

O'REILLY®

Go

идиомы и паттерны
проектирования



Джон Боднер

FIRST EDITION

Learning Go

*An Idiomatic Approach to
Real-World Go Programming*

Jon Bodner

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Go

идиомы и паттерны проектирования

Джон Боднер



Санкт-Петербург • Москва • Минск

2022

ББК 32.988.02-018.1
УДК 004.738.5
Б75

Боднер Джон

Б75 Go: идиомы и паттерны проектирования. — СПб.: Питер, 2022. — 416 с.: ил. — (Серия «Бестселлеры O'Reilly»).
ISBN 978-5-4461-1885-4

Go быстро набирает популярность в качестве языка для создания веб-сервисов. Существует множество учебников по синтаксису Go, но знать его недостаточно. Автор Джон Боднер описывает и объясняет паттерны проектирования, используемые опытными разработчиками. В книге собрана наиболее важная информация, необходимая для написания чистого и идиоматического Go-кода. Вы научитесь думать как Go-разработчик, вне зависимости от предыдущего опыта программирования.

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

ББК 32.988.02-018.1
УДК 004.738.5

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

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги.

Издательство не несет ответственности за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на интернет-ресурсы были действующими.

ISBN 978-1492077213 англ.

Authorized Russian translation of the English edition of Learning Go
ISBN 9781492077213 © 2021 Jon Bodner

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

ISBN 978-5-4461-1885-4

© Перевод на русский язык ООО «Прогресс книга», 2022

© Издание на русском языке, оформление ООО «Прогресс книга», 2022

© Серия «Бестселлеры O'Reilly», 2022

Оглавление

Предисловие	13
Для кого написана книга	14
Условные обозначения.....	14
Использование примеров программного кода.....	15
Благодарности.....	16
От издательства	17
Глава 1. Настройка среды разработки для языка Go	18
Установка средств разработки для языка Go.....	18
Рабочее пространство Go	19
Команда go.....	21
Команды go run и go build	21
Установка сторонних инструментов для Go.....	23
Форматирование кода	24
Выбор инструментов.....	26
Visual Studio Code.....	26
GoLand	28
Онлайн-песочница	29
make-файлы.....	31
Обновление средств разработки	33
Резюме	34
Глава 2. Простые типы и объявление переменных	35
Встроенные типы	35
Нулевое значение.....	35
Литералы.....	36
Логические значения	38
Числовые типы.....	38
Пробуем использовать строки и руны	46
Явное преобразование типов.....	46
var или :=	48
Использование ключевого слова const.....	51

Типизированные и нетипизированные константы	52
Неиспользуемые переменные	53
Именованное переменных и констант	54
Резюме	56

Глава 3. Составные типы.....57

Массивы — слишком строгие для того, чтобы использовать их напрямую.....	57
Срезы	59
Функция len	61
Функция append.....	61
Емкость среза	62
Функция make.....	64
Объявление собственного среза.....	65
Срезание срезов	67
Преобразование массивов в срезы	70
Функция сору.....	70
Строки в сочетании с рунами и байтами	72
Карты	76
Чтение и запись карты.....	79
Идиомы «запятая-ок».....	79
Удаление из карты	80
Использование карты в качестве множества.....	81
Структуры	82
Анонимные структуры	85
Сравнение и преобразование структур	86
Резюме	87

Глава 4. Блоки, затенение переменных и управляющие конструкции88

Блоки.....	88
Затенение переменных.....	89
Выявление затененных переменных.....	91
Оператор if	93
Четыре вида оператора for.....	94
Полный оператор for.....	95
Оператор for, использующий только условие	96
Бесконечный оператор for	96
Ключевые слова break и continue	97
Оператор for-range.....	99
Операторы for с метками.....	104
Выбор подходящего оператора for	105
Оператор switch.....	107
Пустые переключатели	110

Что лучше выбрать: if или switch?	111
Оператор goto	112
Резюме	115
Глава 5. Функции	116
Объявление и вызов функций	116
Имитация именованных и опциональных параметров	117
Вариативные входные параметры и срезы	118
Возврат нескольких значений	119
При возврате нескольких значений всегда возвращается несколько значений	120
Игнорирование возвращаемых значений	121
Именованные возвращаемые значения	122
Никогда не используйте пустые операторы возврата!	123
Функции являются значениями	125
Объявление функциональных типов	127
Анонимные функции	127
Замыкания	128
Передача функций в качестве параметров	129
Возвращение функций из функций	131
Оператор defer	132
Go — язык с передачей параметров по значению	136
Резюме	139
Глава 6. Указатели	140
Общие сведения об указателях	140
Не бойтесь указателей	144
Указатели служат для указания изменяемых параметров	146
Указатели — это крайняя мера	151
Влияние передачи указателей на производительность	152
Различие между нулевым значением и отсутствием значения	153
Различие между картами и срезами	154
Использование срезов в качестве буферов	157
Уменьшение нагрузки на сборщик мусора	159
Резюме	163
Глава 7. Типы, методы и интерфейсы	164
Типы в Go	164
Методы	165
Приемники указателей и приемники значений	166
Пишите код методов с расчетом на экземпляр, равный nil	168
Методы тоже являются функциями	170
Функции или методы?	171

Объявление типа не является наследованием.....	172
Типы являются исполняемой документацией.....	173
Йота (иногда) используется для создания перечислений.....	173
Используйте встраивание для реализации композиции.....	176
Встраивание не является наследованием.....	177
Общее представление об интерфейсах.....	179
Интерфейсы обеспечивают типобезопасную утиную типизацию.....	179
Встраивание и интерфейсы.....	184
Принимайте интерфейсы, возвращайте структуры.....	184
Интерфейсы и значение nil.....	186
Пустой интерфейс ничего не сообщает.....	187
Утверждения типа и переключатели типа.....	188
Используйте утверждения типа и переключатели типа как можно реже.....	191
Функциональные типы — ключ к интерфейсам.....	194
Неявные интерфейсы облегчают внедрение зависимостей.....	195
Утилита Wire.....	200
Go нельзя назвать объектно-ориентированным языком (и это здорово!).....	200
Резюме.....	200
Глава 8. Ошибки.....	201
Как обрабатывать ошибки: основы.....	201
Используйте строки в случае простых ошибок.....	203
Сигнальные ошибки.....	204
Ошибки являются значениями.....	206
Обертывание ошибок.....	209
Функции Is и As.....	212
Обертывание ошибок с помощью оператора defer.....	215
Функции panic и recover.....	216
Извлечение трассировки стека из ошибки.....	219
Резюме.....	219
Глава 9. Модули, пакеты и операции импорта.....	220
Репозитории, модули и пакеты.....	220
Файл go.mod.....	221
Компиляция пакетов.....	222
Операции импорта и экспорта.....	222
Создание и использование пакета.....	222
Именованное пакетов.....	225
Как следует подходить к организации кода модуля.....	226
Переопределение имени пакета.....	227
Комментарии пакета и godoc.....	228
Пакет internal.....	230

По возможности не используйте функцию <code>init</code>	231
Циклические зависимости	232
Переименование и реорганизация API без потери работоспособности	233
Работа с модулями	235
Импортирование стороннего кода	235
Работа с версиями	238
Выбор минимальной версии	241
Обновление до совместимых версий	241
Обновление до несовместимых версий	242
Вендоринг	244
Сайт <code>pkg.go.dev</code>	245
Дополнительная информация	245
Публикация своего модуля	245
Версионирование своего модуля	246
Прокси-серверы модулей	248
Указание прокси-сервера	248
Закрытые репозитории	249
Резюме	250
Глава 10. Конкурентность в Go	251
Когда следует использовать конкурентность	251
Горутины	253
Каналы	255
Чтение, запись и буферизация	255
Цикл <code>for-range</code> и каналы	257
Закрытие канала	257
Различия в поведении каналов	258
Оператор <code>select</code>	259
Принципы и паттерны конкурентного программирования	262
Следите за тем, чтобы конкурентности не было в ваших API	263
Горутины, циклы <code>for</code> и изменяющиеся переменные	263
Всегда закрывайте горутины	264
Паттерн на основе канала <code>done</code>	266
Прекращение выполнения горутин с помощью функции отмены	267
Когда следует использовать буферизованные и небуферизованные каналы	268
Противодавление	269
Отключение ветвей оператора <code>select</code>	270
Как можно установить тайм-аут для кода	271
Использование типа <code>WaitGroup</code>	272
Однократное выполнение кода	274

Собираем инструменты для обеспечения конкурентности	275
Когда вместо каналов следует использовать мьютексы.....	279
Атомарные операции — скорее всего, они вам не понадобятся	284
Где можно найти более подробную информацию о конкурентности	284
Резюме	285
Глава 11. Стандартная библиотека	286
Пакет io и его друзья	286
Пакет time	292
Монотонное время	294
Таймеры и тайм-ауты.....	295
Пакет encoding/json.....	295
Используйте теги структур для добавления метаданных.....	295
Демаршализация и маршализация	297
JSON, считыватели и записыватели.....	298
Кодирование и декодирование JSON-потокoв.....	299
Парсинг пользовательского формата JSON	301
Пакет net/http	303
Клиент	303
Сервер.....	304
Промежуточный слой	307
Резюме	310
Глава 12. Контекст.....	311
Что такое контекст	311
Отмена.....	314
Таймеры	318
Управление отменой контекста в собственном коде.....	321
Значения	322
Резюме	329
Глава 13. Написание тестов.....	330
Основы тестирования	330
Выдача сообщения о неудачном завершении теста	332
Подготовка и заключительная «уборка».....	333
Расположение образцов тестовых данных.....	335
Кэширование результатов теста	335
Тестирование своего публичного API	335
Используйте модуль go-стр для сравнения результатов тестов	337
Табличные тесты.....	339
Проверка степени покрытия кода	341
Сравнительные тесты.....	343

Заглушки в Go	348
Пакет httptest	354
Интеграционные тесты и теги сборки	356
Выявление проблем конкурентности с помощью детектора состояний гонки	358
Резюме	360
Глава 14. Здесь водятся драконы: пакеты reflect, unsafe и cgo	361
Рефлексия позволяет нам работать с типами на этапе выполнения	362
Типы, разновидности и значения	364
Создание новых значений	369
Используйте рефлексию для проверки значения интерфейса на равенство значению nil	370
Используйте рефлексию для создания маршализатора данных	371
Создавайте с помощью рефлексии функции для автоматизации повторяющихся задач	376
Рефлексию можно использовать для создания структур, но лучше этого не делать	378
Рефлексия не позволяет создавать методы	378
Используйте рефлексию только тогда, когда в этом есть смысл	378
Использовать пакет unsafe небезопасно	380
Используйте пакет unsafe для преобразования внешних двоичных данных	381
Пакет unsafe в сочетании со строками и срезами	385
Вспомогательные инструменты для пакета unsafe	386
Пакет cgo предназначен для обеспечения интеграции, а не повышения производительности	387
Резюме	391
Глава 15. Взгляд в будущее: обобщенные типы в Go	392
Обобщенные типы уменьшают количество повторяющегося кода и повышают типобезопасность	392
Добавление обобщенных типов в Go	395
Используйте списки типов для определения операторов	401
Обобщенные функции абстрагируют алгоритмы	402
Списки типов накладывают ограничения на константы и реализации	403
Что остается «за бортом»	407
Идиоматический Go-код и обобщенные типы	409
Какие нововведения нас ожидают	410
Резюме	411
Об авторе	412
Об обложке	413

Go — уникальный язык, и даже опытным программистам приходится «забыть» о том, как делаются некоторые вещи, и научиться иному подходу к программному обеспечению. Автор хорошо объясняет основные возможности языка, сопровождая теорию примерами идиоматического кода и описанием подводных камней и шаблонов проектирования.

Аарон Шлезингер (Aaron Schlesinger), ведущий разработчик, Microsoft

Джон на протяжении многих лет пользуется большим авторитетом в Go-сообществе; его высказывания и статьи часто несут в себе много полезного. Эта книга представляет собой руководство по изучению языка Go, рассчитанное на программистов. Она отлично сбалансирована в том плане, что хорошо объясняет необходимые вещи без пересказывания общеизвестных концепций.

Стив Франция (Steve Francia), создатель Hugo, Cobra и Viper, ведущий менеджер по Go-продуктам, Google

Боднер предлагает нам настоящий Go. В ясной, «живой» манере он учит этому языку от самых его основ до таких продвинутых тем, как рефлексия и взаимодействие с кодом, написанным на языке C. Многочисленные примеры показывают, как следует писать *идиоматический* Go-код, уделяя особое внимание ясности и простоте. Также подробно объясняется, как те или иные базовые концепции влияют на поведение программ, как, например, указатели влияют на распределение памяти и сборку мусора. Эта книга поможет новичкам очень быстро выйти на должный уровень, и даже опытные Go-программисты найдут в ней что-то полезное.

Джонатан Амстердам (Jonathan Amsterdam), сотрудник подразделения Go-разработки, Google

Книга содержит важную вводную информацию о том, что именно делает уникальным язык программирования Go, а также о паттернах проектирования и идиомах, которые делают его чрезвычайно мощным. Джону Боднеру удалось показать связь между базовыми возможностями Go и его философией, направляя читателей писать Go-код так, как это было задумано создателями языка.

Роберт Лебовиц (Robert Liebowitz), разработчик, Morning Consult

Джон написал не просто руководство по Go; издание дает идиоматически и практически ориентированное понимание этого языка. Книга написана на основе богатого опыта Джона в компьютерной отрасли и поможет тем, кто хочет очень быстро научиться эффективно пользоваться этим языком.

Уильям Кеннеди (William Kennedy), управляющий партнер, Ardan Labs

Предисловие

Изначально я хотел назвать эту книгу «Скучный Go», потому что Go — это и правда очень скучно.

Конечно, немного странно писать книгу на скучную тему, но я сейчас все объясню. Язык Go обладает небольшим набором возможностей, что идет вразрез с большинством других современных языков программирования. Хорошо написанная программа на Go, как правило, отличается простотой, а часто и некоторым однообразием. Здесь нет наследования, обобщенных типов (по крайней мере пока), аспектно-ориентированного программирования, перегрузки функций и уж точно нет перегрузки операторов. Здесь нет сравнения с паттерном, нет именованных параметров и исключений, зато есть пугающие многих *указатели*. Модель обмена данными в Go не похожа на ту, что применяется в других языках, и основана на идеях 1970-х годов, как и алгоритм, используемый для сборщика мусора. Другими словами, работа с Go выглядит откатом в прошлое.

Однако если Go скучен, это еще не значит, что он *примитивен*. Для правильного использования этого языка необходимо четко понимать, как его возможности сочетаются друг с другом. Хотя вы вполне можете написать на Go такой код, который будет выглядеть почти так же, как код на Java или Python, но результат вас не слишком обрадует, и вы так и не поймете, к чему была вся эта суета. Вот тут-то и пригодится эта книга. Она познакомит вас с возможностями Go и объяснит, как лучше их использовать для написания расширяемого идиоматического кода.

Когда речь идет о создании чего-то, что должно служить нам долгие годы, такое качество, как «обычность», становится не минусом, а плюсом. Никто не захочет первым прокатиться на своей машине по мосту, созданному с использованием непроверенных технологий, которые инженер посчитал «крутыми». Современный мир зависит от программного обеспечения так же, как и от мостов, если не больше. Тем не менее во многие языки программирования добавляют массу новых компонентов, не задумываясь об их влиянии на «ремонтпригодность» кодовой базы, в то время как Go рассчитан на создание надежных программ, которые смогут улучшать десятки программистов на протяжении десятилетий.

Так что да, Go скучный и это здорово! Я надеюсь, книга научит вас создавать с помощью такого «скучного» кода потрясающие проекты.

Для кого написана книга

Целевая аудитория — разработчики, желающие изучить второй (или пятый) язык программирования. Она подойдет как новичкам, которые знают лишь то, что у Go есть забавный талисман, так и тем, кто уже успел прочитать краткое руководство по этому языку или даже написать на нем несколько фрагментов кода. Эта книга не просто рассказывает, как следует писать программы на Go, она показывает, как можно делать это *идиоматически*. Более опытные Go-разработчики найдут здесь советы по использованию новых возможностей языка.

Предполагается, что читатель уже умеет пользоваться таким инструментом разработчика, как система контроля версий (желательно Git) и IDE. Читатель должен быть знаком с такими базовыми понятиями информатики, как конкурентность и абстракция, поскольку в книге мы будем касаться этих тем применительно к Go. Одни примеры кода можно скачать с GitHub, а десятки других — запустить в онлайн-песочнице *The Go Playground*. Наличие соединения с интернетом необязательно, однако с ним будет проще изучать исполняемые примеры кода. Поскольку язык Go часто используется для создания и вызова HTTP-серверов, для понимания некоторых примеров читатель должен иметь общее представление о протоколе HTTP и связанных с ним концепциях.

Несмотря на то что большинство функций Go встречаются и в других языках, программы, написанные на Go, обладают иной структурой. В книге мы начнем с настройки среды разработки для языка Go, после чего поговорим о переменных, типах, управляющих конструкциях и функциях. Если вам захочется пропустить этот материал, постарайтесь удержаться от этого и все-таки ознакомиться с ним. Зачастую именно такие детали делают Go-код идиоматическим. Простые на первый взгляд вещи могут оказаться очень интересными, если вы присмотритесь к ним получше.

Условные обозначения

В этой книге используются следующие типографские обозначения.

Курсивный шрифт

Используется для выделения новых терминов.

Шрифт без засечек

Применяется для выделения URL-адресов, адресов электронной почты.

Моноширинный шрифт

Используется для записи примеров программ, а также для выделения в тексте таких элементов, как имена переменных и функций, базы данных, типы данных, переменные среды, операторы и ключевые слова, имена и расширения файлов.

Полужирный моноширинный шрифт

Применяется для выделения команд и другого текста, который должен вводиться пользователем без каких-либо изменений.

Курсивный моноширинный шрифт

Используется для обозначения в коде элементов, вместо которых следует подставить предоставленные пользователем значения или значения, зависящие от контекста.



Так обозначается совет или предложение.



Так обозначается примечание общего характера.



Так обозначается предупреждение.

Использование примеров программного кода

Вспомогательные материалы (примеры программного кода, упражнения и т. д.) доступны для скачивания по адресу <https://github.com/learning-go-book>.

При возникновении технических вопросов или проблем, связанных с использованием примеров кода, пожалуйста, обращайтесь по адресу электронной почты bookquestions@oreilly.com.

Книга призвана помочь вам в решении ваших задач. В большинстве случаев вы можете свободно использовать приводимые в этой книге примеры кода в своих программах и документации. Вам не нужно обращаться в издательство

за разрешением, если вы не собираетесь воспроизводить существенную часть программного кода. Например, если вы разрабатываете программу и используете в ней несколько фрагментов кода из книги, вам не нужно обращаться за разрешением. Однако в случае продажи или распространения примеров из этой книги вы должны получить разрешение от издательства O'Reilly. Если вы отвечаете на вопросы, цитируя данную книгу или примеры из нее, то для этого не требуется разрешение. Но при включении значительного объема программного кода из этой книги в документацию вашего продукта необходимо получить разрешение издательства.

Мы не требуем, чтобы вы давали ссылку на первоисточник при цитировании, но будем признательны, если вы будете это делать. При этом обычно указывается название книги, имя ее автора, название издательства и номер ISBN. Например: «Go: идиомы и паттерны проектирования», Джон Боднер (Питер). Copyright 2021 Jon Bodner, 978-5-4461-1885-4».

Если у вас возникнут какие-либо сомнения относительно правомерности использования примеров кода, вы всегда можете связаться с нами по адресу permissions@oreilly.com.

Благодарности

Хоть и считается, что писательство — это индивидуальное занятие, книга не может появиться на свет без помощи множества других людей. Однажды я сказал Кармен Андо (Carmen Andoh), что собираюсь написать книгу по языку Go, и на конференции GopherCon 2019 она познакомила меня с Зан Маккуэйд (Zan McQuade) из компании O'Reilly. Зан помогла мне заключить договор с издательством, после чего регулярно консультировала меня по мере моего прогресса в написании книги. Мишель Кронин (Michele Cronin) редактировала мои тексты, высказывала замечания и выслушивала меня, когда я неизбежно сталкивался с трудностями. Технический редактор Тоня Трибула (Tonya Trybula) и литературный редактор Бет Келли (Beth Kelly) довели мою рукопись до пригодного для печати вида.

По мере написания книги я получал важные замечания (и слова поддержки) от многих людей, в числе которых были Джонатан Алтмэн (Jonathan Altman), Джонатан Амстердам (Jonathan Amsterdam), Джонни Рэй Остин (Johnny Ray Austin), Крис Фойербах (Chris Fauerbach), Крис Хайнс (Chris Hines), Билл Кеннеди (Bill Kennedy), Тони Нельсон (Tony Nelson), Фил Перл (Phil Pearl), Лиз Райс (Liz Rice), Аарон Шлезингер (Aaron Schlesinger), Крис Стайт (Chris Stout), Капил Тхангавелу (Kapil Thangavelu), Клер Тривисонно (Claire Trivisonno), Фолькер Уриг (Volker Uhrig), Джефф Уэндлинг (Jeff Wendling) и Крис Сара-

госа (Kris Zaragoza). Особых слов признательности заслуживает Роб Лебовиц (Rob Liebowitz), подробные замечания и быстрые ответы которого сделали эту книгу гораздо лучше.

Моей семье приходилось мириться с тем, что я проводил вечера и выходные дни за компьютером вместо того, чтобы проводить это время с ними. Моя жена Лора великодушно делала вид, что я не разбудил ее, когда я добирался до кровати в час ночи или еще позже.

Наконец, хочется вспомнить о тех людях, которые направили меня по этому пути четыре десятилетия назад. Первым из них был отец моего друга детства Пол Голдштейн (Paul Goldstein). В 1982 году он показал нам компьютер Commodore PET, ввел команду `PRINT 2 + 2` и нажал клавишу ввода. Я был поражен, когда на экране появилась цифра 4, и заболел этим раз и навсегда. Спустя какое-то время Пол научил меня программировать и даже на несколько недель одолжил свой компьютер. А вторым человеком была моя мама, которую я должен поблагодарить за то, что она поощряла мой интерес к программированию и компьютерам, даже толком не понимая, что это такое. В свое время она купила мне картридж для игровой приставки Atari 2600, позволявший писать программы на языке BASIC, компьютеры Commodore VIC-20 и Commodore 64, а также книги по программированию, после прочтения которых у меня появилась мечта когда-нибудь написать свою собственную книгу.

Спасибо всем вам за то, что помогли мне сделать эту мечту реальностью!

От издательства

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

Мы будем рады узнать ваше мнение!

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

Согласно комментарию автора на странице книги на сайте O'Reilly, инструмент `golint` не поддерживается. По этой причине мы удалили из главы 1 подраздел, посвященный `golint`.

ГЛАВА 1

Настройка среды разработки для языка Go

Любой язык программирования нуждается в среде разработки, и Go не исключение. Если вы уже успели написать на Go одну-две программы, то у вас уже, вероятно, имеется рабочая среда, но при этом вы могли упустить из виду некоторые новые методы и инструменты. Если же вы в первый раз настраиваете свой компьютер для разработки на Go, не беспокойтесь: установка средств поддержки этого языка не представляет большого труда. Настроив среду и убедившись, что это было сделано правильно, мы напишем простую программу и опробуем несколько способов компиляции и запуска Go-кода, после чего рассмотрим несколько инструментов и методов, упрощающих процесс Go-разработки.

Установка средств разработки для языка Go

Чтобы приступить к написанию кода на Go, нужно установить средства разработки. Последнюю версию этих инструментов можно скачать с официального сайта языка Go (<https://golang.org/dl>). Скачайте и установите версию, подходящую для вашей платформы. Установочные пакеты с расширением `.pkg` для Mac и с расширением `.msi` для Windows автоматически установят Go в нужном каталоге, удалят ранее установленные файлы и разместят двоичный файл языка Go в соответствии с путем по умолчанию для исполняемых файлов.



Для платформы Mac средства разработки для языка Go можно установить с помощью менеджера пакетов Homebrew (<https://brew.sh>), выполнив команду `brew install go`. Для Windows это можно сделать с помощью менеджера пакетов Chocolatey (<https://chocolatey.org>), выполнив команду `choco install golang`.

Версии установочных пакетов для Linux и FreeBSD представляют собой архивы с расширением `.tar`, которые распаковываются в каталог с именем `go`. Скопируйте этот каталог в `/usr/local` и добавьте путь `/usr/local/go/bin` в переменную среды `$PATH`, чтобы сделать доступной команду `go`:

```
$ tar -C /usr/local -xzf go1.15.2.linux-amd64.tar.gz
$ echo 'export PATH=$PATH:/usr/local/go/bin' >> $HOME/.profile
$ source $HOME/.profile
```



Go-программы компилируются в один двоичный файл и не требуют установки дополнительного программного обеспечения для их запуска. Устанавливать средства разработки для языка Go нужно только на тех компьютерах, на которых вы будете компилировать Go-программы.

Чтобы убедиться, что ваша среда настроена правильно, откройте окно терминала или командной строки и введите команду:

```
$ go version
```

Если все было настроено правильно, то вы увидите что-то наподобие следующего:

```
go version go1.15.2 darwin/amd64
```

Это сообщение говорит о наличии компилятора Go версии 1.15.2 для Mac OS. (Darwin — это название ядра в Mac OS, а amd64 — название 64-разрядной процессорной архитектуры от AMD и Intel.)

Если вместо сообщения с описанием версии вы увидите сообщение об ошибке, значит, у вас нет файла `go` в каталоге, заданном в качестве пути для исполняемых файлов, или в этом файле находится другая программа с таким же именем. В Mac OS или других Unix-подобных системах можно выяснить, какой файл `go` запускается и запускается ли вообще, с помощью команды `which go`. Если это не файл, расположенный по адресу `/usr/local/go/bin/go`, скорректируйте свой путь для исполняемых файлов.

Если вы работаете в Linux или FreeBSD, ошибка может состоять в том, что вы установили 64-разрядную версию средств разработки для языка Go в 32-разрядной системе или версию для другой процессорной архитектуры.

Рабочее пространство Go

С момента появления Go в 2009 году было введено несколько изменений в способе организации Go-кода и его зависимостей. Из-за этого вы можете

найти множество противоречивых рекомендаций, большая часть которых уже устарела.

Сегодня здесь действует простое правило: вы можете организовывать свои проекты любым удобным вам способом. Однако все же предполагается, что вы будете использовать единое рабочее пространство для сторонних инструментов Go, устанавливаемых с помощью команды `go install` (см. подраздел «Установка сторонних инструментов для Go» на с. 23). По умолчанию это рабочее пространство размещается в каталоге `$HOME/go`; при этом исходный код для этих инструментов хранится в каталоге `$HOME/go/src`, а скомпилированные двоичные файлы — в каталоге `$HOME/go/bin`. Вы можете использовать эти настройки по умолчанию или задать другое расположение рабочего пространства с помощью переменной среды `$GOPATH`.

Вне зависимости от того, какой вариант расположения вы решили использовать, рекомендуется явно определить переменную `GOPATH` и добавить каталог `$GOPATH/bin` в переменную, задающую путь для исполняемых файлов. Явное определение переменной `GOPATH` позволяет ясно указать, где находится рабочее пространство языка Go, а добавление каталога `$GOPATH/bin` в качестве дополнительного пути для исполняемых файлов упрощает запуск сторонних инструментов, которые устанавливаются с помощью команды `go install` и о которых мы поговорим чуть позже.

Если вы работаете с Unix-подобной системой, использующей командную оболочку `bash`, необходимо добавить следующие строки в файл `.profile`. (Если вы используете командную оболочку `zsh`, добавьте эти строки в файл `.zshrc`.)

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

Чтобы эти изменения вступили в силу, выполните команду `source $HOME/.profile` в текущем окне терминала.

Если вы работаете в Windows, выполните следующие команды в окне командной строки:

```
setx GOPATH %GOPATH%\go
setx path "%path%;%USERPROFILE%\bin"
```

Чтобы эти изменения вступили в силу, закройте текущее окно командной строки и откройте новое.

Инструмент `go` принимает во внимание и ряд других переменных среды. Вы можете получить полный список этих переменных вместе с кратким описанием каждой из них, выполнив команду `go env`. Многие из этих переменных управляют низкоуровневым поведением, которое можно спокойно проигнорировать,

однако мы все же рассмотрим некоторые из них при обсуждении модулей и перекрестной компиляции.



В некоторых онлайн-источниках рекомендуется определить переменную среды `GOROOT`. В ней задается расположение установленных средств разработки для языка Go. Теперь в этом уже нет необходимости: инструмент `go` определяет расположение автоматически.

Команда go

В комплект поставки языка Go входит достаточно много средств разработки, запускаемых с помощью команды `go`: компилятор, средства форматирования и статического анализа кода, менеджер зависимостей, средство тестирования и многое другое. Мы будем знакомиться с большинством из них по мере того, как станем учиться создавать высококачественный идиоматический Go-код. Начнем с инструментов, используемых для компиляции Go-кода, и попробуем скомпилировать с помощью команды `go` простое приложение.

Команды `go run` и `go build`

С помощью `go` мы можем выполнить две похожие команды: `go run` и `go build`. Обе принимают на вход отдельный файл Go, список Go-файлов или имя пакета. Создадим простую программу и посмотрим, что произойдет, когда мы воспользуемся этими командами.

Команда `go run`

Начнем с команды `go run`. Создайте каталог с именем `ch1`, после чего откройте текстовый редактор, введите следующий текст и сохраните его в файле с именем `hello.go` внутри каталога `ch1`:

```
package main

import "fmt"

func main() {
    fmt.Println("Hello, world!")
}
```

После этого откройте окно терминала или командной строки и выполните в нем следующую команду:

```
go run hello.go
```

На экран будет выведена фраза `Hello, world!`. Если вы проверите содержимое каталога после выполнения команды `go run`, то увидите, что в нем не появился двоичный файл и он по-прежнему содержит только созданный нами файл `hello.go`. Здесь вы можете спросить: почему так происходит, ведь Go вроде бы компилируемый язык?

На самом деле команда `go run`, как и положено, компилирует ваш код в двоичный файл, но этот файл сохраняется во временном каталоге. Эта команда компилирует двоичный файл, выполняет его из временного каталога и удаляет его после завершения работы программы. В силу этого команду `go run` удобно использовать для тестирования небольших программ или при использовании языка Go в качестве языка сценариев.



Вводите команду `go run`, когда требуется использовать Go-программу в качестве скрипта или выполнить быстрый запуск исходного кода.

Команда `go build`

В большинстве случаев вам потребуется скомпилировать двоичный файл для последующего использования. Здесь вам поможет команда `go build`. Введите в окне терминала такую команду:

```
go build hello.go
```

Она создаст в текущем каталоге исполняемый файл с именем `hello` (или `hello.exe` в Windows). Если вы запустите его, то на экран, как и следовало ожидать, будет выведена фраза `Hello, world!`.

Имя этого двоичного файла совпадает с тем именем файла или пакета, которое вы передали команде. Если вам нужно сохранить приложение под другим именем или в другом каталоге, используйте флаг `-o`. Например, чтобы скомпилировать код в двоичный файл с именем `hello_world`, нужно ввести следующую команду:

```
go build -o hello_world hello.go
```



Используйте команду `go build` для создания распространяемого файла, который будет использоваться другими пользователями. Именно это требуется сделать в большинстве случаев. Используйте флаг `-o`, чтобы задать другое имя или расположение двоичного файла.

Установка сторонних инструментов для Go

Хотя некоторые Go-разработчики и предпочитают распространять свои программы в виде скомпилированных двоичных файлов, написанные на Go инструменты также можно скомпилировать из исходного кода и установить в свое рабочее пространство Go с помощью команды `go install`.

В Go применяется не такой, как в большинстве других языков, метод публикации кода. Вместо того чтобы использовать такой централизованный сервис, как Maven Central в случае Java или реестр NPM в случае JavaScript, Go-разработчики распространяют свои проекты посредством репозитория исходного кода. В качестве аргумента команде `go install` передается расположение репозитория исходного кода интересующего вас проекта, за которым следует символ `@` и нужная версия этого инструмента (если нужно установить последнюю версию, укажите `@latest`). Когда вы запустите эту команду, она скачает, скомпилирует и установит указанный инструмент в каталог `$GOPATH/bin`.

Рассмотрим небольшой пример, используя такой отличный Go-инструмент, как нагрузочное тестирование HTTP-серверов под названием `hey`. Эта программа позволяет протестировать любой выбранный вами сайт или написанное вами приложение. Установить ее можно следующим образом:

```
$ go install github.com/rakyll/hey@latest
go: downloading github.com/rakyll/hey v0.1.4
go: downloading golang.org/x/net v0.0.0-20181017193950-04a2e542c03f
go: downloading golang.org/x/text v0.3.0
```

Команда скачивает программу `hey` и все ее зависимости, компилирует ее и устанавливает двоичный файл в каталоге `$GOPATH/bin`.



Как будет подробно рассказано в разделе «Прокси-серверы модулей» на с. 248, содержимое Go-репозитория кэшируется на прокси-серверах. В зависимости от конкретного репозитория и значений, заданных в переменной среды `GOPROXY`, команда `go install` может производить скачивание либо с прокси-сервера, либо непосредственно из репозитория. При скачивании непосредственно из репозитория эта команда использует установленные на вашей машине инструменты командной строки. Так, например, для скачивания из репозитория, расположенного на сайте GitHub, у вас должна быть установлена система контроля версий Git.

Теперь, когда мы уже скомпилировали и установили программу `hey`, можем запустить ее с помощью следующей команды:

```
$ hey https://www.golang.org
```

Summary:

```
Total:      0.6864 secs
Slowest:    0.3148 secs
Fastest:    0.0696 secs
Average:    0.1198 secs
Requests/sec: 291.3862
```

Если определенный инструмент уже установлен на вашей машине и вам нужно обновить его до последней версии, запустите команду `go install` еще раз, указав в параметре новую версию или слово `latest`:

```
go install github.com/rakyll/hey@latest
```

Разумеется, написанные на Go инструменты совсем не обязательно оставлять в рабочем пространстве Go: это обычные исполняемые двоичные файлы, которые можно разместить в любом каталоге. Точно так же вам не обязательно распространять написанные на Go программы посредством команды `go install`: вместо этого можно предложить для скачивания двоичный файл. Однако эту команду очень удобно использовать для распространения Go-программ среди разработчиков.

Форматирование кода

Создатели языка Go прежде всего хотели создать язык, который позволял бы писать код эффективно. Это означало, что они должны были использовать простой синтаксис и быстрый компилятор. Кроме того, это вынудило создателей языка Go пересмотреть подход к форматированию кода. В то время как другие языки предоставляют вам большую свободу в плане способа организации кода, Go не делает этого. Он обязывает использовать стандартный формат, что существенно облегчает написание инструментов для работы над исходным кодом. Это упрощает компилятор и делает возможным создание продвинутых генераторов кода.

У этого подхода есть и еще одно преимущество. Разработчики раньше тратили очень много времени на «войну форматов». Благодаря тому, что Go определяет стандартный способ форматирования кода, Go-разработчикам не приходится спорить относительно того, какой стиль размещения фигурных скобок лучше использовать или как лучше задавать отступы: с помощью символов табуляции или с помощью пробелов. Так, например, для задания отступов в Go-коде используются символы табуляции, и, если открывающая фигурная скобка не находится в той же строке, что и начинающие этот блок команда или объявление, это считается ошибкой синтаксиса.



Среди Go-разработчиков бытует мнение, что создатели языка Go решили использовать стандартный формат для того, чтобы исключить споры в отношении формата, и уже после этого обнаружили преимущества данного подхода в плане создания инструментов. Однако Расс Кокс (Russ Cox) публично заявил о том, что его исходным мотивом было упрощение процесса создания инструментов (<https://oreil.ly/rZEUv>).

ПРАВИЛО ВСТАВКИ ТОЧКИ С ЗАПЯТОЙ

Команда `go fmt` не исправляет ошибочное размещение фигурной скобки не в той строке, что объясняется наличием *правила вставки точки с запятой*. Подобно языкам C и Java, Go требует, чтобы каждый оператор заканчивался символом «точка с запятой». Однако Go-разработчики никогда не расставляют символы «точка с запятой» вручную, поскольку компилятор языка Go делает это автоматически в соответствии с очень простым правилом, суть которого изложена в кратком руководстве «Эффективный Go» (<https://oreil.ly/hTONU>):

Если символу новой строки предшествует одна из следующих лексем, то лексический анализатор вставляет после нее символ «точка с запятой»:

- идентификатор (включая такие слова, как `int` и `float64`);
- один из базовых литералов, таких как число или строковая константа;
- одна из следующих лексем: `break`, `continue`, `fallthrough`, `return`, `++`, `--`, `)`, `}`.

Зная, что в Go действует это простое правило, легко понять, почему невозможно исправить размещение в неправильном месте фигурной скобки. Если, например, вы напишете следующий код:

```
func main()
{
    fmt.Println("Hello, world!")
}
```

то благодаря правилу вставки точки с запятой будет распознан символ `)` в конце строки `func main()`, а код приведен к следующему виду:

```
func main();
{
    fmt.Println("Hello, world!");
};
```

что не будет корректным Go-кодом.

Правило вставки точки с запятой — одна из тех вещей, которые делают компилятор языка Go проще и быстрее, в то же время обеспечивая единый стиль программирования, что весьма и весьма разумно.

В набор средств разработки языка Go входит команда `go fmt`, которая автоматически форматирует ваш код в соответствии со стандартным форматом. Она позволяет исправить неверно заданные отступы, выравнивает поля в структурах и обеспечивает правильные интервалы между операторами.

