распространенному подходу к обобщению и классификации. На вопрос «Что общего между лебедями, снежными сугробами и облаками?» люди обычно отвечают «Все они белого цвета». Это не означает, что три сущности имеют общего предка (объектно-ориентированный подход). Их объединяет общая особенность, а именно белизна. На этом и основан подход к типам в языке Go. Типы выражают общность, а не наследование.

Язык Go позволяет легко определить, соответствует ли заданный интерфейс другому интерфейсному типу. Получить ответ на этот вопрос можно с помощью преобразования типа, как это показано в следующем листинге.

Листинг 11.4 ❖ Проверка и преобразование типа

```
package main

import (
     "bytes"
     "fmt"
)

func main() {
     b := bytes.NewBuffer([]byte("Hello"))
     if isStringer(b) {
          fmt.Printf("%T is a stringer\n", b)
     }
     i := 123
     if isStringer(i) {
          fmt.Printf("%T is a stringer\n", i)
     }
}

func isStringer(v interface{}) bool {
     _, ok := v.(fmt.Stringer)
     return ok
}
```

Проверка, реализует ли *bytes.Buffer метод fmt.Stringer. Да, реализует

Проверка, реализует ли целое число метод fmt.Stringer. Нет, не реализует

Получение значения типа interface{} и выполнение проверки соответствия нужному интерфейсу

Проверки типов – один из способов проверить, реализует ли данное значение некоторый интерфейс. Но что, если потребуется узнать, реализует ли данный тип интерфейс, который становится известен только во время выполнения? Для этого понадобятся пакет `reflect` и одна маленькая хитрость.

Выше в этой главе были представлены базовые типы из пакета `reflect`. Проницательный читатель должен был заметить, что там чего-то не хватает. Пакет `reflect` не содержит типа `reflect.Interface`. Вместо этого `reflect.Type` (который сам является интерфейсом) предоставляет инструменты для определения, реализует ли заданный тип конкретный интерфейс. Чтобы определить тип интерфейса во время выполнения, можно использовать тип `reflect.Type`, как это показано в следующем листинге.

Листинг 11.5 ❖ Проверка поддержки интерфейса типом

```
package main

import (
     "fmt"
     "io"
```

```
    "reflect"
)

type Name struct {
    First, Last string
}

func (n *Name) String() string {
    return n.First + " " + n.Last
}

func main() {
    n := &Name{First: "Inigo", Last: "Montoya"}
    stringer := (*fmt.Stringer)(nil)
    implements(n, stringer)
    writer := (*io.Writer)(nil)
    implements(n, writer)
}

func implements(concrete interface{}, target interface{}) bool {
    iface := reflect.TypeOf(target).Elem()

    v := reflect.ValueOf(concrete)
    t := v.Type()

    if t.Implements(iface) {
        fmt.Printf("%T is a %s\n", concrete, iface.Name())
        return true
    }
    fmt.Printf("%T is not a %s\n", concrete, iface.Name())
    return false
}
```

Создание типа Name с методом String() и с последующим созданием экземпляра

← Создание указателя на nil типа fmt.Stringer

← Проверка поддержки интерфейса значением n fmt.Stringer (содержит метод String())

← Создание указателя nil типа io.Writer

← Проверка поддержки интерфейса значением n io.Writer (содержит метод Write())

Получение reflect.Type для целевого интерфейса

Получение reflect.Type для указанного типа

← Проверка соответствия экземпляра целевому интерфейсу

В этом примере применяется в определенном смысле обходной путь. Функция `implements()` принимает два значения. Она проверяет, реализует ли первое переданное значение (`concrete`) интерфейс, указанный во втором аргументе (`target`). Если выполнить этот код, он выведет следующее:

```
$ go run implements.go
*main.Name is a Stringer
*main.Name is not a Writer
```

Тип `Name` реализует `fmt.Stringer`, потому что имеет строковый метод `String()`. Но этот тип не реализует интерфейс `io.Writer`, так как в нем отсутствует метод `Write([]bytes) (int, error)`.

Функция `implements()` предполагает, что `target` – это указатель на значение, динамическим типом которого является интерфейс. С помощью нескольких десятков строк можно задействовать механизм рефлексии и удостовериться, что значение является указателем. При том, как это реализовано сейчас, функция `implements()` может вызвать аварию, если передать ей целевой объект, который не соответствует описанию.

Чтобы получить возможность проверить, реализует ли `concrete` интерфейс `target`, необходимо получить тип `reflect.Type` как для `concrete`, так и для `target`. Это можно сделать двумя способами. Первый использует функцию `reflect.TypeOf()`, чтобы получить тип `reflect.Type`, и вызывает метод `Type.Elem()` для получения типа указателя `target`:

```
iface := reflect.TypeOf(target).Elem()
```

Второй способ заключается в получении значения `concrete`, а затем типа `reflect.Type` этого значения. Далее, с помощью метода `Type.Interface()` можно проверить, реализует ли сущность этого типа данный интерфейс:

```
v := reflect.ValueOf(concrete)
t := v.Type()
```

Самое сложное в такой проверке – получение ссылки на интерфейс. Рефлексию типа интерфейса невозможно осуществить напрямую. Интерфейсы так не работают. Невозможно просто создать экземпляр интерфейса и сослаться непосредственно на него.

Вместо этого следует создать нечто, реализующее интерфейс. Проще всего это достичь, сделав то, чего обычно рекомендуется избегать, а именно намеренно создать указатель на `nil`. В предыдущем коде создаются два указателя на `nil`, например: `stringer := (*fmt.Stringer)(nil)`. Это делается только для того, чтобы создать сущность, необходимую лишь для получения информации о ее типе. При передаче такой сущности в функцию `implements()` появляется возможность выполнить рефлексию указателя на `nil` и определить нужный тип. Для получения типа `nil` потребуется функция `Elem()`.

Код в листинге 11.5 демонстрирует, что работа с системой рефлексии в языке Go требует творческого подхода. Задачи, которые на первый взгляд могут показаться элементарными, часто требуют нетривиального взаимодействия с системой типов.

Далее мы рассмотрим использование системы рефлексии для получения структур и программного доступа к их полям.