Существует расширенная версия команды `go fmt` под названием `goimports`, которая также приводит в порядок операторы импорта. Она располагает их в алфавитном порядке, удаляет неиспользуемые операторы и добавляет операторы импорта недостающих, по ее мнению, пакетов. Поскольку она не всегда точно угадывает, каких пакетов вам не хватает, добавлять операторы импорта лучше вручную.

Скачать `goimports` можно с помощью команды `go install golang.org/x/tools/cmd/goimports@latest`. Чтобы применить ее к файлам своего проекта, выполните следующую команду:

```
goimports -l -w .
```

Флаг `-l` дает `goimports` указание вывести в консоль список неправильно отформатированных файлов. Флаг `-w` дает указание непосредственно изменить эти файлы. Символ «точка» указывает, что нужно проверить все файлы, находящиеся в текущем каталоге и во всех подкаталогах.



Всегда используйте `go fmt` или программу `goimports` перед тем, как компилировать свой код!

Выбор инструментов

Хотя, как мы уже успели убедиться, написать небольшую программу на Go можно, используя только текстовый редактор и команду `go`, для работы над более крупными проектами вам, вероятно, потребуются более продвинутые инструменты. К счастью, к настоящему времени уже созданы отличные средства поддержки языка Go для большинства существующих текстовых редакторов и интегрированных сред разработки (<https://oreil.ly/MwCWT>). Если вы еще не определились с выбором среды разработки, то двумя наиболее популярными вариантами в случае языка Go являются редактор кода Visual Studio Code и интегрированная среда разработки Goland.

Visual Studio Code

Если вы хотите найти бесплатную среду разработки, то лучшим выбором будет редактор кода Visual Studio Code от компании Microsoft (<https://oreil.ly/zktT8>). За время, прошедшее с момента его появления в 2015 году, VS Code успел при-

обрести чрезвычайно большую популярность среди разработчиков. Хотя поддержка языка Go не входит в его «комплект поставки», его можно превратить в среду разработки для языка Go, скачав расширение для поддержки этого языка из галереи расширений.

Поддержка языка Go в редакторе VS Code обеспечивается с помощью сторонних инструментов. Это включает в себя комплект средств разработки языка Go, отладчик Delve (<https://oreil.ly/sosLu>) и gopls (<https://oreil.ly/TLapT>) — языковой сервер для языка Go, созданный командой разработчиков этого языка. При этом комплект средств разработки языка Go вы должны установить самостоятельно, а Delve и gopls для вас установит Go-расширение редактора.

Установив и настроив этот инструмент, можете открыть свой проект и приступить к работе над ним. Окно вашего проекта должно выглядеть примерно так, как показано на рис. 1.1. Основы работы с Go-расширением редактора VS Code демонстрируются во вводном видео «Приступая к работе с VS Code Go» (<https://oreil.ly/XhoeB>).

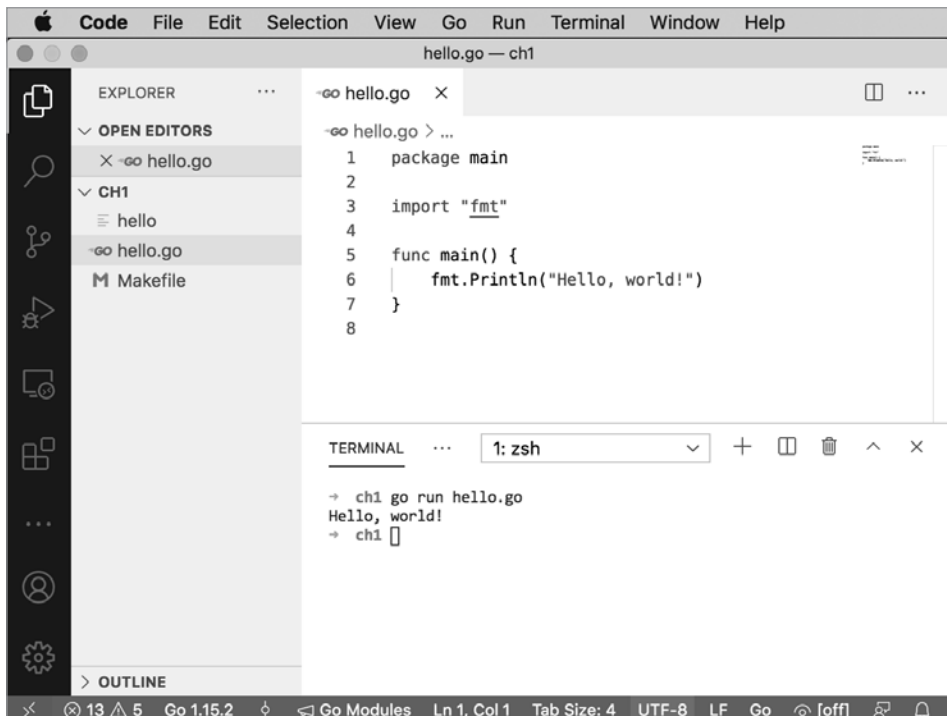


Рис. 1.1. Visual Studio Code



Что такое языковой сервер? Это стандартная спецификация API, позволяющего редакторам реализовать такие интеллектуальные функции редактирования, как автодополнение и статический анализ кода, поиск мест использования методов и т. д. Для большей информации ознакомьтесь с протоколом языкового сервера (<https://oreil.ly/2T2fw>).

GoLand

GoLand (<https://oreil.ly/6cXjL>) — это ориентированная на язык Go интегрированная среда разработки (IDE) от компании JetBrains. Хотя компания JetBrains в первую очередь славится своими инструментами для разработки на Java, это не мешает GoLand быть прекрасной средой разработки для языка Go. Как можно убедиться, взглянув на рис. 1.2, пользовательский интерфейс среды разработки GoLand выглядит практически так же, как интерфейс сред разработки IntelliJ, PyCharm, RubyMine, WebStorm и Android Studio или любой другой IDE от компании JetBrains. Поддержка Go в GoLand включает в себя такие вещи, как рефакторинг, выделение синтаксиса, автодополнение и навигация по коду, всплывающие подсказки с описанием типов и функций, отладчик, отслеживание покрытия кода и многое другое. Помимо поддержки языка Go, среда разработки

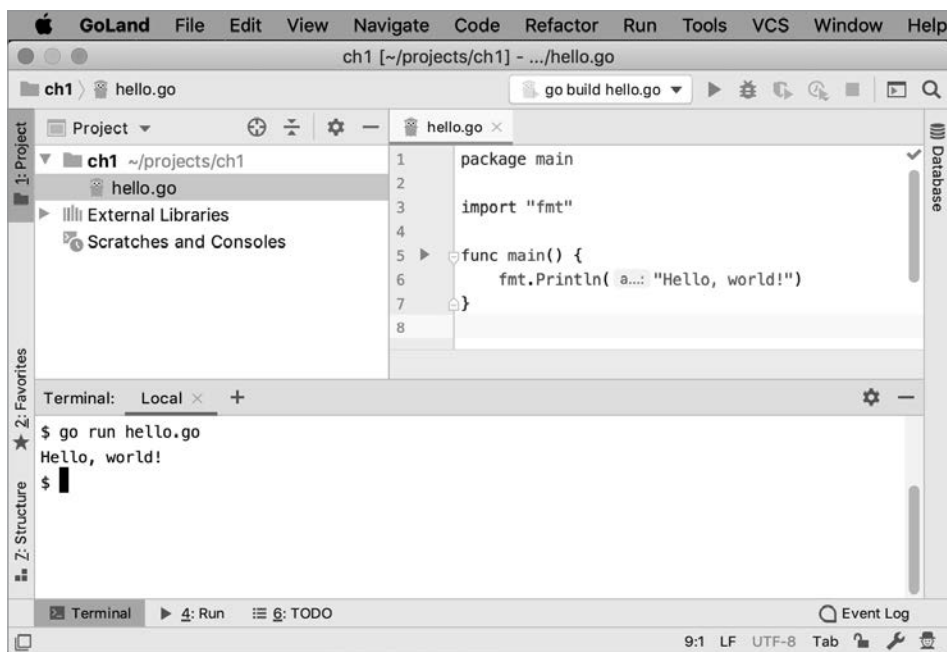


Рис. 1.2. GoLand

GoLand также предлагает инструменты для работы с JavaScript/HTML/CSS и базами данных SQL. В отличие от редактора кода VS Code, для использования среды разработки GoLand вам не потребуется скачивать никаких дополнительных инструментов.

Если у вас уже есть подписка на IntelliJ IDEA Ultimate (или право на бесплатное использование этого продукта), можете добавить поддержку Go в него, установив соответствующий плагин. В противном случае придется заплатить за среду разработки GoLand, у которой нет бесплатной версии.

Онлайн-песочница

Существует еще один важный инструмент для разработки на языке Go, который к тому же не требует установки. Перейдя по адресу <http://play.golang.org>, вы попадете на страницу онлайн-песочницы для языка Go, внешний вид которой показан на рис. 1.3. Если вам приходилось использовать такие инструменты командной строки, как `irb`, `node` или `python`, то вы увидите, что работа с этой онлайн-песочницей осуществляется во многом так же. Она позволяет вам запускать и показывать другим пользователям небольшие программы. Введите свою программу в поле ввода и нажмите кнопку Run (Выполнить) для ее выполнения. Нажатие кнопки Format (Форматировать) запустит для вашей программы

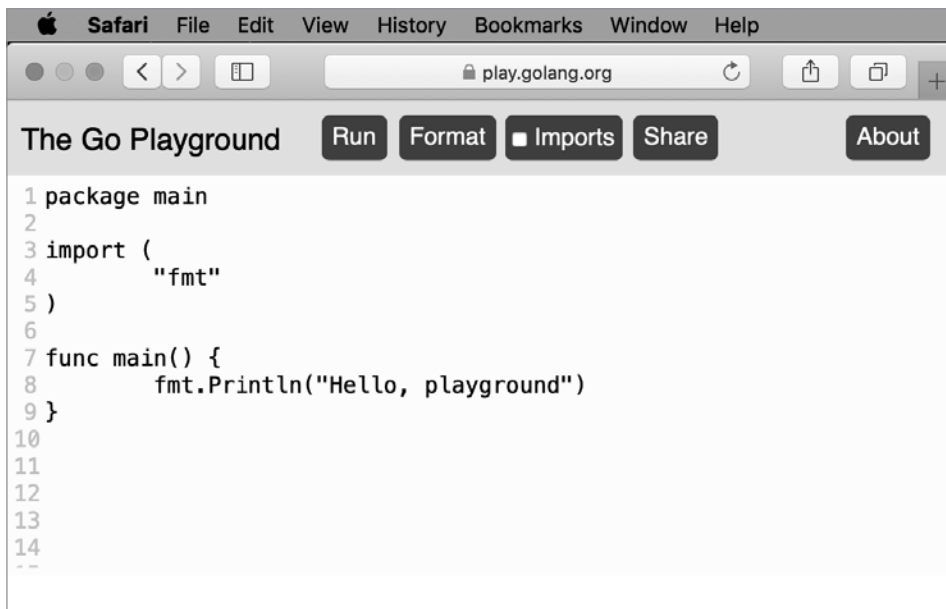


Рис. 1.3. Онлайн-песочница

команду `go fmt` и, если при этом будет установлен флажок **Imports** (Импорты), также приведет в порядок операторы импорта, как это делает утилита `goimports`. Кнопка **Share** (Поделиться) позволяет создать уникальный URL-адрес для того, чтобы предоставить ссылку на программу другим пользователям или вернуться к работе над ней в дальнейшем (хотя эти ссылки и сохраняются в течение долгого времени, я все же не стал бы использовать онлайн-песочницу в качестве репозитория исходного кода).

Как показывает рис. 1.4, вы даже можете симитировать работу с несколькими файлами, отделив каждый файл с помощью строки вида `-- filename.go --`.

Используя онлайн-песочницу, не забывайте о том, что она работает на другой машине (а если быть точнее, на машине компании Google), что ограничивает вашу свободу действий. При этом всегда используется последняя стабильная версия языка Go. Вы не можете устанавливать сетевые соединения, и, кроме того, ваш процесс может быть остановлен, если он будет работать слишком долго или задействовать слишком много памяти. Если в вашей программе используется время, установите в качестве начала отсчета 10 ноября 2009 года, 23:00:00 UTC (это дата и время первой официальной презентации языка Go). Однако даже эти ограничения не мешают онлайн-песочнице быть очень полезным инструментом,

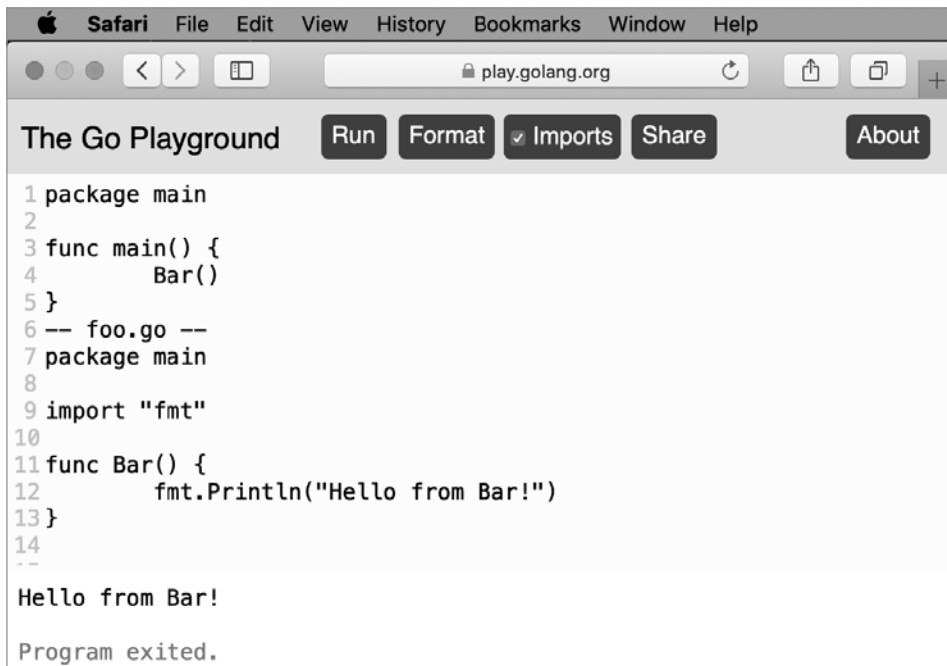


Рис. 1.4. Онлайн-песочница поддерживает использование нескольких файлов

который позволяет опробовать новые идеи без создания нового проекта на своей машине. В этой книге я буду часто использовать ссылки на онлайн-песочницу, чтобы вы могли запускать примеры кода, не копируя их на свой компьютер.



Никогда не размещайте в онлайн-песочнице такую конфиденциальную информацию, как сведения о вашей личности, пароли и секретные ключи! Если вы нажмете кнопку Share (Поделиться), то эта информация будет сохранена на серверах компании Google и станет доступной для всех пользователей, получивших соответствующую ссылку. Если вы по ошибке предоставили доступ к такой информации, свяжитесь с компанией Google по адресу security@golang.org и сообщите, какой URL-адрес вам нужно удалить и по какой причине это нужно сделать.

make-файлы

IDE удобно использовать, но трудно автоматизировать. Сегодня процесс программной разработки подразумевает использование воспроизводимых и автоматизируемых операций компиляции, которые может выполнять кто угодно, где угодно и когда угодно, что позволяет не обращать внимания на распространенную отговорку разработчиков: «На моей машине все работает!» Реализовать такой подход можно с помощью скрипта, в котором будут определены этапы процесса компиляции. Go-разработчики для этой цели используют утилиту `make`. Если вы не знакомы с этой утилитой, напомним, что она с 1976 года применяется для компиляции программ в операционных системах Unix.

Например, в случае нашего простого проекта можно использовать следующий make-файл:

```
.DEFAULT_GOAL := build

fmt:
    go fmt ./...
.PHONY:fmt

lint: fmt
    golint ./...
.PHONY:lint

vet: fmt
    go vet ./...
.PHONY:vet

build: vet
    go build hello.go
.PHONY:build
```

Даже если вам не приходилось использовать make-файлы раньше, будет совсем не трудно разобраться в том, что здесь происходит. Каждая из выполняемых операций называется *целью*. Директива цели по умолчанию `.DEFAULT_GOAL` указывает, какая цель должна выполняться в том случае, если не будет указано ни одной цели. В данном случае в качестве цели по умолчанию задана операция `build`. Далее следуют определения целей. В каждом из них сначала указывается имя цели, а за ним, после знака двоеточия (:), — имена целей, запущенных перед выполнением данной цели (как, например, `vet` в определении `build: vet`). Задачи, которые выполняются целью, находятся в строках с отступом после цели. (Директива фиктивной цели `.PHONY` не дает утилите `make` запутаться в том случае, если в вашем проекте будет создан каталог с таким же именем, как у цели.)

Разместив этот make-файл в каталоге `ch1`, выполните следующую команду:

```
make
```

На экран будет выведено следующее:

```
go fmt ./...
go vet ./...
go build hello.go
```

Таким образом, с помощью всего одной команды вы обеспечите правильное форматирование кода, проверите его на наличие неочевидных программных ошибок и скомпилируете его. Вы также можете запустить только линтер с помощью команды `make lint`, только утилиту `vet` с помощью команды `make vet` или только средство форматирования с помощью команды `make fmt`. Возможно, это и нельзя назвать большим улучшением, однако гарантированный запуск средств форматирования и статического анализа перед тем, как разработчик (или скрипт, запущенный сервером непрерывной интеграции) запустит операцию компиляции, означает, что вы никогда не пропустите ни одного шага.

Одним из недостатков make-файлов является то, что они требуют определенной внимательности: каждую из указанных для цели задач нужно *обязательно* снабдить отступом с помощью символа табуляции. Поддержка этих файлов также не входит в число стандартных возможностей операционной системы Windows. Если вы собираетесь писать Go-код на машине с Windows, вам сначала потребуется установить утилиту `make`. Самый простой способ это сделать сводится к тому, чтобы сначала установить менеджер пакетов наподобие Chocolatey (<https://chocolatey.org>), а затем установить утилиту `make` с его помощью (в случае Chocolatey это можно сделать с помощью команды `choco install make`).

Обновление средств разработки

Как и в случае любого другого языка программирования, средства разработки языка Go регулярно подвергаются изменениям. Поскольку Go-программы представляют собой файлы нативного двоичного кода, для запуска которых не требуется отдельная среда выполнения, вы можете не бояться того, что после обновления ваших средств разработки перестанут работать ранее развернутые программы. Вы можете совершенно спокойно запускать на одном компьютере или в одной виртуальной машине программы, скомпилированные с использованием разных версий языка Go.

Начиная с версии Go 1.2, крупные релизы выходят с интервалом примерно шесть месяцев. По мере необходимости также выпускаются небольшие релизы с исправлениями программных ошибок и проблем безопасности. В силу того, что команда разработчиков языка Go применяет быстрые циклы разработки и старается обеспечивать обратную совместимость, релизы языка Go обычно носят инкрементный характер и вносят не слишком много изменений. В статье «Обязательства по совместимости языка Go» (https://oreil.ly/p_NMY) подробно объясняется, каким образом команда разработчиков языка Go планирует не допускать изменений, способных нарушить работоспособность имеющегося Go-кода. Там говорится, что команда разработчиков не будет вносить в язык или стандартную библиотеку изменения, ломающие обратную совместимость, в случае любой версии языка Go, начинающейся с 1, за исключением изменений, необходимых для исправления программных ошибок или проблем безопасности. В то же время команда разработчиков может вносить (и уже вносила) несовместимые изменения во флаги или функциональность команд `go`.

Несмотря на гарантии обратной совместимости, при внесении изменений не исключены программные ошибки, поэтому, вполне естественно, у вас возникнет желание убедиться, что новый релиз не нарушает работоспособность ваших программ. Один из способов это сделать состоит в том, чтобы установить вторую рабочую среду для языка Go. Например, если сейчас вы используете версию 1.5.2 и хотели бы опробовать версию 1.15.6, можно выполнить следующие команды:

```
$ go get golang.org/dl/go.1.15.6
$ go1.15.6 download
```

После этого вместо команды `go` можно будет применить команду `go1.15.6`, чтобы посмотреть, будут ли работать ваши программы при использовании версии 1.15.6:

```
$ go1.15.6 build
```

Убедившись в том, что ваш код будет работать, вы можете удалить вторую рабочую среду. Для этого нужно найти и удалить соответствующую переменную `GOROOT`, а затем удалить двоичный файл этой среды из каталога `$GOPATH/bin`. В Mac OS, Linux и BSD это можно сделать следующим образом:

```
$ go1.15.6 env GOROOT
/Users/gobook/sdk/go1.15.6
$ rm -rf $(go1.15.6 env GOROOT)
$ rm $(go env GOPATH)/bin/go1.15.6
```

Когда вы будете готовы к тому, чтобы обновить установленные на вашей машине средства разработки языка Go, проще всего это будет сделать на платформах Mac и Windows. Если вы производили установку с помощью менеджера пакетов `brew` или `chocolatey`, то с его помощью можно выполнить и обновление. Если же вы скачивали установочный пакет со страницы <https://golang.org/dl>, воспользуйтесь последней версией этого пакета, которая в ходе установки удалит с машины старую версию.

На машинах с Linux и BSD необходимо скачать последнюю версию, переместить старую версию в резервный каталог, распаковать новую версию и удалить старую версию.

```
$ mv /usr/local/go /usr/local/old-go
$ tar -C /usr/local -xzf go1.15.2.linux-amd64.tar.gz
$ rm -rf /usr/local/old-go
```

Резюме

В этой главе вы узнали, как установить и настроить среду разработки для языка Go. Мы рассмотрели инструменты для компиляции Go-программ и обеспечения хорошего уровня качества кода. Теперь, когда у нас есть готовая среда, можем перейти к следующей главе, где рассмотрим встроенные типы языка Go и способы объявления переменных.

Простые типы и объявление переменных

Теперь, когда мы уже настроили среду разработки, пришло время перейти к изучению возможностей языка Go и наилучших способов их использования. Под наилучшими здесь имеется в виду то, что код будет удовлетворять главному принципу: программа должна быть написана так, чтобы сразу было понятно, для чего она предназначена. По мере изучения вы познакомитесь с различными подходами и поймете, почему тот или иной подход позволяет получить более ясный код.

Начнем с рассмотрения простых типов и переменных. Хотя эти понятия и знакомы каждому программисту, в Go некоторые вещи делаются по-другому, что отличает его от других языков.

Встроенные типы

В языке Go есть много тех же встроенных типов, что и в других языках: логические значения, целые числа, числа с плавающей точкой и строки. Идиоматическое использование этих типов иногда вызывает затруднения у разработчиков, ранее писавших на других языках. Вы познакомитесь с этими типами и узнаете, как они применяются в Go. Но перед этим остановимся на ряде концепций, которые применимы ко всем типам.

Нулевое значение

Go, как и большинство других современных языков, по умолчанию присваивает *нулевое значение* любой переменной, которая объявлена, но не имеет значения. Наличие четко заданного нулевого значения делает код более ясным и устраняет

источник программных ошибок, часто встречающихся в программах на С и С++. При этом у каждого типа есть свое нулевое значение.

Литералы

Под *литералом* в Go понимается запись числа, символа или строки. Существует четыре вида литералов (но есть и очень редкий пятый литерал, о котором мы поговорим при обсуждении комплексных чисел).

Целочисленные литералы — это последовательности цифр. Обычно это числа с основанием 10, но с помощью специального префикса можно задать и другое основание: префикс `0b` означает двоичную систему счисления (с основанием 2), префикс `0o` — восьмеричную (с основанием 8) и префикс `0x` — шестнадцатеричную (с основанием 16). Префикс может быть записан как в верхнем, так и в нижнем регистре. Для обозначения восьмеричного литерала также можно использовать цифру 0 без какой-либо буквы после нее, но лучше никогда этого не делать, чтобы не было путаницы.

Go позволяет разбивать литерал на несколько частей с помощью символов подчеркивания, чтобы упростить чтение длинных целочисленных литералов. Это позволяет, например, отделять разряд тысяч в числах с основанием 10 (`1_234`). Эти символы подчеркивания не влияют на значение числа. Единственное ограничение состоит в том, что их нельзя размещать в начале или в конце числа и рядом друг с другом. Вы можете отделить друг от друга все разряды числа (`1_2_3_4`), но лучше не поступайте таким образом. Используйте символы подчеркивания для улучшения восприятия, например отделяя разряд тысяч в десятичных числах или разбивая двоичные, восьмеричные и шестнадцатеричные числа на группы из одного, двух или четырех байт.

Литералы чисел с плавающей запятой содержат десятичную запятую, отделяющую дробную часть значения. Они также могут содержать показатель степени, обозначаемый буквой `e`, за которой следует положительное или отрицательное число (например, `6.03e23`). Их можно записывать в шестнадцатеричном виде, используя префикс `0x` и букву `p` для обозначения показателя степени. Как и целочисленные литералы, литералы чисел с плавающей точкой можно форматировать с помощью символов подчеркивания.

Литералы рун представляют собой символы и заключаются в одинарные кавычки. В отличие от многих других языков в Go одинарные и двойные кавычки не являются взаимозаменяемыми. Литералы рун можно записывать в виде одиночных символов стандарта Unicode (`'a'`), восьмиразрядных восьмеричных чисел (`'\141'`), восьмиразрядных шестнадцатеричных чисел (`'\x61'`), 16-разрядных шестнадцатеричных чисел (`'\u0061'`) и 32-разрядных кодов стандарта Unicode

(`'\u00000061'`). Существует также ряд рунных литералов, экранированных символом обратной косой черты, из которых чаще всего используются символ новой строки (`'\n'`), символ табуляции (`'\t'`), одинарная кавычка (`'\''`), двойная кавычка (`'\"'`) и обратная косая черта (`'\\'`).

На практике рекомендуется использовать десятичную систему счисления для представления числовых литералов и, если это не объясняется контекстом, по возможности не применять шестнадцатеричные экранированные рунные литералы. В редких случаях может использоваться восьмеричное представление, главным образом для представления значений флагов разрешения стандарта POSIX (например, восьмеричное значение `0o777` для флагов `gwxgwxgwx`). Шестнадцатеричный и двоичный форматы представления иногда используются в битовых фильтрах или сетевых и инфраструктурных приложениях.

Существует два способа обозначения *строковых литералов*. В большинстве случаев следует использовать *интерпретируемый строковый литерал*, который создается с помощью двойных кавычек (например, `"Greetings and Salutations"`). Такой литерал может содержать несколько рунных литералов в любом из допустимых форматов или не содержать их вовсе. Единственные символы, которые не могут здесь появиться, — символы обратной косой черты, новой строки и двойных кавычек. Если, допустим, нам нужно, чтобы второе слово фразы выводилось на новой строке и было заключено в кавычки, можно написать так: `"Greetings and\n\"Salutations\""`.

Если же необходимо включить в строку символы обратной косой черты, двойных кавычек или новой строки, используйте *необработанный строковый литерал*. Такие литералы заключаются в символы обратной одинарной кавычки (```) и могут содержать любые символы, за исключением символа обратной кавычки. Используя необработанный строковый литерал, мы можем записать нашу фразу в двух строках следующим образом:

```
`Greetings and
"Salutations"``
```

Как вы увидите в подразделе «Явное преобразование типов» на с. 46, Go даже не позволяет складывать значения двух целочисленных переменных, если они объявлены как целые числа разной размерности. В то же время можно использовать целочисленный литерал в выражениях с плавающей запятой и даже присваивать целочисленный литерал самой переменной с плавающей запятой. Это объясняется тем, что литералы в Go представляют собой нетипизированные значения и потому могут взаимодействовать с любой переменной совместимого с литералами типа. В главе 7 вы увидите, что литералы можно использовать даже совместно с типами, определяемыми пользователем на основе простых типов. Однако это все, что позволяет сделать нетипизированный характер литералов:

невозможно присвоить строковый литерал переменной числового типа и, наоборот, числовой литерал — строковой переменной либо присвоить литерал числа с плавающей точкой целочисленной переменной. Все подобные действия компилятор посчитает ошибкой.

Литералы являются нетипизированными по той причине, что Go ориентирован на практику. Нет смысла относить литерал к какому-либо типу, если разработчик еще не указал этот тип. При этом существуют ограничения в плане размера. Вы, конечно, можете записать числовой литерал, размер которого превышает размерность любого целочисленного типа, но получите ошибку времени компиляции, если попытаетесь присвоить переменной чрезмерно большое значение литерала, как, например, в случае присвоения литерала 1000 переменной типа `byte`.

Как вы увидите в разделе, посвященном присвоению значений переменным, иногда в Go-коде не производится явное указание типа. В таких случаях Go использует для литерала *тип по умолчанию*: если в выражении ничто не указывает на то, к какому типу следует отнести литерал, то этот литерал относится к типу по умолчанию. Мы еще коснемся отнесения литералов к типу по умолчанию при обсуждении различных встроенных типов.

Логические значения

Для представления логических значений используется тип `bool`. Переменные типа `bool` могут иметь одно из двух значений: `true` или `false`. Нулевым значением для типа `bool` является значение `false`:

```
var flag bool // переменной не присвоено значение, поэтому она равна false
var isAwesome = true
```

Нужно сказать, что говорить о типах переменных достаточно сложно, не обсудив сначала способы их объявления, и наоборот. Мы сначала рассмотрим несколько примеров объявления переменных и обсудим их подробнее в разделе «`var` или `:=`» на с. 48.

Числовые типы

В Go много числовых типов (12, а также несколько специальных имен для этих типов), которые разделены на три категории. Если раньше вы использовали такой язык, как JavaScript, в котором лишь один числовой тип, то вас может немного удивить такое количество. При этом, на самом деле, одни типы используются довольно часто, а другие — крайне редко. Начнем с целочисленных типов, после чего рассмотрим типы чисел с плавающей запятой и редко используемые типы комплексных чисел.

Целочисленные типы

Язык Go предлагает типы для знаковых и беззнаковых целых чисел с размерностью от одного до восьми байт. Эти типы перечислены в табл. 2.1.

Таблица 2.1. Целочисленные типы языка Go

Имя типа	Диапазон значений
int8	От -128 до 127
int16	От -32 768 до 32 767
int32	От -2 147 483 648 до 2 147 483 647
int64	От -9 223 372 036 854 775 808 до 9 223 372 036 854 775 807
uint8	От 0 до 255
uint16	От 0 до 65 535
uint32	От 0 до 4 294 967 295
uint64	От 0 до 18 446 744 073 709 551 615

Нулевым значением для всех целочисленных типов, очевидно, является значение 0.

Специальные целочисленные типы

В Go предусмотрено несколько специальных имен для целочисленных типов. Так, `byte` — это псевдоним для типа `uint8`; значения типов `byte` и `uint8` можно присваивать друг другу, сравнивать и использовать в одной математической операции. Однако вы редко увидите имя `uint8` в Go-коде: вместо него принято применять имя `byte`.

Вторым специальным именем является имя `int`. На машинах с 32-разрядным процессором `int` означает 32-разрядное знаковое целое число, как и `int32`. На машинах с 64-разрядным процессором `int` означает 64-разрядное знаковое целое число, как и `int64`. Поскольку размерность типа `int` зависит от используемой платформы, вы получите ошибку времени компиляции, если попытаетесь присвоить друг другу, сравнить или применить в одной математической операции числа типов `int` и `int32` (или типов `int` и `int64`) без преобразования типов (подробнее об этом будет рассказано в подразделе «Явное преобразование типов» на с. 46). Целочисленные литералы по умолчанию относятся к типу `int`.



Вряде нетипичных 64-разрядных процессорных архитектур в качестве типа `int` используются 32-разрядные знаковые целые числа. Go поддерживает три такие архитектуры: `amd64p32`, `mips64p32` и `mips64p32le`.

Третьим специальным именем является имя `uint`. Для него действуют те же правила, что и в случае типа `int`, с тем лишь отличием, что это беззнаковый тип (то есть значение этого типа представляет собой либо 0, либо положительное число).

Существует еще два имени для целочисленных типов — `rune` и `uintptr`. Мы уже сталкивались с рунными литералами выше и подробно обсудим тип `rune` в подразделе «Пробуем использовать строки и руны» с. 46. Тип `uintptr` будет рассмотрен в главе 14.

Выбор подходящего целочисленного типа

Go предлагает больше целочисленных типов, чем многие другие языки. Такое богатство выбора может вызвать вопрос: когда лучше использовать каждый из этих типов? Здесь следует использовать три простых правила.

- Если вы работаете с файлами двоичного формата или с сетевым протоколом, использующим целые числа определенной размерности или знака, выбирайте соответствующий целочисленный тип.
- Если вы пишете библиотечную функцию, которая должна работать с любым целочисленным типом, напишите две функции, одна из которых будет использовать для параметров и переменных тип `int64`, а вторая — тип `uint64`. (О функциях и их параметрах мы подробно поговорим в главе 5.)



В данном случае идиоматический подход сводится к применению типов `int64` и `uint64` из-за того, что в языке Go нет (по крайней мере пока) универсальных функций и перегрузки. Без этих возможностей вам пришлось бы написать несколько функций для разных типов данных. Применение `int64` и `uint64` позволит написать код один раз, а затем пользоваться преобразованием типов для изменения данных по необходимости.

Этот подход применяется в стандартной библиотеке Go с функциями `FormatInt/FormatUint` и `ParseInt/ParseUint` из пакета `strconv`. Возможны и другие ситуации: например, в пакете `math/bits` размер целого числа имеет значение. В таких случаях необходимо написать отдельную функцию для каждого целочисленного типа.

- Во всех остальных случаях следует использовать тип `int`.



Если нет *необходимости* в явном указании размерности или знака целочисленных значений из соображений производительности или обеспечения интеграции, используйте тип `int`. Пока не доказано иное, использование любого другого типа следует считать преждевременной оптимизацией.

Целочисленные операторы

Целые числа в Go поддерживают обычный набор арифметических операторов: `+`, `-`, `*`, `/` и `%` (деление по модулю). Результатом целочисленного деления является целое число, и, чтобы получить в качестве результата число с плавающей запятой, необходимо использовать преобразование типов. Старайтесь не допускать деления целого числа на 0: это вызовет панику (подробнее о паниках будет сказано в разделе «Функции `panic` и `recover`» на с. 216).



Округление при целочисленном делении в Go производится путем отбрасывания дробной части; подробности можно найти в разделе документации языка Go, посвященном арифметическим операторам (<https://oreil.ly/zp3OJ>).

Для изменения переменной можно сочетать любой из арифметических операторов `s =`: например, `+=`, `-=`, `*=`, `/=`, `%=`. Так, после выполнения следующего кода переменная `x` будет иметь значение `20`:

```
var x int = 10
x *= 2
```

Для сравнения целых чисел можно использовать операторы `==`, `!=`, `>`, `>=`, `<` и `<=`.

В Go также есть операторы побитовых манипуляций для целых чисел. Вы можете выполнять побитовый сдвиг влево и вправо с помощью операторов `<<` и `>>` или применять битовые маски с помощью операторов `&` (логическое И), `|` (логическое ИЛИ), `^` (логическое исключающее ИЛИ) и `&^` (логическое И-НЕ). Как и арифметические операторы, для изменения переменной все логические операторы могут быть объединены `s =`: `&=`, `|=`, `^=`, `&^=`, `<<=`, `>>=`.

Типы чисел с плавающей запятой

В Go есть два типа чисел с плавающей запятой, которые представлены в табл. 2.2.

Таблица 2.2. Типы чисел с плавающей запятой, используемые в языке Go

Имя типа	Наибольшее абсолютное значение	Наименьшее (ненулевое) абсолютное значение
<code>float32</code>	3,40282346638528859811704183484516925440e+38	1,401298464324817070923729583289916131280e-45
<code>float64</code>	1,797693134862315708145274237317043567981e+308	4,940656458412465441765687928682213723651e-324

Как и для целочисленных типов, для типов чисел с плавающей точкой нулевым значением является 0.

Работа с числами с плавающей запятой в Go происходит практически так же, как в других языках. В Go используется формат представления таких чисел, основанный на спецификации IEEE 754, которая обеспечивает широкий диапазон и ограниченную степень точности. Выбрать подходящий тип достаточно просто: за исключением тех случаев, когда нужно обеспечить совместимость с имеющимся форматом, используется тип `float64`. Литералы чисел с плавающей запятой по умолчанию относятся к типу `float64`, поэтому самым простым решением будет всегда использовать тип `float64`. Это также позволяет уменьшить проблему точности чисел с плавающей запятой, связанную с тем, что точность типа `float32` ограничивается только шестью или семью десятичными знаками после запятой. При этом не стоит волноваться о разнице в расходе памяти, за исключением того случая, когда профайлер явно укажет, что расход памяти является значительным источником проблем. (Тестирование и профайлинг мы подробно обсудим в главе 13.)

Более важный вопрос состоит в том, стоит ли вообще использовать число с плавающей запятой. В большинстве случаев ответ на него будет отрицательным. Как и в других языках, в Go числа с плавающей запятой охватывают огромный диапазон, но не могут поддерживать каждое значение в этом диапазоне, позволяя сохранить лишь ближайшее приближение. Поскольку числа с плавающей запятой не относятся к точным, их можно использовать только в ситуациях, когда допускаются приблизительные значения или известны правила их применения. Поэтому их практически не используют в компьютерной графике и научных расчетах.



Числа с плавающей запятой не могут обеспечить точное представление десятичных значений. Не используйте их для представления денежных сумм или других значений, которые требуют точного десятичного представления!