РЕЦЕПТ 68 Доступ к полям структуры

Структуры в языке Go являются наиболее часто используемым средством описания структурированных данных. Эффективность применения структур определяется тем, что язык Go имеет доступ к сведениям о содержимом структур еще на этапе компиляции. Во время выполнения также можно получать определенную информацию о структуре, в том числе сведения о полях и присвоенных им значениях.

Проблема

Требуется получить данные о структуре во время выполнения, включая информацию о ее полях.

Решение

Осуществим рефлексию структуры и используем `reflect.Value` и `reflect.Type` для извлечения информации.

Обсуждение

В последних нескольких рецептах вы могли видеть, как, имея значение, получить информацию о нем, его виде и типе. Теперь пришло время собрать воедино эти технологии и с их помощью исследовать структуры.

Создадим инструмент – простую программу, которая будет читать значения и выводить информацию о них в консоль. Заложенные в ней принципы пригодятся в следующем разделе, где при работе с системой аннотаций языка Go будут использоваться аналогичные приемы.

Начнем с анализа нескольких типов.

Листинг 11.6 ❖ Типы для анализа

```
package main

import (
     "fmt"
     "reflect"
     "strings"
)

type MyInt int

type Person struct {
     Name *Name Address *Address
}

type Name struct {
```

```
    Title, First, Last string
}

type Address struct {
    Street, Region string
}
```

Здесь определяются целочисленный тип и несколько структур. Далее напишем код для анализа этих типов, как показано в следующем листинге.

Листинг 11.7 ❖ Рекурсивный анализ значений

```
func main() {
    fmt.Println("Walking a simple integer")
    var one MyInt = 1
    walk(one, 0)

    fmt.Println("Walking a simple struct")
    two := struct{ Name string }{"foo"}
    walk(two, 0)

    fmt.Println("Walking a struct with struct fields")
    p := &Person{
        Name:   &Name{"Count", "Tyrone", "Rugen"},
        Address: &Address{"Humperdink Castle", "Florian"},
    }
    walk(p, 0)
}

type MyInt int

type Person struct {
    Name *Name Address *Address
}

type Name struct {
    Title, First, Last string
}

type Address struct {
    Street, Region string
}

func walk(u interface{}, depth int) {
    val := reflect.Indirect(reflect.ValueOf(u))
    t := val.Type()
    tabs := strings.Repeat("\t", depth+1)
    fmt.Printf("%sValue is type %q (%s)\n", tabs, t, val.Kind())
```

Вывод сведений о простом типе

Вывод сведений о простой структуре

Вывод сведений о полях структуры

Получение reflect.Value для неизвестного значения u. Если получен указатель, он разыменовывается

Функция walk() принимает любое значение и глубину (для целей форматирования)

Получение типа значения

Глубина помогает организовать отступы для форматирования вывода

```
    if val.Kind() == reflect.Struct {
          for i := 0; i < t.NumField(); i++ {
                field := t.Field(i)
                fieldVal := reflect.Indirect(val.Field(i))

                tabs := strings.Repeat("\t", depth+2)
                fmt.Printf("%sField %q is type %q (%s)\n",
                      tabs, field.Name, field.Type, fieldVal.Kind())

                if fieldVal.Kind() == reflect.Struct {
                      walk(fieldVal.Interface(), depth+1)
                }
          }
    }
}
```

Если вид является структурой, анализируются ее поля

Для каждого поля необходимы reflect.StructField и reflect.Value

Если и поле является структурой, выполняется рекурсивный вызов функции walk()

Предыдущий пример объединяет все, что вы узнали о рефлексии. Типы, значения и виды – все это используется здесь в процессе анализа значения и информации, возвращаемой механизмом рефлексии. Если запустить эту программу, она выведет следующее:

```
$ go run structwalker.go
Walking a simple integer
     Value is type "main.MyInt" (int)
Walking a simple struct
     Value is type "struct { Name string }" (struct)
           Field "Name" is type "string" (string)
Walking a struct with struct fields
     Value is type "main.Person" (struct)
           Field "Name" is type "*main.Name" (struct)
     Value is type "main.Name" (struct)
           Field "Title" is type "string" (string)
           Field "First" is type "string" (string)
           Field "Last" is type "string" (string)
           Field "Address" is type "*main.Address" (struct)
     Value is type "main.Address" (struct)
           Field "Street" is type "string" (string)
           Field "Region" is type "string" (string)
```

Как видите, программа проанализировала все переданные ей значения. Сначала она разобрала значение с типом `MyInt` (вид `int`). Затем выполнила обход простой структуры. И наконец, осуществила обход более сложной структуры с рекурсивным погружением до достижения видов, не являющихся структурами.

Основную работу в этой программе выполняет функция `walk()`. Она начинается с проверки неизвестного значения `u`. При работе

с неизвестным значением следует учесть следующие нюансы. После того как функция reflect.ValueOf() получит указатель, она вернет тип reflect.Value, описывающий этот указатель. В данном случае это не интересно. Вместо этого нужно получить значение, на которое ссылается указатель, поэтому далее вызывается метод reflect.Indirect(), чтобы получить тип reflect.Value, описывающий значение, на которое ссылается указатель. Метод reflect.Indirect() удобен тем, что при передаче значения, которое не является указателем, он вернет reflect.Value переданного значения, поэтому ему можно безопасно передавать любые значения:

```
val := reflect.Indirect(reflect.ValueOf(u))
```

Кроме значения переменной u, необходимо получить информацию о ее типе и виде. В этом примере используются все три типа рефлексии:

- значение (указатель будет автоматически разыменован);
- тип;
- вид.

В этом конкретном случае особый интерес представляют виды. Некоторые виды, в частности срезы, массивы, словари и структуры, могут содержать элементы. В этом случае нужно сосредоточиться на получении информации об архитектуре заданного значения (u). В этом примере не требуется перечислять значения в словарях, срезах и массивах, но требуется проанализировать структуры. Если значение имеет вид reflect.Struct, необходимо просмотреть поля структуры.

Самый простой способ перечислить поля структуры – определить тип структуры и затем обойти все поля этого типа с помощью функций Type.NumField() (возвращает количество полей) и Type.Field(). Метод Type.Field() возвращает объект reflect.StructField, описывающий поле. Из него можно узнать тип данных поля и его имя.

Но когда дело доходит до извлечения значения из поля структуры, оказывается, что его невозможно получить ни из reflect.Type (описывающего тип данных), ни из reflect.StructField (описывающего поле типа структуры). Однако это значение можно получить из reflect.Value, описывающего значение структуры. К счастью, сведения о типе и значении можно объединить, чтобы получить числовой индекс типа поля, соответствующего числовому индексу значения поля структуры. Методу Value.Field() можно передать тот же номер поля, что передается в Type.Field(), и получить его значение. И снова, если поле

является указателем, вы получаете ссылку на значение, а не само значение, поэтому вызывается метод `reflect.Indirect()`, чтобы получить значение поля. Все эти действия иллюстрирует вывод предыдущей программы:

```
Field "Name" is type "*main.Name" (struct)
Value is type "main.Name" (struct)
```

Поле `Name` имеет тип `*main.Name`. Но при работе с указателем будет получено значение с типом `main.Name`. В этой небольшой программе обобщено все, что было рассмотрено выше:

- для типа `interface{}` можно использовать метод `reflect.ValueOf()`, чтобы получить `reflect.Value`;
- полученное значение может быть указателем. Чтобы разыменовать указатель и получить `reflect.Value` сущности, на которую он ссылается, вызывается метод `reflect.Indirect()`;
- из `reflect.Value` можно получить тип и вид;
- из структуры (`kind == reflect.Struct`), с помощью функции `Type.NumField()` можно узнать количество полей, а вызовом метода `Type.Field()` – извлечь описания полей (`reflect.StructField`);
- аналогично, с помощью объектов `reflect.Value`, можно получить значения полей в структуре, вызывая метод `Value.Field()`.

При необходимости получения другой информации пакет `reflect` предоставляет средства для анализа методов структур, элементов словарей, списков и массивов, а также сведения о каналах для отправки и получения данных. При всей элегантности языка Go освоение и использование пакета `reflect` часто вызывает сложности, поскольку многие из его функций и методов возбуждают аварии вместо возврата ошибок.

Описанный здесь пример показал одну из самых любимых авторами особенностей языка Go. Далее мы познакомимся с системой аннотаций в языке Go. Вы узнаете, как создавать и использовать собственные теги для структур.

11.2. Структуры, теги и аннотации

В языке Go отсутствуют макросы, и, в отличие от таких языков, как Java и Python, язык Go обеспечивает лишь минимальную поддержку аннотаций. В языке Go можно добавлять аннотации только к полям структур. Такое аннотирование уже использовалось при обработке данных в формате JSON. Рассмотрим пример.

11.2.1. Аннотирование структур

В предыдущей главе вы видели пример использования аннотаций в структуре JSON-кодировщиком. Например, можно взять структуру из листинга 11.5 и добавить в нее аннотации для кодировщика JSON, как показано в следующем листинге.

Листинг 11.8 ❖ Простая JSON-структура

```
package main

import (
     "encoding/json"
     "fmt"
)

type Name struct {
     First string `json:"firstName"`
     Last  string `json:"lastName "`
}

func main() {
     n := &Name{"Inigo", "Montoya"}
     data, _ := json.Marshal(n)
     fmt.Printf("%s\n", data)B
}
```

Аннотирование полей структуры для JSON-кодирования и декодирования

Преобразование n в JSON и вывод результата

Этот код объявляет структуру `Name`, аннотированную для преобразования в формат JSON. Грубо говоря, такое аннотирование позволяет сопоставить элемент структуры `First` с JSON-полем `firstName` и элемент `Last` с полем `lastName`. Ниже показан результат выполнения этого кода:

```
$ go run json.go
{"firstName":"Inigo","lastName":"Montoya"}
```

Аннотирование структур позволяет контролировать JSON-данные. Теги обеспечивают удобный способ добавления небольших фрагментов данных, описывающих поля структуры.

С точки зрения синтаксиса, аннотации – это произвольные строки, заключенные в кавычки, которые следуют за объявлениями типов полей структуры.

Аннотации не оказывают никакого влияния на компиляцию, но к ним можно получить доступ во время выполнения с помощью механизма рефлексии. Это позволяет помещать в аннотации любую ин-

формацию, которой сможет воспользоваться синтаксический анализатор. Например, можно изменить предыдущий код, включив в него различные формы аннотаций, как показано в следующем листинге.

Листинг 11.9 ❖ Различные виды аннотаций

```
type Name struct {
     First string `json:"firstName" xml:"FirstName"`
     Last  string `json:"lastName,omitempty"`
     Other string `not,even.a=tag`
}
```

Все эти аннотации вполне допустимы, в том смысле, что анализатор языка Go способен правильно их обработать. А JSON-кодировщик будет иметь возможность выбрать аннотации, относящиеся к нему. Он пропустит тег `xml`, а также непонятно как отформатированную аннотацию поля `Other`.

Просмотрев теги в листинге 11,9, можно заметить, что аннотации не имеют фиксированного формата. Допускается использовать любые строки. Тем не менее в Go-сообществе устоялся определенный формат аннотаций, и теперь он является стандартом де-факто. Go-разработчики называют соответствующими ему аннотации *тегами*.

11.2.2. Использование тегов

Пример JSON-структуры, представленный выше, содержат аннотации в форме `` `json: "NAME,DATA"` ``, где `NAME` определяет имя поля (в JSON-документах), а `DATA` – список дополнительных сведений о поле (`omitempty` или вид данных). На рис. 11.2 изображен пример структуры, аннотированной для преобразования в форматы XML и JSON.

Аналогично, если заглянуть в пакет `encoding/xml`, можно увидеть аннотации с похожей структурой, предназначенные для преобразования данных в формат XML и обратно. Теги XML определяются как `` `xml:"body"` `` или `` `xml:"href,attr"` ``. Как видите, формат напоминает формат тегов JSON: `` `xml:"NAME,DATA"` ``, где `NAME` определяет имя поля, а `DATA` содержит список сведений о поле (хотя XML-аннотации несколько сложнее, чем JSON-аннотации).

```
type Person struct {
     FirstName string `json:"first" xml:"firstName,attr"`
     LastName  string `json:"last" xml:"lastName"`
}
```

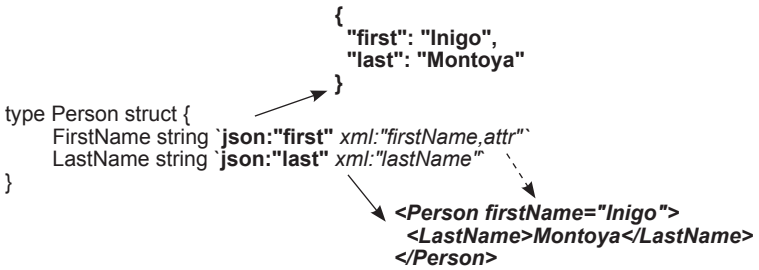


Рис. 11.2 ❖ *Структура, подготовленная для преобразования в форматы JSON и XML*

Аннотации для проверки допустимости

Одним из самых интересных применений аннотаций является проверка допустимости значений полей структуры. Помещая регулярные выражения в теги (`` `validate:"^[a-z]+$"` ``) и добавляя код для сопоставления этих регулярных выражений с данными в структуре, легко можно организовать простую и компактную проверку допустимости. Соответствующий пример можно найти в проекте Deis Router, на странице https://github.com/deis/router.