В случае чисел с плавающей запятой можно использовать все стандартные операторы сравнения и математических действий, за исключением оператора `%`. При этом стоит обратить внимание на пару интересных особенностей операции деления чисел с плавающей запятой. Деление на ноль ненулевого значения этого типа чисел возвращает в качестве результата `+Inf` или `-Inf` (плюс или минус бесконечность), в зависимости от знака числа. Деление на ноль нулевого значения с плавающей запятой возвращает в качестве результата значение `NaN` (Not a Number, «не число»).

Хотя Go и позволяет сравнивать числа с плавающей запятой с помощью операторов `==` и `!=`, лучше не делайте этого. Из-за неточности два, казалось бы, равных числа с плавающей запятой могут оказаться не равны друг другу при их сравнении. Вместо того чтобы сравнивать два числа с плавающей запятой, следует определить максимально допустимое отклонение и посмотреть, не превышает ли его разница между этими числами. Величина этого отклонения (которое иногда называют «*эпсилон*») зависит от требуемой точности. Здесь я могу дать лишь одну простую рекомендацию: если вы не знаете, каким оно должно быть, обратитесь за советом к ближайшему знакомому математику.

IEEE 754

Как уже упоминалось выше, в Go (и в большинстве других языков программирования) используется формат представления чисел с плавающей запятой, основанный на спецификации IEEE 754.

Знакомство с этими правилами выходит за рамки этой книги, и надо сказать, что они не отличаются простотой. Например, при сохранении числа $-3,1415$ в виде значения типа `float64` его 64-разрядное представление в памяти будет выглядеть так:

```
1100000000001001001000011100101011000000100000110001001001101111
```

что, если быть точным, равно $-3,141500000000000018118839761883$.

Большинство программистов рано или поздно начинают понимать принцип представления целых чисел в двоичном виде (крайний правый разряд — это 1, следующий разряд — 2, следующий — 4 и т. д.). Принцип представления чисел с плавающей запятой выглядит совершенно иначе. Из 64 разрядов представленного выше числа один разряд используется для представления знака числа (положительного или отрицательного), 11 разрядов — для представления показателя степени, и 52 разряда — для представления нормализованной формы числа (или, как ее еще называют, *мантиссы*).

Более подробное описание стандарта IEEE 754 можно найти в Википедии (<https://oreil.ly/Gc05u>).

Типы комплексных чисел (возможно, они вам не потребуются)

Существует еще один редко используемый числовой тип. В языке Go превосходно реализована поддержка комплексных чисел. Если вы не знаете, что такое

комплексные числа, то, видимо, вам и не требуется эта функциональная возможность, поэтому можете спокойно пропустить данный раздел.

О поддержке комплексных чисел в Go можно сказать не так уж много. В Go определены два типа комплексных чисел: для представления действительной и мнимой части `complex64` использует значения типа `float32`, а `complex128` — значения типа `float64`. Оба этих типа объявляются с помощью встроенной функции `complex`. В Go существует несколько правил для определения типа вывода функции.

- Если оба параметра функции `complex` представляют собой нетипизированные константы или литералы, то функция возвратит нетипизированный литерал комплексного числа, по умолчанию обладающий типом.
- Если оба параметра функции `complex` представляют собой значения типа `float32`, то функция возвратит значение типа `complex64`.
- Если один параметр функции `complex` представляет собой значение типа `float32`, а второй параметр является нетипизированной константой или литералом, не выходящим за рамки диапазона типа `float32`, то функция возвратит значение типа `complex64`.
- Во всех остальных случаях функция возвратит значение типа `complex128`.

Для работы с комплексными числами можно использовать все стандартные арифметические операторы. Как и числа с плавающей запятой, комплексные числа можно сравнивать с помощью операторов `==` и `!=`, но из-за тех же проблем точности вместо этого лучше проверять величину разницы между числами. Для извлечения действительной и мнимой части комплексного числа можно использовать встроенные функции `real` и `imag` соответственно. В пакете `math/cmplx` также имеется ряд дополнительных функций для работы со значениями типа `complex128`.

Нулевым значением для обоих типов комплексных чисел является значение, и действительной, и мнимой части которого присвоено значение `0`.

В примере 2.1 показана простая программа, демонстрирующая основные приемы работы с комплексными числами. Вы можете запустить этот код в онлайн-песочнице (<https://oreil.ly/fuyIu>).

Пример 2.1. Комплексные числа

```
func main() {  
    x := complex(2.5, 3.1)  
    y := complex(10.2, 2)  
    fmt.Println(x + y)  
    fmt.Println(x - y)  
}
```

```

fmt.Println(x * y)
fmt.Println(x / y)
fmt.Println(real(x))
fmt.Println(imag(x))
fmt.Println(cmplx.Abs(x))
}

```

Запустив этот код, вы получите следующие результаты:

```

(12.7+5.1i)
(-7.699999999999999+1.1i)
(19.3+36.62i)
(0.2934098482043688+0.24639022584228065i)
2.5
3.1
3.982461550347975

```

Здесь также можно увидеть неточность чисел с плавающей запятой.

Если вам интересно, что собой представляет пятая разновидность используемых в Go литералов простых типов, то сообщу, что это мнимые литералы, которые служат для представления мнимой части комплексного числа. Они выглядят так же, как литералы чисел с плавающей запятой, отличаясь от них лишь наличием суффикса `i`.

Несмотря на встроенную поддержку комплексных чисел, язык Go не приобрел популярности в качестве языка для числовых вычислений. Его ограниченное применение связано с тем, что другие возможности (например, поддержка матриц) не входят в состав языка, а библиотеки вынуждены использовать менее эффективные средства, такие как срезы. (О срезах и о том, как они реализованы в Go, мы подробно поговорим в главах 3 и 6 соответственно.) Но если вдруг вам потребуется вычислять множество Мандельброта в определенном месте большой программы или создать программу для решения квадратных уравнений, вы всегда сможете воспользоваться встроенной поддержкой комплексных чисел.



Если вы собираетесь писать на Go приложения для числовых вычислений, то для этой цели можно использовать сторонний пакет `Gonum` (<https://www.gonum.org>). Этот пакет в полной мере использует возможности комплексных чисел и предлагает удобные библиотеки для таких вещей, как линейная алгебра, матрицы, интегралы и статистика. Однако перед тем, как его применять, рассмотрите вариант использования других языков.

Возможно, у вас возник вопрос: зачем вообще в Go была включена поддержка комплексных чисел? Ответ прост: они показались интересными Кену Томпсону (Ken Thompson), одному из создателей языка Go (а также операционной системы

Unix) (см. соответствующее обсуждение по адресу <https://oreil.ly/eBmkq>). Сейчас ведется дискуссия о возможном удалении комплексных чисел из будущей версии Go (<https://oreil.ly/Q76EV>), однако легче всего будет просто игнорировать существование такой возможности.

Пробуем использовать строки и руны

Пришло время поговорить о строках. Как и в большинстве других современных языков, строки присутствуют в Go в качестве одного из встроенных типов. Нулевым значением в случае строк является пустая строка. Go поддерживает стандарт Unicode: как вы уже видели в разделе, посвященном строковым литералам, в строку можно поместить любой символ стандарта Unicode. Подобно целым числам и числам с плавающей запятой, для сравнения строк используют операторы `==` и `!=`, а для упорядочения — операторы `>`, `>=`, `<` и `<=`. Объединение (конкатенация) строк производится с помощью `+`.

Строки в Go неизменяемы: вы можете повторно присвоить значение строковой переменной, но не можете изменить присвоенное ей строковое значение.

В Go также имеется тип для представления отдельной кодовой точки, а именно так называемые *руны* (*rune*). Тип `rune` представляет собой просто псевдоним для типа `int32`, так же как тип `byte` является псевдонимом для типа `uint8`. Как вы уже могли догадаться, рунные литералы по умолчанию относятся к типу `rune`, а строковые — к типу `string`.



При использовании символа используйте для него тип `rune`, а не тип `int32`. Хотя компилятор не видит между ними разницы, это позволяет четко выразить цель использования переменной.

Мы еще вернемся к строкам в следующей главе, где обсудим некоторые детали их реализации, их взаимосвязь с байтами и рунами, а также некоторые продвинутые возможности и подводные камни при работе с ними.

Явное преобразование типов

В большинстве языков, использующих несколько числовых типов, по мере необходимости производится автоматическое преобразование одного типа в другой. Такой подход называется *автоматическим повышением типов*, и хотя это кажется очень удобным, на практике эти правила преобразования одного типа в другой порой становятся очень сложными и могут привести к неожиданным результатам. Поскольку в языке Go ценится хорошая читабельность и четкое

выражение намерений, в нем не допускается автоматическое повышение типа переменных. При несовпадении типа переменных вы должны использовать *преобразование типов*, причем в один тип необходимо преобразовывать даже разноразмерные целые числа и числа с плавающей запятой. Это обеспечивает ясность в отношении того, какой тип вам нужен, без необходимости запоминать какие-либо правила преобразования типов (пример 2.2).

Пример 2.2. Преобразование типов

```
var x int = 10
var y float64 = 30.2
var z float64 = float64(x) + y
var d int = x + int(y)
fmt.Println(z, d)
```

В данном примере кода определены четыре переменные. Переменная `x` относится к типу `int` и содержит значение `10`; переменная `y` относится к типу `float64` и содержит значение `30.2`. Поскольку это переменные разного типа, для того чтобы их сложить, необходимо их преобразовать к одному типу. В случае переменной `z` мы преобразовали переменную `x` в тип `float64`, используя преобразование в тип `float64`, а в случае переменной `d` мы преобразовали переменную `y` в тип `int` с помощью преобразования в тип `int`. После выполнения этого кода переменные `z` и `d` будут хранить значения `40.2` и `40` соответственно.

Столь строгое отношение к типам влечет за собой и ряд других последствий. Поскольку все преобразования типов в Go выполняются явным образом, вы не можете интерпретировать значение второй переменной как логическое значение. Во многих языках ненулевое число или непустая строка могут интерпретироваться как логическое значение `true`. Как и в случае автоматического повышения типа, правила такой интерпретации могут варьироваться от языка к языку и требуют большой внимательности. Поэтому, как и следовало ожидать, в Go не допускается их использование. Это значит, что *ни один другой тип не может быть преобразован в логический тип явным или неявным образом*. Если вам нужно преобразовать значение другого типа данных в логический тип, используйте для этого операторы сравнения (`==`, `!=`, `>`, `<`, `<=`, `>=`). Например, вы можете проверить, не равна ли нулю переменная `x`, используя код `x == 0`. Если нужно выяснить, не является ли пустой строка `s`, используйте код `s == ""`.



Преобразование типов является одним из тех мест, в которых язык Go сделан чуть более многословным в обмен на большую ясность и простоту. Вы еще не раз увидите примеры такого подхода в этом языке. Делая выбор между ясностью выражения и краткостью кода, идиоматический Go выбирает первое.

var или :=

Для такого небольшого языка, каким является Go, в нем довольно много способов объявления переменных. Это объясняется тем, что каждый из имеющихся стилей объявления в определенной мере отражает и способ использования переменной. Что ж, посмотрим, как можно объявлять переменные в Go и когда будет уместен тот или иной способ.

Наиболее многословный способ объявления переменной в Go сводится к тому, чтобы использовать ключевое слово `var`, явный тип и присваивание:

```
var x int = 10
```

Если значение, стоящее справа от знака `=`, относится к нужному вам типу, то вы можете не указывать тип слева от знака `=`. Поскольку целочисленный литерал по умолчанию относится к типу `int`, объявляемая ниже переменная `x` будет отнесена к типу `int`:

```
var x = 10
```

С другой стороны, если требуется объявить переменную с присвоением ей нулевого значения, это можно сделать, указав лишь тип переменной и опустив стоящую правее операцию присваивания.

```
var x int
```

С помощью ключевого слова `var` можно объявить сразу несколько переменных одного типа:

```
var x, y int = 10, 20
```

несколько переменных одного типа с нулевыми значениями:

```
var x, y int
```

или несколько переменных разного типа:

```
var x, y = 10, "hello"
```

Наконец, существует еще один способ использования ключевого слова `var`. Если требуется объявить сразу несколько переменных, эти объявления можно сгруппировать в общий *список объявлений*:

```
var (  
  x      int  
  y      = 20  
  z      int = 30
```



```
d, e      = 40, "hello"
    f, g string
)
```

Go также поддерживает краткий формат объявления переменных. Внутри функций вместо объявления переменных с помощью ключевого слова `var` можно использовать оператор `:=`, который производит автоматический вывод типа переменной. Например, следующие две инструкции делают в точности одно и то же: они объявляют переменную `x` типа `int` со значением `10`:

```
var x = 10
x := 10
```

Как и в случае ключевого слова `var`, с помощью оператора `:=` можно объявить сразу несколько переменных. Так, обе представленные ниже строки объявляют переменные `x` и `y` со значениями `10` и `"hello"`:

```
var x, y = 10, "hello"
x, y := 10, "hello"
```

В то же время оператор `:=` способен на один трюк, который недоступен при использовании ключевого слова `var`: с его помощью можно присвоить значение существующей переменной. Если слева от оператора `:=` будет указана хотя бы одна новая переменная, то любая из остальных переменных может представлять собой уже существующую переменную:

```
x := 10
x, y := 30, "hello"
```

Оператор `:=` обладает одним ограничением: при объявлении переменной на уровне пакета необходимо использовать ключевое слово `var`, поскольку оператор `:=` нельзя использовать вне функций.

Какой же из этих стилей лучше использовать? Как и в любом другом случае, используйте тот способ, который позволяет как можно яснее выразить свои намерения. В большинстве случаев внутри функций переменные лучше объявлять с помощью оператора `:=`. Вне функций используйте списки объявлений в тех редких случаях, когда требуется объявить сразу несколько переменных на уровне пакета.

В ряде случаев будет лучше воздержаться от использования оператора `:=` внутри функций.

- Когда требуется инициализировать переменную нулевым значением, используйте форму `var x int`. Тем самым вы ясно покажете, что хотели создать переменную с нулевым значением.

- Когда переменной присваивается нетипизированная константа или литерал и эта константа или литерал по умолчанию относится не к тому типу, которым должна обладать переменная, используйте длинную форму объявления `var` с указанием типа. Хотя ничего не мешает использовать оператор `:=`, указав тип переменной с помощью преобразования типа: `x := byte(20)`, идиоматический подход сводится к тому, чтобы записать это в виде `var x byte = 20`.
- Поскольку оператор `:=` позволяет присваивать значения одновременно и новым, и существующим переменным, иногда он создает новые переменные вместо использования уже существующих так, как вы задумывали (подробнее об этом будет рассказано в подразделе «Затенение переменных» на с. 89). В таком случае лучше явно объявить все новые переменные с помощью ключевого слова `var`, чтобы однозначно указать, какие переменные являются новыми, а затем присвоить значения одновременно и новым, и старым переменным, используя оператор присваивания (`=`).

Хотя и ключевое слово `var`, и оператор `:=` позволяют объявлять сразу несколько переменных в одной строке, следует использовать этот стиль только в случае присвоения переменным нескольких возвращаемых функцией значений или при использовании идиомы «запятая-ок» (см. главу 5 и подраздел «Идиома “запятая-ок”» на с. 79).

Старайтесь по возможности не объявлять переменные вне функций в так называемом *блоке пакета* (см. раздел «Блоки» на с. 88). Объявление на уровне пакета переменных с изменяющимися значениями будет не очень удачной идеей. При объявлении переменной вне какой-либо функции, как правило, трудно отслеживать вносимые в нее изменения, что, в свою очередь, затрудняет анализ существующих в программе потоков данных. Это может вести к трудноуловимым программным ошибкам. Обычно рекомендуется объявлять в блоке пакета только те переменные, значения которых практически не изменяются.



Старайтесь не объявлять переменные вне функций, поскольку это усложняет анализ потоков данных.

Здесь вы можете спросить: а есть ли в Go способ *гарантировать* неизменяемость значения? Да, в Go существует такая возможность, однако она немного отличается от того, что вы могли видеть в других языках программирования. Пришло время познакомиться с ключевым словом `const`.

Использование ключевого слова `const`

Начиная изучение нового языка программирования, разработчики обычно пытаются найти в нем знакомые концепции. Определенный способ объявления неизменяемости значения есть во многих других языках. В Go это делается с помощью ключевого слова `const`. На первый взгляд, константы в Go ведут себя так же, как в других языках. Попробуйте запустить в онлайн-песочнице код из примера 2.3 (<https://oreil.ly/FdG-W>).

Пример 2.3. Объявление констант

```
const x int64 = 10

const (
    idKey = "id"
    nameKey = "name"
)

const z = 20 * 10

func main() {
    const y = "hello"

    fmt.Println(x)
    fmt.Println(y)

    x = x + 1
    y = "bye"

    fmt.Println(x)
    fmt.Println(y)
}
```

Если вы попытаетесь запустить этот код, компиляция завершится неудачно со следующими сообщениями об ошибках:

```
./const.go:20:4: cannot assign to x
./const.go:21:4: cannot assign to y
```

Как видите, константа объявляется на уровне пакета или внутри функции. Как и в случае ключевого слова `var`, с помощью ключевого слова `const` можно (и нужно) объявлять целые группы взаимосвязанных констант, используя круглые скобки.

Однако возможности `const` в Go очень ограничены. Константы в Go — это средство присвоения имен литералам. Они способны содержать только те значения, которые компилятор может вычислить на этапе компиляции. Это значит, что им допускается присваивать:

- числовые литералы;
- значения `true` и `false`;
- строки;
- руны;
- встроенные функции `complex`, `real`, `imag`, `len` и `cap`;
- выражения, состоящие из операторов и перечисленных выше видов значений.



Функции `len` и `cap` мы рассмотрим в следующей главе. Вместе с ключевым словом `const` также можно использовать такой вид значений, как `iota`, который мы рассмотрим, когда будем обсуждать создание собственных типов в главе 7.

В Go вы не можете сделать неизменяемым значение, вычисляемое на этапе выполнения программы. Как будет показано в следующей главе, в этом языке нет неизменяемых массивов, срезов, карт или структур и нет способа сделать неизменяемым определенное поле структуры. Однако это не настолько серьезные ограничения, как может показаться. Внутри функции сразу понятно, изменяется ли переменная, поэтому неизменяемость в Go не играет большой роли. Как мы увидим в разделе «Go — язык с передачей параметров по значению» на с. 136, в Go не допускается изменение переменных, передаваемых функциям в качестве параметров.



Константы в Go представляют собой лишь способ присвоения имен литералам. В этом языке нельзя сделать переменную неизменяемой.

Типизированные и нетипизированные константы

Константы могут быть типизированными или нетипизированными. Нетипизированная константа ведет себя совершенно так же, как литерал: она не обладает собственным типом, но относится к типу по умолчанию в том случае, когда невозможно определить тип путем вывода типов. Типизированной константе можно лишь непосредственно присвоить значение соответствующего типа.

Делать константу типизированной или нет, зависит от того, с какой целью она объявляется. Если вы объявляете математическую константу, которая будет использоваться с разными числовыми типами, то лучше оставить ее нетипизи-

рованной. В большинстве случаев использование нетипизированной константы дает больше гибкости. Однако в некоторых ситуациях константа должна быть отнесена к определенному типу. Мы воспользуемся типизированными константами, когда будем рассматривать создание перечислений с помощью ключевого слова `iota` в подразделе «Йота (иногда) используется для создания перечислений» на с. 173.

Вот так выглядит объявление нетипизированной константы:

```
const x = 10
```

При этом будет допустимой любая из следующих операций присваивания:

```
var y int = x
var z float64 = x
var d byte = x
```

А так объявляется типизированная константа:

```
const typedX int = 10
```

Этой константе можно присвоить только значение типа `int`. Попытка присвоить ей значение какого-либо другого типа приведет к ошибке на этапе компиляции:

```
cannot use typedX (type int) as type float64 in assignment
```

Неиспользуемые переменные

Одна из целей языка Go состоит в том, чтобы упростить большим группам разработчиков совместную работу над программами. Для этого в Go предусмотрен ряд правил, отличающих его от других языков программирования. Как уже было упомянуто в главе 1, Go-программы должны форматироваться с помощью команды `go fmt`, чтобы упростить создание средств обработки кода и обеспечить единообразное оформление кода. Еще одно требование состоит в том, что *каждая объявленная локальная переменная должна быть прочитана*. Объявление локальной переменной без последующего чтения ее значения приведет к *ошибке на этапе компиляции*.

Проверка на наличие неиспользуемых переменных выполняется компилятором не очень тщательно. До тех пор, пока переменная не будет прочитана хотя бы один раз, компилятор не будет жаловаться, даже если некоторые из записанных в эту переменную значений не будут прочитаны. Следующий пример кода является вполне допустимой Go-программой, которую можно запустить в онлайн-песочнице (<https://oreil.ly/8JLA6>):

```
func main() {
    x := 10
    x = 20
    fmt.Println(x)
    x = 30
}
```

Хотя компилятор и команда `go vet` не заметят, что переменной `x` присваиваются неиспользуемые значения `10` и `30`, инструмент `golangci-lint` выявит эти операции присваивания:

```
$ golangci-lint run
unused.go:6:2: ineffectual assignment to `x` (ineffassign)
    x := 10
    ^
unused.go:9:2: ineffectual assignment to `x` (ineffassign)
    x = 30
    ^
```



Компилятор языка Go не мешает созданию неиспользуемых переменных на уровне пакета. Это еще одна причина, по которой необходимо избегать создания переменных на этом уровне.

НЕИСПОЛЬЗУЕМЫЕ КОНСТАНТЫ

Удивительно, но компилятор языка Go позволяет создавать неиспользуемые константы с помощью ключевого слова `const`. Это объясняется тем, что константы в Go вычисляются на этапе компиляции и не производят никаких побочных эффектов. Это позволяет легко их удалять: если константа нигде не используется, она просто не включается в компилируемый двоичный файл.

Именование переменных и констант

Есть определенная разница между тем, какие правила именования переменных установлены в языке Go, и тем, какой стиль именования переменных и констант обычно используют Go-разработчики. Как и в большинстве других языков, в Go имена идентификаторов должны начинаться с буквы или символа подчеркивания и могут содержать цифры, буквы и символы подчеркивания. В Go понятия «буква» и «цифра» понимаются чуть шире, чем в других языках, и могут представлять собой любой буквенно-цифровой символ стандарта Unicode. В результате этого в Go будет вполне допустимым использование таких переменных, какие показаны в примере 2.4.

Пример 2.4. Имена переменных, которые не стоит использовать

```
_0 := 0_0
_1 := 20
n := 3
a := "hello" // Символ Unicode U+FF41
fmt.Println(_0)
fmt.Println(_1)
fmt.Println(n)
fmt.Println(a)
```

Хотя такой код будет работать, *никогда* не давайте своим переменным такие имена. Они считаются неидиоматическими, поскольку идут вразрез с базовым принципом четкого выражения в коде его назначения. Они трудны для понимания, и их сложно вводить на большинстве клавиатур. Особенно опасно при этом использовать кодовые точки Unicode, которые при таком же внешнем виде будут обозначать совершенно другую переменную. Попробуйте запустить в онлайн-песочнице код из примера 2.5 (<https://oreil.ly/hrvb6>).

Пример 2.5. Использование в именах переменных кодовых точек Unicode, которые похожи на стандартные символы

```
func main() {
    a := "hello"    // символ Unicode U+FF41
    a := "goodbye" // стандартная строчная буква "a" (символ Unicode U+0061)
    fmt.Println(a)
    fmt.Println(a)
}
```

Запустив этот код, вы увидите на экране следующее:

```
hello
goodbye
```

Хотя в именах переменных допускается применение символа подчеркивания, он используется достаточно редко, поскольку в идиоматическом Go-коде не принято задействовать «змеиный стиль» написания имен (такие имена, как `index_counter` или `number_tries`). Вместо этого в Go принято использовать «верблюжий стиль» написания (такие имена, как `indexCounter` или `numberTries`) в тех случаях, когда имя идентификатора состоит из нескольких слов.

Во многих языках константы записываются буквами верхнего регистра с разделением слов символами подчеркивания (то есть используются такие имена, как `INDEX_COUNTER` или `NUMBER_TRIES`). В Go такой стиль не используется. Это объясняется тем, что в Go регистр первой буквы в имени элемента, объявляемого на уровне пакета, определяет его доступность за пределами пакета. Мы еще вернемся к этому вопросу, когда будем говорить о пакетах в главе 9.



В Go символ подчеркивания () сам по себе является специальным именем идентификатора; мы поговорим об этом подробнее, когда будем обсуждать функции в главе 5.

Внутри функций старайтесь использовать короткие имена переменных. Чем меньше область видимости переменной, тем более коротким должно быть ее имя. В Go-коде можно часто встретить однобуквенные имена переменных. Например, в цикле `for-range` часто используются переменные с именами `k` и `v` (что является сокращением от слов `key` — «ключ» и `value` — «значение»). В стандартном цикле `for` в качестве индексной переменной обычно используются переменные `i` и `j`. Существуют и другие идиоматические способы именования переменных распространенных типов; мы рассмотрим их после знакомства с содержимым стандартной библиотеки.

Некоторые языки с менее строгой системой типов поощряют разработчиков включать ожидаемый тип переменной в ее имя. Но здесь в этом нет необходимости, поскольку Go является языком со строгой типизацией. В то же время в случае однобуквенных имен в Go принято использовать в качестве имени переменной первую букву имени типа (то есть `i` для целых чисел, `f` для чисел с плавающей запятой и `b` для логических значений). Тот же принцип действует и при определении собственных типов.

Такие короткие имена служат двум целям. Во-первых, это позволяет вводить меньше повторяющихся слов и делает код короче. Во-вторых, это не допускает чрезмерного усложнения кода. Если у вас начинают возникать трудности с пониманием того, что обозначают переменные с короткими именами, то, возможно, данный блок кода выполняет слишком большой объем работы.

При объявлении переменных и констант в блоке пакета старайтесь использовать более описательные имена. При этом по-прежнему не нужно указывать тип, но из-за широкого применения следует использовать более полное имя, чтобы было ясно, что представляет собой значение.

Резюме

Вы проделали большой объем работы: поняли, как использовать встроенные типы, объявлять переменные, работать с присваиваниями и операторами. В следующей главе будут рассмотрены составные типы языка Go: массивы, срезы, карты и структуры. Мы также еще раз поговорим о строках и рунах, и вы узнаете, что такое кодировки.

Составные типы

В предыдущей главе были рассмотрены простые типы: числа, булевы значения и строки. В этой главе вы узнаете о составных типах языка Go, встроенных функциях для их поддержки и рекомендуемых способах работы с ними.

Массивы — слишком строгие для того, чтобы использовать их напрямую

Как и в большинстве других языков программирования, в Go есть массивы, однако они редко используются напрямую. Вы узнаете чуть позже, чем это объясняется, а пока ненадолго остановимся на том, как выглядит синтаксис объявления массивов и как с ними работать.

Все элементы массива должны относиться к указанному типу (однако это не значит, что все элементы должны относиться к одному и тому же типу). Существует несколько стилей объявления массивов. Первый стиль сводится к тому, чтобы указать размер массива и тип его элементов:

```
var x [3]int
```

Это объявление создает массив из трех элементов типа `int`. Поскольку значения не были указаны, все элементы (`x[0]`, `x[1]` и `x[2]`) инициализируются нулевыми значениями для типа `int`, то есть 0. При наличии начальных значений массива их следует указать в *литерале массива*.

```
var x = [3]int{10, 20, 30}
```

В случае *разреженного массива* (у которого большинство элементов имеют нулевое значение) в литерале массива можно указать только индексы отдельных элементов с соответствующими значениями:

```
var x = [12]int{1, 5: 4, 6, 10: 100, 15}
```

Это объявление создает массив из 12 элементов типа `int` со следующими значениями: `[1, 0, 0, 0, 0, 4, 6, 0, 0, 0, 100, 15]`.

Если массив инициализируется с помощью литерала массива, вместо количества элементов можно поставить многоточие (`...`):

```
var x = [...]int{10, 20, 30}
```

Для сравнения массивов используют операторы сравнения `==` и `!=`:

```
var x = [...]int{1, 2, 3}
var y = [3]int{1, 2, 3}
fmt.Println(x == y) // выводит true
```

Хотя в Go есть только одномерные массивы, с их помощью можно симитировать и многомерные массивы:

```
var x [2][3]int
```

Этот код объявляет массив `x` из двух элементов, представляющих собой массивы из трех элементов типа `int`. Возможно, это покажется лишней подробностью, но некоторые языки предлагают реальную поддержку многомерных массивов, и Go не относится к их числу.

Как и в большинстве других языков, для чтения и записи элементов массива в Go используются квадратные скобки:

```
x[0] = 10
fmt.Println(x[2])
```

При чтении и записи нельзя выходить за границы массива или использовать отрицательный индекс. Если такая ошибка будет допущена при использовании константы или литерала в качестве индекса, это приведет к ошибке на этапе компиляции. Если же выход за границы массива будет допущен при использовании переменной в качестве индекса, то такой код скомпилируется, но выдаст *панику* во время выполнения (что такое паника, будет подробно рассказано в разделе «Функции `panic` и `recover`» на с. 216).

Наконец, встроенная функция `len` принимает на вход массив и возвращает его длину:

```
fmt.Println(len(x))
```

Как уже было сказано выше, массивы в Go редко используются напрямую. Это объясняется тем, что они обладают необычным ограничением: в Go *размер* массива считается составной частью его *типа*. То есть массивы, объявленные с помощью объявлений `[3]int` и `[4]int`, будут представлять собой массивы

разного типа. Это также означает, что вы не можете использовать переменную для указания размера массива, поскольку типы должны определяться еще на этапе компиляции, а не во время выполнения.

Более того, *вы не можете использовать преобразование типов для преобразования друг в друга массивов разного размера*. Поскольку нельзя преобразовывать друг в друга массивы разного размера, невозможно написать функцию, способную работать с массивами любого размера или присваивать массивы разного размера одной и той же переменной.



Мы подробнее остановимся на внутреннем устройстве массивов в главе 6, когда будем обсуждать схему распределения памяти.

Из-за этих ограничений следует использовать массивы только с определенным размером. Так, например, некоторые из криптографических функций в стандартной библиотеке возвращают массивы, потому что размеры контрольных сумм определены самим алгоритмом. Но такое поведение следует считать исключением, а не правилом.

Зачем же нужно было включать в язык настолько ограниченный элемент? Основная причина в том, что они служат в качестве вспомогательного хранилища для *срез*ов, которые являются одними из самых полезных элементов этого языка.

Срезы

В большинстве случаев, когда требуется структура данных для размещения последовательности значений, следует использовать срез. Основная причина такой популярности срезов в том, что их длина *не является* составной частью типа. Это устраняет свойственное массивам ограничение. Мы можем написать функцию, способную обрабатывать срезы любого размера (о том, как в Go пишутся функции, будет рассказано в главе 5) и увеличивать срезы по мере необходимости. Начнем с основ работы со срезами в Go, после чего рассмотрим рекомендуемые способы их использования.

Работа со срезами во многом напоминает работу с массивами, но все же имеет ряд небольших отличий. Первое отличие состоит в том, что при объявлении среза вам не нужно указывать его размер:

```
var x = []int{10, 20, 30}
```



Если использовать [...], то получится массив. Срез формируется с помощью [].

В коде, приведенном выше, создается массив из трех элементов типа `int` с помощью *литерала среза*. Как и в случае массивов, в литерале среза также можно указать только индексы отдельных элементов с соответствующими значениями:

```
var x = []int{1, 5: 4, 6, 10: 100, 15}
```

Здесь создается срез из 12 элементов типа `int` со следующими значениями: [1, 0, 0, 0, 0, 4, 6, 0, 0, 0, 100, 15].

Вы можете имитировать многомерные срезы, создавая срезы срезов:

```
var x [][]int
```

Для чтения и записи элементов среза используются квадратные скобки, и, как и в случае массивов, при этом нельзя выходить за границы среза или использовать отрицательный индекс:

```
x[0] = 10  
fmt.Println(x[2])
```

До сих пор срезы вели себя практически так же, как и массивы. Различия между срезами и массивами становятся заметны при объявлении среза без использования литерала:

```
var x []int
```

Так мы получим срез элементов типа `int`. Поскольку мы не предоставили никаких значений, срезу `x` присваивается нулевое значение, которым в случае срезов является пока неизвестное нам значение `nil`. Что собой представляет это значение, будет подробно рассказано в главе 6, а пока можно лишь отметить, что оно немного отличается от значения `null`, используемого в других языках. Значение `nil` языка Go представляет собой идентификатор, представляющий отсутствие значения для ряда типов. Как и нетипизированные числовые константы, рассмотренные в предыдущей главе, значение `nil` не обладает типом, что позволяет присваивать или сравнивать его со значениями разных типов. Срез, равный `nil`, не содержит элементов.

В отличие от всех рассмотренных до сих пор типов, срез является *несравнимым* типом. Попытка узнать, являются ли два среза одинаковыми, с помощью

оператора сравнения `==` или `!=` приведет к ошибке на этапе компиляции. Единственное, с чем можно сравнить срез, — это значение `nil`:

```
fmt.Println(x == nil) // выводит true
```



В пакете `reflect` есть функция `DeepEqual`, которая может сравнивать практически все что угодно, включая срезы. В первую очередь она предназначена для тестирования, но при необходимости ее можно использовать для сравнения срезов. Мы рассмотрим ее подробнее при обсуждении рефлексии в главе 14.

Функция `len`

Язык Go предлагает несколько встроенных функций для работы со встроенными типами. Вы уже видели примеры использования встроенных функций `complex`, `real` и `imag` для создания комплексных чисел и извлечения их действительной и мнимой части. Для срезов также предусмотрено несколько встроенных функций. Встроенная функция `len` уже упоминалась при рассмотрении массивов. Ее можно использовать для срезов, кроме того, она возвращает 0 при передаче ей среза, равного `nil`.



Такие функции, как `len`, встроены в язык Go в силу того, что выполняемые ими действия невозможно осуществить с помощью функций, написанных программистом. Как вы уже видели, функция `len` может принимать на вход любой массив или срез. Чуть позже мы убедимся, что она также может работать со строками и картами. В разделе «Каналы» на с. 255 будет показано, как ее можно использовать для работы с каналами. Попытка передать функции `len` переменную любого другого типа приведет к ошибке на этапе компиляции. Go не позволяет разработчикам писать функции, которые могли бы вести себя таким же образом. В главе 5 будет возможность еще раз в этом убедиться.

Функция `append`

Встроенная функция `append` используется для увеличения срезов:

```
var x []int
x = append(x, 10)
```

Эта функция принимает как минимум два параметра: срез с элементами любого типа и отдельный элемент этого типа. Возвращает она срез того же типа. Возвращаемый срез нужно снова присвоить тому срезу, который был

передан функции. В данном примере мы добавляем элемент в срез, равный `nil`, однако точно так же можно добавлять элементы и в срез, который уже содержит элементы:

```
var x = []int{1, 2, 3}
x = append(x, 4)
```

За один раз можно добавить не один, а сразу несколько элементов:

```
x = append(x, 5, 6, 7)
```

Один срез добавляется к другому с помощью оператора `...` для расширения исходного среза на отдельные значения (подробнее об этом операторе будет рассказано в подразделе «Вариативные входные параметры и срезы» на с. 118):

```
y := []int{20, 30, 40}
x = append(x, y...)
```

Если вы забудете присвоить значение, возвращаемое функцией `append`, это приведет к ошибке на этапе компиляции. Возможно, вам непонятно, зачем это нужно делать, поскольку такой подход кажется несколько многословным. Мы остановимся на этом подробнее в главе 5, а пока можно лишь отметить, что в языке Go используется *передача параметров по значению*. Каждый раз, когда вы передаете параметр функции, Go создает копию передаваемого значения. Поэтому, когда срез передается функции `append`, на самом деле она получает его копию. Функция добавляет значения в копию среза и возвращает ее. После этого возвращенный срез нужно снова присвоить той же переменной, которой была передана функции.

Емкость среза

Как уже было сказано, срез представляет собой последовательность значений. Элементам среза выделяются последовательные ячейки памяти, что ускоряет чтение и запись этих значений. Каждый срез обладает определенной *емкостью*, под чем понимается количество зарезервированных последовательных ячеек памяти. Емкость может быть больше длины. При каждом добавлении элементов в срез одно или несколько значений добавляются в конец среза. Каждое из добавляемых значений увеличивает на единицу длину среза. Когда длина становится равной емкости, в срезе уже не остается места для размещения новых значений. В этом случае при попытке добавить новые значения функция `append` даст указание среде выполнения языка Go выделить новый срез большей емкости. После этого она скопирует значения исходного среза в новый срез, добавит новые значения в его конец и возвратит его в качестве результата.

СРЕДА ВЫПОЛНЕНИЯ ЯЗЫКА GO

Выполнение программ, написанных на любом высокоуровневом языке, обеспечивается с помощью некоторого набора библиотек, и язык Go не является исключением в этом плане. Среда выполнения языка Go берет на себя такие задачи, как выделение памяти и сборка мусора, поддержка конкурентности, работа с сетью и реализация встроенных типов и функций.

Среда выполнения Go включается в состав каждого компилируемого двоичного файла этого языка. В этом язык Go отличается от языков, использующих виртуальную машину, которая должна быть установлена дополнительно, чтобы могли работать программы, написанные на этих языках. Включение среды выполнения в состав двоичных файлов упрощает распространение программ, написанных на языке Go, и исключает возможность несовместимости между средой выполнения и программой.