Однако этот формат не закреплен в определении аннотации структуры. Это просто соглашение, оказавшееся полезным и получившее широкое распространение. Даже пакет `reflect` облегчает работу с тегами, как будет показано ниже.

РЕЦЕПТ 69 Обработка тегов в структурах

Аннотации могут пригодиться в самых разных ситуациях. Предыдущие примеры продемонстрировали их использование в кодировщиках. Аннотации также можно использовать для сопоставления структур с полями баз данных или для форматирования перед выводом. Авторам знакомы даже случаи использования аннотаций для определения порядка передачи значений из структур через функции фильтрации.

И поскольку формат аннотаций не определен, перед созданием аннотаций требуется принять решение об их формате, а затем написать соответствующую реализацию.

Проблема

Требуется создать собственные аннотации и реализовать программный доступ к данным в аннотациях во время выполнения.

Решение

Определим формат аннотаций (предпочтительнее с использованием синтаксиса тегов, описанного выше). Затем используем пакет `reflect` для создания инструмента, извлекающего сведения из аннотаций.

Обсуждение

Допустим, что нужно написать кодировщик для файла с простым синтаксисом в виде набора пар имя-значение. Этот формат напоминает старый формат INI. Пример такого файла выглядит следующим образом:

```
total=247
running=2
sleeping=245
threads=1189
load=70.87
```

Здесь имена находятся слева от знака равенства, а значения справа. Предположим, что нужно создать структуру для представления этих данных. Она показана в следующем листинге.

Листинг 11.10 ❖ Простая структура `Processes`

```
type Processes struct {
     Total    int
     Running  int
     Sleeping int
     Threads  int
     Load     float32
}
```

Для преобразования обычного текстового файла такого формата в структуру можно создать нужные теги и аннотировать ими структуру (как показано в следующем листинге).

Листинг 11.11 ❖ Структура `Processes` с аннотациями

```
type Processes struct {
     Total    int     `ini:"total"`
     Running  int     `ini:"running"`
     Sleeping int     `ini:"sleeping"`
     Threads  int     `ini:"threads"`
     Load     float32 `ini:"load"`
}
```

Эта структура соответствует тому же соглашению, что и JSON- и XML-теги, описанному выше. Но в языке Go отсутствует встроенная функция синтаксического анализа INI-файлов и автоматического определения порядка заполнения структуры по аннотациям свойств. Эту работу должен проделать разработчик.

При разработке имеет смысл учитывать существующие соглашения. Кодеры и декодеры в языке Go, как правило, предоставляют методы `marshal()` и `unmarshal()` с достаточно предсказуемым набором параметров и возвращаемых значений. Поэтому в декодере INI-файла мы реализуем тот же шаблон, представленный в следующем листинге.

Листинг 11.12 ❖ Шаблон маршалинга и демаршалинга

```
func Marshal(v interface{}) ([]byte, error) {}
func Unmarshal(data []byte, v interface{}) error {}
```

Основной частью этих функций является анализ значений `interface{}` и определение, как извлекать данные или заполнять данными эти значения. Для краткости в следующем примере будут показаны только функции маршалинга и демаршалинга структур.

Использование рефлексии, как правило, требует большого объема кода, поэтому разделим код программы на фрагменты, начав с определения структуры для INI-файла и функции `main()`. В первой части определим новый тип (`Processes`), а затем создадим структуру `Processes` в функции `main()`. Выполним маршалинг структуры в INI-формат, а затем демаршалинг в новую структуру `Processes`.

Листинг 11.13 ❖ Структура `Processes` и функция `main()`

```
package main

import (  ← Большинство из импортируемых пакетов будет использовано позднее
     "bufio"
     "bytes"
     "errors"
     "fmt"
     "reflect"
     "strconv"
     "strings"
)

type Processes struct {  ← Структура из листинга 11.11
     Total    int     `ini:"total"`
     Running  int     `ini:"running"`
```

```
    Sleeping int     `ini:"sleeping"`
    Threads  int     `ini:"threads"`
    Load     float64 `ini:"load"`
}

func main() {
    fmt.Println("Write a struct to output:")
    proc := &Processes{
            Total:    23,
            Running:  3,
            Sleeping: 20,
            Threads:  34,
            Load:     1.8,
    }
    data, err := Marshal(proc)
    if err != nil {
          panic(err)
    }
    fmt.Println(string(data))

    fmt.Println("Read the data back into a struct")
    proc2 := &Processes{}
    if err := Unmarshal(data, proc2); err != nil {
          panic(err)
    }
    fmt.Printf("Struct: %#v", proc2)
}
```

Создание экземпляра структуры Processes

← Маршалинг структуры в массив байтов []byte

← Вывод результата

Создание новой структуры Processes и демаршалинг данных в нее

← Вывод структуры

Код верхнего уровня довольно прост. Он начинается с создания экземпляра структуры с последующим ее преобразованием в массив байтов. При выводе результатов они будут отображены в формате INI-файла. Затем выполняется обратная процедура: INI-данные распаковываются в новую структуру `Processes`. Ниже показан результат выполнения программы:

```
$ go run load.go
Write a struct to a output:
total=23
running=3
sleeping=20
threads=34
load=1.8

Read the data back into a struct
Struct: &main.Processes{Total:23, Running:3, Sleeping:20, Threads:34,
Load:1.8}
```

Здесь сначала выводятся данные, полученные в результате маршалинга, а затем – структура, заполненная в ходе демаршалинга. Ниже приводится код функции `Marshal()`, в которой используется многое из того, что вы узнали о механизме рефлексии.

Листинг 11.14 ❖ Функция `Marshal`

```
func fieldName(field reflect.StructField) string {
     if t := field.Tag.Get("ini"); t != "" {
            return t
     }
     return field.Name
}

func Marshal(v interface{}) ([]byte, error) {
     var b bytes.Buffer
     val := reflect.Indirect(reflect.ValueOf(v))
     if val.Kind() != reflect.Struct {
            return []byte{}, errors.New("unmarshal can only take structs")
     }

     t := val.Type()
     for i := 0; i < t.NumField(); i++ {
            f := t.Field(i)
            name := fieldName(f)
            raw := val.Field(i).Interface()
     }
     fmt.Fprintf(&b, "%s=%v\n", name, raw)
     return b.Bytes(), nil
}
```

Получение тега поля структуры

Вспомогательная функция для чтения тегов полей структуры

Если тега нет, возвращается имя поля

Получение reflect.Value текущего интерфейса. Разыменование указателей

← Обработка для этой программы

← Обход всех полей структуры

← Получение имени из tagName

← Возврат содержимого буфера

Использование форматирования для вывода необработанных данных в буфер

Функция `Marshal()` принимает структуру `v interface{}` и читает ее поля. В процессе исследования выполняется обход всех полей структуры и извлекаются аннотации из всех полей (с помощью `StructField.Tag()`). При обходе полей структуры также извлекаются их значения. Преобразование значений полей в строки поручается функции `fmt.Fprintf()`.

Следует отметить, что функция `fieldName()` использует возможность языка Go автоматически анализировать теги. Вы можете (при желании) помещать в аннотации любые строковые данные, но помните, что в Go имеется возможность автоматического анализа тегов. Если тег имеет формат `NAME:"VALUE"`, его значение можно получить с помощью функции `StructField.Tag.Get()`. Она возвращает необ-

работанное значение. Это правило распространяется и на значения тегов, содержащие списки параметров, разделенных запятыми (`json:"myField,omitempty"`). В данном случае поле `VALUE` будет содержать единственное значение. И наконец, в отсутствие тега возвращается имя поля структуры.

Игнорирование полей структуры с аннотациями

Иногда требуется указать кодировщику, что нужно игнорировать некоторые поля в структуре. Общепринятой идиомой для этого является использование дефиса (-) в поле имени аннотации (`json:"-"`). Эта возможность не поддерживается в предыдущем коде, но его можно расширить и реализовать пропуск полей с именем -.

Функция `Marshal()` не обладает особой гибкостью. Например, она работает только со структурами. Словари, которые легко можно преобразовать в INI-поля, не поддерживаются. Кроме того, функция `Marshal()` работает только с определенными типами данных. Она, к примеру, не в состоянии произвести хоть сколько-нибудь полезный результат для полей, значения которых являются структурами, каналами, словарями, срезами или массивами. Однако даже при том, что эта доработка потребует написать большой объем кода, нет ничего особенно сложного в расширении функции `Marshal()` для поддержки более широкого множества типов.

В следующем листинге приводится код, получающий фрагмент INI-данных и преобразующий их в структуру. Он также использует аннотации и подсистему рефлексии.

Листинг 11.15 ❖ Функция `Unmarshal`

```
func Unmarshal(data []byte, v interface{}) error {

    val := reflect.Indirect(reflect.ValueOf(v))
    t := val.Type()

    b := bytes.NewBuffer(data)
    scanner := bufio.NewScanner(b)
    for scanner.Scan() {
        line := scanner.Text()
        pair := strings.SplitN(line, "=", 2)
        if len(pair) < 2 {
            // Пропуск неподходящих строк.
            continue
```

И снова работа начинается с извлечения (разыменования) типа reflect.Value

Полученный из данных сканер используется для чтения строк INI-данных

← Разбивает строку по знаку "="

```
            }
            setField(pair[0], pair[1], t, val)
      }
      return nil
}
```

Установка значения с помощью функции setField()

Функция `Unmarshal()` читает данные из массива `[]byte` и пытается преобразовать найденные в нем поля в соответствующие поля в типе `v interface{}`. Парсинг INI-данных выполняется тривиально просто: строки, извлекаемые из файла, разбиваются на пары имя/значение. Но запись полученных значений в структуру требует уже достаточно серьезных усилий.

Функция `Unmarshal()` опирается на вспомогательную функцию `setField()`, использующую большую часть приемов рефлексии, рассмотренных в этой главе. И снова здесь применяется ветвление на основе видов. Рассмотрим следующий листинг.

Листинг 11.16 ❖ Вспомогательная функция `setField`

Функция setField получает необработанное имя и значение из INI-данных, а также тип и значение самой структуры

```
func setField(name, value string, t reflect.Type, v reflect.Value) {
     for i := 0; i < t.NumField(); i++ {
           field := t.Field(i)
           if name == fieldName(field) {
                 var dest reflect.Value
                 switch field.Type.Kind() {
                 default:
                       fmt.Printf("Kind %s not supported.\n",
                       field.Type.Kind())
                       continue
                 case reflect.Int:
                       ival, err := strconv.Atoi(value)
                       if err != nil {
                             fmt.Printf(
                             "Could not convert %q to int: %s\n", value, err)
                             continue
                       }
                       dest = reflect.ValueOf(ival)
                 case reflect.Float64:
                       fval, err := strconv.ParseFloat(value, 64)
                       if err != nil {
                             fmt.Printf(
                             "Could not convert %q to float64: %s\n", value,
                             err)
```

Обход всех полей структуры, получение поля, чье имя соответствует имени поля в INI-данных

Ветвление на основе вида для преобразования строкового значения в нужный тип

Если тип неизвестен, поле просто пропускается. Это не является ошибкой

Эта версия поддерживает только несколько видов значений. Поддержка других типов не вызывает особых сложностей, но привносит массу повторяющегося кода

После преобразования исходного значения в значение требуемого типа осуществляется обертывание значения

```
                                continue
                        }
                        dest = reflect.ValueOf(fval)
                case reflect.String:
                        dest = reflect.ValueOf(value)
                case reflect.Bool:
                        bval, err := strconv.ParseBool(value)
                        if err != nil {
                                fmt.Printf(
                                "Could not convert %q to bool: %s\n",
                                 value, err)
                                 continue
                        }
                        dest = reflect.ValueOf(bval)
                }
                v.Field(i).Set(dest)   ← Запись значения в соответствующее
        }                                поле структуры
    }
}
```