При увеличении среза с помощью функции `append` среде выполнения языка Go требуется некоторое время на то, чтобы выделить новую область памяти и скопировать в нее уже имеющиеся данные из старой области памяти. Также нужно освободить старую область памяти с помощью сборки мусора. Из-за этого каждый раз, когда емкость среза становится недостаточной, среда выполнения языка Go увеличивает ее сразу на несколько единиц. Начиная с версии Go 1.14, действует следующее правило: емкость среза увеличивается в два раза, пока не начинает превышать 1024, после чего она каждый раз увеличивается минимум на 25 %.

Точно так же, как встроенная функция `len` возвращает текущую длину среза, встроенная функция `cap` возвращает текущую емкость среза. Эта функция используется гораздо реже функции `len`. Обычно ее применяют для проверки среза на предмет того, достаточно ли в нем места для размещения новых данных, или требуется создать новый срез с помощью функции `make`.

Функции `cap` можно передать и массив, но в таком случае она всегда возвращает то же значение, что и функция `len`. Не стоит использовать этот трюк в своем коде; лучше просто запомните это как любопытную особенность языка Go.

Посмотрим, как изменяются длина и емкость среза по мере добавления в него элементов. Запустите на своей машине или в онлайн-песочнице (<https://oreil.ly/yiHu->) код из примера 3.1.

Пример 3.1. Как изменяется емкость среза

```
var x []int
fmt.Println(x, len(x), cap(x))
x = append(x, 10)
fmt.Println(x, len(x), cap(x))
x = append(x, 20)
fmt.Println(x, len(x), cap(x))
x = append(x, 30)
fmt.Println(x, len(x), cap(x))
x = append(x, 40)
fmt.Println(x, len(x), cap(x))
x = append(x, 50)
fmt.Println(x, len(x), cap(x))
```

Скомпилировав и запустив этот код, вы увидите представленные ниже результаты. Обратите внимание на то, как и когда увеличивается емкость среза.

```
[] 0 0
[10] 1 1
[10 20] 2 2
[10 20 30] 3 4
[10 20 30 40] 4 4
[10 20 30 40 50] 5 8
```

Как бы ни было удобно полагаться на автоматическое увеличение срезов, будет гораздо эффективнее один раз задать их размер. Если вы заранее знаете, сколько элементов будет размещено в срезе, то лучше сразу выберите подходящую емкость при его создании. Это можно сделать с помощью функции `make`.

Функция `make`

Мы уже видели два способа объявления среза: с использованием литерала среза и нулевого значения `nil`. Будучи вполне удобными, они, однако, не позволяют создавать пустой срез с заданной длиной или емкостью. Для этой цели используется встроенная функция `make`. Она позволяет указать тип, длину и — опционально — емкость среза. Вот как это выглядит:

```
x := make([]int, 5)
```

Этот код создает срез элементов типа `int`, длина и емкость которого равны 5. Поскольку длина равна 5, допустимыми являются элементы `x[0]`–`x[4]`, и все они инициализируются значением `0`.

Частая ошибка новичков здесь сводится к тому, чтобы попытаться заполнить такой срез с помощью функции `append`:

```
x := make([]int, 5)
```



```
x = append(x, 10)
```

Число 10 добавляется в конец среза, *после* нулевых значений с индексами 0–4, поскольку функция `append` всегда увеличивает длину среза. Теперь срез `x` содержит элементы `[0 0 0 0 0 10]`, его длина равна 6, а емкость — 10 (емкость была увеличена в два раза из-за добавления шестого элемента).

Функция `make` также позволяет указать исходную емкость среза:

```
x := make([]int, 5, 10)
```

Этот код создает срез элементов типа `int` длиной 5 и емкостью 10.

Аналогичным образом можно создать срез с нулевой длиной, но ненулевой емкостью:

```
x := make([]int, 0, 10)
```

Здесь мы получаем не равный `nil` срез длиной 0 и емкостью 10. Поскольку длина равна 0, невозможно обращаться к элементам среза по индексу; можно только добавлять в него значения с помощью функции `append`:

```
x := make([]int, 0, 10)
x = append(x, 5, 6, 7, 8)
```

Теперь срез `x` содержит элементы `[5 6 7 8]`, его длина равна 4, а емкость — 10.



Указанная вами емкость никогда не должна быть меньше длины! Если вы укажете такое значение с помощью константы или числового литерала, это приведет к ошибке при компиляции. Если сделать это с помощью переменной, то во время выполнения программа выдаст панику.

Объявление собственного среза

Теперь, когда были рассмотрены все возможные способы создания срезов, необходимо определиться со стилем объявления. При решении этой задачи нужно прежде всего постараться сделать так, чтобы срез как можно реже приходилось увеличивать. Если существует вероятность того, что срез вообще не потребуется увеличивать (например, в силу того, что функция не возвратит результаты), объявите его с помощью ключевого слова `var` без присваиваемого значения, как показано в примере 3.2, чтобы создать срез, равный `nil`.

Пример 3.2. Объявление среза, который может остаться равным `nil`

```
var data []int
```



Вы также можете создать срез, используя пустой литерал среза:

```
var x = []int{}
```

Этот код создает срез нулевой длины, который не является нулевым (сравнение его со значением `nil` возвращает значение `false`). Во всех остальных отношениях срез нулевой длины ведет себя точно так же, как и срез, равный `nil`. Единственным случаем, когда может потребоваться такой срез нулевой длины, является преобразование среза в формат JSON. Мы поговорим об этом подробнее в разделе «Пакет `encoding/json`» на с. 295.

Если у вас есть некоторые начальные значения или если известно, что значения среза не будут изменяться, лучше будет объявить срез, используя литерал среза (пример 3.3).

Пример 3.3. Объявление среза с использованием исходных значений

```
data := []int{2, 4, 6, 8} // известные нам числа
```

Если при написании программы сразу известно, насколько большим должен быть срез, но нет информации, какие именно значения он будет содержать, используйте функцию `make`. Но что следует указать в вызове функции `make`: ненулевую длину или нулевую длину и ненулевую емкость? Здесь возможны три варианта.

- Если срез используется в качестве буфера (мы коснемся этой темы в разделе «Пакет `io` и его друзья» на с. 286), то лучше указать ненулевую длину.
- Если точно известно, каким должен быть размер среза, можно указать его длину и задать значения, обращаясь к его элементам по индексу. Так часто делают в том случае, когда нужно преобразовать значение одного среза и сохранить его в другом срезе. Недостатком такого подхода является то, что в случае ошибки с определением размера среза можно получить нулевые значения в конце среза или панику из-за попытки обратиться к несуществующим элементам.
- Во всех прочих случаях в вызове функции `make` лучше указать нулевую длину и ненулевую емкость. Это позволяет добавлять элементы с помощью функции `append`. Если реальное количество элементов будет меньше указанной емкости, вы не получите лишние нулевые значения в конце среза. Если количество элементов превысит указанную емкость, ваш код не выдаст панику.

Go-сообщество разделилось на сторонников второго и третьего подхода. Я предпочитаю использовать функцию `append` в сочетании со срезом, изначально имеющим нулевую длину. Хотя такой подход и оказывается достаточно медленным в некоторых ситуациях, меньше вероятность возникновения ошибки.



Не забывайте о том, что функция `append` всегда увеличивает длину среза! Если вы указали длину среза в вызове функции `take`, то перед тем, как использовать функцию `append`, убедитесь, что это именно то, что вам нужно. В противном случае вы можете получить ненужные вам нулевые значения в начале среза.

Срезание срезов

Можно создать срез на основе среза. В таком случае он заключается в квадратных скобки и включает в себя значения начального и конечного смещения, разделенные знаком двоеточия (`:`). Если опускается начальное смещение, оно принимается равным 0. Сходным образом, если опускается конечное смещение, вместо него подставляется конец среза. Вы можете посмотреть, как это работает, запустив в онлайн-песочнице код из примера 3.4 (https://oreil.ly/DW_FU).

Пример 3.4. Срезание срезов

```
x := []int{1, 2, 3, 4}
y := x[:2]
z := x[1:]
d := x[1:3]
e := x[:]
fmt.Println("x:", x)
fmt.Println("y:", y)
fmt.Println("z:", z)
fmt.Println("d:", d)
fmt.Println("e:", e)
```

Этот код выведет следующее:

```
x: [1 2 3 4]
y: [1 2]
z: [2 3 4]
d: [2 3]
e: [1 2 3 4]
```

Иногда срезы используют общую область памяти

При создании среза на основе среза копия этих данных *не создается*. На самом деле вы создаете дополнительную переменную, использующую ту же область памяти. Это значит, что изменение элемента среза будет затрагивать все срезы, которые используют этот элемент. Посмотрим, что произойдет, если мы попробуем изменить значения. Для этого запустите в онлайн-песочнице код из примера 3.5 (<https://oreil.ly/mHxe4>).

Пример 3.5. Срезы с общей памятью

```
x := []int{1, 2, 3, 4}
y := x[:2]
z := x[1:]
x[1] = 20
y[0] = 10
z[1] = 30
fmt.Println("x:", x)
fmt.Println("y:", y)
fmt.Println("z:", z)
```

Результат работы кода будет таким:

```
x: [10 20 30 4]
y: [10 20]
z: [20 30 4]
```

Изменение среза `x` привело к изменению срезов `y` и `z`, а изменение срезов `y` и `z` привело к изменению среза `x`.

Срезание срезов может стать чрезвычайно запутанным, если сочетать его с функцией `append`. Попробуйте запустить в онлайн-песочнице код из примера 3.6 (<https://oreil.ly/2mB59>).

Пример 3.6. Использование функции `append` усложняет понимание кода, использующего срезы с общей памятью

```
x := []int{1, 2, 3, 4}
y := x[:2]
fmt.Println(cap(x), cap(y))
y = append(y, 30)
fmt.Println("x:", x)
fmt.Println("y:", y)
```

Этот код выведет следующее:

```
4 4
x: [1 2 30 4]
y: [1 2 30]
```

Что же здесь произошло? Всякий раз, когда срез создается на основе другого среза, емкость этого подсреза устанавливается равной емкости исходного среза, за вычетом используемого подсрезом смещения внутри исходного среза. Это значит, что любой подсрез также занимает всю неиспользуемую емкость исходного среза.

Когда мы создаем срез `y` на основе среза `x`, его длина устанавливается равной 2, но его емкость устанавливается равной 4, как и у среза `x`. И поскольку емкость

равна 4, то при добавлении элемента в конец среза `y` это значение помещается в третью позицию среза `x`.

В силу такого поведения функции `append` ее использование может давать очень неожиданные результаты, когда добавление элементов в один срез ведет к перезаписи элементов другого среза. Попробуйте, например, догадаться, что выведет код из примера 3.7, а затем проверьте правильность своих предположений, запустив этот код в онлайн-песочнице (https://oreil.ly/1u_tO).

Пример 3.7. Еще более запутанный пример использования срезов

```
x := make([]int, 0, 5)
x = append(x, 1, 2, 3, 4)
y := x[:2]
z := x[2:]
fmt.Println(cap(x), cap(y), cap(z))
y = append(y, 30, 40, 50)
x = append(x, 60)
z = append(z, 70)
fmt.Println("x:", x)
fmt.Println("y:", y)
fmt.Println("z:", z)
```

Чтобы не усложнять себе жизнь при использовании срезов, следует либо никогда не применять функцию `append` к подсрезам, либо исключить вероятность того, что эта функция приведет к перезаписи данных, используя *полное выражение среза*. Будучи немного странным на вид, это выражение, однако, показывает, сколько памяти совместно используют родительский срез и подсрез. Полное выражение среза дополнительно содержит третий элемент, указывающий последнюю позицию в рамках емкости родительского среза, которая доступна для подсреза. Чтобы получить емкость подсреза, нужно отнять от этой цифры начальное смещение. В примере 3.8 показано, как можно изменить третью и четвертую строки предыдущего примера для использования выражения среза.

Пример 3.8. Выражение полного среза позволяет защититься от функции `append`

```
y := x[:2:2]
z := x[2:4:4]
```

Попробуйте запустить этот код в онлайн-песочнице (<https://oreil.ly/Cn2cX>). Здесь и у среза `y`, и у среза `z` емкость равна 2. Поскольку мы ограничили емкость подсрезом до их длины, добавление дополнительных элементов в срезы `y` и `z` ведет к созданию новых срезов, никак не влияющих на другие срезы. После выполнения этого кода срез `x` будет содержать элементы [1 2 3 4 60], срез `y` — элементы [1 2 30 40 50], а срез `z` — элементы [3 4 70].



Будьте очень внимательны, когда создаете срез на основе среза! При этом оба среза будут использовать общую область памяти, и изменение одного среза будет затрагивать и второй. Старайтесь не изменять срезы после создания среза на их основе или после их получения путем срезания. Чтобы функция `append` не могла использовать общую емкость срезов, используйте трехэлементное выражение среза.

Преобразование массивов в срезы

Срезанию могут подвергаться не только срезы. Если у вас есть массив, вы можете создать срез на его основе, используя для этого выражение среза. Это может быть удобным способом преобразования массива в функцию, которая принимает на вход только срезы. Но необходимо помнить о том, что в случае взятия среза в массиве новый срез будет точно так же использовать общую память, как и в случае взятия среза в срезе. Если вы запустите следующий код в онлайн-песочнице (<https://oreil.ly/kliaJ>):

```
x := [4]int{5, 6, 7, 8}
y := x[:2]
z := x[2:]
x[0] = 10
fmt.Println("x:", x)
fmt.Println("y:", y)
fmt.Println("z:", z)
```

на экран будет выведено следующее:

```
x: [10 6 7 8]
y: [10 6]
z: [7 8]
```

Функция `copy`

Если вам нужно создать срез так, чтобы он не зависел от исходного среза, используйте встроенную функцию `copy`. Рассмотрим следующий простой пример, который можно запустить в онлайн-песочнице (<https://oreil.ly/iIMNY>):

```
x := []int{1, 2, 3, 4}
y := make([]int, 4)
num := copy(y, x)
fmt.Println(y, num)
```

На экран будет выведено следующее:

```
[1 2 3 4] 4
```

Функция `copy` принимает два параметра: целевой срез в качестве первого параметра и исходный срез в качестве второго. Она копирует из исходного среза в целевой максимально возможное количество элементов, которое определяется размером меньшего среза, и возвращает это количество скопированных элементов. При этом не играет роли *емкость* срезов `x` и `y`: важность представляет лишь их длина.

Не обязательно копировать весь срез. Следующий код копирует в двухэлементный срез первые два элемента четырехэлементного среза:

```
x := []int{1, 2, 3, 4}
y := make([]int, 2)
num = copy(y, x)
```

Переменная `y` теперь содержит срез `[1 2]`, а переменная `num` — число 2.

Можно также скопировать элементы из середины исходного среза:

```
x := []int{1, 2, 3, 4}
y := make([]int, 2)
copy(y, x[2:])
```

Здесь мы копируем третий и четвертый элементы среза `x` путем взятия среза в этом срезе. Обратите также внимание, что результат функции `copy` здесь не присваивается переменной. Если не требуется количество скопированных элементов, то его можно и не присваивать.

Функция `copy` позволяет выполнять копирование между двумя срезами, которые охватывают пересекающиеся области базового среза:

```
x := []int{1, 2, 3, 4}
num = copy(x[:3], x[1:])
fmt.Println(x, num)
```

В данном случае последние три элемента среза `x` копируются в первые три элемента этого среза. На экран будет выведено `[2 3 4] 3`.

Функцию `copy` можно использовать и с массивами путем взятия среза в массиве. При этом массив может выступать и в качестве источника, и в качестве цели копирования. Попробуйте запустить в онлайн-песочнице следующий код (<https://oreil.ly/mhRW>):

```
x := []int{1, 2, 3, 4}
d := [4]int{5, 6, 7, 8}
y := make([]int, 2)
copy(y, d[:])
fmt.Println(y)
copy(d[:], x)
fmt.Println(d)
```

Первый вызов функции `copy` копирует в срез `y` первые два элемента массива `d`. Второй вызов этой функции копирует в массив `d` все элементы среза `x`. На экран будет выведено следующее:

```
[5 6]
[1 2 3 4]
```

Строки в сочетании с рунами и байтами

Теперь, когда вы уже познакомились со срезами, вернемся назад и еще раз поговорим о строках. Можно было бы предположить, что строки в Go состоят из рун, но в действительности это не так. На самом деле для представления строк в Go используются последовательности байтов. При этом не предъявляется требование о том, чтобы эти байты были символами в какой-либо конкретной кодировке, однако несколько библиотечных функций языка Go (а также цикл `for-range`, который мы рассмотрим в следующей главе) исходят из предположения, что строка состоит из кодовых точек кодировки UTF-8.



Согласно спецификации языка Go, исходный код программ на этом языке должен всегда записываться в кодировке UTF-8. За исключением шестнадцатеричных экранирующих последовательностей, строковые литералы также должны записываться в кодировке UTF-8.

В точности так же, как элемент извлекается из массива или среза, можно извлечь одно значение из строки, используя для этого *индексное выражение*:

```
var s string = "Hello there"
var b byte = s[6]
```

Как и в случае массивов и срезов, индексы строк отсчитываются от 0. В данном примере байту `b` присваивается значение седьмого элемента строки `s`, то есть `t`.

Со строками можно использовать и нотацию выражения среза, которую мы уже использовали с массивами и срезами:

```
var s string = "Hello there"
var s2 string = s[4:7]
var s3 string = s[:5]
var s4 string = s[6:]
```

В данном случае переменной `s2` присваивается строка `"o t"`, переменной `s3` — строка `"Hello"`, а переменной `s4` — строка `"there"`.

Хоть очень удобно, что Go позволяет использовать нотацию взятия среза для получения подстрок и индексную нотацию для извлечения отдельных элементов строки, и в том и в другом случае нужно быть очень внимательными. Поскольку строки являются неизменяемыми, они лишены проблем с модификацией элементов, свойственных срезам срезов. Однако в их случае имеется другая проблема: в то время как строка представляет собой последовательность байтов, размер кодовой точки в кодировке UTF-8 может составлять от одного до четырех байт. В нашем предыдущем примере все сработало, как ожидалось, потому что мы использовали исключительно кодовые точки кодировки UTF-8, длина которых равна одному байту. Однако при работе с текстами на других языках, помимо английского, или с символами эмоций вы будете иметь дело с кодовыми точками кодировки UTF-8, длина которых составляет больше одного байта:

```
var s string = "Hello 🌞 "  
var s2 string = s[4:7]  
var s3 string = s[:5]  
var s4 string = s[6:]
```

В данном примере переменной `s3`, как и раньше, присваивается строка "Hello". Переменной `s4` присваивается символ эмоций в виде солнца. Однако в переменную `s2` вместо строки "o 🌞" заносится строка "o 📀". Это объясняется тем, что мы копируем только первый байт символа эмоций, что дает некорректный результат.

Go позволяет нам узнать длину строки, передав ее встроенной функции `len`. Как вы уже знаете, и в случае индекса строк, и в случае выражения среза отсчет позиций производится в байтах, поэтому вряд ли стоит удивляться, что и длину строки эта функция измеряет в байтах, а не в кодовых точках:

```
var s string = "Hello 🌞 "  
fmt.Println(len(s))
```

Этот код выводит `10`, а не `7`, потому что для представления символа эмоций в виде улыбающегося солнца в кодировке UTF-8 требуется 4 байта.



Хоть язык Go и позволяет использовать со строками синтаксис взятия среза и обращения по индексу, эти возможности следует использовать только в том случае, когда точно известно, что строка содержит лишь однобайтовые символы.

Из-за этих сложных отношений между рунами, строками и байтами в Go возможны несколько интересных видов преобразования типов между этими типами. Таким образом одну руну или один байт можно преобразовать в строку:

```
var a rune    = 'x'
var s string  = string(a)
var b byte    = 'y'
var s2 string = string(b)
```



Распространенная ошибка начинающих Go-разработчиков — пытаться преобразовать число типа `int` в строку с помощью преобразования типов:

```
var x int = 65
var y = string(x)
fmt.Println(y)
```

В данном случае в переменную `y` заносится строка `A`, а не строка `65`. Начиная с версии Go 1.15, команда `go vet` блокирует преобразование в строку любого другого целочисленного типа, помимо типов `rune` и `byte`.

Вы можете преобразовать строку в срез байтов или срез рун и выполнить обратное преобразование. Попробуйте выполнить в онлайн-песочнице код из примера 3.9 (<https://oreil.ly/N7fOB>).

Пример 3.9. Преобразование строк в срезы

```
var s string = "Hello, 🌍"
var bs []byte = []byte(s)
var rs []rune = []rune(s)
fmt.Println(bs)
fmt.Println(rs)
```

Запустив этот код, вы увидите на экране следующее:

```
[72 101 108 108 111 44 32 240 159 140 158]
[72 101 108 108 111 44 32 127774]
```

В первой строке выведен результат преобразования исходной строки в байты в кодировке UTF-8. Во второй строке — результат преобразования исходной строки в руны.

Поскольку большинство данных в языке Go считывается и записывается как последовательность байтов, для строк наиболее часто используется преобразование строки в срез байтов и обратно. Срезы рун используются сравнительно редко.

Для извлечения из строки подстрок и кодовых точек рекомендуется использовать не выражения среза и индекса, а функции из пакетов `strings` и `unicode/utf8` стандартной библиотеки. В следующей главе вы узнаете, как можно использовать цикл `for-range` для обхода кодовых точек строки.

UTF-8

UTF-8 — это наиболее широко используемая кодировка для стандарта Unicode. В этом стандарте для представления каждой *кодовой точки*, то есть каждого символа или модификатора, используется 4 байта (32 бита). По этой причине простейший способ представления кодовых точек стандарта Unicode состоит в том, чтобы сохранять по 4 байта для каждой кодовой точки. Такой способ кодирования получил название UTF-32 и практически не используется из-за большого расхода места на диске и в памяти. В силу особенностей реализации стандарта Unicode, 11 из 32 бит всегда содержат нули. Еще одной распространенной кодировкой является UTF-16, в которой для представления каждой кодовой точки используется одна или две 16-битные (двухбайтовые) последовательности. Однако и этот подход является слишком расточительным, поскольку подавляющая часть создаваемого в мире контента использует кодовые точки, способные поместиться в одном байте. Здесь в игру вступает кодировка UTF-8.

Эта кодировка очень продуманна. Она использует один байт для представления символов стандарта Unicode со значениями не выше 128 (что включает в себя буквы, цифры и знаки пунктуации, используемые в английском языке), но расширяется до 4 байт, когда нужно представить кодовые точки стандарта Unicode с более высокими значениями. В результате этого UTF-8 занимает столько же места, сколько UTF-32, только в самом худшем случае. У этой кодировки есть и другие приятные особенности. В отличие от кодировок UTF-32 и UTF-16, вам не нужно беспокоиться о различиях между «младшеконечным» и «старшеконечным» форматами следования байтов. В кодировке UTF-8 также можно взглянуть на любой байт в последовательности и определить, в каком месте последовательности вы находитесь: в ее начале или где-то посередине. Это значит, что вы не сможете случайно прочитать символ неверным образом.

Единственным недостатком является невозможность произвольного доступа к строке, представленной в формате UTF-8. Хотя вы сможете определить, что символ находится где-то в середине строки, при этом нельзя будет сказать, сколько символов расположено перед ним. Для этого нужно посчитать эти символы, начав с начала строки. Язык Go не требует представления строки в кодировке UTF-8, но всячески это поддерживает. В следующих главах будет показано, как следует работать со строками в кодировке UTF-8.

Стоит отметить еще один интересный факт: кодировку UTF-8 в 1992 году создали Кен Томпсон (Ken Thompson) и Роб Пайк (Rob Pike), которые впоследствии создали и язык Go.

Карты

Срезы полезны при работе с последовательными данными. Как и большинство других языков, Go также предлагает встроенный тип данных, когда нужно связать одно значение с другим. Это карты, которые записываются так: `map[типКлюча]типЗначения`. Рассмотрим возможные способы объявления карт. Прежде всего вы можете использовать объявление с помощью ключевого слова `var`, чтобы создать переменную карты с нулевым значением:

```
var nilMap map[string]int
```

В этом случае объявляется карта `nilMap` с ключами типа `string` и значениями типа `int`. Нулевым значением карты является `nil`. Карта, равная `nil`, обладает нулевой длиной. Попытка чтения карты, равной `nil`, всегда возвращает нулевое значение того типа, к которому относятся значения карты. Однако попытка выполнить запись в переменную `nil` карты вызовет панику.

Объявление с помощью оператора `:=` позволяет создать переменную карты путем присвоения ей *литерала карты*:

```
totalWins := map[string]int{}
```

Здесь используется пустой литерал карты. Однако созданная таким образом карта отличается от карты, равной `nil`. Хотя ее длина тоже равна 0, для карты, созданной путем присвоения пустого литерала карты, допустимы операции чтения и записи. А вот как выглядит непустой литерал карты:

```
teams := map[string][]string {
    "Orcas": []string{"Fred", "Ralph", "Bijou"},
    "Lions": []string{"Sarah", "Peter", "Billie"},
    "Kittens": []string{"Waldo", "Raul", "Ze"},
}
```

В теле литерала карты записываются ключи и соответствующие значения, между которыми ставится знак двоеточия (`:`). После каждой пары «ключ — значение» ставится запятая, даже на последней строке. В данном примере каждое значение представляет собой срез строк. В качестве значений карты можно использовать значения любого типа. Некоторые ограничения накладываются только на тип ключей; мы обсудим это чуть позже.

Если заранее известно, сколько пар «ключ — значение» нужно будет разместить в карте, но нет информации, какими именно будут эти значения, можно создать карту с заданным исходным размером, используя функцию `make`:

```
ages := make(map[int][]string, 10)
```

Карты, созданные с помощью функции `make`, все равно обладают нулевой длиной и не ограничены в своем росте изначально указанным размером.

У карт есть много общего со срезами.

- Карты автоматически увеличиваются по мере добавления в них пар «ключ — значение».
- Если заранее известно, сколько пар «ключ — значение» нужно будет разместить в карте, можно создать карту с заданным исходным размером, используя функцию `make`.
- Вы можете узнать, сколько пар «ключ — значение» содержится в карте, передав ее функции `len`.
- Нулевым значением в случае карты является `nil`.
- Карты являются несравнимым типом. Вы можете убедиться, что карта не равна значению `nil`, но вы не проверите, содержат ли две карты одинаковые или разные ключи и значения, используя оператор `==` или оператор `!=` соответственно.

Ключ карты может быть любого сравнимого типа. Это значит, что в качестве ключей карты нельзя использовать срезы или карты.

Когда же лучше использовать карту, а когда — срез? Срезы следует использовать для списков данных, особенно когда эти данные обрабатываются последовательно. Карты удобно использовать при работе с данными, способ организации которых определяется значениями, не следующими в строгом порядке возрастания.



Используйте карту, когда порядок элементов не имеет значения. Используйте срез, когда порядок элементов играет важную роль.

ЧТО ТАКОЕ ХЕШ-КАРТА

В сфере компьютерных технологий под *картой* понимается структура данных, которая связывает (или, иначе говоря, сопоставляет) одно значение с другим. Существует несколько способов реализации карт, каждый из которых обладает своими преимуществами и недостатками. Используемые в Go карты представляют собой *хеш-карту* (или *хеш-таблицу*).

Если вы еще не знакомы с этой концепцией, вот краткое объяснение, что это такое.

Хеш-карта позволяет быстро извлекать значения по ключу. «За кулисами» это реализуется с помощью массива. При добавлении нового ключа и значения ключ преобразуется в число с помощью *алгоритма хеширования*. Эти числа могут повторяться, поскольку алгоритм хеширования может преобразовывать разные ключи в одно и то же число. Полученное таким образом число затем используется в качестве индекса массива. В качестве элементов массива выступают так называемые *ведра*. Затем производится сохранение пары «ключ — значение» в ведре. Если ведро уже содержит такой же ключ, то предыдущее значение заменяется новым.

Каждое ведро тоже является массивом и может содержать несколько значений. При так называемой *коллизии*, когда два ключа отображаются на одно и то же ведро, в этом ведре сохраняются обе пары ключей и значений.

Сходным образом осуществляется и чтение из хеш-карты. Для этого указанный ключ нужно преобразовать в число с помощью алгоритма хеширования, найти соответствующее ведро и, перебрав все ключи этого ведра, найти среди них ключ, совпадающий с указанным. При наличии такого ключа возвращается связанное с ним значение.

Количество коллизий не должно быть слишком большим, потому что чем больше будет коллизий, тем медленнее будет работать хеш-карта, поскольку при поиске нужного ключа приходится перебирать все ключи, отображенные на одно и то же ведро. Свести количество коллизий к минимуму можно путем использования хорошо продуманных алгоритмов хеширования. Они меняют размеры хеш-карты после добавления достаточного количества элементов, чтобы обеспечить равномерное заполнение ведер и сделать возможным добавление дополнительных элементов.

Хеш-карты — очень полезный инструмент, который трудно создать самим. Чтобы подробнее узнать, как они реализованы в Go, просмотрите видеозапись доклада по этой теме, представленного на конференции GopherCon 2016 (<https://oreil.ly/kIeJM>).

Go не требует и даже не позволяет вам определить собственный алгоритм хеширования или определение равенства. Вместо этого среда выполнения языка Go, которая включается в состав каждой компилируемой программы на этом языке, предоставляет реализацию алгоритмов хеширования для всех типов, выступающих в качестве ключей.

Чтение и запись карты

Рассмотрим небольшую программу, которая объявляет карту, а затем производит ее запись и чтение. Для этого запустите в онлайн-песочнице код из примера 3.10 (<https://oreil.ly/gBMvf>).

Пример 3.10. Использование карты

```
totalWins := map[string]int{}
totalWins["Orcas"] = 1
totalWins["Lions"] = 2
fmt.Println(totalWins["Orcas"])
fmt.Println(totalWins["Kittens"])
totalWins["Kittens"]++
fmt.Println(totalWins["Kittens"])
totalWins["Lions"] = 3
fmt.Println(totalWins["Lions"])
```

Запустив эту программу, вы увидите следующие результаты:

```
1
0
1
3
```

Чтобы присвоить ключу карты значение, нужно поместить ключ в квадратные скобки и указать присваиваемое значение, используя оператор `=`; чтобы прочесть присвоенное ключу значение, нужно поместить ключ в квадратные скобки. Обратите внимание, что для присваивания значения ключу карты нельзя использовать оператор `:=`.

Если вы попытаетесь прочесть значение ключа, которому еще не было присвоено значение, карта возвратит нулевое значение того типа, к которому относятся значения карты. В данном случае мы получили `0`, поскольку значения карты относятся к типу `int`. Используя оператор `++`, можно увеличить числовое значение, связанное с ключом карты. Поскольку карта по умолчанию возвращает свое нулевое значение, это работает даже в том случае, когда у ключа нет ассоциированного с ним значения.

Идиома «запятая-ок»

Как вы уже видели, при запросе значения, ассоциированного с ключом, которого еще нет в карте, карта возвращает нулевое значение. Это удобно в тех случаях, когда нужно реализовать что-то наподобие показанного выше счетчика. Однако иногда сначала нужно выяснить, содержит ли карта опре-

деленный ключ. Для таких случаев в Go есть *идиома «запятая-ок»*, которая позволяет отличить ключ, с которым связано нулевое значение, от ключа, отсутствующего в карте:

```
m := map[string]int{
    "hello": 5,
    "world": 0,
}
v, ok := m["hello"]
fmt.Println(v, ok)

v, ok = m["world"]
fmt.Println(v, ok)

v, ok = m["goodbye"]
fmt.Println(v, ok)
```

С помощью идиомы «запятая-ок» результаты чтения карты присваиваются не одной, а двум переменным. В первую переменную заносится связанное с ключом значение. Во вторую переменную, которую принято называть *ok*, заносится второе возвращаемое значение булева типа. Если переменная *ok* равна *true*, то этот ключ присутствует в карте. Если переменная *ok* равна *false*, то этого ключа в карте нет. В данном случае на экран будет выведено *5 true, 0 true и 0 false*.



Идиома «запятая-ок» используется в Go в тех случаях, когда требуется провести разницу между считыванием значения и возвращением нулевого значения. Мы еще встретимся с ней, когда будем обсуждать чтение из каналов в главе 10 и использование утверждений типа в главе 7.

Удаление из карты

Для удаления из карты пар «ключ — значение» используется встроенная функция *delete*:

```
m := map[string]int{
    "hello": 5,
    "world": 10,
}
delete(m, "hello")
```

Функция *delete* принимает на вход карту и ключ и удаляет пару «ключ — значение» с указанным ключом. Если указанного ключа нет в карте или если карта равна *nil*, то ничего не происходит. Функция *delete* не возвращает значение.

Использование карты в качестве множества

В стандартной библиотеке многих языков имеется такой тип данных, как *множество* (*set*). Множество обеспечивает неповторяемость элементов, не давая никаких гарантий в отношении порядка их расположения. Проверка наличия определенного элемента во множестве осуществляется очень быстро, независимо от количества содержащихся в нем элементов. (В срезе такая проверка начинает занимать больше времени по мере увеличения количества элементов.)

В языке Go нет множеств, но вы можете имитировать некоторые их свойства с помощью карт. Создайте карту с ключами того типа, к которому должны относиться элементы множества, и значениями типа `bool`. Этот подход демонстрирует код из примера 3.11; вы можете запустить его в онлайн-песочнице (<https://oreil.ly/wC6XK>).

Пример 3.11. Использование карты в качестве множества

```
intSet := map[int]bool{}
vals := []int{5, 10, 2, 5, 8, 7, 3, 9, 1, 2, 10}
for _, v := range vals {
    intSet[v] = true
}
fmt.Println(len(vals), len(intSet))
fmt.Println(intSet[5])
fmt.Println(intSet[500])
if intSet[100] {
    fmt.Println("100 is in the set")
}
```

Поскольку нам требуется множество элементов типа `int`, мы создаем карту с ключами типа `int` и значениями типа `bool`. Затем мы перебираем значения среза `vals` с помощью цикла `for-range` (о котором мы поговорим в подразделе «Оператор `for-range`» на с. 99) и помещаем их в карту `intSet`, сопоставляя с каждым значением `int` булево значение `true`.

После того как мы записали в карту `intSet` 11 значений, ее длина стала равной 8, поскольку карты не допускают дублирования ключей. Если мы обратимся к карте `intSet`, используя ключ 5, она вернет значение `true`, поскольку у нас есть ключ 5. Но при попытке обратиться к ней с ключом 500 или 100 она вернет значение `false`. Это объясняется тем, что у карты `intSet` нет таких ключей, и потому она возвращает нулевое значение того типа, к которому относятся ее значения, то есть `false` в случае типа `bool`.

Если вам нужны множества с поддержкой операций объединения, пересечения и вычитания, то вы можете либо воспользоваться одной из множества сторон-

них библиотек, предлагающих эту функциональность, либо реализовать ее самостоятельно. (Использование сторонних библиотек мы подробно обсудим в главе 9.)



При реализации множества с помощью карты некоторые разработчики предпочитают использовать в качестве значений пустую структуру: `struct{}`. (Что такое структуры, вы узнаете в следующем разделе.) Преимущество такого подхода в том, что пустая структура не занимает ни одного байта, в то время как булево значение занимает один байт.

Недостатком является то, что выражение `struct{}` делает код более громоздким. Присваивание становится менее очевидным, и для проверки наличия значения в множестве приходится использовать идиому «запятая-ok»:

```
intSet := map[int]struct{}{}
vals := []int{5, 10, 2, 5, 8, 7, 3, 9, 1, 2, 10}
for _, v := range vals {
    intSet[v] = struct{}
}
if _, ok := intSet[5]; ok {
    fmt.Println("5 is in the set")
}
```

Если речь не идет о множествах очень большого размера, то разница в объеме занимаемой памяти обычно не настолько значительна для того, чтобы перевешивать эти недостатки.

Структуры

Представляя собой удобный способ сохранения некоторых видов данных, карты в то же время обладают рядом ограничений. Они не позволяют определить API, поскольку карта не может быть настроена так, чтобы допускать только определенные ключи. Кроме того, все значения карты должны относиться к одному и тому же типу. Из-за этого карты не очень подходят для того, чтобы передавать данные из одной функции в другую. Когда требуется сгруппировать некоторые взаимосвязанные данные, следует определять *структуру*.



Если вы уже знакомы с каким-либо объектно-ориентированным языком, то вас, возможно, интересует, чем структуры отличаются от классов. На это можно дать очень простой ответ: в языке Go нет классов, потому что в нем нет наследования. В то же время некоторые возможности объектно-ориентированных языков есть и в Go, только реализуются немного по-другому. Эти объектно-ориентированные возможности будут подробно рассмотрены в главе 7.

Поскольку подобная концепция принята во многих языках программирования, используемый в Go синтаксис чтения и записи структур не должен выглядеть для вас чем-то абсолютно новым:

```
type person struct {
    name string
    age  int
    pet  string
}
```

Определение структурного типа включает в себя ключевое слово `type`, имя структурного типа, ключевое слово `struct` и пару фигурных скобок (`{}`). Внутри фигурных скобок перечисляются поля структуры. Подобно тому как в объявлении `var` сначала указывается имя переменной, а затем — ее тип, здесь тоже сначала указывается имя поля структуры, а затем — его тип. Обратите также внимание, что, в отличие от литералов карты, в объявлении структуры поля не разделяются запятыми. Структурный тип можно определить внутри или за пределами функции. Если структурный тип задан внутри функции, то его можно использовать только в ее пределах. (Подробнее о функциях мы поговорим в главе 5.)



Строго говоря, область видимости определения структуры может быть ограничена до любого уровня блоков. Подробнее о блоках будет рассказано в главе 4.

После объявления структурного типа мы можем определить переменные этого типа:

```
var fred person
```

В данном случае используется объявление `var`. Поскольку значение не присваивается переменной `fred`, она получает нулевое значение для структурного типа `person`. У структуры, равной нулевому значению, каждое поле содержит нулевое значение того типа, к которому относится это поле.

Вы также можете присвоить переменной *литерал структуры*.

```
bob := person{}
```

В отличие от карт, в структурах нет никакой разницы между присвоением переменной пустого литерала структуры и объявлением переменной без присвоения значения. И в том и в другом случае все поля структуры будут инициализированы нулевыми значениями соответствующего типа. В случае непустого литерала структуры можно использовать два стиля записи. Первый стиль сводится

к тому, чтобы перечислить внутри фигурных скобок значения полей, разделив их запятыми:

```
julia := person{
    "Julia",
    40,
    "cat",
}
```

При использовании такого формата литерала структуры необходимо указывать значения всех полей структуры в том же порядке, в каком они объявляются в определении структуры.

Второй стиль записи литерала структуры выглядит так же, как и стиль записи литерала карты:

```
beth := person{
    age: 30,
    name: "Beth",
}
```

Имена полей в структуре используются для указания их значений. Применяя этот стиль, можно не указывать значения некоторых полей и перечислять поля в любом порядке. Всем неуказанным полям будет присвоено нулевое значение соответствующего типа. Эти два стиля записи литерала структуры нельзя сочетать друг с другом: либо все поля должны указываться с ключами, либо ни одно из них. В случае небольших структур, у которых всегда указываются имена всех полей, будет вполне уместным более простой стиль записи. Во всех остальных случаях лучше использовать имена ключей. Хотя этот стиль более многословен, он позволяет четко указать, какое значение присваивается какому полю, без необходимости сверяться с определением структуры. Кроме того, литерал структуры в таком формате проще поддерживать. Если вы будете инициализировать структуру, не используя имена полей, то добавление в структуру дополнительных полей в одной из новых версий программы приведет к ошибке при компиляции.

Для доступа к полям структуры используется точечная нотация:

```
bob.name = "Bob"
fmt.Println(beth.name)
```

Точечная нотация используется для чтения и записи полей структуры в точности таким же образом, как квадратные скобки применяются для чтения и записи значений карты.

Анонимные структуры

Вы также можете объявить, что переменная реализует структурный тип, без предварительного присвоения имени этому структурному типу. Такие структуры называют *анонимными*:

```
var person struct {
    name string
    age  int
    pet  string
}

person.name = "bob"
person.age = 50
person.pet = "dog"

pet := struct {
    name string
    kind string
}{
    name: "Fido",
    kind: "dog",
}
```

В данном примере переменные `person` и `pet` относятся к анонимному структурному типу. Присвоение значений полям анонимной структуры (и их чтение) выполняется точно так же, как и в случае именованного структурного типа. Подобно тому как экземпляр именованной структуры можно инициализировать с помощью литерала структуры, то же самое можно сделать и в случае анонимной структуры.

Здесь может возникнуть вопрос: зачем может понадобиться тип данных, связанный только с одним экземпляром? Анонимные структуры удобно использовать в двух распространенных случаях. Первым случаем является преобразование внешних данных в структуру или, наоборот, структуры во внешние данные (например, данные в формате JSON или буферы протоколов). Эти виды преобразований называют *демаршаллингом* и *маршаллингом* данных. Мы подробно поговорим об их использовании в разделе «Пакет `encoding/json`» на с. 295.

Второй областью применения анонимных структур является написание тестов. Мы будем использовать срез анонимных структур при написании табличных тестов в главе 13.

Сравнение и преобразование структур

Структурный тип может быть сравниваемым или несравниваемым в зависимости от того, к каким типам относятся его поля. Если все поля структуры относятся к сравниваемым типам, то и сама структура является сравниваемой. Если же в качестве некоторых полей используются срезы или карты (или функции и каналы, как мы увидим в последующих главах), то такая структура является несравниваемой.

В отличие от таких языков, как Python и Ruby, в Go нет «магического» метода, переопределив который можно было бы заставить работать операторы `==` и `!=` для несравниваемых структур. Хотя, конечно, ничто не мешает вам написать собственную функцию и сравнивать структуры с ее помощью.

Подобно тому как в Go нельзя сравнивать переменные, относящиеся к разным простым типам, в этом языке невозможно сравнивать и переменные, относящиеся к разным структурным типам. В то же время в Go доступно преобразование из одного структурного типа в другой, *если поля обеих структур обладают одинаковыми именами и типами и расположены в том же порядке*. Посмотрим, что это означает на практике. Например, у нас есть следующая структура:

```
type firstPerson struct {  
    name string  
    age  int  
}
```

Мы можем преобразовать экземпляр типа `firstPerson` в экземпляр типа `secondPerson`, используя преобразование типов, но не можем сравнить экземпляр типа `firstPerson` с экземпляром типа `secondPerson`, используя оператор `==`, поскольку они относятся к разным типам:

```
type secondPerson struct {  
    name string  
    age  int  
}
```

Невозможно преобразовать экземпляр типа `firstPerson` в экземпляр типа `thirdPerson`, поскольку поля этих структур расположены в разном порядке:

```
type thirdPerson struct {  
    age  int  
    name string  
}
```

Мы не можем преобразовать экземпляр типа `firstPerson` в экземпляр типа `fourthPerson`, поскольку у этих структур не совпадают имена полей:

```
type fourthPerson struct {  
    firstName string
```

```
    age      int
}
```

Наконец, мы не преобразуем экземпляр типа `firstPerson` в экземпляр типа `fifthPerson`, поскольку у второй структуры есть дополнительное поле:

```
type fifthPerson struct {
    name      string
    age       int
    favoriteColor string
}
```

У анонимных структур здесь имеется небольшая дополнительная особенность: если из двух переменных структурного типа как минимум одна относится к анонимному структурному типу, то их можно сравнивать без преобразования типов, если поля обеих структур обладают одинаковыми именами и типами и расположены в том же порядке. Вы также можете выполнять присваивание между переменными именованного и анонимного структурных типов, если поля обеих структур обладают одинаковыми именами и типами и расположены в том же порядке.

```
type firstPerson struct {
    name string
    age  int
}
f := firstPerson{
    name: "Bob",
    age:  50,
}
var g struct {
    name string
    age  int
}
```

```
// компилируется без проблем — можно использовать операторы = и ==
// между одинаковыми именованными и анонимными структурами
g = f
fmt.Println(f == g)
```

Резюме

В этой главе вы многое узнали об используемых в Go контейнерных типах, получили больше информации о строках и о том, как использовать встроенные общие типы контейнеров, срезы и карты. Вы также научились создавать собственные составные типы с помощью структур. В следующей главе мы рассмотрим управляющие конструкции языка Go: операторы `for`, `if/else` и `switch`. Вы узнаете о том, как в Go производится организация кода в блоки и как наличие нескольких уровней блоков может приводить к неожиданным результатам.

ГЛАВА 4

Блоки, затенение переменных и управляющие конструкции

Теперь, когда мы уже рассмотрели переменные, константы и встроенные типы, пора перейти к рассмотрению программной логики и способов организации кода. Сначала вы узнаете, что собой представляют блоки и как они влияют на доступность идентификаторов. После этого мы рассмотрим управляющие конструкции языка Go, а именно операторы `if`, `for` и `switch`. Наконец, мы поговорим об операторе `goto` и о том единственном случае, в котором его следует использовать.

Блоки

Go позволяет вам объявлять переменные в разных местах: их можно объявлять вне функций, в качестве параметров функции или в качестве локальной переменной внутри функции.



До сих пор мы использовали только функцию `main`, но в следующей главе начнем использовать функции с параметрами.

Каждое из тех мест, в которых мы размещаем то или иное объявление, называется *блоком*. При объявлении переменных, констант, типов и функций вне какой-либо функции они размещаются в *блоке пакета*. В своих программах мы использовали операторы `import` для доступа к функциям вывода на экран и математическим функциям (подробнее о которых будет рассказано в главе 9). Операторы `import` определяют, имена каких других пакетов допускается использовать внутри содержащего их файла. Эти имена находятся в *блоке файлов*. Все переменные, определяемые на верхнем уровне функции (включая параметры функции), находятся в отдельном блоке. Внутри функции каждая пара фигур-

ных скобок ({}) определяет дополнительный блок, и, как мы вскоре увидим, управляющие конструкции языка Go также определяют собственные блоки.

К идентификатору, определенному в любом внешнем блоке, можно получить доступ из любого внутреннего блока. Это порождает следующий вопрос: что произойдет, если во вложенном блоке будет объявлен идентификатор с таким же именем, как и во внешнем блоке? Это приведет к *затенению* идентификатора, созданного во внешнем блоке.

Затенение переменных

Прежде чем приступить к разговору о том, что такое затенение, рассмотрим небольшой пример кода (пример 4.1). Вы можете запустить его в онлайн-песочнице (<https://oreil.ly/50t6b>).

Пример 4.1. Затенение переменных

```
func main() {
    x := 10
    if x > 5 {
        fmt.Println(x)
        x := 5
        fmt.Println(x)
    }
    fmt.Println(x)
}
```

Перед тем как запускать этот код, попробуйте догадаться, что он выведет на экран:

- не выведет ничего, поскольку не сможет успешно скомпилироваться;
- 10 в первой строке, 5 во второй строке и 5 в третьей строке;
- 10 в первой строке, 5 во второй строке и 10 в третьей строке.

На самом деле этот код выведет следующее:

```
10
5
10
```

Переменная является затеняющей, если ее имя совпадает с именем переменной, определенной во вещающем блоке. При наличии затеняющей переменной вы не можете получить доступ к затененной переменной.

В данном случае мы, очевидно, не собирались создавать новую переменную `x` внутри оператора `if`. Вместо этого мы хотели присвоить 5 переменной `x`,

объявленной на верхнем уровне блока функции. При первом вызове функции `fmt.Println` внутри оператора `if` еще есть возможность получить доступ к переменной `x`, объявленной на верхнем уровне блока функции. Однако в следующей строке переменная `x` *затеняется путем объявления новой переменной с таким же именем* внутри блока, образованного телом оператора `if`. При втором вызове функции `fmt.Println` обращение к переменной с именем `x` выводит затеняющую переменную, которая содержит значение 5. Закрывающая фигурная скобка тела оператора `if` завершает блок, в котором присутствует затеняющая переменная `x`, и поэтому при третьем вызове функции `fmt.Println` обращение к переменной с именем `x` выводит переменную, объявленную на верхнем уровне блока функции и содержащую значение 10. Обратите внимание, что эта переменная `x` никуда не исчезла и не получила новое значение: мы просто не могли получить к ней доступ, поскольку она была затенена во внутреннем блоке.

В предыдущей главе я говорил о том, что стараюсь не использовать оператор `:=` в тех случаях, когда это может привести к неясности в отношении того, какая именно переменная применяется. Это объясняется тем, что при использовании оператора `:=` легко случайно затенить переменную. Как вы помните, с помощью оператора `:=` можно создавать сразу несколько переменных с присвоением значения. Кроме того, оператор `:=` применяется даже в том случае, когда не все переменные слева от него являются новыми. Достаточно, чтобы хотя бы одна из указанных слева переменных была новой. Рассмотрим еще одну программу (пример 4.2), которую вы можете запустить в онлайн-песочнице (https://oreil.ly/U_m4B).

Пример 4.2. Затенение в случае присвоения нескольких значений

```
func main() {
    x := 10
    if x > 5 {
        x, y := 5, 20
        fmt.Println(x, y)
    }
    fmt.Println(x)
}
```

Запустив этот код, вы получите следующие результаты:

```
5 20
10
```

Переменная `x` затеняется внутри оператора `if`, несмотря на наличие определения переменной `x` во внешнем блоке. Это объясняется тем, что оператор `:=` заново использует переменные, объявляемые в текущем блоке. Поэтому при применении оператора `:=` следует убедиться в том, что слева от него не указаны

переменные, объявленные во внешней области видимости, если у вас нет намерения их затенить.

Также нужно проследить за тем, чтобы не был затенен импорт пакета. Об импорте пакетов мы подробно поговорим в главе 9, однако уже сейчас воспользуемся пакетом `fmt` для вывода на экран результатов своих программ. Посмотрим, что произойдет, если мы объявим переменную с именем `fmt` внутри функции `main`, как показано в примере 4.3. Попробуйте запустить этот код в онлайн-песочнице (<https://oreil.ly/CKQvm>).

Пример 4.3. Затенение имен пакетов

```
func main() {
    x := 10
    fmt.Println(x)
    fmt := "oops"
    fmt.Println(fmt)
}
```

Попытавшись выполнить этот код, мы получим сообщение об ошибке:

```
fmt.Println undefined (type string has no field or method Println)
```

Обратите внимание, что проблема состоит не в присвоении переменной имени `fmt`, а в попытке обращения к тому, чего нет у локальной переменной `fmt`. Как только локальная переменная `fmt` объявлена, она затеняет пакет с именем `fmt` в блоке файлов, делая невозможным использование этого пакета в оставшейся части функции `main`.

Выявление затененных переменных

Учитывая то, насколько трудноуловимыми могут быть вносимые затенением ошибки, будет очень полезно убедиться в отсутствии затененных переменных в ваших программах. Хотя ни команда `go vet`, ни инструмент `golangci-lint` не предлагают вам инструментов для выявления затенения, вы можете включить это выявление в свой процесс компиляции, установив на своей машине линтер `shadow`:

```
$ go install golang.org/x/tools/go/analysis/passes/shadow/cmd/shadow@latest
```

При компиляции с помощью `make`-файла включите линтер `shadow` в перечень задач цели `vet`:

```
vet:
    go vet ./...
    shadow ./...
.PHONY:vet
```

Если вы попытаете выполнить команду `make vet` для приведенного выше кода, то увидите, что затененная переменная будет обнаружена:

```
declaration of "x" shadows declaration at line 6
```

ВСЕОБЩИЙ БЛОК

На самом деле существует еще одна, немного странная разновидность блоков: «всеобщий блок» (universe block). Как вы помните, Go — небольшой язык, в котором имеется лишь 25 ключевых слов. Что интересно, в этот список не входят встроенные типы (такие как `int` и `string`), константы (`true` и `false`), функции (`make` и `close`) и значение `nil`. В таком случае где же они?

Все это считается в Go не ключевыми словами, а *предопределенными идентификаторами* и определено во всеобщем блоке, включающем в себя все остальные блоки.

Тот факт, что эти имена объявлены во всеобщем блоке, означает, что их можно затенить в других областях видимости. Как это может происходить, можно увидеть, запустив в онлайн-песочнице код из примера 4.4 (<https://oreil.ly/eoU2A>).

Пример 4.4. Затенение значения true

```
fmt.Println(true)
true := 10
fmt.Println(true)
```

Запустив этот код, вы увидите на экране следующее:

```
true
10
```

Никогда не допускайте переопределения идентификаторов, определенных во всеобщем блоке! Если вы случайно это сделаете, ваш код будет вести себя совсем не так, как вы ожидали. В лучшем случае это приведет к ошибкам на этапе компиляции. В более тяжелом случае вам придется долго выискивать источник своих проблем.

Если вы подумали, что ошибки со столь серьезными последствиями должны выявляться средствами статического анализа (линтерами), то хочу вам сообщить, что, как ни удивительно, они этого не делают. Даже линтер `shadow` не выявляет затенение идентификаторов, объявленных во всеобщем блоке.

Оператор if

Оператор `if` в языке Go ведет себя практически так же, как в других языках программирования. Учитывая то, насколько общеизвестным является этот оператор, я уже использовал его в предыдущих примерах кода, не волнуясь о том, что кто-то не поймет его назначение. Пример 4.5 демонстрирует более полный образец его использования.

Пример 4.5. Оператор `if` в сочетании с оператором `else`

```
n := rand.Intn(10)
if n == 0 {
    fmt.Println("That's too low")
} else if n > 5 {
    fmt.Println("That's too big:", n)
} else {
    fmt.Println("That's a good number:", n)
}
```



Запустив этот код, вы увидите, что он всегда присваивает переменной `n` единицу. Это объясняется тем, что в пакете `math/rand` жестко задано используемое по умолчанию начальное значение для генерирования случайных чисел. В подразделе «Переопределение имени пакета» на с. 227 вы узнаете, как можно задать надлежащее начальное значение для генерирования случайных чисел, а также как следует устранять конфликты между именами пакетов.

Наиболее заметное отличие оператора `if` языка Go от других языков состоит в том, что здесь не нужно заключать условие в круглые скобки. Однако у оператора `if` в Go есть еще одна особенность, которая позволяет лучше управлять переменными.

Как уже говорилось в разделе, посвященном затенению переменных, любая переменная, объявленная внутри фигурных скобок оператора `if` или `else`, существует только внутри этого блока. В этом нет ничего необычного; так же обстоит дело и в большинстве других языков. Однако, в отличие от других языков, Go также позволяет объявить переменные, область видимости которых будет включать себя условие и блоки операторов `if` и `else`. Посмотрим, как будет выглядеть предыдущий пример, если мы перепишем его, используя эту возможность (пример 4.6).

Пример 4.6. Объявление переменной внутри оператора `if`

```
if n := rand.Intn(10); n == 0 {
    fmt.Println("That's too low")
}
```

```
} else if n > 5 {
    fmt.Println("That's too big:", n)
} else {
    fmt.Println("That's a good number:", n)
}
```

Наличие этой особой области видимости может быть очень полезным. Это позволяет создавать переменные, которые будут доступны только там, где они нужны. В коде, следующем за цепочкой операторов `if/else`, переменная `n` становится неопределенной. Вы можете убедиться в этом, запустив в онлайн-песочнице код из примера 4.7 (<https://oreil.ly/rz671>).

Пример 4.7. Попытка обращения к переменной за пределами области видимости

```
if n := rand.Intn(10); n == 0 {
    fmt.Println("That's too low")
} else if n > 5 {
    fmt.Println("That's too big:", n)
} else {
    fmt.Println("That's a good number:", n)
}
fmt.Println(n)
```

Попытка выполнить этот код приведет к ошибке компиляции:

undefined: n



В принципе, перед условием оператора `if` можно разместить любой простой оператор, включая такие вещи, как вызов не возвращающей значение функции или присвоение нового значения существующей переменной. Однако так поступать не стоит. Во избежание путаницы используйте эту возможность только для определения новых переменных, область видимости которых будет ограничиваться операторами `if/else`.

Также не забывайте о том, что, как и в любом другом блоке, переменная, объявленная внутри оператора `if`, будет затенять переменные с таким же именем, объявленные в других блоках.

Четыре вида оператора `for`

Как и другие С-подобные языки, Go использует для организации циклов оператор `for`. Однако, в отличие от других языков, в Go применение оператора `for` является *единственным* способом организации циклов. Это обеспечивается за счет использования четырех разновидностей оператора `for`.

- Полная форма оператора for в стиле языка C.
- Оператор for, использующий только условие.
- Бесконечная форма оператора for.
- Оператор for-range.

Полный оператор for

Прежде всего рассмотрим полную форму оператора for, которую вы уже могли видеть в таких языках, как C, Java или JavaScript (пример 4.8).

Пример 4.8. Полный оператор for

```
for i := 0; i < 10; i++ {  
    fmt.Println(i)  
}
```

Вы, вероятно, уже догадались, что эта программа выведет на экран числа от 0 до 9 включительно.

Как и в случае оператора if, выражения оператора for в Go не нужно заключать в круглые скобки. В остальном это совершенно обычный оператор for. Он содержит три выражения, отделенные друг от друга символами точки с запятой. Первое выражение представляет собой выражение инициализации, которое присваивает значения одной или нескольким переменным перед выполнением цикла. В отношении этого выражения следует отметить два важных момента. Во-первых, для инициализации переменных здесь *в обязательном порядке* используется оператор :=; объявление переменных с помощью ключевого слова var здесь *не допускается*. Во-вторых, как и в случае объявления переменных в операторе if, здесь тоже можно затенить переменные.

Второе выражение представляет собой выражение сравнения, которое должно давать в результате логическое значение. Результат этого выражения проверяется непосредственно *перед* выполнением тела цикла, после выполнения инициализации и после выполнения каждой итерации. Тело цикла выполняется, если результат этого выражения равен true.

Последнее выражение стандартного оператора for представляет собой выражение инкремента. Обычно здесь стоит что-то вроде i++, но вы можете использовать здесь любую операцию присваивания. Это выражение выполняется непосредственно после каждой итерации цикла, перед проверкой условия.

Оператор `for`, использующий только условие

Go позволяет опустить в операторе `for` и выражение инициализации, и выражение инкремента. В результате остается оператор `for`, работающий так же, как `while` в C, Java, JavaScript, Python, Ruby и многих других языках. Как выглядит такой оператор, показано в примере 4.9.

Пример 4.9. Оператор `for`, использующий только условие

```
i := 1
for i < 100 {
    fmt.Println(i)
    i = i * 2
}
```

Бесконечный оператор `for`

В третьей разновидности оператора `for` убрано условие. Go позволяет использовать цикл `for`, способный выполняться бесконечно. Если вы учились программировать в 1980-е годы, то ваша первая программа, вероятно, представляла собой бесконечный цикл на языке BASIC, который непрерывно выводил на экран слово HELLO:

```
10 PRINT "HELLO"
20 GOTO 10
```

Пример 4.10 показывает, как будет выглядеть Go-версия такой программы. Попробуйте запустить этот код на своей машине или в онлайн-песочнице (<https://oreil.ly/whOi->).

Пример 4.10. Старый добрый бесконечный цикл

```
package main

import "fmt"

func main() {
    for {
        fmt.Println("Hello")
    }
}
```

Запустив эту программу, вы увидите те же бесконечные строки, которыми когда-то заполняли экраны миллионы компьютеров Commodore 64 и Apple:

```
Hello
Hello
```



```
Hello
Hello
Hello
Hello
Hello
...
```

Вполне насладившись этим путешествием в прошлое, нажмите сочетание клавиш Ctrl+C.



Если вы запустите эту программу в онлайн-песочнице, то ее выполнение будет прекращено через несколько секунд. Представляя собой совместно используемый ресурс, онлайн-песочница не позволяет ни одной программе выполняться слишком долго.

Ключевые слова `break` и `continue`

Каким же образом можно выйти из бесконечного цикла `for`, не используя клавиатуру и не выключая компьютер? Это можно сделать с помощью оператора `break`. Как и в других языках, он позволяет немедленно выйти из цикла. Также надо сказать, что оператор `break` можно использовать не только в бесконечной, но и в других разновидностях оператора `for`.



В Go нет аналога для ключевого слова `do`, существующего в Java, C и JavaScript. Если вам нужно, чтобы цикл выполнялся как минимум один раз, то самый «чистый» способ это сделать состоит в том, чтобы использовать бесконечный цикл `for`, в конце которого стоит оператор `if`. Например, если у вас есть Java-код с циклом `do/while` следующего вида:

```
do {
    // действия, выполняемые в цикле
} while (CONDITION);
```

то его Go-версия будет выглядеть так:

```
for {
    // действия, выполняемые в цикле
    if !CONDITION {
        break
    }
}
```

Обратите внимание, что перед условием поставлен знак «!», чтобы инвертировать условие, используемое в Java-коде. В Go-коде указывается условие *выхода из цикла*, в то время как в Java-коде указывается условие выполнения цикла.

В языке Go также есть ключевое слово `continue`, которое позволяет сразу перейти к следующей итерации, не выполняя оставшуюся часть тела цикла `for`. В принципе, можно обойтись и без оператора `continue`, организовав цикл так, как показано в примере 4.11.

Пример 4.11. Цикл, в котором трудно разобраться

```
for i := 1; i <= 100; i++ {
    if i%3 == 0 {
        if i%5 == 0 {
            fmt.Println("FizzBuzz")
        } else {
            fmt.Println("Fizz")
        }
    } else if i%5 == 0 {
        fmt.Println("Buzz")
    } else {
        fmt.Println(i)
    }
}
```

Однако такой подход считается в Go неидиоматическим. В Go рекомендуется использовать операторы `if` с небольшим телом и минимальным отступом вправо. Вложенный код труден для понимания, и использование оператора `continue` позволяет сделать его более понятным. Пример 4.12 показывает, как будет выглядеть код из предыдущего примера, если мы его перепишем, используя оператор `continue`.

Пример 4.12. Приведение кода к более понятному виду с помощью оператора `continue`

```
for i := 1; i <= 100; i++ {
    if i%3 == 0 && i%5 == 0 {
        fmt.Println("FizzBuzz")
        continue
    }
    if i%3 == 0 {
        fmt.Println("Fizz")
        continue
    }
    if i%5 == 0 {
        fmt.Println("Buzz")
        continue
    }
    fmt.Println(i)
}
```

Как видите, заменив операторы `if/else` на операторы `if`, использующие оператор `continue`, мы смогли выстроить условия в один ряд. Такое, более удачное расположение условий облегчило код для чтения и понимания.

Оператор for-range

Четвертой разновидностью оператора `for` является оператор `for`, выполняющий обход элементов одного из встроенных типов языка Go. Это оператор `for-range`, который напоминает имеющиеся в других языках итераторы. В данном разделе мы рассмотрим способы использования цикла `for-range` для обхода строк, массивов, срезов и карт. При обсуждении каналов в главе 10 вы также узнаете, как циклы `for-range` используются для работы с каналами.



Цикл `for-range` можно использовать только для обхода встроенных составных типов или основанных на них пользовательских типов.

Сначала посмотрим, как цикл `for-range` можно использовать для обхода срезов. Запустите в онлайн-песочнице код из примера 4.13 (<https://oreil.ly/XwuTL>).

Пример 4.13. Цикл `for-range`

```
evenVals := []int{2, 4, 6, 8, 10, 12}
for i, v := range evenVals {
    fmt.Println(i, v)
}
```

Этот код выведет на экран следующее:

```
0 2
1 4
2 6
3 8
4 10
5 12
```

Интересной особенностью цикла `for-range` является то, что вы получаете две переменные цикла. Первая переменная содержит текущую позицию в той структуре данных, которую вы обходите, а вторая — значение в этой позиции. Выбор идиоматических имен для этих переменных зависит от того, что именно обрабатывается в цикле. В случае массива, среза или строки *индексы* принято обозначать буквой `i`. При обработке карты вместо нее используется буква `k`, которая подразумевает *ключи*.

Второй переменной обычно дают имя `v`, что означает *value* — «значение», однако иногда ей присваивают имя, основанное на типе перебираемых значений. В то же время ничто не мешает вам назвать эти переменные как угодно иначе. Если тело цикла содержит небольшое количество операторов, этим переменным

можно присвоить однобуквенные имена. Для более длинных (или вложенных) циклов лучше использовать более описательные имена.

Но что, если вам не нужно применять ключи в цикле `for-range`? Как вы помните, Go требует, чтобы вы использовали все объявленные переменные, и это правило касается и тех переменных, которые объявляются в цикле `for`. Если вам не нужно использовать ключи, поставьте вместо имени переменной символ подчеркивания (`_`). Это укажет Go проигнорировать значение. Перепишем наш код для обхода среза таким образом, чтобы он не выводил индексы элементов. Запустите в онлайн-песочнице код из примера 4.14 (<https://oreil.ly/2fO12>).

Пример 4.14. Игнорирование ключей в цикле `for-range`

```
evenVals := []int{2, 4, 6, 8, 10, 12}
for _, v := range evenVals {
    fmt.Println(v)
}
```

Этот код выведет на экран следующее:

```
2
4
6
8
10
12
```



Всякий раз, когда вам нужно проигнорировать возвращаемое значение, скрывайте его с помощью символа подчеркивания. Вы еще не раз увидите примеры использования этого символа, когда мы будем обсуждать функции в главе 5 и пакеты в главе 9.

А что, если вам требуются ключи, но не нужны значения элементов? В таком случае Go позволяет просто опустить вторую переменную. Вот пример вполне допустимого Go-кода:

```
uniqueNames := map[string]bool{"Fred": true, "Raul": true, "Wilma": true}
for k := range uniqueNames {
    fmt.Println(k)
}
```

Чаще всего перебор ключей применяется при использовании карты в качестве множества. В таком случае значения элементов не играют большой роли. Однако эти значения также можно опускать и при обходе массивов и срезов. Такое случается достаточно редко, поскольку обход линейной структуры данных обычно производится для доступа к значениям ее элементов. Если вы используете такой

формат для массива или среза, то вполне вероятно, что вы неправильно выбрали структуру данных и стоит подумать о рефакторинге.



При обсуждении каналов в главе 10 вы также познакомитесь с примером ситуации, когда нужно, чтобы на каждой итерации цикл `for-range` возвращал только значение элемента.

Обход элементов карты

В том, как цикл `for-range` выполняет обход элементов карты, есть интересная особенность. Запустите в онлайн-песочнице код из примера 4.15 (<https://oreil.ly/VpInA>).

Пример 4.15. Порядок обхода элементов карты может варьироваться

```
m := map[string]int{
    "a": 1,
    "c": 3,
    "b": 2,
}

for i := 0; i < 3; i++ {
    fmt.Println("Loop", i)
    for k, v := range m {
        fmt.Println(k, v)
    }
}
```

Скомпилировав и запустив эту программу, вы увидите, что ее вывод будет меняться. Вот как выглядит один из возможных вариантов вывода:

```
Loop 0
c 3
b 2
a 1
Loop 1
a 1
c 3
b 2
Loop 2
b 2
a 1
c 3
```

Порядок ключей и значений будет меняться, иногда повторяясь. На самом деле такое поведение обусловлено соображениями безопасности. В более ранних

версиях языка Go порядок обхода ключей у карт с одинаковыми элементами обычно (но не всегда) был одинаковым. Это порождало две проблемы.

- Разработчики писали код с расчетом на фиксированный порядок обхода, но он иногда менялся, и это приводило к сбоям в самый неподходящий момент.
- Если карта всегда хеширует элементы в одни и те же значения и злоумышленнику известно, что сервер сохраняет некоторые пользовательские данные в виде карты, то можно добиться реального замедления работы сервера с помощью *DoS-атаки на основе хеш-коллизий (Hash DoS)*, отправив серверу специально подготовленные данные, все ключи которых хешируются в одно и то же ведро.

Чтобы устранить обе эти проблемы, разработчики языка Go внесли два изменения в реализацию карты. Во-первых, они модифицировали хеш-алгоритм для карт таким образом, чтобы при каждом создании переменной карты генерировалось случайное число. Во-вторых, они сделали так, чтобы при каждом обходе карты с помощью цикла `for-range` порядок обхода элементов немного варьировался. Эти два изменения существенно усложнили проведение DoS-атаки на основе хеш-коллизий.



Из этого правила есть одно исключение. Чтобы упростить процесс отладки и ведение журналов карт, функции форматирования (такие как `fmt.Println`) всегда выводят карты в порядке возрастания ключей.

Обход элементов строки

Как уже упоминалось выше, цикл `for-range` также можно использовать для элементов строки. Посмотрим, как это выглядит. Запустите на своей машине или в онлайн-песочнице код из примера 4.16 (<https://oreil.ly/C3LRy>).

Пример 4.16. Обход элементов строки

```
samples := []string{"hello", "apple_π!"}
for _, sample := range samples {
    for i, r := range sample {
        fmt.Println(i, r, string(r))
    }
    fmt.Println()
}
```

При обходе слова `hello` мы получаем вполне ожидаемый результат:

```
0 104 h
1 101 e
```

```
2 108 l
3 108 l
4 111 o
```

В первом столбце выводится индекс, во втором — числовое значение буквы, а в третьем — результат преобразования в строку числового значения буквы.

Результат обхода слова `apple_п!` выглядит уже более интересно:

```
0 97 a
1 112 p
2 112 p
3 108 l
4 101 e
5 95 _
6 960 п
8 33 !
```

Здесь следует отметить два момента. Во-первых, обратите внимание, что в первом столбце пропущено число 7. Во-вторых, рядом с индексом 6 выведено значение 960. Это намного больше, чем может поместиться в байте. Однако в главе 3 говорилось, что строки состоят из байтов. Что же здесь происходит?

То, что мы здесь наблюдаем, является уникальной особенностью обхода строки с помощью цикла `for-range`. Этот цикл перебирает *руны*, а не *байты*. Всякий раз, когда цикл `for-range` встречает в строке руну из нескольких байтов, он преобразует это представление в формате UTF-8 в одно 32-разрядное число и присваивает его переменной. Смещение при этом увеличивается на то количество байтов, которое содержится в руне. Когда цикл `for-range` встречает байт, который не является одним из допустимых значений в формате UTF-8, вместо него возвращается символ подстановки Unicode (шестнадцатеричное значение 0xfffd).



Используйте цикл `for-range` для последовательного доступа к рунам в строке. Ключ при этом представляет собой смещение в байтах от начала строки, но значения элементов относятся к рунному типу.

Цикл `for-range` копирует значения элементов

Следует иметь в виду, что при обходе любого составного типа цикл `for-range` *копирует* значение из этого составного типа в значение переменной. *Изменение значения переменной не приведет к изменению соответствующего значения в составном типе.* Эту особенность демонстрирует пример 4.17. Попробуйте запустить этот код в онлайн-песочнице (<https://oreil.ly/ShwR0>).

Пример 4.17. Изменение значения переменной не ведет к изменению исходного значения

```
evenVals := []int{2, 4, 6, 8, 10, 12}
for _, v := range evenVals {
    v *= 2
}
fmt.Println(evenVals)
```

Этот код выведет следующее:

```
[2 4 6 8 10 12]
```

Такое поведение цикла `for-range` иногда приводит к неожиданным последствиям. Как мы увидим при обсуждении горутин в главе 10, при их запуске в цикле `for-range` нужно быть крайне внимательными в отношении способа передачи индексов и значений горутине, чтобы не получить совершенно неожиданные результаты.

Как и три другие разновидности оператора `for`, оператор `for-range` позволяет использовать ключевые слова `break` и `continue`.

Операторы `for` с метками

По умолчанию действие ключевых слов `break` и `continue` распространяется на тот цикл `for`, который непосредственно их содержит. Но что, если вы используете вложенный цикл `for` и вам нужно выйти из внешнего цикла или перейти к его следующей итерации? Рассмотрим небольшой пример. Мы изменим рассмотренную ранее программу для обхода строки таким образом, чтобы она прекращала обход строки, встретив букву `l`. Запустите в онлайн-песочнице код из примера 4.18 (<https://oreil.ly/ToDkq>).

Пример 4.18. Использование меток

```
func main() {
    samples := []string{"hello", "apple_π!"}
outer:
    for _, sample := range samples {
        for i, r := range sample {
            fmt.Println(i, r, string(r))
            if r == 'l' {
                continue outer
            }
        }
        fmt.Println()
    }
}
```


Обратите внимание, что команда `go fmt` снабдила метку `outer` таким же отступом, как у содержащей ее функции. Метки всегда снабжаются таким же уровнем отступа, как у фигурных скобок блока. Это делает их более заметными. Запустив эту программу, вы увидите на экране следующее:

```
0 104 h
1 101 e
2 108 l
0 97 a
1 112 p
2 112 p
3 108 l
```

Вложенные циклы очень редко снабжаются метками. Обычно метки используются для реализации алгоритмов, подобных приведенному ниже псевдокоду:

```
outer:
    for _, outerVal := range outerValues {
        for _, innerVal := range outerVal {
            // обработка значений innerVal
            if invalidSituation(innerVal) {
                continue outer
            }
        }
        // здесь располагается код, выполняемый в случае
        // успешного завершения обработки всех значений innerVal
    }
}
```

Выбор подходящего оператора for

Теперь, когда вы уже изучили все разновидности оператора `for`, вы можете задаться вопросом, когда лучше использовать каждую из них. В большинстве случаев следует использовать оператор `for-range`. Использование цикла `for-range` является наилучшим способом обхода строки, поскольку этот цикл выдает руны, а не байты. Как мы уже видели, цикл `for-range` также удобно использовать для обхода срезов и карт, и в главе 10 будет показано, что он хорошо подходит и для работы с каналами.



Отдавайте предпочтение циклу `for-range`, когда требуется перебирать содержимое экземпляра одного из встроенных составных типов. Это позволяет обойтись без громоздкого шаблонного кода, который требуется писать при обходе массивов, срезов и карт с помощью других разновидностей цикла `for`.

А когда следует использовать полную форму цикла `for`? Ее лучше использовать в тех случаях, когда не требуется обходить все содержимое составного типа с первого по последний элемент. Можно, конечно, использовать определенную комбинацию операторов `if`, `continue` и `break` внутри цикла `for-range`, но стандартный цикл `for` позволяет более четко указать диапазон перебираемых элементов. Сравним две версии кода, обходящего массив, начиная со второго и заканчивая предпоследним элементом. Вот как это будет выглядеть при использовании цикла `for-range`:

```
evenVals := []int{2, 4, 6, 8, 10}
for i, v := range evenVals {
    if i == 0 {
        continue
    }
    if i == len(evenVals)-1 {
        break
    }
    fmt.Println(i, v)
}
```

А вот как этот же код будет выглядеть при использовании стандартного цикла `for`:

```
evenVals := []int{2, 4, 6, 8, 10}
for i := 1; i < len(evenVals)-1; i++ {
    fmt.Println(i, evenVals[i])
}
```

Как видим, версия со стандартным циклом `for` и короче, и проще для понимания.



Этот паттерн не подходит для случая, когда нужно пропустить начало строки. Как вы помните, стандартный цикл `for` не умеет перебирать многобайтовые символы. Если вам нужно пропустить несколько рун в строке, используйте цикл `for-range`, чтобы должным образом перебрать руны.

Остальные две разновидности оператора `for` применяются гораздо реже. Цикл `for`, использующий только условие, подобно циклу `while`, вместо которого он используется, будет уместен в тех случаях, когда процесс выполнения цикла зависит от вычисляемого значения.