Функция `setField()` принимает необработанную пару имя/значение, а также `reflect.Value` и `reflect.Type` структуры и пытается отыскать поле структуры, соответствующее паре. (И снова работа ведется только со структурами, хотя есть возможность обрабатывать и отражения.) Получить соответствующее имя поля достаточно просто, поскольку для этого можно воспользоваться функцией `fieldName()` из листинга 11.14. Но, чтобы получить значение, необходимо преобразовать данные из исходной строковой формы в тип данных поля структуры. Для краткости код в листинге 11.14 обрабатывает только несколько типов данных (`int`, `float64`, `string` и `bool`). Кроме того, не анализируются типы, расширяющие базовые типы. Но в шаблон, представленный здесь, легко можно добавить обработку других типов. И наконец, преобразованные значения сначала обертываются функцией `reflect.Value()` и затем помещаются в соответствующие поля структуры.

Просматривая код, реализующий описанный прием, можно сделать вывод, что строгая система типов в языке Go требует писать массу шаблонного кода для преобразований между типами. Иногда для этого можно воспользоваться одним из встроенных инструментов (например, `fmt.Fprintf()`). А иногда приходится писать такой код. В некоторых случаях можно выбрать другой подход – вместо механизма рефлексии использовать генератор кода Go. В следующем раз-

деле вы увидите пример создания генератора, выполняющего ту же работу, для которой обычно требуется осуществлять проверку типов и использовать механизм рефлексии во время выполнения.

11.3. Генерация Go-кода с помощью Go-кода

Новички в Go часто задаются одними и теми же вопросами. Как без средств обобщенного программирования создавать коллекции конкретного типа? Есть ли более простой способ записи в типизированные коллекции, чем использование рефлексии? Механизм рефлексии имеет значительные накладные расходы. Есть ли способ писать более производительный код? Аннотации имеют ограниченные возможности. Есть ли другие способы реализации преобразований? Как уже упоминалось выше, язык Go не поддерживает макросов, а возможности аннотаций в нем весьма ограничены. Существует ли другой способ трансформации кода? Как писать метапрограммы на языке Go?

Часто упускаемой из виду особенностью языка Go является возможность генерации кода. Язык Go поставляется с инструментом `go generate`, разработанным именно для этой цели. Фактически метапрограммирование с помощью генераторов является мощным ответом на все поставленные выше вопросы. Сгенерированный код (который затем компилируется) выполняется намного быстрее, чем код, использующий механизм рефлексии. Кроме того, он, как правило, гораздо проще. Генераторы значительно облегчают задачу создания большого числа повторяющихся типизированных объектов. И хотя многие программисты с предубеждением относятся к метапрограммированию, на практике генераторы кода используются довольно часто. Фреймворки, Protobuf, gRPC и Thrift, представленные в предыдущей главе, используют генераторы. Многие SQL-библиотеки являются генераторами. Некоторые языки внутренне используют генераторы для реализации макросов, средств обобщенного программирования и коллекций. Это здорово, что в языке Go предусмотрены мощные встроенные средства генерации кода.

Средства генерации кода в языке Go сосредоточены в простом инструменте `go generate`. Как и другие инструменты языка Go, инструмент `go generate` является частью окружения Go и может работать с файлами и пакетами. Идея выглядит поразительно просто.

Инструмент обходит указанные файлы и просматривает первые строки в каждом из них. Обнаружив определенный шаблон, он выполняет программу. Шаблон выглядит следующим образом:

```
//go:generate COMMAND [ARGUMENT...]
```

Генератор ищет этот комментарий в начале каждого из указанных Go-файлов. Если заголовок отсутствует, такой файл пропускается. Если заголовок имеется, выполняется команда `COMMAND`, которая может быть любым инструментом командной строки, доступным генератору. Команде можно передать любое количество аргументов. Рассмотрим простой пример в следующем листинге.

Листинг 11.17 ❖ Простейший генератор

```
//go:generate echo hello
package main
func main() {
     println("Goodbyte")
}
```

Это вполне допустимый Go-код. Если скомпилировать и выполнить его, он выведет в консоль строку `Goodbyte`. Но в первой строке он содержит команду для генератора: `echo` – обычную команду UNIX, которая выводит содержимое своих аргументов в поток стандартного вывода. Ей передается один аргумент, строка `hello`. Запустим генератор и посмотрим, что произойдет:

```
$ go generate simple.go
hello
```

Все, что сделал здесь генератор, – это выполнил команду, которая вывела `hello` в консоль. Хотя эта реализация очень проста, но она иллюстрирует идею, заключающуюся в том, что имеется возможность добавлять команды для генерации кода. Это будет продемонстрировано в следующей технологии.

РЕЦЕПТ 70 Генерация кода с помощью инструмента go generate

Создание пользовательских типизированных коллекций, создание структур из таблиц базы данных, преобразование JSON-схем в код, создание множества схожих объектов – это лишь несколько примеров применения генераторов. Иногда Go-разработчики используют для генерации Go-кода пакет Abstract Syntax Tree (AST) или инструмент

yacc. Но мы считаем, что генераторы – более простой и интересный подход к созданию кода, заключающийся в написании Go-шаблонов, генерирующих Go-код.

Проблема

Необходимо получить возможность создавать коллекции определенного типа, такие как очереди для произвольного числа типов. При этом во время выполнения не должно возникать проблем с безопасностью и производительностью, связанных с операциями над типами.

Решение

Создадим генератор, способный создавать очереди, а затем используем заголовки генератора для создания очередей по мере необходимости.

Обсуждение

Очередь – достаточно простая структура: вы помещаете данные в ее конец, а извлекаете из начала. Первое поступившее значение также будет излечено первым (первым пришел – первым вышел, FIFO). Как правило, очереди имеют два метода: `insert` (или `enqueue`) для добавления данных в конец очереди и `remove` (или `dequeue`) для извлечения из начала очереди.

Нам требуется организовать автоматическое создание очередей для хранения определенных типов. Очереди должны соответствовать шаблону, представленному в следующем листинге.

Листинг 11.18 ❖ Простая очередь

```
package main

type MyTypeQueue struct {
     q []MyType
}

func NewMyTypeQueue() *MyTypeQueue {
     return &MyTypeQueue{
            q: []MyType{},
     }
}

func (o *MyTypeQueue) Insert(v MyType) {
     o.q = append(o.q, v)
}

func (o *MyTypeQueue) Remove() MyType {
     if len(o.q) == 0 {
```

← Простая очередь типизированных элементов

← Добавление элемента в конец очереди

← Удаление элемента из начала очереди

```
        panic("Oops.")   ← В действующем коде аварию следует заменить ошибкой.
    }                      Здесь она используется для сокращения генерируемого кода
    first := o.q[0]
    o.q = o.q[1:]
    return first
}
```

Этот код является демонстрацией, что должно быть сгенерировано. В нем имеются определенные детали, которые нужно будет задать в момент генерации. Очевидным примером является тип. Но также желательно автоматически подставлять имя пакета. Следующий наш шаг – преобразование приведенного выше кода в Go-шаблон. В следующем листинге приводится начало реализации генератора очереди.

Листинг 11.19 ❖ Шаблон очереди

```
package main

import (
    "fmt"
    "os"
    "strings"
    "text/template"
)

var tpl = `package {{.Package}}   ← .Package – заполнитель для подстановки имени пакета

type {{.MyType}}Queue struct {   ← .MyType – заполнитель для подстановки типа
    q []{{.MyType}}
}

func New{{.MyType}}Queue() *{{.MyType}}Queue {
    return &{{.MyType}}Queue{
        q: []{{.MyType}}{},
    }
}

func (o *{{.MyType}}Queue) Insert(v {{.MyType}}) {
    o.q = append(o.q, v)
}

func (o *{{.MyType}}Queue) Remove() {{.MyType}} {
    if len(o.q) == 0 {
        panic("Oops.")
    }
    first := o.q[0]
    o.q = o.q[1:]
return first
}
`
```

Шаблон практически полностью совпадает с написанным ранее целевым кодом, но имя пакета заменено на `{{.Package}}`, а префикс `MyType` заменен на `{{. MyType}}`. Теперь нужно написать код, выполняющий генерацию. Это будет инструмент командной строки, соответствующий шаблону `//go:generate COMMAND ARGUMENT...`. В идеале желательно иметь возможность написать что-то вроде:

```
//go:generate queue MyInt
```

В результате генератор должен создать реализацию `MyIntQueue`. Еще лучше было бы иметь возможность создавать очереди сразу для нескольких типов:

```
//go:generate queue MyInt MyFloat64
```

Это тоже легко реализовать, как показано в следующем листинге.

Листинг 11.20 ❖ Функция `main` генератора очередей

```
func main() {
    tt := template.Must(template.New("queue").Parse(tpl))
    for i := 1; i < len(os.Args); i++ {
        dest := strings.ToLower(os.Args[i]) + "_queue.go"
        file, err := os.Create(dest)
        if err != nil {
               fmt.Printf("Could not create %s: %s (skip)\n", dest, err)
               continue
        }
        vals := map[string]string{
                 "MyType": os.Args[i],
                 "Package": os.Getenv("GOPACKAGE"),
         }
         tt.Execute(file, vals)
         file.Close()
     }
}
```

Обход аргументов с созданием для каждого из них файла TYPE_queue.go

Компиляция шаблона генератора

Присваивание параметру .MyType типа, указанного в переданном аргументе

Присваивание параметру .Package значения переменной окружения $GOPACKAGE

Выполнение шаблона, сохранение результатов в файл

Поскольку предусматривается обработка нескольких типов в командной строке, сначала надо выполнить обход `os.Args`. Для каждого из аргументов автоматически создается выходной файл `TYPE_queue.go`. В соответствии с соглашением об именах файлов в Go имя типа должно быть указано строчными буквами.

Шаблон содержит только две переменные. Одна из них определяет обрабатываемый тип данных, а другая – имя пакета. Команда `go generate` дает возможность определить несколько переменных окружения с полезной информацией о местонахождении файла генератора. Переменной окружения `$GOPACKAGE` присваивается имя пакета, где был найден заголовок `go:generate`.

Функции обработки шаблона передаются словарь `vals` и файл, куда следует сохранить готовый Go-файл.

Теперь, чтобы сгенерировать код, необходимо добавить заголовок `go:generate` в соответствующий файл, как показано в следующем листинге.

Листинг 11.21 ❖ Использование генератора

```
//go:generate ./queue MyInt  ← Заголовок генератора для генерации очереди типа MyInt
package main

import "fmt"

type MyInt int  ← Определение типа MyInt

func main() {
     var one, two, three MyInt = 1, 2, 3
     q := NewMyIntQueue()
     q.Insert(one)
     q.Insert(two)
     q.Insert(three)

     fmt.Printf("First value: %d\n", q.Remove())
}
```

Использование MyIntQueue

Это хороший пример типичного использования генератора в языке Go. В одном файле определяются используемый генератор и тип. Очевидно, что этот код не будет скомпилирован, пока не выполнится генератор. Но, поскольку генератор не зависит от компиляции кода, можно (и нужно) выполнить его перед созданием пакета.

В результате возникает важный вопрос, связанный с генераторами: должны ли они использоваться в качестве инструментов разработки. Авторы языка Go утверждают, что генераторы являются частью цикла разработки, а не сборки или выполнения. Например, всегда следует создавать код и передавать сгенерированный код системе управления версиями. Пользователи не должны выполнять генератор (даже если пользователи являются сторонними разработчиками).

Чтобы выполнить предыдущий генератор, необходимо произвести несколько действий. Во-первых, скомпилировать инструмент генерации из листингов 11.18 и 11.19. Чтобы строки генератора сработали, необходимо поместить скомпилированный инструмент в локальный каталог (потому что мы ссылаемся на него как `./queue`). Программу `queue` можно также сохранить в любом каталоге, указанном в переменной пути (`$PATH`), но обычно она помещается в `$GOPATH/bin` и вызывается как `//go:generate queue`.

Скомпилировав `queue` и поместив выполняемый файл туда, где `go generate` сможет найти его, необходимо запустить инструмент `go generate`, а затем можно выполнить основную программу, использующую сгенерированный код:

```
$ ls
myint.go
$ go generate
$ ls
myint.go
myint_queue.go
$ go run myint.go myint_queue.go
First value: 1
```

После того как генератор создал код с реализацией очереди, можно запустить программу из листинга 11.21. Все необходимое уже было сгенерировано.