В некоторых случаях будет уместен и бесконечный цикл `for`. Тело такого цикла всегда должно содержать оператор `break`, поскольку бесконечное выполнение цикла требуется лишь в очень редких случаях. Реально используемые программы должны корректно выходить из цикла, когда операции не могут быть завершены. Как было показано ранее, применяя бесконечный цикл `for` в сочетании с оператором `if`, можно имитировать оператор `do`, присутствующий в других языках.

Бесконечный цикл `for` также используется для реализации некоторых версий паттерна «*итератор*», который будет рассмотрен при обсуждении стандартной библиотеки в разделе «Пакет `io` и его друзья» на с. 286.

Оператор switch

Как и во многих других языках, ведущих свое начало от языка C, в языке Go есть оператор `switch`. Разработчики, как правило, стараются не применять операторы `switch` из-за ограничений в использовании переключающих значений и из-за того, что по умолчанию в этих языках происходит «проваливание» из одной ветви оператора `switch` в другую. В отличие от этих языков в Go операторы `switch` могут быть очень полезными.



С расчетом на тех читателей, которые уже хорошо знакомы с языком Go, в этой главе будет рассмотрена такая разновидность операторов `switch`, как переключатели выражений (expression switch). При обсуждении интерфейсов в главе 7 мы также рассмотрим переключатели типов (type switch).

На первый взгляд оператор `switch` языка Go выглядит практически так же, как в C/C++, Java или JavaScript, однако он таит в себе несколько сюрпризов. Рассмотрим пример его использования. Запустите в онлайн-песочнице код из примера 4.19 (<https://oreil.ly/VKf4N>).

Пример 4.19. Оператор switch

```
words := []string{"a", "cow", "smile", "gopher",
    "octopus", "anthropologist"}
for _, word := range words {
    switch size := len(word); size {
    case 1, 2, 3, 4:
        fmt.Println(word, "is a short word!")
    case 5:
        wordLen := len(word)
        fmt.Println(word, "is exactly the right length:", wordLen)
    case 6, 7, 8, 9:
    default:
        fmt.Println(word, "is a long word!")
    }
}
```

Запустив этот код, мы увидим на экране следующее:

```
a is a short word!
cow is a short word!
smile is exactly the right length: 5
```

anthropologist is a long word!

Теперь посмотрим, за счет каких своих особенностей оператор `switch` обеспечил нам такой результат. Как и в случае оператора `if`, проверяемое оператором `switch` значение не нужно заключать в круглые скобки. Еще одно сходство с оператором `if` состоит в том, что вы можете объявить переменную, область видимости которой будет включать в себя все ветви оператора `switch`. В данном случае в пределах всех ветвей оператора `switch` объявляется переменная `size`.

Все ветви `case` (и необязательная ветвь `default`) заключаются в фигурные скобки. Однако заметьте, что здесь не нужно заключать в фигурные скобки содержимое каждой ветви `case`. Внутри каждой ветви `case` (и ветви `default`) можно разместить несколько строк кода, и все эти строки будут считаться частью единого блока.

Внутри ветви `case 5`: объявляется новая переменная `wordLen`. Мы объявляем здесь новые переменные, поскольку это новый блок. Как в случае любого другого блока, переменные, объявленные внутри блока ветви `case`, доступны только внутри этого блока.

Если вы привыкли размещать оператор `break` в конце каждой ветви `case` своих операторов `switch`, то вас должен обрадовать тот факт, что в Go этого делать не нужно. В Go по умолчанию не происходит «проваливание» из одной ветви оператора `switch` в другую. В этом Go будет больше напоминать вам Ruby или (если вы программист старой школы) Pascal.

Здесь вы можете спросить: если у нас нет «проваливания» между ветвями, то как следует поступать в том случае, когда несколько значений должны запускать в точности одинаковую логику? В таком случае Go позволяет указать через запятую несколько значений, как мы перечислили здесь значения 1, 2, 3, 4 или 6, 7, 8, 9. Именно поэтому мы получили одинаковый результат для строк `a` и `cow`.

Это приводит к следующему вопросу: если нет «проваливания» между ветвями, то что произойдет в случае выбора пустой ветви (как это происходит в нашей программе, когда длина строки составляет 6, 7, 8 или 9 символов)? В Go *при выборе пустой ветви не производится никаких действий*. Именно поэтому наша программа не выдала никаких результатов для строк `octopus` и `gopher`.



Для полноты картины следует сказать, что в языке Go есть ключевое слово `fallthrough`, которое позволяет перейти в следующую ветвь после выполнения текущей ветви. Однако прежде, чем приступать к реализации алгоритма, использующего это ключевое слово, лучше внимательно рассмотрите его еще раз. Если ваша логика требует использования ключевого слова `fallthrough`, попробуйте перестроить ее таким образом, чтобы ветви не зависели друг от друга.

В данной программе мы выполняем переключение на основе целочисленного значения, но это также можно делать на основе значения любого другого типа, который можно сравнивать с помощью оператора `==`. Это включает в себя все встроенные типы, за исключением срезов, карт, каналов и функций, а также структур с полями этих типов.

Хоть вам и не нужно размещать оператор `break` в конце каждой ветви `case`, вы можете использовать этот оператор в том случае, когда требуется выйти из ветви `case` раньше времени. Однако потребность в использовании оператора `break` может быть признаком того, что вы используете слишком сложный код и стоит подумать о рефакторинге.

Еще один случай, в котором может потребоваться оператор `break` внутри ветви `case` оператора `switch`, сводится к следующему. Если оператор `switch` вложен в цикл `for` и вам нужно выйти из этого цикла, снабдите меткой цикл `for` и укажите эту метку в операторе `break`. Когда вы не указываете метку, Go считает, что вы хотите выйти из процесса. Рассмотрим небольшой пример. Запустите в онлайн-песочнице код из примера 4.20 (<https://oreil.ly/o2xg2>).

Пример 4.20. Вариант без метки

```
func main() {
    for i := 0; i < 10; i++ {
        switch {
            case i%2 == 0:
                fmt.Println(i, "is even")
            case i%3 == 0:
                fmt.Println(i, "is divisible by 3 but not 2")
            case i%7 == 0:
                fmt.Println("exit the loop!")
                break
            default:
                fmt.Println(i, "is boring")
        }
    }
}
```

Вы получите такой результат:

```
0 is even
1 is boring
2 is even
3 is divisible by 3 but not 2
4 is even
5 is boring
6 is even
exit the loop!
8 is even
9 is divisible by 3 but not 2
```

Это не совсем то, что мы хотели получить. Нам нужно было выйти из цикла `for`, когда значение переменной станет равным 7. Чтобы добиться этого, мы должны использовать метку, как это делалось в случае выхода из вложенного цикла `for`. Это значит, что сначала нужно снабдить меткой оператор `for`:

```
loop:
    for i := 0; i < 10; i++ {
```

А затем нужно указать эту метку в операторе `break`:

```
break loop
```

Запустите модифицированный код в онлайн-песочнице (<https://oreil.ly/gA0O3>). На этот раз мы получим нужный нам результат:

```
0 is even
1 is boring
2 is even
3 is divisible by 3 but not 2
4 is even
5 is boring
6 is even
exit the loop!
```

Пустые переключатели

Существует еще один способ использования оператора `switch`, который дает более мощные возможности. Подобно тому как в Go можно опускать различные части определения оператора `for`, вы также можете использовать оператор `switch`, не указывая проверяемое значение. Такой оператор `switch` называют *пустым переключателем* (*blank switch*). В то время как обычный оператор `switch` позволяет только выполнять проверку на равенство значению, пустой оператор `switch` позволяет использовать для каждой ветви `case` любую операцию сравнения, дающую в результате логическое значение. Попробуйте запустить в онлайн-песочнице код из примера 4.21 (<https://oreil.ly/v7qI5>).

Пример 4.21. Пустой оператор `switch`

```
words := []string{"hi", "salutations", "hello"}
for _, word := range words {
    switch wordLen := len(word); {
        case wordLen < 5:
            fmt.Println(word, "is a short word!")
        case wordLen > 10:
            fmt.Println(word, "is a long word!")
        default:
            fmt.Println(word, "is exactly the right length.")
    }
```

```
    }
}
```

Запустив эту программу, вы увидите на экране следующее:

```
hi is a short word!
salutations is a long word!
hello is exactly the right length.
```

Как и в случае обычного оператора `switch`, в пустом операторе `switch` при желании можно использовать краткий вариант объявления переменной. Однако, в отличие от обычного оператора `switch`, можно использовать для каждой ветви `case` отдельную операцию сравнения. Таким образом, пустые переключатели являются довольно мощным инструментом, но в то же время не стоит ими злоупотреблять. Если в записанном вами пустом операторе `switch` во всех ветвях одна и та же переменная проверяется на равенство значению:

```
switch {
case a == 2:
    fmt.Println("a is 2")
case a == 3:
    fmt.Println("a is 3")
case a == 4:
    fmt.Println("a is 4")
default:
    fmt.Println("a is ", a)
}
```

то лучше замените этот оператор `switch` на переключатель выражений:

```
switch a {
case 2:
    fmt.Println("a is 2")
case 3:
    fmt.Println("a is 3")
case 4:
    fmt.Println("a is 4")
default:
    fmt.Println("a is ", a)
}
```

Что лучше выбрать: if или switch?

С точки зрения функциональности между использованием цепочки операторов `if/else` и пустого оператора `switch` нет большой разницы. И первый и второй подход позволяет вам использовать несколько операций сравнения. Так когда, в таком случае следует использовать оператор `switch`, а когда — цепочку

операторов `if/else`? Использование оператора `switch`, в том числе пустой разновидности этого оператора, говорит о наличии определенной взаимосвязи между значениями или операциями сравнения, используемыми в каждой ветви `case`. Чтобы продемонстрировать эту разницу, давайте перепишем код для классификации случайных чисел из раздела, посвященного оператору `if`, используя на этот раз оператор `switch`, как показано в примере 4.22.

Пример 4.22. Использование пустого оператора `switch` вместо операторов `if/else`

```
switch n := rand.Intn(10); {  
case n == 0:  
    fmt.Println("That's too low")  
case n > 5:  
    fmt.Println("That's too big:", n)  
default:  
    fmt.Println("That's a good number:", n)  
}
```

Мало кто будет спорить с тем, что этот вариант является более читабельным. Сравнимое значение располагается в отдельной строке, а все варианты условий выровнены по левому краю. Такое, более единообразное расположение операций сравнения облегчает их чтение и модификацию.

Конечно, ничто не мешает вам выполнять в разных ветвях пустого оператора `switch` совершенно не связанные друг с другом операции сравнения. Однако такой подход считается в Go неидиоматическим. В таком случае лучше воспользуйтесь цепочкой операторов `if/else` (или сделайте рефакторинг кода).



Используйте пустой оператор `switch` вместо цепочки операторов `if/else` в том случае, когда вам нужно выполнять несколько взаимосвязанных операций сравнения. Использование оператора `switch` делает операции сравнения более заметными и подчеркивает наличие взаимосвязи между ними.

Оператор `goto`

Хотя в Go имеется еще одна, четвертая управляющая конструкция, очень возможно, что вы ею никогда не воспользуетесь. С тех пор как в 1968 году Эдгар Дейкстра (Edgar Dijkstra) написал статью «О вреде оператора Go To» (<https://oreil.ly/YK2tl>), оператор `goto` считается чем-то вроде «паршивой овцы» в семье средств программирования, и для этого есть очень веские основания. Оператор `goto` всегда принято было считать опасным, поскольку он позволял переходить практически в любое место программы: вы могли перейти в цикл или выйти из

цикла, пропустить определение переменных или перейти в середину цепочки операторов, расположенной внутри оператора `if`. Из-за этого было очень трудно понять, что делала программа с операторами `goto`.

Хотя в большинстве современных языков нет оператора `goto`, Go позволяет вам его использовать. Но все равно рекомендуется его избегать. В то же время этот оператор имеет ряд областей применения, и, кроме того, Go заставляет его лучше вписываться в парадигму структурного программирования, накладывая на него ряд ограничений.

В Go оператор `goto` указывает некоторую снабженную меткой строку кода, в которую должно перейти выполнение программы. Однако при этом нельзя перейти в абсолютно любое место программы. В Go не допускаются переходы с пропуском объявлений переменных и переходы во вложенный или параллельный блок.

Эти недопустимые способы использования оператора `goto` демонстрирует программа, представленная в примере 4.23. Попробуйте запустить этот код в онлайн-песочнице (<https://oreil.ly/I016p>).

Пример 4.23. Используя оператор `goto` в Go, нужно соблюдать определенные правила

```
func main() {
    a := 10
    goto skip
    b := 20
skip:
    c := 30
    fmt.Println(a, b, c)
    if c > a {
        goto inner
    }
    if a < b {
inner:
        fmt.Println("a is less than b")
    }
}
```

Попытавшись выполнить эту программу, вы получите следующие сообщения об ошибках:

```
goto skip jumps over declaration of b at ./main.go:8:4
goto inner jumps into block starting at ./main.go:15:11
```

Так когда же использовать оператор `goto`? Обычно никогда. Для выхода из глубоко вложенных циклов и пропуска итераций циклов можно использовать операторы `break` и `continue` с указанием метки. Один из немногих сценариев

допустимого использования оператора `goto` демонстрирует программа, представленная в примере 4.24.

Пример 4.24. Пример ситуации, требующей использования оператора `goto`

```
func main() {
    a := rand.Intn(10)
    for a < 100 {
        if a%5 == 0 {
            goto done
        }
        a = a*2 + 1
    }
    fmt.Println("do something when the loop completes normally")
done:
    fmt.Println("do complicated stuff no matter why we left the loop")
    fmt.Println(a)
}
```

Несмотря на некоторую надуманность, этот пример показывает, как использование оператора `goto` может сделать код программы более «прозрачным». В этом простом примере нам нужно пропустить определенную логику в середине функции, выполнив при этом ее заключительную часть. Конечно, это можно сделать и без использования оператора `goto`. Например, мы могли бы добавить логический флаг или продублировать расположенный после метки код, но оба эти подхода обладают определенными недостатками. Управление порядком выполнения логики с помощью логических флагов фактически дает нам ту же функциональность, что и использование оператора `goto`, отличаясь лишь большей громоздкостью. Дублирование помеченного кода, в свою очередь, делает более трудоемким процесс поддержки кода. Конечно, такое случается очень редко, но если в таком случае не удастся каким-либо образом перестроить логику, то подобное использование оператора `goto` на самом деле позволит сделать код лучше.

Если вы хотите увидеть пример реально используемого кода с операторами `goto`, взгляните на код метода `floatBits` в файле `atof.go` из пакета `strconv` стандартной библиотеки. Я не буду приводить код этого метода целиком из-за его большого объема, но вот как выглядит его заключительная часть:

```
overflow:
    // ±Inf
    mant = 0
    exp = 1<<flt.expbits - 1 + flt.bias
    overflow = true

out:
    // Сборка битов
    bits := mant & (uint64(1)<<flt.mantbits - 1)
```

```
bits |= uint64((exp-flt.bias)&(1<<flt.expbits-1)) << flt.mantbits
if d.neg {
    bits |= 1 << flt.mantbits << flt.expbits
}
return bits, overflow
```

Перед этими строками проверяется несколько условий. При выполнении одних условий нужно выполнить код после метки `overflow`, а при выполнении других — пропустить этот код и сразу перейти к метке `out`. Соответственно, после каждого условия производится переход к метке `overflow` или метке `out` с помощью оператора `goto`. При этом, вероятно, можно было обойтись без использования операторов `goto`, но все эти способы делают код более трудным для понимания.



Старайтесь использовать оператор `goto` как можно реже. Однако в тех редких случаях, когда он делает код более читабельным, это вполне допустимо.

Резюме

В этой главе мы рассмотрели много важных вопросов, имеющих отношение к написанию идиоматического Go-кода. Вы познакомились с блоками, затенением переменных и управляющими конструкциями и изучили надлежащие способы использования последних. Вы уже умеете писать простые Go-программы, размещая код внутри функции `main`. Пришло время узнать, как можно создавать более крупные программы, применяя функции для организации кода.

ГЛАВА 5

Функции

До сих пор наши программы представляли собой не более чем несколько строк кода внутри функции `main`. Пришла пора «повзрослеть». В этой главе вы узнаете, как в языке Go можно писать функции и что можно сделать с их помощью.

Объявление и вызов функций

Основы работы с функциями в Go будут выглядеть для вас знакомо, если вам приходилось работать с функциями первого класса в таких языках, как C, Python, Ruby и JavaScript. (В Go также есть методы, о которых мы поговорим в главе 7.) Как и в случае управляющих конструкций, при использовании функций в Go вы можете задействовать ряд уникальных возможностей. Некоторые из них являются улучшением, а некоторые — «пробой пера» с весьма сомнительными преимуществами. В этой главе будут рассмотрены и первые и вторые.

Вы уже видели примеры объявления и использования функций. Каждая из написанных здесь программ содержала функцию `main`, которая является точкой входа в любой Go-программе, и мы уже вызывали функцию `fmt.Println` для вывода на экран результатов вычислений. Поскольку функция `main` не принимает параметров и не возвращает никаких значений, рассмотрим пример функции, которая это делает:

```
func div(numerator int, denominator int) int {  
    if denominator == 0 {  
        return 0  
    }  
    return numerator / denominator  
}
```

Итак, что же нового в этом коде? Объявление функции включает в себя четыре элемента: ключевое слово `func`, имя функции, список входных параметров и тип

возвращаемого значения. Входные параметры заключаются в круглые скобки и разделяются запятыми; при этом сначала указывается имя параметра, а затем — его тип. Поскольку Go — типизированный язык, необходимо указывать тип параметров. Тип возвращаемого значения записывается после круглых скобок со списком входных параметров, перед открывающей фигурной скобкой тела функции.

Как и в других языках, в Go для возврата значений из функции используется ключевое слово `return`. Если функция возвращает значение, вы *должны* добавить оператор `return`. Если функция ничего не возвращает, то записывать оператор `return` в ее конце не нужно. В функции, не возвращающей значение, оператор `return` требуется только при необходимости выйти из нее раньше последней строки.

У функции `main` нет входных параметров и возвращаемых значений. Когда у функции нет входных параметров, после ее имени ставится пустая пара круглых скобок: `()`. Если у функции нет возвращаемых значений, вы просто ничего не пишете в промежутке между круглыми скобками входных параметров и открывающей фигурной скобкой тела функции:

```
func main() {
    result := div(5, 2)
    fmt.Println(result)
}
```

Выполняемый здесь вызов функции должен выглядеть знакомо для опытных разработчиков. Справа от оператора `:=` мы вызываем определенную нами функцию `div`, передавая ей значения 5 и 2. Слева от оператора `:=` мы присваиваем возвращаемое значение переменной `result`.



Когда функция принимает несколько параметров одинакового типа, их тип можно указывать следующим образом:

```
func div(numerator, denominator int) int {
```

Имитация именованных и опциональных параметров

Прежде чем переходить к обсуждению того, какими уникальными возможностями обладают функции в Go, следует упомянуть, какими двумя возможностями они *не обладают* в Go: этот язык не позволяет использовать именованные и опциональные входные параметры. За исключением одного случая, который мы рассмотрим в следующем разделе, вы всегда должны указывать все параметры функции. При желании возможно симитировать использование именованных и опциональных параметров, определив структуру, поля которой представляют

собой нужные вам параметры, и передав эту структуру функции. Этот паттерн демонстрирует программа, представленная в примере 5.1.

Пример 5.1. Использование структуры для имитации именованных параметров

```
type MyFuncOpts struct {
    FirstName string
    LastName string
    Age int
}

func MyFunc(opts MyFuncOpts) error {
    // выполнение каких-либо действий
}

func main() {
    MyFunc(MyFuncOpts {
        LastName: "Patel",
        Age: 50,
    })
    MyFunc(MyFuncOpts {
        FirstName: "Joe",
        LastName: "Smith",
    })
}
```

На самом деле отсутствие именованных и опциональных параметров не является ограничением, поскольку рекомендуется, чтобы функция не принимала слишком много параметров, а именованные и опциональные параметры удобно использовать именно в случае большого количества входных параметров. Но если функция принимает много параметров, то вполне вероятно, что она является слишком сложной.

Вариативные входные параметры и срезы

Когда мы использовали функцию `fmt.Println` для вывода на экран результатов вычислений, вы могли заметить, что эта функция может принимать любое количество входных параметров. Каким образом ей удастся это делать? Как и многие другие языки, Go позволяет использовать *вариативные параметры*. Вариативный параметр *должен* быть последним (или единственным) параметром в списке входных параметров. Он обозначается путем размещения трех точек (...) перед именем типа. При этом создаваемая внутри функции переменная представляет собой срез указанного типа, который можно использовать как любой другой срез. Чтобы посмотреть, как используются такие параметры, напомним программу, которая будет добавлять к некоторому базовому числу переменное количество параметров и возвращать результат в виде среза элементов типа `int`. Вы можете

запустить эту программу в онлайн-песочнице (<https://oreil.ly/nSad4>). Сначала мы напишем нашу вариативную функцию:

```
func addTo(base int, vals ...int) []int {
    out := make([]int, 0, len(vals))
    for _, v := range vals {
        out = append(out, base+v)
    }
    return out
}
```

А затем опробуем несколько разных способов ее вызова:

```
func main() {
    fmt.Println(addTo(3))
    fmt.Println(addTo(3, 2))
    fmt.Println(addTo(3, 2, 4, 6, 8))
    a := []int{4, 3}
    fmt.Println(addTo(3, a...))
    fmt.Println(addTo(3, []int{1, 2, 3, 4, 5}...))
}
```

Как видите, в качестве вариативного параметра можно предоставить любое нужное вам количество значений либо ни одного значения. Поскольку вариативный параметр преобразуется в срез, вы можете предоставить срез в качестве входных данных. Однако при этом нужно поставить три точки (...) *после* имени переменной или литерала среза; в противном случае вы получите ошибку времени компиляции.

Скомпилировав и запустив эту программу, вы увидите на экране следующее:

```
[]
[5]
[5 7 9 11]
[7 6]
[4 5 6 7 8]
```

Возврат нескольких значений

Первым отличием языка Go от других языков является то, что он поддерживает возврат нескольких значений. Немного подправим рассмотренную нами выше программу для деления целых чисел. Пусть наша функция возвращает и результат деления двух чисел, и остаток от деления. Вот как она будет выглядеть после внесения этих изменений:

```
func divAndRemainder(numerator int, denominator int) (int, int, error) {
    if denominator == 0 {
        return 0, 0, errors.New("cannot divide by zero")
    }
}
```

```

    }
    return numerator / denominator, numerator % denominator, nil
}

```

Чтобы обеспечить возврат нескольких значений, нужно внести несколько изменений. Когда функция в Go возвращает несколько значений, необходимо перечислить типы возвращаемых значений, заключив их в круглые скобки и разделив запятыми. Кроме того, все эти значения должны быть указаны в операторе `return` через запятую. Однако при этом возвращаемые значения не нужно заключать в круглые скобки: это приведет к ошибке времени компиляции.

В данном примере можно заметить еще кое-что новое: создание и возвращение ошибки (значения типа `error`). Подробнее об ошибках будет рассказано в главе 8. Пока просто запомните, что возможность возврата нескольких значений может использоваться в Go для возврата ошибки в том случае, когда в ходе выполнения функции происходит какой-либо сбой. После успешного выполнения функции в качестве значения ошибки возвращается значение `nil`. Согласно общепринятому соглашению значение типа `error` всегда является последним (или единственным) значением в списке возвращаемых значений.

Вызов обновленной функции производится следующим образом:

```

func main() {
    result, remainder, err := divAndRemainder(5, 2)
    if err != nil {
        fmt.Println(err)
        os.Exit(1)
    }
    fmt.Println(result, remainder)
}

```

В разделе «`var` или `:=`» на с. 48 мы уже говорили об одновременном присвоении сразу нескольких значений. Здесь эта возможность используется для присвоения результатов вызова функции трем переменным. Справа от оператора `:=` мы вызываем нашу функцию `divAndRemainder`, передавая ей значения 5 и 2. Слева от оператора `:=` мы присваиваем возвращаемые значения переменным `result`, `remainder` и `err`. Затем мы проверяем, не произошла ли ошибка, сравнивая переменную `err` со значением `nil`.

При возврате нескольких значений всегда возвращается несколько значений

Если вы знакомы с языком Python, то, возможно, подумали, что возврат нескольких значений в Go похож на возврат кортежа из функции в Python, где кортеж по желанию может быть разделен, если его значения присваиваются

нескольким переменным. В примере 5.2 показано выполнение такого кода интерпретатором языка Python.

Пример 5.2. Возврат нескольких значений в Python производится путем разделения кортежа

```
>>> def div_and_remainder(n,d):
...     if d == 0:
...         raise Exception("cannot divide by zero")
...     return n / d, n % d
>>> v = div_and_remainder(5,2)
>>> v
(2.5, 1)
>>> result, remainder = div_and_remainder(5,2)
>>> result
2.5
>>> remainder
1
```

Однако в Go все происходит иначе. Здесь вы должны присвоить каждое возвращаемое значение. Попытка присвоить несколько возвращаемых значений одной переменной приведет к ошибке времени компиляции.

Игнорирование возвращаемых значений

Но что, если при вызове функции вам не нужно использовать все возвращаемые значения? Как уже говорилось в разделе «Неиспользуемые переменные» на с. 53, Go не допускает наличия неиспользуемых переменных. Если функция возвращает несколько значений, но вам не нужно считывать некоторые из них, присвойте неиспользуемые значения пустому идентификатору `_`. Например, если нам не потребуется переменная `remainder`, то операцию присваивания нужно будет записать так: `result, _ := divAndRemainder(5,2)`.

Что удивительно, Go позволяет неявно проигнорировать *все* возвращаемые значения функции. Вы можете записать только вызов функции: `divAndRemainder(5,2)`, и все возвращаемые значения будут отброшены. На самом деле мы уже делаем это, начиная с самых первых примеров: функция `fmt.Println` возвращает два значения, но идиоматический подход сводится к тому, чтобы их игнорировать. За исключением этого случая, практически всегда необходимо явно обозначать игнорирование возвращаемых значений с помощью символов подчеркивания.



Используйте идентификатор `_` всякий раз, когда вам не требуется возвращаемое функцией значение.

Именованные возвращаемые значения

Наряду с возвращением из функции нескольких значений, Go также позволяет указывать *имена* возвращаемых значений. Перепишем нашу функцию `divAndRemainder` еще раз, но теперь используем именованные возвращаемые значения:

```
func divAndRemainder(numerator int, denominator int) (result int, remainder int,
                                                         err error) {
    if denominator == 0 {
        err = errors.New("cannot divide by zero")
        return result, remainder, err
    }
    result, remainder = numerator/denominator, numerator%denominator
    return result, remainder, err
}
```

Дополняя именами возвращаемые значения, вы фактически заранее объявляете переменные, которые будут использоваться внутри функции для сохранения возвращаемых значений. При этом возвращаемые значения должны быть записаны в круглых скобках через запятую. Круглые скобки нужно использовать даже в том случае, если возвращается лишь одно такое значение. Именованные возвращаемые значения инициализируются в момент создания соответствующими нулевыми значениями. Это означает, что их можно возвращать еще до явного их использования или присвоения им значений.

Важно также отметить, что имена возвращаемых значений действуют только внутри функции, не оказывая никакого влияния за ее пределами. Ничто не мешает вам присвоить возвращаемые значения переменным с другими именами:

```
func main() {
    x, y, z := divAndRemainder(5, 2)
    fmt.Println(x, y, z)
}
```



Если вы хотите дать имена только некоторым возвращаемым значениям, можете указать пустой идентификатор `_` в качестве имени тех возвращаемых значений, которые нужно оставить безымянными.

Хотя именованные возвращаемые значения часто позволяют сделать код более ясным, их использование может вызывать и определенные проблемы. Прежде всего, здесь нужно помнить о проблеме затенения переменных. Как и любую другую переменную, именованное возвращаемое значение можно нечаянно затенить. Поэтому необходимо внимательно следить за тем, чтобы при при-

сваивании возвращаемых значений внутри функции не происходило затенения переменных.

Еще одна проблема при использовании именованных возвращаемых значений заключается в том, что их можно не возвращать. Рассмотрим еще одну версию функции `divAndRemainder`. Вы можете запустить этот код в онлайн-песочнице (<https://oreil.ly/FzUkw>):

```
func divAndRemainder(numerator, denominator int) (result int, remainder int,
                                                    err error) {
    // присвоение значений
    result, remainder = 20, 30
    if denominator == 0 {
        return 0, 0, errors.New("cannot divide by zero")
    }
    return numerator / denominator, numerator % denominator, nil
}
```

Обратите внимание, что здесь мы присваиваем значения переменным `result` и `remainder`, а затем напрямую возвращаем другие значения. Перед запуском этого кода попробуйте угадать, каким будет результат, если мы передадим этой функции значения 5 и 2. Реальный результат может вас удивить:

2 1

Оператор `return` возвращает значения, несмотря на то что они не были присвоены именованным возвращаемым параметрам. Это объясняется тем, что компилятор языка Go автоматически добавляет код, который присваивает возвращаемым параметрам возвращаемые нами значения. Именованные возвращаемые параметры позволяют вам указать, с какой целью вы собираетесь применять переменные для сохранения возвращаемых значений, но *не требуют*, чтобы вы их использовали.

Некоторые разработчики любят использовать именованные возвращаемые параметры как дополнительное средство документирования. Однако я нахожу их не слишком полезными из-за возможной путаницы в случае их затенения или игнорирования. В то же время именованные возвращаемые параметры играют важную роль в одном случае, который мы обсудим чуть позже в этой главе, когда будем говорить об операторе `defer`.

Никогда не используйте пустые операторы возврата!

При использовании именованных возвращаемых значений важно помнить об одном серьезном недостатке языка Go, который заключается в том, что он позволяет использовать пустой (или «голый») оператор возврата. При наличии

именованных возвращаемых значений вы можете просто записать оператор `return`, не указывая никаких возвращаемых значений. При этом функция возвращает то, что содержат именованные возвращаемые значения к концу ее выполнения. Перепишем нашу функцию `divAndRemainder` еще один, последний раз, используя пустые операторы возврата:

```
func divAndRemainder(numerator, denominator int) (result int, remainder int,
                                                    err error) {
    if denominator == 0 {
        err = errors.New("cannot divide by zero")
        return
    }
    result, remainder = numerator/denominator, numerator%denominator
    return
}
```

Чтобы использовать пустые операторы возврата, нам потребовалось внести в функцию и ряд других изменений. В случае некорректных входных данных мы сразу же производим возврат. Поскольку это делается еще до присвоения значений переменным `result` и `remainder`, возвращаются соответствующие нулевые значения. Если в качестве именованных возвращаемых значений возвращаются нулевые значения соответствующего типа, нужно проследить за тем, чтобы этот случай имел смысл. Также обратите внимание, что при этом вы все равно должны разместить оператор `return` в конце функции. Функция возвращает значения даже при использовании пустых операторов возврата. Если в функции не будет оператора `return`, вы получите сообщение об ошибке на этапе компиляции.

На первый взгляд, использование пустых операторов возврата может показаться удобным, поскольку это позволяет вводить меньше кода. Однако большинство Go-разработчиков считает использование пустых операторов возврата плохой практикой, поскольку это усложняет анализ потоков данных. Хорошая программа должна быть простой для понимания и читабельной: тому, кто читает ее код, должно быть понятно, что она делает. Но при использовании пустого оператора возврата для того, чтобы понять, что именно возвращает функция, потребуется просмотреть расположенный выше код и найти то место, где возвращаемым параметрам в последний раз присваиваются значения.



Если ваша функция возвращает значения, *никогда* не используйте пустой оператор возврата. Это может существенно усложнить понимание того, какое именно значение она возвращает.

Функции являются значениями

Как и во многих других языках, в Go функции являются значениями. Тип функции составляется из ключевого слова `func` и типов параметров и возвращаемых значений. Эту комбинацию называют *сигнатурой* функции. Если две функции обладают в точности одинаковым количеством и типом параметров, то их сигнатуры типов совпадают.

Тот факт, что функции являются значениями, позволяет нам использовать ряд интересных приемов. Например, мы можем создать простейший калькулятор, используя функции в качестве элементов карты. Посмотрим, как это выглядит на практике. Вы можете запустить этот код в онлайн-песочнице (<https://oreil.ly/L59VY>). Сначала мы создаем несколько функций с одинаковой сигнатурой:

```
func add(i int, j int) int { return i + j }

func sub(i int, j int) int { return i - j }

func mul(i int, j int) int { return i * j }

func div(i int, j int) int { return i / j }
```

Затем мы создаем карту, которая сопоставляет математические операторы с каждой из этих функций:

```
var opMap = map[string]func(int, int) int{
    "+": add,
    "-": sub,
    "*": mul,
    "/": div,
}
```

После этого попробуем применить наш калькулятор для вычисления нескольких выражений:

```
func main() {
    expressions := [][]string{
        []string{"2", "+", "3"},
        []string{"2", "-", "3"},
        []string{"2", "*", "3"},
        []string{"2", "/", "3"},
        []string{"2", "%", "3"},
        []string{"two", "+", "three"},
        []string{"5"},
    }
    for _, expression := range expressions {
```

```

    if len(expression) != 3 {
        fmt.Println("invalid expression:", expression)
        continue
    }
    p1, err := strconv.Atoi(expression[0])
    if err != nil {
        fmt.Println(err)
        continue
    }
    op := expression[1]
    opFunc, ok := opMap[op]
    if !ok {
        fmt.Println("unsupported operator:", op)
        continue
    }
    p2, err := strconv.Atoi(expression[2])
    if err != nil {
        fmt.Println(err)
        continue
    }
    result := opFunc(p1, p2)
    fmt.Println(result)
}
}

```

Для преобразования типа `string` в тип `int` здесь используется функция стандартной библиотеки `strconv.Atoi`. В качестве второго значения эта функция возвращает значение типа `error`. Как и раньше, мы проверяем, не произошел ли сбой при выполнении функции, чтобы корректно обрабатывать ошибки.

Мы используем переменную `op` в качестве ключа карты `opMap` и присваиваем ассоциированное с этим ключом значение переменной `opFunc`. Переменная `opFunc` обладает типом `func(int, int) int`. Если в карте не будет функции, ассоциированной с предоставленным ключом, мы выведем сообщение об ошибке и сразу перейдем к следующей итерации. Затем мы вызываем функцию, присвоенную переменной `opFunc` с помощью переменных `p1` и `p2`, которые мы расшифровали ранее. Вызов функции, присвоенной переменной, выглядит точно так же, как вызов функцию напрямую.

Запустив эту программу, вы увидите наш простейший калькулятор в деле:

```

5
-1
6
0
unsupported operator: %
strconv.Atoi: parsing "two": invalid syntax
invalid expression: [5]

```



При написании программ старайтесь исключать возможность сбоев. Основная логика данной программы занимает сравнительно небольшой объем. Из 22 строк кода, размещенных внутри цикла `for`, лишь 6 строк обеспечивают непосредственную реализацию алгоритма, в то время как остальные 16 строк обеспечивают проверку на ошибки и проверку корректности данных. Если у вас возникает соблазн не выполнять проверку входных данных на корректность или проверку на наличие ошибок, помните о том, что при таком подходе вы получите нестабильно работающий и трудно поддерживаемый код. Именно наличие обработки ошибок отличает профессиональный подход от любительского.

Объявление функциональных типов

Ключевое слово `type` можно использовать не только для определения структуры (`struct`), но и для определения функционального типа (подробнее об определении типов будет рассказано в главе 7):

```
type opFuncType func(int,int) int
```

После этого мы можем переписать объявление переменной `opMap` следующим образом:

```
var opMap = map[string]opFuncType {  
    // без изменений  
}
```

При этом не нужно как-либо модифицировать функции. Любая функция с двумя входными параметрами типа `int` и одним возвращаемым значением типа `int` автоматически считается соответствующей требуемому типу и может быть использована в качестве значения карты.

Но что дает объявление функционального типа? Одно из преимуществ состоит в упрощении документирования. Если вы собираетесь упоминать что-то много раз, полезно дать этой сущности имя. Еще одно преимущество будет показано в разделе «Функциональные типы — это ключ к интерфейсам» на с. 194.

Анонимные функции

Помимо простого присвоения функций переменным, вы можете присваивать переменным функции, определенные внутри других функций.

Такие вложенные функции называются *анонимными функциями*. Они не носят какого-либо имени. Их также можно не присваивать переменной. Вы можете просто записать определение такой функции в строке и тут же ее вызвать. Рассмотрим следующий простой пример из онлайн-песочницы (<https://oreil.ly/EnkN6>):

```
func main() {  
    for i := 0; i < 5; i++ {  
        func(j int) {  
            fmt.Println("printing", j, "from inside of an anonymous function")  
        }(i)  
    }  
}
```

Анонимная функция объявляется с помощью ключевого слова `func`, за которым следуют входные параметры, возвращаемые значения и открывающая фигурная скобка. Попытка разместить имя функции между ключевым словом `func` и входными параметрами приведет к ошибке времени компиляции.

Как и любая другая функция, анонимная функция вызывается с помощью круглых скобок. В данном случае мы передаем анонимной функции переменную `i` внутри цикла `for`. Она присваивается входному параметру `j` анонимной функции.

Запустив эту программу, вы увидите на экране следующее:

```
printing 0 from inside of an anonymous function  
printing 1 from inside of an anonymous function  
printing 2 from inside of an anonymous function  
printing 3 from inside of an anonymous function  
printing 4 from inside of an anonymous function
```

Однако такой подход является чем-то из ряда вон выходящим. Если вы выполняете анонимную функцию сразу после ее объявления, то вы с тем же успехом можете просто выполнить этот код, не используя анонимную функцию. Тем не менее такое объявление анонимной функции без присвоения ее переменной может быть полезным в следующих двух случаях: при использовании операторов `defer` и при запуске горутин. Об операторах `defer` мы поговорим чуть позже в этой главе, а использование горутин будет подробно рассмотрено в главе 10.