Здесь используется чрезмерно упрощенная реализация генератора. Вместо полноценной обработки ошибок она создает аварию. Не предусмотрена обработка случая с очередью указателей. Но эти проблемы легко исправить обычными методами. Например, для добавления указателей необходимо предусмотреть еще один параметр шаблона для вставки в нужные места символа звездочки (`*`).

И снова Go-шаблоны – не единственный способ генерации кода. Пакет `go/ast` позволяет генерировать код путем программирования дерева абстрактного синтаксиса. Также можно написать код на Python, C или Erlang, генерирующий Go-код. Можно даже не ограничиваться генерацией Go-кода. Например, можно генерировать SQL-операторы `CREATE` с помощью генератора, читающего структуры. Одной из особенностей, которые делают `go generate` столь элегантным, является универсальность при всей его простоте.

Под впечатлением от инструмента go generate

Элегантность Go-генератора произвела на авторов столь глубокое впечатление, что при разработке диспетчера пакетов Helm для Kubernetes (http://helm.sh) мы реализовали похожее решение для преобразования шаблонов в файлы манифестов Kubernetes.

Таким образом, генераторы могут пригодиться в случаях, когда требуется писать повторяющийся код и вместо них пришлось бы использовать рефлексию. Механизм рефлексии в языке Go обладает некоторыми достоинствами, но он громоздок, имеет ограниченные возможности и отрицательно влияет на производительность приложения. Это не означает, что рефлексию вообще не надо использовать. Скорее, это инструмент, предназначенный для решения специфичных проблем.

С другой стороны, генерация кода не всегда оказывается лучшим решением. Метапрограммирование может затруднять отладку. Оно привносит дополнительные действия в процесс разработки. Как и рефлексию, генерацию следует использовать только тогда, когда без нее не обойтись. Она не является панацеей и средством для обхода строгой типизации языка Go.

11.4. Итоги

Эта глава была посвящена более сложным темам: рефлексии и метапрограммированию. В процессе знакомства с рефлексией вы увидели, как использовать типы, значения, виды, поля и теги для решения различных задач программирования. При обсуждении метапрограммирования было показано, как с помощью инструмента `go generate` писать код, создающий другой код.

Эта глава знакомила со следующими возможностями:

- использование видов для получения важных сведений о типах;
- определение реализации типом заданного интерфейса во время выполнения;
- доступ к полям структуры во время выполнения;
- работа с аннотациями;
- анализ тегов в аннотациях структур;
- написание функций для маршалинга и демаршалинга;
- использование инструмента `go generate`;
- написание шаблонов Go, генерирующих Go-код.

В этой книге был охвачен широкий круг вопросов. Мы надеемся, что представленные в ней технологии, которыми пользуются авторы, вы будете использовать при написании собственного кода. Go – фантастический системный язык. Отчасти это обусловлено простотой его синтаксиса и семантики. Кроме того, разработчики языка Go при его создании ориентировались не на академическое применение, а на решение реальных проблем наиболее элегантным способом. Мы надеемся, что в этой книге нам удалось передать элегантность языка Go и его практичность.

Предметный указатель

Г

Д

Ж

З

И

К

Л

М

Н

О

П

P

С

Книги издательства «ДМК Пресс» можно заказать
в торгово-издательском холдинге «Планета Альянс» наложенным платежом,
выслав открытку или письмо по почтовому адресу:
115487, г. Москва, 2-й Нагатинский пр-д, д. 6А.
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: **www.alians-kniga.ru**.
Оптовые закупки: тел. **(499) 782-38-89**.
Электронный адрес: **books@alians-kniga.ru**.

Мэтт Батчер, Мэтт Фарина

Go на практике

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Научный редактор *Киселев А. Н.*
Перевод *Рагимов Р. Н.*
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Формат 60×90 1/16.
Гарнитура «Петербург». Печать офсетная.
Усл. печ. л. 35,0625. Тираж 200 экз.

Веб-сайт издательства: www.дмк.рф

Go на практике

Go– превосходный системный язык. Созданный для удобной разработки современных приложений с параллельной обработкой, Go предоставляет встроенный набор инструментов для быстрого создания облачных, системных и веб-приложений. Знакомые с такими языками, как Java или C#, быстро освоят Go – достаточно лишь немного попрактиковаться, чтобы научиться писать профессиональный код.
Книга содержит решения десятков типовых задач в ключевых областях. Следуя стилю сборника рецептов – проблема/решение/обсуждение – это практическое руководство опирается на основополагающие концепции языка Go и знакомит с конкретными приемами использования Go в облаке, тестирования и отладки, маршрутизации, а также создания веб-служб, сетевых и многих других приложений.

«Очень важные сведения, помогающие поднять разработку приложений на новый уровень.»

— Брайан Кэтелсен, автор предисловия, один из авторов книги «Go in Action»

«Бесценные сведения, которые можно сразу начать использовать для создания высокопроизводительных и практичных веб-приложений.»

— Гэри А. Стаффорд, ThoughtWorks

«Прекрасное сочетание простых примеров с подробным объяснением практических понятий языка Go.»

— Брэндон Титус, Mercury

Краткое содержание:

- десятки конкретных практических приемов программирования на Go;
- использование языка Go для создания обычных и облачных приложений;
- разработка веб-служб RESTful и микрослужб;
- практические приемы веб-разработки.

Мэтт Батчер (Matt Butcher) *– архитектор программного обеспечения компании Deis.*

Мэтт Фарина (Matt Farina) *– ведущий инженер группы передовых технологий в компании Hewlett Packard Enterprise. Оба являются техническими писателями, лекторами и активными участниками проектов с открытым исходным кодом.*

Издание адресовано опытным разработчикам, уже начавшим изучать язык Go и желающим научится эффективно использовать его в своей профессиональной деятельности.

ISBN 978-5-97060-477-9

Интернет-магазин: www.dmkpress.com
Книга – почтой: orders@alians-kniga.ru
Оптовая продажа: "Альянс-книга"
тел.(499)782-3889. books@alians-kniga.ru

ДМК ПРЕСС ИЗДАТЕЛЬСТВО
www.дмк.рф