Замыкания

Функции, объявляемые внутри других функций, представляют собой особую разновидность функций, называемую *замыканиями*. В теории вычислительных машин под замыканием понимается функция, которая объявлена внутри другой функции и может использовать и изменять переменные, объявленные во внешней функции.

На первый взгляд все эти вложенные функции и замыкания кажутся крайне малополезными. Какой прок может быть от создания мини-функций внутри более крупной функции? Для чего в языке Go предусмотрена такая возможность?

Помимо прочего, замыкания позволяют ограничивать область видимости функции. Если вы собираетесь многократно вызывать функцию только из одной другой функции, то вызываемую функцию можно скрыть путем использования вложенной функции. Это позволяет уменьшить количество объявлений, производимых на уровне пакета, и тем самым упростить поиск неиспользуемых имен.

Однако действительно интересными замыкания делает возможность их использования в качестве входных параметров или возвращаемых значений других функций. При этом они позволяют использовать объявленные внутри функции переменные *за ее пределами*.

Передача функций в качестве параметров

Поскольку функции являются значениями, а их тип можно определять типом входных параметров и возвращаемых значений, вы можете передавать функции другим функциям в качестве параметров. Если вам не приходилось раньше использовать функции в качестве данных, подумайте немного о том, к каким последствиям может привести создание замыкания, использующего локальные переменные, с последующей передачей этого замыкания другой функции. Примеры использования этого весьма полезного паттерна можно не раз встретить в стандартной библиотеке.

Одним из этих примеров является сортировка срезов. В пакете `sort` стандартной библиотеки есть функция `sort.Slice`, которая принимает в качестве параметров срез и функцию, используемую для сортировки этого среза. Посмотрим, как она работает, на примере сортировки среза структур, содержащих поля двух типов.



Поскольку в Go (по крайней мере пока) нет обобщенных типов, функции `sort.Slice` приходится прибегать к некоторой собственной «магии», чтобы ей можно было передавать срез любого типа. О том, что это за «магия», мы поговорим в главе 14.

Посмотрим, как с помощью замыканий можно реализовать различные способы сортировки одних и тех же данных. Вы можете запустить этот код в онлайн-песочнице (<https://oreil.ly/3kjg3>). Сначала мы определяем простой структурный тип и срез значений этого типа, после чего выводим на экран его исходное содержимое:

```
type Person struct {  
    FirstName string  
    LastName  string  
    Age       int  
}
```

```
people := []Person{
    {"Pat", "Patterson", 37},
    {"Tracy", "Bobbert", 23},
    {"Fred", "Fredson", 18},
}
fmt.Println(people)
```

Затем мы производим сортировку среза по фамилии и выводим полученные результаты:

```
// сортировка по фамилии
sort.Slice(people, func(i int, j int) bool {
    return people[i].LastName < people[j].LastName
})
fmt.Println(people)
```

Хотя замыкание, которое передается функции `sort.Slice`, принимает два параметра, `i` и `j`, внутри этого замыкания мы можем обращаться к срезу `people`, что позволяет отсортировать его по полю `LastName`. Выражаясь в терминах теории вычислительных машин, замыкание *перехватывает* срез `people`. Затем мы точно таким же образом производим сортировку по полю `Age`:

```
// сортировка по возрасту
sort.Slice(people, func(i int, j int) bool {
    return people[i].Age < people[j].Age
})
fmt.Println(people)
```

Запустив этот код, вы увидите на экране следующее:

```
[{Pat Patterson 37} {Tracy Bobbert 23} {Fred Fredson 18}]
[{Tracy Bobbert 23} {Fred Fredson 18} {Pat Patterson 37}]
[{Fred Fredson 18} {Tracy Bobbert 23} {Pat Patterson 37}]
```

Как видим, после вызова функции `sort.Slice` в срезе `people` меняется порядок элементов. О том, каким образом это обеспечивается, будет кратко рассказано в разделе «Go — язык с передачей параметров по значению» на с. 136 и более подробно в следующей главе.



Передача функций в качестве параметров другим функциям часто используется для выполнения различных операций над данными одного и того же типа.

Возвращение функций из функций

С помощью замыканий можно не только передавать состояние одной функции в другую, но и возвращать одну функцию из другой. Посмотрим, как это делается, на примере функции, возвращающей функцию для перемножения чисел. Вы можете запустить эту программу в онлайн-песочнице (<https://oreil.ly/8tpbN>). Вот как здесь выглядит определение функции, возвращающей замыкание:

```
func makeMult(base int) func(int) int {
    return func(factor int) int {
        return base * factor
    }
}
```

И вот как мы ее используем:

```
func main() {
    twoBase := makeMult(2)
    threeBase := makeMult(3)
    for i := 0; i < 3; i++ {
        fmt.Println(twoBase(i), threeBase(i))
    }
}
```

Запустив эту программу, вы увидите на экране следующее:

```
0 0
2 3
4 6
```

Теперь, когда вы уже видели замыкания в деле, у вас может возникнуть вопрос: насколько часто их используют Go-разработчики? Удивительно, но их применяют достаточно часто. Здесь было показано, как замыкания используются для сортировки срезов. Они также применяются и для эффективного поиска в отсортированном срезе с помощью функции `sort.Search`. Что касается возвращения замыканий, то с примером использования этого паттерна можно будет познакомиться во время создания промежуточного слоя для веб-сервера в подразделе «Промежуточный слой» на с. 307. Еще одной областью применения замыканий в Go является реализация освобождения ресурсов с помощью ключевого слова `defer`.



Если вам приходилось общаться с программистами, использующими функциональные языки программирования наподобие Haskell, то вы, вероятно, слышали о таком понятии, как «функции высшего порядка». На самом деле это лишь более замысловатый способ сказать о том, что функция использует другую функцию в качестве входного параметра или возвращаемого значения. Так что не волнуйтесь, Go-разработчики не уступают в крутизне вашему заумным товарищам!

Оператор defer

Программы часто создают такие временные ресурсы, как файлы или сетевые соединения, которые необходимо очищать. Эти ресурсы должны высвобождаться вне зависимости от того, сколько точек выхода имеет функция и заканчивается ли успехом ее выполнение. В Go код для очистки добавляется в функцию с помощью ключевого слова `defer`.

Что ж, посмотрим, как можно использовать оператор `defer` для освобождения ресурсов. Для этого мы напишем простую версию утилиты `cat`, используемой в операционной системе Unix для вывода на экран содержимого файла. Мы не откроем файлы в онлайн-песочнице, но вы можете найти код этого примера на сайте GitHub по адресу <https://oreil.ly/P4RuC>, в каталоге `simple_cat`:

```
func main() {
    if len(os.Args) < 2 {
        log.Fatal("no file specified")
    }
    f, err := os.Open(os.Args[1])
    if err != nil {
        log.Fatal(err)
    }
    defer f.Close()
    data := make([]byte, 2048)
    for {
        count, err := f.Read(data)
        os.Stdout.Write(data[:count])
        if err != nil {
            if err != io.EOF {
                log.Fatal(err)
            }
            break
        }
    }
}
```

Этот пример демонстрирует несколько новых элементов, которые мы подробно рассмотрим в последующих главах. При желании вы можете сразу перейти к чтению этих глав.

Прежде всего необходимо убедиться в том, что в командной строке было указано имя файла, путем проверки длины среза `os.Args` из пакета `os`, содержащего имя запускаемой программы и переданные ей аргументы. При недостаточном количестве аргументов мы используем функцию `Fatal` из пакета `log`, чтобы вывести сообщение об ошибке и выйти из программы. Затем мы получаем доступный только для чтения дескриптор файла с помощью функции `Open` из пакета `os`. В качестве второго значения функция `Open` возвращает ошибку. В случае сбоя на

этапе открытия файла мы выводим сообщение об ошибке и выходим из программы. Как уже говорилось ранее, ошибки будут подробно рассмотрены в главе 8.

Убедившись в том, что у нас есть корректный дескриптор файла, мы должны закрыть его после использования, вне зависимости от того, каким образом будет осуществлен выход из функции. Чтобы обеспечить гарантированный запуск кода для освобождения ресурсов, перед вызовом функции или метода следует записать ключевое слово `defer`. В данном случае мы используем метод `Close` в файловой переменной. (О применении методов в Go будет рассказано в главе 7.) Если обычно вызов функции производится немедленно, то при использовании ключевого слова `defer` вызов откладывается до момента выхода из содержащей его функции.

Мы производим чтение из дескриптора файла, передавая срез байтов методу `Read` в файловой переменной. Об использовании метода `Read` будет подробно рассказано в разделе «Пакет `io` и его друзья» на с. 286, а пока можно сказать, что он возвращает количество считанных в срез байтов и ошибку. Если ошибка не равна `nil`, мы проверяем, не содержит ли она метку конца файла. Если мы достигли конца файла, то используем оператор `break`, чтобы выйти из цикла `for`. В случае любой другой ошибки мы выдаем сообщение об ошибке и производим немедленный выход с помощью функции `log.Fatal`. Мы еще кратко коснемся темы срезов и параметров функций в разделе «Go — язык с передачей параметров по значению» на с. 136, а также подробно рассмотрим этот паттерн при обсуждении указателей в следующей главе.

Скомпилировав и запустив программу из каталога `simple_cat`, вы получите следующий результат:

```
$ go build
$ ./simple_cat simple_cat.go
package main

import (
    "fmt"
    "os"
)
...
```

В отношении оператора `defer` следует отметить еще несколько моментов. Во-первых, с его помощью можно отложить выполнение внутри функции нескольких замыканий. Эти замыкания выполняются по принципу «последним вошел — первым вышел», то есть оператор `defer`, указанный последним, выполняется первым.

Код таких отложенных замыканий выполняется *после выполнения* оператора возврата. Как уже говорилось выше, в операторе `defer` также можно указать

функцию с некоторыми входными параметрами. При этом определение значений переменных, передаваемых отложенному замыканию, также откладывается до момента запуска замыкания.



Хотя в операторе `defer` также можно указать функцию с некоторыми возвращаемыми значениями, вы никак не сможете прочитать эти значения.

```
func example() {
    defer func() int {
        return 2 // это значение невозможно прочитать
    }()
}
```

Возможно, вас интересует, может ли отложенная функция каким-либо образом прочитать или модифицировать возвращаемые значения той функции, которая ее содержит. Такая возможность существует и является одной из главных причин использования именованных возвращаемых значений. Это позволяет нам действовать в зависимости от наличия ошибки. При обсуждении ошибок в главе 8 мы рассмотрим паттерн, в котором оператор `defer` используется для дополнения возвращаемой из функции ошибки информацией о контексте. Посмотрим, как с помощью именованных возвращаемых значений и оператора `defer` можно высвобождать транзакции базы данных:

```
func DoSomeInserts(ctx context.Context, db *sql.DB, value1, value2 string)
    (err error) {
    tx, err := db.BeginTx(ctx, nil)
    if err != nil {
        return err
    }
    defer func() {
        if err == nil {
            err = tx.Commit()
        }
        if err != nil {
            tx.Rollback()
        }
    }()
    _, err = tx.ExecContext(ctx, "INSERT INTO FOO (val) values $1", value1)
    if err != nil {
        return err
    }
    // здесь можно выполнить еще ряд операций вставки, используя tx
    return nil
}
```

В этой книге мы не будем касаться того, как в Go реализована поддержка баз данных, однако следует сказать, что пакет `database/sql` стандартной библиотеки предоставляет широкий набор функций для работы с ними. В представленной

выше функции мы создаем транзакцию для выполнения ряда операций вставки в базу данных. Если любая из этих операций закончится неудачно, необходимо произвести откат (то есть отказаться от модификации базы данных). В случае успешного выполнения всех операций транзакция фиксируется (то есть сохраняются внесенные изменения). Отложенное замыкание `defer` позволяет проверить, не было ли присвоено ненулевое значение переменной `err`. Если этого не произошло, мы запускаем метод `tx.Commit()`, который также может вернуть ошибку. Если он это сделает, значение переменной `err` изменится. Если какое-либо из взаимодействий с базой данных возвращает ошибку, мы вызываем метод `tx.Rollback()`.



Начинающие Go-разработчики часто забывают ставить круглые скобки, когда указывают замыкание в операторе `defer`. Поскольку отсутствие этих скобок вызывает ошибку времени компиляции, постепенно вы привыкнете к правильному варианту. При этом полезно помнить о том, что круглые скобки нужны для того, чтобы указывать значения, передаваемые замыканию при его запуске.

В Go широко используется следующий паттерн: функция, которая выделяет некоторый ресурс, также возвращает замыкание, которое высвобождает этот ресурс. В каталоге `simple_cat_cancel` нашего проекта на сайте GitHub можно найти второй вариант простой программы `cat`, где используется данный подход. Сначала мы определяем вспомогательную функцию, которая будет открывать файл и возвращать замыкание:

```
func getFile(name string) (*os.File, func(), error) {
    file, err := os.Open(name)
    if err != nil {
        return nil, nil, err
    }
    return file, func() {
        file.Close()
    }, err
}
```

Эта вспомогательная функция возвращает файл, функцию и ошибку. Знак `*` здесь означает, что ссылка на файл в Go является указателем. Подробнее об этом будет рассказано в следующей главе.

После этого мы используем нашу функцию `getFile` внутри функции `main`:

```
f, closer, err := getFile(os.Args[1])
if err != nil {
    log.Fatal(err)
}
defer closer()
```

Поскольку в Go не допускается наличие неиспользуемых переменных, возврат функции `closer` из другой функции означает, что программа успешно скомпилируется лишь в том случае, если будет вызвана эта функция. Это является еще одним доводом в пользу использования оператора `defer`. Как уже говорилось ранее, указывая функцию `closer` в операторе `defer`, вы должны поставить и круглые скобки.



Такое использование оператора `defer` может показаться вам немного странным, если до этого вы использовали язык, в котором момент высвобождения ресурсов определяется с помощью размещаемых внутри функции блоков, таких как блоки `try/catch/finally` в Java, JavaScript и Python или блоки `begin/rescue/ensure` в Ruby.

Недостатком такого подхода является то, что эти блоки для высвобождения ресурсов создают дополнительный уровень отступа внутри функции, что делает код менее читабельным. Наличие вложенности усложняет понимание кода, и это не только мое личное мнение. В статье, опубликованной в 2017 году в журнале *Empirical Software Engineering* (<https://oreil.ly/VcYrR>), Вард Антинян (Vard Antinyan), Мирослав Старон (Miroslaw Staron) и Анна Сандберг (Anna Sandberg) описали результаты исследования, согласно которому «из одиннадцати предложенных характеристик кода только две заметно повышают степень его сложности: глубина вложенности и недостаточная структуризация».

Исследования в отношении того, что делает программы более читабельными и понятными, многократно проводились и ранее. Вы можете найти статьи по этой теме, опубликованные несколько десятилетий назад. В одной из этих работ, опубликованной в 1983 году (<https://oreil.ly/s0xcq>), Ричард Миара (Richard Miara), Джойс Муссеман (Joyce Musseman), Хуан Наварро (Juan Navarro) и Бен Шнайдерман (Ben Shneiderman) попытались определить наиболее приемлемую величину отступа (согласно их исследованиям отступ должен составлять от двух до четырех пробелов).

Go — язык с передачей параметров по значению

Возможно, вы уже слышали, что Go еще называют языком с *передачей параметров по значению*, и вас интересует, что же это означает. Это означает, что при передаче переменной функции в качестве параметра Go *всегда* создает копию значения переменной. Рассмотрим соответствующий пример кода, который доступен для запуска в онлайн-песочнице (https://oreil.ly/yo_rY). Сначала мы определяем простую структуру:

```
type person struct {
    age int
    name string
}
```


Затем мы определяем функцию, которая пытается изменить передаваемые ей значения типов `int`, `string` и `person`:

```
func modifyFails(i int, s string, p person) {
    i = i * 2
    s = "Goodbye"
    p.name = "Bob"
}
```

Теперь вызовем эту функцию внутри функции `main` и посмотрим, удастся ли ей произвести модификацию:

```
func main() {
    p := person{}
    i := 2
    s := "Hello"
    modifyFails(i, s, p)
    fmt.Println(i, s, p)
}
```

Запустив этот код, мы можем убедиться, что этой функции не удастся изменить значения, передаваемые ей в качестве параметров:

```
2 Hello {0 }
```

Я добавил в данный пример структуру `person`, чтобы показать, что так себя ведут не только простые типы. Если вам приходилось писать программы на Java, JavaScript, Python или Ruby, то вам может показаться странным такое поведение структур, поскольку эти языки позволяют модифицировать поля объекта, передаваемого функции в качестве параметра. Отличие языка Go в этом отношении объясняется причинами, которые мы обсудим, когда будем говорить об указателях.

Карты и срезы в такой ситуации ведут себя немного иначе. Посмотрим, что произойдет, если мы попробуем модифицировать их внутри функции. Вы можете запустить соответствующий код в онлайн-песочнице (<https://oreil.ly/kKL4R>). Здесь мы определяем две функции, одна из которых модифицирует передаваемую ей карту, а вторая модифицирует передаваемый ей срез:

```
func modMap(m map[int]string) {
    m[2] = "hello"
    m[3] = "goodbye"
    delete(m, 1)
}

func modSlice(s []int) {
    for k, v := range s {
        s[k] = v * 2
    }
    s = append(s, 10)
}
```

После этого мы вызываем эти функции внутри функции `main`:

```
func main() {  
    m := map[int]string{  
        1: "first",  
        2: "second",  
    }  
    modMap(m)  
    fmt.Println(m)  
  
    s := []int{1, 2, 3}  
    modSlice(s)  
    fmt.Println(s)  
}
```

Запустив этот код, мы получим достаточно интересный результат:

```
map[2:hello 3:goodbye]  
[2 4 6]
```

В случае карты мы можем легко объяснить полученный результат: все изменения, внесенные в переданную в качестве параметра карту, были отражены в исходной переменной. В случае среза дело обстоит немного сложнее: вы можете модифицировать любой элемент среза, но не можете увеличить его длину. Так ведут себя те карты и срезы, которые передаются в функцию напрямую, а также те карты и срезы, которые передаются в качестве полей структуры.

Данный пример кода рождает следующий вопрос: почему карты и срезы ведут себя не так, как другие типы? Это объясняется тем, что и карты, и срезы реализуются с помощью указателей. Подробнее об этом будет рассказано в следующей главе.



В Go каждый тип представляет собой значимый тип, просто иногда в качестве значения выступает указатель.

Именно передачей параметров по значению, помимо прочего, объясняется то, что ограниченную поддержку констант в Go не следует считать серьезным препятствием. Передача параметров по значению дает вам уверенность в том, что вызов функции не приведет к модификации той переменной, которая ей передается (если это не срез или карта). Такой подход будет полезен в большинстве существующих случаев. Когда функции не изменяют свои входные параметры и вместо этого возвращают вновь вычисленные значения, проще производить анализ существующих в программе потоков данных.

С другой стороны, хотя такой подход упрощает понимание кода, в некоторых случаях вам все же потребуется передать функции что-то способное изменяться. Что же следует делать в таких случаях? В таких случаях следует использовать указатель.

Резюме

В этой главе вы познакомились с функциями в Go и узнали, чем они похожи на функции в других языках, а также их уникальные особенности. В следующей главе, которая будет посвящена указателям, вы убедитесь, что они совсем не так страшны, как думают многие начинающие Go-разработчики, и узнаете, как можно использовать их преимущества для написания эффективных программ.

ГЛАВА 6

Указатели

Теперь, уже зная, как выглядят переменные и функции, перейдем к теме синтаксиса указателей. Затем мы подробнее рассмотрим поведение указателей в Go, сравнив его с поведением классов в других языках. Вы также узнаете, как и когда следует использовать указатели, правила использования памяти в Go, а также способы сделать Go-программы более быстрыми и эффективными за счет правильного использования указателей и значений.

Общие сведения об указателях

Указатель — это просто переменная, содержащая адрес ячейки памяти, в которой размещено значение. Если вы посещали лекции по теории вычислительных машин, то, вероятно, уже видели графические схемы размещения переменных в памяти. Схема размещения в памяти следующих двух переменных выглядит так, как показано на рис. 6.1.

```
var x int32 = 10
var y bool = true
```

Значение	0	0	0	10	1
Адрес	1	2	3	4	5
Переменная	x				y

Рис. 6.1. Размещение в памяти двух переменных

Каждая переменная хранится в одной или нескольких последовательных ячейках памяти с определенными адресами. При этом переменные, относящиеся к разным типам данных, могут занимать разный объем памяти. В данном примере у нас есть две переменные: переменная `x`, содержащая 32-разрядное целое число, и переменная `y`, содержащая булево значение. Поскольку для размещения

в памяти 32-разрядного целого числа требуется четыре байта, значение переменной `x` размещается в четырех байтах с адресами 1–4. Для размещения в памяти булевого значения требуется только один байт (строго говоря, для представления значений `true` и `false` нужен лишь один бит, но наименьший объем независимо адресуемой памяти составляет один байт), поэтому значение переменной `y` размещается в одном байте с адресом 5, где значение `true` представляется в виде единицы.

Указатель — это просто переменная, содержащая адрес, по которому расположена другая переменная. Как указатели размещаются в памяти, показано на рис. 6.2.

```
var x int32 = 10
var y bool = true
pointerX := &x
pointerY := &y
var pointerZ *string
```

Значение	0 _i	0 _i	0 _i	10	1	0 _i	0 _i	0 _i	1	0 _i	0 _i	0 _i	5	0 _i	0 _i	0 _i	0
Адрес	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Переменная	x				y	pointerX				pointerY				pointerZ			

Рис. 6.2. Размещение в памяти указателей

В то время как переменные разного типа могут занимать разное количество ячеек памяти, каждый указатель занимает одно и то же количество ячеек памяти, необходимое для размещения адреса той ячейки, которая содержит данные. В этом случае указатель на переменную `x`, `pointerX`, расположен в ячейке с адресом 6 и содержит значение 1 — адрес переменной `x`. Таким же образом указатель на переменную `y`, `pointerY`, расположен в ячейке с адресом 10 и содержит значение 5 — адрес переменной `y`. Последний указатель, `pointerZ`, расположен в ячейке с адресом 14 и содержит значение 0, поскольку он ни на что не указывает.

Нулевым значением указателей является значение `nil`. Мы уже несколько раз сталкивались со значением `nil` и знаем, что оно используется в качестве нулевого значения для срезов, карт и функций. Все эти типы реализованы с помощью указателей. (С помощью указателей также реализованы еще два типа: каналы и интерфейсы, которые будут подробно рассмотрены в разделе «Общее представление об интерфейсах» на с. 179 и в разделе «Каналы» на с. 255.) Как уже говорилось в главе 3, значение `nil` языка Go представляет собой нетипизированный идентификатор, представляющий случай отсутствия значения для определенных типов. В отличие от значения `NULL` языка C, значение `nil` не является просто псевдонимом для значения 0: вы не можете преобразовать значение `nil` в число и, наоборот, преобразовать число в значение `nil`.



Как упоминалось в главе 4, значение `nil` определено во всеобщем блоке. Это означает, что его можно затенить. Никогда не давайте переменной или функции имя `nil`. Исключением может быть лишь тот случай, когда вы хотите подшутить над коллегой и вас не волнует, какими будут результаты ежегодной оценки работы сотрудников.

Используемый в Go синтаксис указателей был частично заимствован из языков C и C++. Наличие в Go сборщика мусора позволило устранить практически все сложные моменты, связанные с управлением памятью. Кроме того, в Go нельзя использовать многие из тех возможностей, которые доступны при работе с указателями в C и C++, включая, в частности, *адресную арифметику*.



Вы все же можете выполнять над структурами данных некоторые низкоуровневые операции, используя пакет `unsafe` стандартной библиотеки. Но, в отличие от языка C, где манипуляции с указателями используются для выполнения распространенных операций, в Go пакет `unsafe` используется крайне редко. Этот пакет будет кратко рассмотрен в главе 14.

Символом `&` обозначается *оператор взятия адреса*. Он ставится перед типом значения и возвращает адрес ячейки памяти, в которой находится значение:

```
x := "hello"
pointerToX := &x
```

Символом `*` обозначается *оператор разыменования*. Он ставится перед переменной указательного типа и возвращает то значение, на которое она указывает, то есть производит так называемое *разыменование*:

```
x := 10
pointerToX := &x
fmt.Println(pointerToX) // выводит адрес в памяти
fmt.Println(*pointerToX) // выводит 10
z := 5 + *pointerToX
fmt.Println(z)           // выводит 15
```

Перед разыменованием указателя необходимо убедиться в том, что он не является нулевым. Если вы попытаетесь разыменовать указатель, равный `nil`, ваша программа выдаст панику:

```
var x *int
fmt.Println(x == nil) // выводит true
fmt.Println(*x)       // выдает панику
```

Указательный тип служит для представления указателя и обозначается с помощью символа `*` перед именем типа. Такой тип можно создать на основе любого типа данных:

```
x := 10
var pointerToX *int
pointerToX = &x
```

Встроенная функция `new` создает переменную указательного типа. Она возвращает указатель на экземпляр нулевого значения для заданного типа.

```
var x = new(int)
fmt.Println(x == nil) // выводит false
fmt.Println(*x)       // выводит 0
```

Функция `new` используется редко. В структурах экземпляр указателя следует создавать с помощью оператора `&` перед литералом структуры. Оператор `&` нельзя использовать перед литералами простых типов (числами, булевыми значениями и строками) или константами, потому что они не обладают адресом в памяти и существуют только во время компиляции. В случае, когда необходимо получить указатель на значение простого типа, следует объявить переменную и создать указывающий на нее указатель.

```
x := &Foo{}
var y string
z := &y
```

Отсутствие возможности получить адрес константы иногда причиняет неудобства. Если у вас есть структура, одно из полей которой содержит указатель на значение простого типа, вы не можете напрямую присвоить этому полю литерал:

```
type person struct {
    FirstName string
    MiddleName *string
    LastName  string
}

p := person{
    FirstName: "Pat",
    MiddleName: "Perry", // Эта строка не скомпилируется
    LastName:  "Peterson",
}
```

При попытке скомпилировать этот код вы увидите сообщение об ошибке:

```
cannot use "Perry" (type string) as type *string in field value
```

Если вы попытаетесь поставить оператор `&` перед строкой `"Perry"`, то увидите следующее сообщение об ошибке:

```
cannot take the address of "Perry"
```

Обойти эту проблему можно двумя способами. Первый способ сводится к тому, чтобы, как было показано выше, создать переменную, содержащую константное значение. Второй способ сводится к тому, чтобы написать вспомогательную функцию, которая будет принимать булево значение, число или строку и возвращать указатель на это значение:

```
func stringp(s string) *string {
    return &s
}
```

Определив такую функцию, можно написать следующее:

```
p := person{
    FirstName: "Pat",
    MiddleName: stringp("Perry"), // Это работает
    LastName: "Peterson",
}
```

Почему этот код работает? Когда мы передаем константу функции, эта константа копируется в параметр, который представляет собой переменную. Эта переменная уже обладает определенным адресом в памяти. Затем функция возвращает этот адрес.



Используйте вспомогательную функцию, когда нужно преобразовать константное значение в указатель.

Не бойтесь указателей

Первое правило в отношении указателей сводится к тому, что их не стоит бояться. Если раньше вы писали код на Java, JavaScript, Python или Ruby, то, возможно, немного побаиваетесь указателей. Однако на самом деле знакомые всем классы ведут себя практически так же. Действительно необычной вещью в Go являются не указатели, а неуказательные структурные типы.

В Java и JavaScript существует различие в поведении между простыми типами и классами (языки Python и Ruby не используют простые типы, а имитируют их с помощью неизменяемых экземпляров). Как показано в примере 6.1, когда значение простого типа присваивается другой переменной или передается функции или методу, вносимые в другую переменную изменения не отражаются в оригинале.

Пример 6.1. Присваивание значений простых типов в Java не ведет к совместному использованию памяти

```
int x = 10;
int y = x;
y = 20;
System.out.println(x); // выводит 10
```

Однако посмотрим, что произойдет в том случае, если мы присвоим другой переменной или передадим функции или методу экземпляр класса (в примере 6.2 код написан на Python, но на сайте GitHub можно найти и другие версии такого кода, написанные на Java, JavaScript и Ruby (<https://oreil.ly/9IpUK>)).

Пример 6.2. Передача в функцию экземпляра класса

```
class Foo:
    def __init__(self, x):
        self.x = x
```

```
def outer():
    f = Foo(10)
    inner1(f)
    print(f.x)
    inner2(f)
    print(f.x)
    g = None
    inner2(g)
    print(g is None)
```

```
def inner1(f):
    f.x = 20
```

```
def inner2(f):
    f = Foo(30)
```

```
outer()
```

Запустив этот код, мы увидим на экране следующее:

```
20
20
True
```

Это объясняется тем, что в Java, Python, JavaScript и Ruby справедливо следующее.

- Если вы передаете функции экземпляр класса и изменяете значение одного из полей, то изменение отражается в той переменной, которая была передана функции.

- Если вы заново присваиваете экземпляр класса параметру, это изменение *не* отражается в той переменной, которая была передана функции.
- Если в качестве параметра передается значение `nil/null/None`, то присвоение параметру нового значения не ведет к изменению переменной в вызывающей функции.

Иногда такое поведение объясняют тем, что в этих языках экземпляры классов передаются по ссылке, однако в действительности это не так. Если бы экземпляры классов передавались по ссылке, то во втором и третьем случаях происходило бы изменение переменной в вызывающей функции. Как и в Go, в этих языках параметры всегда передаются по значению.

На самом деле такое поведение объясняется тем, что каждый экземпляр класса в этих языках реализуется как указатель. Когда экземпляр класса передается функции или методу, копируемое значение представляет собой указатель на экземпляр. И поскольку функции `outer` и `inner1` ссылаются на одну и ту же область памяти, изменение полей переменной `f` внутри функции `inner1` также отражается в переменной `f` внутри функции `outer`. Когда функция `inner2` присваивает переменной `f` новый экземпляр класса, создается отдельный экземпляр, что не оказывает влияния на переменную `f` внутри функции `outer`.

При использовании переменной или параметра указательного типа в Go вы увидите в точности такое же поведение. Отличие языка Go от других языков здесь заключается в том, что и в случае простых, и в случае структурных типов вы можете *по своему выбору* использовать либо указатели, либо значения. В большинстве случаев следует использовать значения. Они облегчают понимание того, как и когда меняются данные. Еще одно преимущество состоит в том, что использование значений уменьшает объем работы, выполняемой сборщиком мусора. Мы обсудим это подробнее в разделе «Уменьшение нагрузки на сборщик мусора» на с. 159.

Указатели служат для указания изменяемых параметров

Как мы уже видели, в Go константы представляют собой имена для литеральных выражений, которые могут быть вычислены на этапе компиляции. В этом языке не предусмотрено никаких способов объявления неизменяемости других разновидностей значений. Однако в современной программной разработке поощряется неизменяемость. Основные доводы в пользу этого изложены в курсе по разработке ПО от Массачусетского технологического института (MIT) (<https://oreil.ly/FbUTJ>): «Неизменяемые типы меньше подвержены ошибкам, легче

поддаются пониманию и в большей мере готовы к модификации. Изменяемость затрудняет понимание того, что делает программа, и существенно усложняет соблюдение контрактов».

Отсутствие в Go способов объявления неизменяемости может показаться проблемой, но при этом у вас есть возможность выбирать для параметров либо значимый, либо указательный тип. Как объясняется далее в том же курсе по разработке ПО от MIT, «использование изменяемых объектов не представляет проблем в том случае, если они используются исключительно локально внутри методов и только с одной ссылкой на объект». Вместо того чтобы объявлять о том, что определенные переменные и параметры являются неизменяемыми, Go-разработчики указывают, что определенные параметры являются изменяемыми, используя для этого указатели.

Поскольку Go — язык с передачей параметров по значению, передаваемые функциям значения представляют собой копии. В случае таких неуказательных типов, как простые типы, структуры и массивы, это означает, что вызываемая функция не может изменить оригинал. Вызываемая функция получает копию исходных данных, что гарантирует неизменность исходных данных.



О том, что происходит в случае передачи функциям карт и срезов, будет рассказано в разделе «Различие между картами и срезами» на с. 154.

Однако при передаче указателя функция получает копию указателя, который по-прежнему указывает на исходные данные. Это означает, что в таком случае вызываемая функция может изменить оригинал.

Это влечет за собой два следствия.

Первое следствие состоит в том, что при передаче функции указателя, равного `nil`, невозможно сделать это значение не равным `nil`. Если указателю уже присвоено некоторое значение, вы можете только заново присвоить ему другое значение. Это может немного сбивать с толку поначалу, но если подумать об этом, то получается вполне логично. Поскольку функции по значению передается адрес ячейки памяти, мы не можем изменить этот адрес в памяти, подобно тому как мы не могли изменить значение при передаче указателя на значение типа `int`. Эту особенность демонстрирует следующая программа:

```
func failedUpdate(g *int) {  
    x := 10  
    g = &x  
}
```

```
func main() {
    var f *int // переменная f содержит nil
    failedUpdate(f)
    fmt.Println(f) // выводит nil
}
```

Что происходит по мере выполнения этого кода, показано на рис. 6.3.

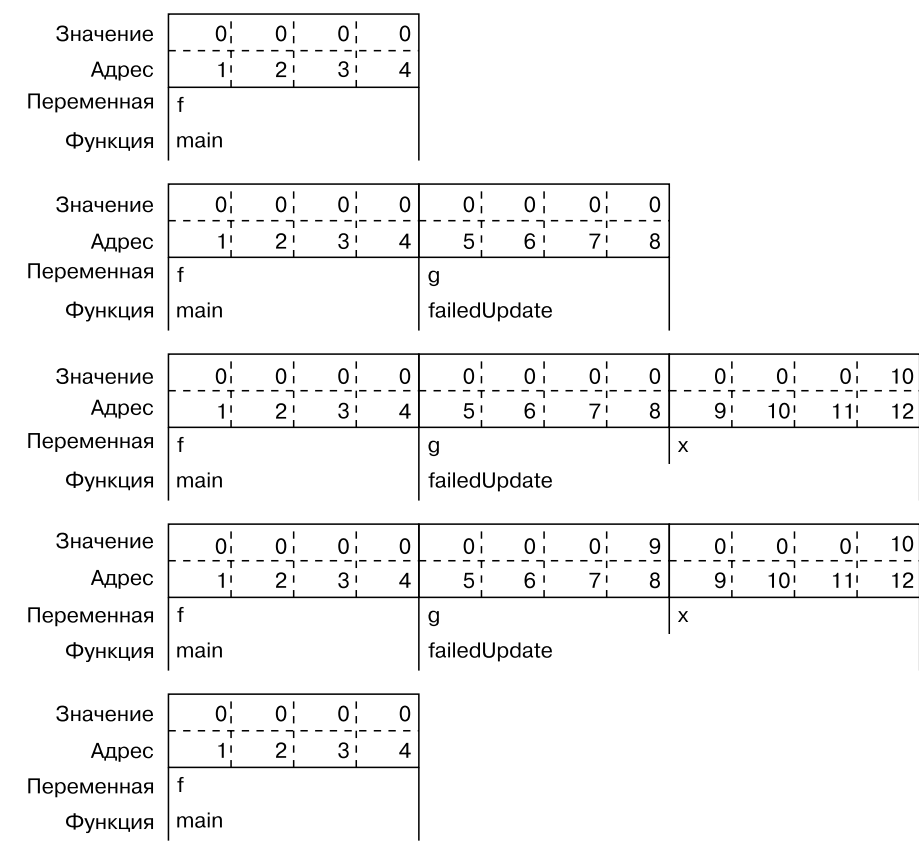


Рис. 6.3. Неудачное завершение попытки обновить указатель, равный nil

Мы начинаем здесь с объявления нулевой переменной `f` внутри функции `main`. При вызове функции `failedUpdate` значение переменной `f`, то есть `nil`, копируется в параметр `g`. Это значит, что параметр `g` тоже становится равным `nil`. После этого внутри функции `failedUpdate` объявляется новая переменная `x`, которой присваивается значение `10`. Затем внутри функции `failedUpdate` мы

изменяем переменную `g` таким образом, чтобы она указывала на переменную `x`. Это не ведет к изменению переменной `f` внутри функции `main`, и после выхода из функции `failedUpdate` обратно в функцию `main` переменная `f` по-прежнему остается равной `nil`.

Еще одно следствие из того факта, что функции передается копия указателя, сводится к следующему. Если вам нужно, чтобы значение, присваиваемое параметру указательного типа, не «пропадало» после выхода из функции, необходимо разыменовать указатель и присвоить ему значение. Если вы измените указатель, то изменится копия, а не оригинал. Однако, разыменовав указатель, вы поместите новое значение в ту ячейку памяти, на которую указывают и оригинал, и копия указателя. Как это работает, показывает следующая небольшая программа:

```
func failedUpdate(px *int) {
    x2 := 20
    px = &x2
}

func update(px *int) {
    *px = 20
}

func main() {
    x := 10
    failedUpdate(&x)
    fmt.Println(x) // выводит 10
    update(&x)
    fmt.Println(x) // выводит 20
}
```

Рисунок 6.4 показывает, что происходит по мере выполнения этого кода.

В данном примере мы начинаем с объявления внутри функции `main` переменной `x`, равной `10`. Когда мы вызываем функцию `failedUpdate`, адрес переменной `x` копируется в параметр `px`. После этого мы объявляем внутри функции `failedUpdate` переменную `x2` и присваиваем ей значение `20`. Затем внутри функции `failedUpdate` мы заносим в переменную `px` адрес переменной `x2`. После возвращения в функцию `main` значение переменной `x` остается неизменным. Вызывая функцию `update`, мы снова копируем адрес переменной `x` в параметр `px`. Однако на этот раз изменяется значение той переменной, на которую указывает переменная `px` внутри функции `update`, то есть переменной `x`, определенной внутри функции `main`. Соответственно, после возвращения в функцию `main` значение переменной `x` изменяется.

Значение	0	0	0	10
Адрес	1	2	3	4
Переменная	x			
Функция	main			

Значение	0	0	0	10	0	0	0	1
Адрес	1	2	3	4	5	6	7	8
Переменная	x				px			
Функция	main				failedUpdate			

Значение	0	0	0	10	0	0	0	1	0	0	0	20
Адрес	1	2	3	4	5	6	7	8	9	10	11	12
Переменная	x				px				x2			
Функция	main				failedUpdate							

Значение	0	0	0	10	0	0	0	9	0	0	0	20
Адрес	1	2	3	4	5	6	7	8	9	10	11	12
Переменная	x				px				x2			
Функция	main				failedUpdate							

Значение	0	0	0	10
Адрес	1	2	3	4
Переменная	x			
Функция	main			

Значение	0	0	0	10	0	0	0	1
Адрес	1	2	3	4	5	6	7	8
Переменная	x				px			
Функция	main				Update			

Значение	0	0	0	20	0	0	0	1
Адрес	1	2	3	4	5	6	7	8
Переменная	x				px			
Функция	main				Update			

Значение	0	0	0	20
Адрес	1	2	3	4
Переменная	x			
Функция	main			

Рис. 6.4. Неправильный и правильный способы обновления указателей

Указатели — это крайняя мера

Тем не менее использовать указатели в Go нужно крайне осмотрительно. Как уже упоминалось ранее, их использование затрудняет анализ потоков данных и может увеличивать объем работы, выполняемой сборщиком мусора. Вместо того чтобы заполнять структуру, передавая функции указатель на структуру, используйте функцию, создающую и возвращающую экземпляр структуры (примеры 6.3 и 6.4).

Пример 6.3. Не делайте так

```
func MakeFoo(f *Foo) error {
    f.Field1 = "val"
    f.Field2 = 20
    return nil
}
```

Пример 6.4. Делайте так

```
func MakeFoo() (Foo, error) {
    f := Foo{
        Field1: "val",
        Field2: 20,
    }
    return f, nil
}
```

Использовать параметр указательного типа для изменения переменной следует лишь в том случае, когда функция принимает интерфейс. Этот паттерн, в частности, используется при работе с форматом JSON (подробнее о поддержке формата JSON в стандартной библиотеке языка Go будет рассказано в разделе «Пакет `encoding/json`» на с. 295):

```
f := struct {
    Name string `json:"name"`
    Age int `json:"age"`
}{}
err := json.Unmarshal([]byte(`{"name": "Bob", "age": 30}`), &f)
```

Функция `Unmarshal` заполняет переменную из среза байтов, содержащего данные в формате JSON. Она принимает срез байтов и параметр типа `interface{}`. При этом в качестве параметра `interface{}` должен передаваться указатель; в противном случае эта функция возвращает ошибку. Использование такого паттерна обусловлено тем, что в Go нет обобщенных типов. Это означает, что в этом языке отсутствует удобный способ передачи типа в функцию, чтобы указать ей, что необходимо разгруппировать. И в принципе нет какого-либо способа указать нужный тип возвращаемого значения для разных типов.

Из-за широкого использования формата JSON начинающие Go-разработчики часто думают, что используемый в данном API подход является типичным, в то время как на самом деле его следует считать исключением.



Чтобы представить тип в виде переменной, в Go можно использовать тип `Туре` из пакета `reflect`. Однако пакет `reflect` следует применять лишь в тех случаях, когда задачу невозможно решить каким-либо другим способом. О рефлексии мы подробно поговорим в главе 14.

Для возвращения значений из функции старайтесь использовать значимые типы. Использовать указательный тип в качестве типа возвращаемых значений следует лишь в том случае, когда требуется изменить состояние типа данных. При обсуждении ввода-вывода в разделе «Пакет `io` и его друзья» на с. 286 будет показано, как это делается, на примере использования буферов для чтения или записи данных. Кроме того, в Go есть несколько типов данных, которые используются для реализации конкурентности и всегда должны передаваться как указатели. Вы познакомитесь с ними в главе 10.

Влияние передачи указателей на производительность

В случае достаточно больших структур использование указателя на структуру в качестве входного параметра или возвращаемого значения дает некоторый прирост производительности. Время, необходимое на передачу указателя функции, является неизменным для данных любых размеров и составляет приблизительно одну наносекунду. Это вполне логично, поскольку размер указателя является одинаковым для всех типов данных. Передача функции значения занимает все больше времени по мере увеличения размера данных и составляет примерно 1 миллисекунду, когда размер значения доходит до 10 Мбайт.

Возврат указателя дает более любопытный результат, чем возврат значения. Когда размер структуры данных составляет менее одного мегабайта, возвращение указательного типа вместо значимого на самом деле *снижает* производительность. Так, например, возвращение структуры данных размером 100 байт занимает около 10 наносекунд, а возвращение указателя на такую структуру данных — около 30 наносекунд. Однако когда размер структуры данных превышает один мегабайт, использование указателей, наоборот, дает положительный эффект. Таким образом, для возвращения данных размером 10 Мбайт требуется

почти 2 миллисекунды, а для возвращения указателя на такие данные — чуть больше половины миллисекунды.

В то же время имейте в виду, что это очень короткие промежутки времени. В подавляющем большинстве случаев различие в производительности между использованием указателей и значений никак не скажется на общей производительности программы. Но при передаче из одной функции в другую мегабайтов данных подумайте об использовании указателя, даже если эти данные не должны быть изменены.

Все приведенные здесь цифры были получены на компьютере с процессором i7-8700 и ОЗУ объемом 32 Гбайт. Вы можете провести собственное тестирование производительности, используя код с сайта GitHub (<https://oreil.ly/uVEin>).

Различие между нулевым значением и отсутствием значения

Указатели в Go также могут использоваться для обозначения различий между переменной или полем, которым было присвоено нулевое значение, и переменной или полем, которым вообще не было присвоено значение. Если это различие играет в программе важную роль, то для представления переменной или поля структуры, которым не было присвоено значение, следует использовать указатель, равный `nil`.

Поскольку указатели также служат для обозначения изменяемости, при использовании этого паттерна следует быть крайне внимательными. Вместо того чтобы возвращать из функции указатель, равный `nil`, воспользуйтесь идиомой «запятая-ок», с которой мы познакомились при обсуждении карт, и возвращайте значимый тип и булево значение.

Помните о том, что, если вы передаете функции указатель, равный `nil`, через параметр или поле параметра, вы не сможете присвоить ему значение внутри функции, поскольку это значение нигде будет разместить. Если вы передаете функции указатель, не равный `nil`, не изменяйте его; в противном случае следует задокументировать его поведение.

Преобразование JSON-данных — это исключение, которое подтверждает правило. При преобразовании данных в формат JSON и обратно (как уже говорилось, подробнее о поддержке формата JSON в стандартной библиотеке языка Go будет рассказано в разделе «Пакет `encoding/json`» на с. 295) часто требуется каким-то образом провести различие между нулевым значением и отсутствием

присвоенного значения. Используйте указательный тип для полей структуры, допускающих обновление.

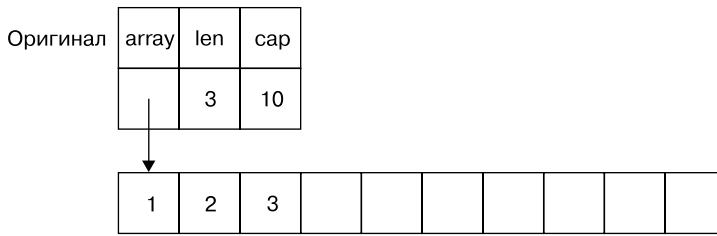
Если вы не работаете с форматом JSON (или другими внешними протоколами), старайтесь не использовать указательный тип для обозначения отсутствия значений. Использование указателя является удобным способом обозначения отсутствия значений, но если вы не собираетесь изменять значение, то вместо указателя следует использовать значимый тип и булево значение.

Различие между картами и срезами

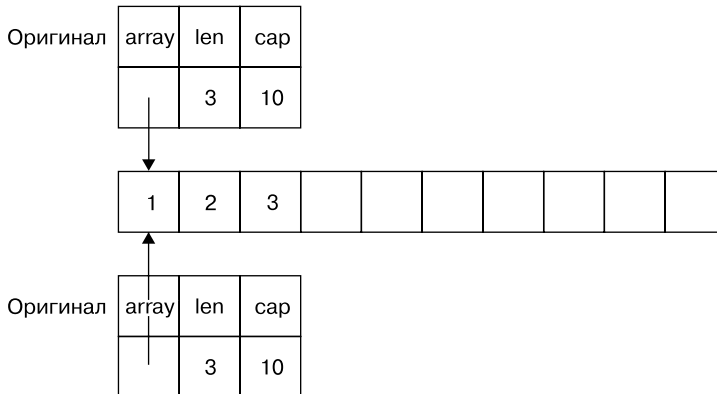
Как мы уже знаем из предыдущей главы, при модификации переданной функции карты все изменения отражаются в исходной переменной, которая была передана функции. Теперь, после знакомства с указателями, вы можете понять, почему так происходит: это объясняется тем, что в среде выполнения языка Go карта реализована как указатель на структуру. Передавая карту функции, вы фактически копируете указатель.

По этой причине карты не следует использовать в качестве входных параметров или возвращаемых значений, особенно при создании публичных API. С точки зрения дизайна API использование карт является плохой идеей из-за того, что они не дают никакой информации о том, какие значения в них содержатся; у вас нет никаких явных указаний о том, какие ключи содержит карта, и выяснить это можно, только проследив порядок выполнения предыдущего кода. Использовать карты нежелательно и с точки зрения неизменяемости данных, поскольку узнать, что будет содержать карта в итоге, можно только путем отслеживания всех взаимодействий функций с этой картой. Это не позволяет сделать API самодокументируемым. Если раньше вы использовали динамические языки, не используйте карту вместо отсутствующей в других языках структуры. Go — язык со строгой типизацией, и вместо того, чтобы передавать данные в виде карты, в нем следует использовать структуры. (Еще один довод в пользу применения структур будет приведен при обсуждении размещения данных в памяти в разделе «Уменьшение нагрузки на сборщик мусора» на с. 159.)

В то же время при передаче функции среза приходится иметь дело с более сложным поведением: хотя любое изменение содержимого среза отражается в исходной переменной, в ней не отражается изменение длины среза с помощью функции `append`, даже если емкость среза превышает его длину. Это объясняется тем, что срез в Go реализован как структура, содержащая три поля: поле типа `int` для длины среза, поле типа `int` для емкости и указатель на блок памяти. Как это выглядит, показано на рис. 6.5.

**Рис. 6.5.** Схема размещения в памяти среза

Когда срез копируется в другую переменную или передается функции, создаваемая копия включает в себя длину, емкость и указатель. Как показано на рис. 6.6, при этом обе переменные среза будут указывать на одну и ту же область памяти.

**Рис. 6.6.** Схема размещения в памяти среза и его копии

Изменение значений элементов среза ведет к изменениям в той области памяти, на которую указывает указатель, и потому эти изменения будут видны и в копии, и в оригинале. Что при этом происходит в памяти, показано на рис. 6.7.

Изменение длины и емкости среза не отражается в оригинале, поскольку при этом изменяется только копия. Изменение емкости означает, что указатель теперь будет указывать на новый, более крупный блок памяти. Как показано на рис. 6.8, теперь переменные среза будут указывать на разные блоки памяти.

Если в копию среза будут добавлены новые значения и при этом емкость среза будет достаточной для того, чтобы не выделять новый срез, то длина копии изменится, а новые значения будут сохранены в блоке памяти, совместно

используемом копией и оригиналом. Однако длина исходного среза при этом останется неизменной. Это значит, что среда выполнения языка Go не даст исходному срезу увидеть эти значения, поскольку они находятся за пределами его длины.

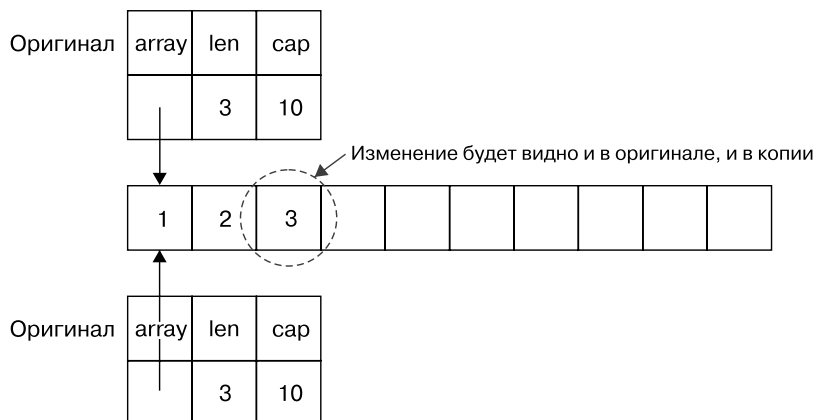


Рис. 6.7. Изменение содержимого среза

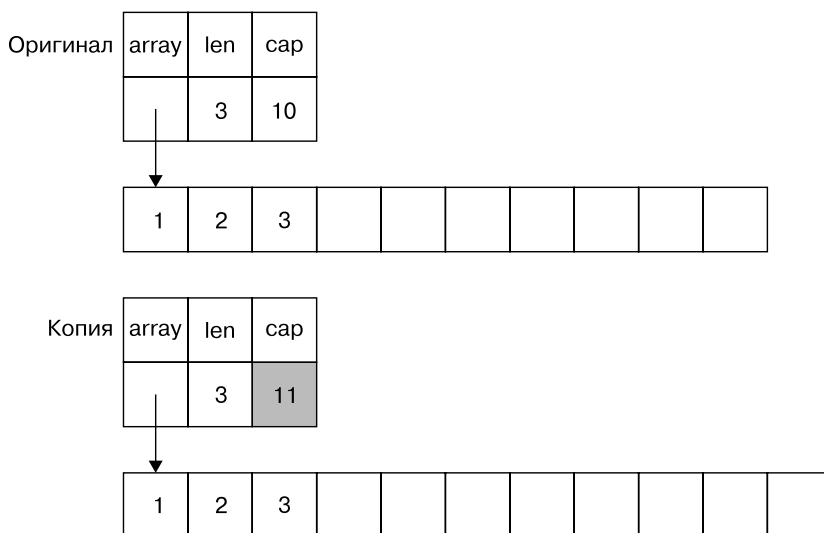


Рис. 6.8. Изменение емкости ведет к изменению объема выделенной памяти

На рис. 6.9 показано, какие значения будут видны в одной переменной среза, но не видны в другой.

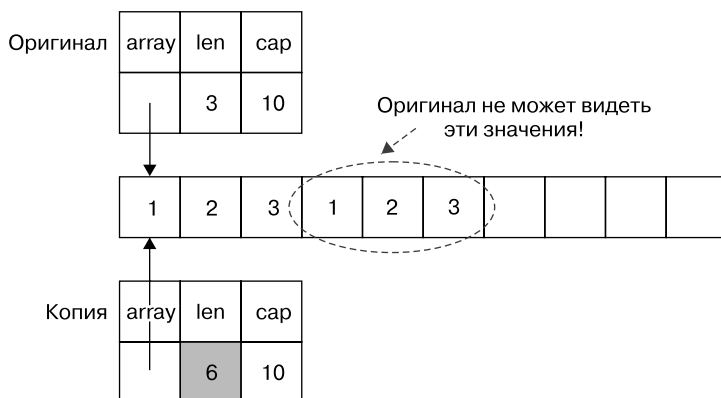


Рис. 6.9. Изменение длины не отражается в оригинале

В итоге при передаче функции среза вы можете изменить его содержимое, но не можете изменить его размер. Будучи единственной пригодной для использования линейной структурой данных, срезы часто передаются функциям в Go-программах. При этом по умолчанию предполагается, что срез не должен изменяться функцией. Если функция меняет содержимое среза, это должно быть явно указано в документации функции.



Вы можете передать функции срез любого размера, потому что при этом всегда передаются и те же данные: два значения типа `int` и указатель. Написать функцию, способную принимать массив любого размера, невозможно, так как при этом передается весь массив, а не только указатель на данные.

Использование срезов в качестве входных параметров может пригодиться еще в одном случае: их очень удобно применять в качестве многократно используемых буферов.

Использование срезов в качестве буферов

При чтении данных из внешнего ресурса (такого как файл или сетевое соединение) во многих языках используется код следующего вида:

```
r = open_resource()
while r.has_data() {
```

```
    data_chunk = r.next_chunk()
    process(data_chunk)
}
close(r)
```

Недостатком этого паттерна является то, что на каждой итерации цикла `while` в памяти размещается новая переменная `data_chunk`, которая используется всего один раз. В результате выполняется много лишних операций выделения памяти. Хотя в языках со сборкой мусора вам и не приходится управлять всем этим распределением памяти, при этом все равно проводится определенная работа по освобождению памяти после обработки данных.

Несмотря на то что Go является языком со сборкой мусора, идиоматический подход к написанию Go-кода подразумевает исключение ненужного распределения памяти. Вместо того чтобы выделять новую память при выполнении каждой операции чтения из источника данных, необходимо один раз создать срез байтов и использовать его в качестве буфера для чтения из источника данных:

```
file, err := os.Open(fileName)
if err != nil {
    return err
}
defer file.Close()
data := make([]byte, 100)
for {
    count, err := file.Read(data)
    if err != nil {
        return err
    }
    if count == 0 {
        return nil
    }
    process(data[:count])
}
```

Как вы помните, когда мы передаем срез функции, у нас нет возможности изменять его длину или емкость, но при этом мы можем изменять его содержимое в пределах текущей длины. В данном примере создается буфер из 100 байт и на каждой итерации цикла в срез копируется следующий блок байтов (до 100 байт). После этого заполненная часть буфера передается функции `process`. Более подробно о вводе-выводе будет рассказано в разделе «Пакет `io` и его друзья» на с. 286.

Уменьшение нагрузки на сборщик мусора

Использование буферов является лишь одним из примеров снижения объема работы, выполняемой сборщиком мусора. Под мусором в программировании понимаются данные, на которые больше не указывает ни один указатель. Когда уже не остается указателей, указывающих на определенные данные, занимаемую ими область памяти можно освободить для повторного использования. Если не производить такое освобождение, объем используемой программой памяти будет возрастать до тех пор, пока не будет исчерпана вся доступная на компьютере оперативная память. Задача сборщика мусора сводится к тому, чтобы выявлять неиспользуемые области памяти и высвобождать их для повторного использования. Наличие сборщика мусора в языке Go является его большим плюсом, поскольку, как показывает многолетняя практика, обеспечение надлежащего управления памятью вручную часто вызывает у программистов затруднения. Однако наличие в Go сборщика мусора совсем не означает, что мы должны производить неограниченно большое количество мусора.

Если вам уже приходилось изучать то, каким образом реализуются языки программирования, то вы, вероятно, уже знаете, что такое *куча* и *стек*. Если еще нет, то я кратко напомним, как работает стек. Стек представляет собой непрерывный блок памяти, который совместно используется всеми вызовами функций в потоке выполнения. Выделение памяти в стеке — быстрый и простой процесс. *Указатель стека* указывает на то место, где память выделялась в последний раз, и дополнительная память добавляется путем смещения указателя стека. В момент вызова функции для ее данных создается новый *стековый кадр*. В стеке сохраняются локальные переменные и передаваемые функции параметры. При этом каждая новая переменная смещает указатель стека на длину своего значения. Когда функция завершает свою работу, возвращаемые ею значения копируются назад в вызывающую функцию с помощью стека и указатель стека возвращается в начало стекового кадра для закончившей работу функции, высвобождая ту часть стековой памяти, которая была занята локальными переменными и параметрами этой функции.



Необычной особенностью языка Go является то, что он фактически позволяет увеличить размер стека во время выполнения программы. Это становится возможным благодаря тому, что каждая горутина обладает собственным стеком, и управление горутинами осуществляется средой выполнения языка Go, а не операционной системой (о горутинах мы поговорим подробнее в главе 10, посвященной конкурентности). Это дает свои плюсы и минусы. К плюсам можно отнести то, что стек в Go изначально обладает меньшим размером и занимает меньше памяти, а к минусам — то, что при увеличении стека требуется копировать все его содержимое и это отнимает много времени. При этом в самом худшем случае вы можете написать код, который будет попеременно заставлять стек расти и уменьшаться.

Чтобы сохранить что-либо в стеке, необходимо точно знать его размер на этапе компиляции. Если мы взглянем на значимые типы языка Go (простые типы, массивы и структуры), то увидим, что все они дают четкое представление об объеме памяти, занимаемой данными на этапе компиляции. Именно поэтому размер считается частью типа массива. Когда размер известен, можно выделить память в стеке, а не в куче. По этой причине указатели тоже размещаются в стеке.

Ситуация немного усложняется, когда дело касается данных, на которые указывает указатель. Для размещения таких данных в стеке должны выполняться несколько условий. Это должна быть локальная переменная с точно определенным размером содержащихся в ней данных на этапе компиляции. Указатель при этом не может быть возвращен из функции. Если указатель передается в функцию, то компилятор по-прежнему должен быть способным проследить за соблюдением этих условий. Когда размер данных не определен, вы не можете выделить для них пространство путем простого смещения указателя стека. Если указатель возвращается из функции, то данные, на которые он указывает, уже не будут корректными после выхода из функции. Когда компилятор определяет, что данные, на которые указывает указатель, невозможно разместить в стеке, то мы говорим, что данные покидают стек и сохраняются в куче.

Куча — это память, управление которой осуществляется сборщиком мусора (или вручную в таких языках, как C и C++). Мы не будем вдаваться здесь в детали того, как реализуется алгоритм сборщика мусора; отмечу лишь, что его реализация гораздо сложнее, чем простое смещение указателя стека. Размещенные в куче данные остаются корректными до тех пор, пока их можно проследить до размещенной в стеке переменной указательного типа. Когда уже не остается указателей, указывающих на эти данные (или на данные, указывающие на эти данные), эти данные становятся *мусором*, который должен быть удален сборщиком мусора.



Распространенным источником ошибок в программах на C является возвращение из функции указателя на локальную переменную. В языке C это дает в результате указатель, указывающий на некорректные данные в памяти. Компилятор Go более сообразителен. Заметив, что функция возвращает указатель на локальную переменную, он размещает значение этой переменной в куче.

Выполняя *escape-анализ* (анализ на предмет «убегания» памяти в кучу), компилятор делает это далеко не идеально. В некоторых случаях данные убегают в кучу, хотя их можно было бы разместить в стеке. Однако компилятор вынужден проявлять сдержанность, поскольку не может допустить того, чтобы в стеке оказалось значение, которое должно находиться в куче; в противном случае ссылка на недоступные данные приведет к нарушению целостности данных

в памяти. С выходом новых релизов языка Go эффективность escape-анализа постепенно повышается.

Вы можете спросить: а что плохого в том, чтобы размещать данные в куче? Это влечет за собой две проблемы, которые негативно влияют на производительность. Первая проблема состоит в том, что процесс сборки мусора занимает определенное время. Контроль за тем, какие области памяти в куче еще свободны и какие из используемых блоков памяти еще обладают корректными указателями, — далеко не простая задача. Чем больше времени тратится на решение этой задачи, тем меньше остается на выполнение той обработки, для выполнения которой была написана ваша программа. Множество написанных к настоящему времени алгоритмов сборки мусора можно грубо разделить на две категории: алгоритмы, призванные обеспечить максимальную пропускную способность (то есть выявление максимального количества мусора за одно сканирование), и алгоритмы, призванные обеспечить минимальную величину задержки (то есть максимально быстрое сканирование на наличие мусора). В статье, опубликованной в 2013 году (<https://oreil.ly/cvLpa>), одним из авторов которой был Джефф Дин (Jeff Dean), чей гениальный ум стоял за многими успешными разработками компании Google, рекомендуется оптимизировать системы в отношении времени задержки, чтобы обеспечить минимальное время отклика. Сборщик мусора, используемый средой выполнения языка Go, прежде всего стремится обеспечить минимальное время задержки. Каждый цикл сборки мусора при этом занимает не более 500 микросекунд. Однако если Go-программа создает много мусора, сборщик мусора не может выявлять весь мусор за один цикл, что замедляет работу сборщика и повышает расход памяти.



Если вы хотите узнать чуть больше о том, как реализована сборка мусора в Go, ознакомьтесь с докладом, представленным Риком Хадсоном (Rick Hudson) на Международном симпозиуме по управлению памятью (International Symposium on Memory Management, ISMM) в 2018 году, в котором он коснулся истории и деталей реализации сборщика мусора языка Go (<https://oreil.ly/UUhGK>).

Вторая проблема обусловлена особенностями аппаратного обеспечения компьютеров. Хотя оперативная память и позволяет осуществлять произвольный доступ, она обеспечивает более высокую скорость при последовательном чтении данных. Срез структур в Go обеспечивает последовательное размещение данных в памяти, что ускоряет их чтение и обработку. Однако когда мы имеем дело со срезом указателей на структуры (или со срезом структур, поля которых являются указателями), данные разбросаны по всей оперативной памяти, что замедляет их чтение и обработку. Форрест Смит (Forrest Smith) опубликовал в своем блоге статью (https://oreil.ly/v_urr), в которой проводится глубокий анализ того, насколько это может влиять на производительность. Приводимые им

цифры показывают, что при произвольном доступе к данным через указатели скорость снижается примерно на два порядка.

Такой подход, при котором программное обеспечение создается с учетом особенностей аппаратного обеспечения, предназначенного для его запуска, называется *механической симпатией*. Данный термин пришел из мира автомобильных гонок, где он означает, что водитель, который понимает, как работает его автомобиль, может наилучшим образом выжать из него максимальную мощность. В 2011 году Мартин Томпсон (Martin Thompson) начал использовать этот принцип в отношении программной разработки. Используя рекомендуемые методы работы в Go, вы обеспечите соблюдение этого принципа автоматически.

Если мы сравним используемый в Go подход с подходом Java, то увидим, что в Java, как и в Go, локальные переменные и параметры сохраняются в стеке. Однако, как уже упоминалось, объекты в Java реализованы как указатели. Это означает, что в случае любого экземпляра объектной переменной в стеке размещается только указатель на этот объект, а его данные размещаются в куче. Целиком в стеке размещаются только значения простых типов (числа, булевы значения и символы). Это означает, что сборщику мусора в Java приходится выполнять очень большой объем работы. Еще один вывод состоит в том, что такие вещи, как список (list), в Java на самом деле представляют собой указатель на массив указателей. Хотя такой список выглядит как линейная структура данных, для чтения его данных потребуется «прыгать» из одной части памяти в другую, что крайне неэффективно. Примеры подобного поведения также присутствуют и в языках Python, Ruby и JavaScript. Для избавления от всей этой неэффективности в виртуальную машину Java (Java Virtual Machine, JVM) включен ряд мощных сборщиков мусора, способных справляться с большим объемом работы. Некоторые из них оптимизированы для получения максимальной пропускной способности, другие — для получения минимальной задержки, и все они обладают параметрами конфигурации, которые можно настраивать для получения наилучшей производительности. Виртуальные машины языков Python, Ruby и JavaScript не могут похвастаться такой степенью оптимизации и, соответственно, демонстрируют более низкую производительность.

Теперь становится понятно, почему в Go рекомендуется использовать указатели как можно реже. Тем самым снижается нагрузка на сборщик мусора за счет того, что максимально возможная часть данных сохраняется в стеке. При использовании срезов структур или простых типов данные располагаются в памяти последовательно, что обеспечивает высокую скорость доступа. А когда сборщик мусора все же принимается за свою работу, он стремится затратить на нее как можно меньше времени, вместо того чтобы пытаться собрать как можно больше мусора. Ключ к повышению эффективности этого подхода в создании меньшего количества мусора. Хотя кто-то может посчитать такую оптимизацию операций

выделения памяти преждевременной, следует отметить, что для обеспечения максимальной эффективности в Go достаточно просто придерживаться идиоматического подхода.

Если вы хотите узнать больше об особенностях escape-анализа в Go и различиях между распределением памяти в куче и в стеке, то в интернете можно найти отличные статьи по этой теме. В частности, рекомендую вам прочитать статью Билла Кеннеди (Bill Kennedy) из компании Arden Labs (<https://oreil.ly/juu44>) и статью Ахилла Руссела (Achille Roussel) и Рика Брэнсона (Rick Branson) из компании Segment (https://oreil.ly/c_gvC).

Резюме

В этой главе мы немного коснулись того, что «скрывается за кулисами», чтобы иметь более четкое представление о том, что собой представляют указатели, как их следует использовать и, что важнее всего, когда их следует использовать. В следующей главе вы узнаете, как в Go реализованы методы, интерфейсы и типы, чем они отличаются от других языков и какими возможностями они вас наделяют.

ГЛАВА 7

Типы, методы и интерфейсы

Как мы видели в предыдущих главах, Go — статически типизированный язык, позволяющий использовать как встроенные, так и пользовательские типы. Подобно большинству современных языков, Go позволяет привязывать к типам методы. В этом языке также есть абстракция типов, что дает возможность вызывать методы в коде без явного указания реализации.

Однако подход Go к методам, интерфейсам и типам сильно отличается от подходов большинства других широко используемых языков. Язык Go создан с расчетом на применение практик, рекомендуемых разработчиками программного обеспечения, которые исключают наследование, поощряя при этом использование композиции. В этой главе вы познакомитесь с типами, методами и интерфейсами и узнаете, как их следует использовать для создания программ, легко поддающихся тестированию и сопровождению.

Типы в Go

В разделе «Структуры» на с. 82 было показано, как в Go определяются структурные типы:

```
type Person struct {  
    FirstName string  
    LastName  string  
    Age      int  
}
```

Этот код представляет собой объявление пользовательского типа с именем `Person`, базовым типом которого является литерал структуры указанного вида. Помимо литерала структуры, для определения конкретного типа можно использовать любой простой тип или литерал составного типа. Вот несколько примеров:

```
type Score int  
type Converter func(string)Score  
type TeamScores map[string]Score
```

Go позволяет объявить тип на любом уровне блоков вплоть до уровня блока пакета. Однако получить доступ к типу можно только внутри его области видимости. Единственным исключением из этого правила являются экспортируемые типы, объявленные на уровне блока пакета. Подробнее о них будет рассказано в главе 9.



Чтобы нам было легче вести разговор о типах, дам определение следующим двум понятиям. Абстрактный тип указывает, что должен делать тип, но не указывает, как это должно быть сделано. Конкретный тип указывает, что и как должен делать тип. Это значит, что он обладает определенным методом сохранения своих данных и предоставляет реализацию для всех объявленных в нем методов. В то время как в Go все типы являются либо абстрактными, либо конкретными, в некоторых языках можно использовать комбинированные типы, такие как абстрактные классы или интерфейсы с методами по умолчанию в Java.

Методы

Подобно большинству современных языков, Go позволяет дополнять пользовательские типы методами. Определение методов для типа производится на уровне блока пакета:

```
type Person struct {
    FirstName string
    LastName  string
    Age       int
}

func (p Person) String() string {
    return fmt.Sprintf("%s %s, age %d", p.FirstName, p.LastName, p.Age)
}
```

Объявление метода выглядит так же, как объявление функции, но с одним отличием: здесь дополнительно указывается *приемник метода*. Приемник метода указывается после ключевого слова `func` перед именем метода. Как и при объявлении любой другой переменной, сначала указывается имя приемника, а затем — его тип. Согласно общепринятому соглашению имя приемника должно представлять собой сокращение от имени типа: обычно используется только первая буква имени типа. Использование в качестве имени приемника слова `this` или `self` не соответствует идиоматическому подходу.

Как и имена функций, имена методов нельзя перегружать. Вы можете использовать одно и то же имя метода для разных типов, но нельзя использовать одно и то же имя метода для определения двух разных методов для одного и того же

типа. При переходе с языков, допускающих перегрузку методов, такой подход может восприниматься как ограничение, но отказ от повторного использования имен вполне согласуется с заложенным в Go принципом четкого выражения в коде его назначения.

Подробнее о пакетах будет рассказано в главе 9, а пока я упомяну лишь, что методы должны объявляться в том же пакете, где объявляется соответствующий тип: Go не позволяет добавлять методы в неконтролируемые вами типы. Хотя вы можете определить метод в другом файле в пределах того же пакета, в котором определяется тип, рекомендуется определять типы и связанные с ними методы в одном месте, чтобы сделать код реализации более понятным.

Вызов метода не будет выглядеть для вас как что-то новое, если вам приходилось использовать методы в других языках:

```
p := Person {
    FirstName: "Fred",
    LastName: "Fredson",
    Age: 52,
}
output := p.String()
```

Приемники указателей и приемники значений

Как мы уже говорили в главе 6, в Go параметры указательного типа используются для обозначения, что параметр может быть изменен функцией. Те же правила действуют и в отношении приемников методов. Они могут представлять собой *приемники указателей* (когда используется указательный тип) или *приемники значений* (когда используется значимый тип). Определиться с тем, когда следует использовать тот или иной вид приемника, вам помогут следующие правила.

- Если метод вносит изменения в приемник, *необходимо* использовать приемник указателей.
- Если метод должен учитывать вероятность того, что экземпляр будет равен `nil` (см. подраздел «Пишите код методов с расчетом на экземпляр, равный `nil`» на с. 168), *необходимо* использовать приемник указателей.
- Если метод не вносит изменения в приемник, *можно* использовать приемник значений.

Следует ли использовать приемник значений для метода, не изменяющего этот приемник, зависит от того, какие еще методы объявляются для данного типа. Если *хотя бы один* из определяемых для типа методов использует приемник указателей, для единообразия рекомендуется использовать приемник указателей во *всех* методах, включая те из них, которые не меняют приемник.

Использование приемника указателей и приемника значений демонстрирует представленный ниже простой пример кода. Сначала мы определяем тип и два метода для него, один из которых использует приемник указателей, а второй — приемник значений:

```
type Counter struct {
    total      int
    lastUpdated time.Time
}

func (c *Counter) Increment() {
    c.total++
    c.lastUpdated = time.Now()
}

func (c Counter) String() string {
    return fmt.Sprintf("total: %d, last updated: %v", c.total, c.lastUpdated)
}
```

После этого мы можем опробовать эти методы в деле, как показано ниже. Вы можете запустить этот код в онлайн-песочнице (<https://oreil.ly/aqY0i>):

```
var c Counter
fmt.Println(c.String())
c.Increment()
fmt.Println(c.String())
```

На экран будет выведено следующее:

```
total: 0, last updated: 0001-01-01 00:00:00 +0000 UTC
total: 1, last updated: 2009-11-10 23:00:00 +0000 UTC m=+0.000000001
```

Как вы могли заметить, мы смогли вызвать метод, использующий приемник указателей, несмотря на то что переменная с обладает значимым типом. При использовании приемника указателей для локальной переменной значимого типа Go автоматически преобразует эту переменную в указательный тип. То есть в данном случае `c.Increment()` преобразуется в `(&c).Increment()`.

Однако имейте в виду, что здесь по-прежнему действуют правила в отношении передачи значений функциям. Если вы передадите значимый тип функции и вызовете для переданного значения метод, использующий приемник указателей, то этот метод будет вызван для *копии*. Попробуйте запустить в онлайн-песочнице следующий код (<https://oreil.ly/bGdDi>):

```
func doUpdateWrong(c Counter) {
    c.Increment()
    fmt.Println("in doUpdateWrong:", c.String())
}
```

```
func doUpdateRight(c *Counter) {
    c.Increment()
    fmt.Println("in doUpdateRight:", c.String())
}

func main() {
    var c Counter
    doUpdateWrong(c)
    fmt.Println("in main:", c.String())
    doUpdateRight(&c)
    fmt.Println("in main:", c.String())
}
```

После запуска кода вы получите следующий результат:

```
in doUpdateWrong: total: 1, last updated: 2009-11-10 23:00:00 +0000 UTC
m=+0.000000001
in main: total: 0, last updated: 0001-01-01 00:00:00 +0000 UTC
in doUpdateRight: total: 1, last updated: 2009-11-10 23:00:00 +0000 UTC
m=+0.000000001
in main: total: 1, last updated: 2009-11-10 23:00:00 +0000 UTC m=+0.000000001
```

Параметр метода `doUpdateRight` относится к типу `*Counter`, то есть является экземпляром указательного типа. Как видите, мы можем вызвать для него и метод `Increment`, и метод `String`. Go включает в *набор методов* экземпляра указательного типа и те методы, которые используют приемник указателей, и те методы, которые используют приемник значений. В случае экземпляра значимого типа в набор методов включаются только методы с приемником значений. В данный момент эти подробности могут показаться излишними, но они понадобятся нам чуть позже, когда мы коснемся интерфейсов.

Еще одно, последнее замечание: создавать методы-получатели и методы-установщики для структур в Go следует лишь в том случае, если это требуется для того, чтобы обеспечить соответствие интерфейсу (об интерфейсах мы начнем говорить в разделе «Общее представление об интерфейсах» на с. 179). Доступ к полям в Go рекомендуется выполнять напрямую, используя методы только для бизнес-логики. Исключением из этого правила являются только те случаи, когда требуется обновить сразу несколько полей или когда обновление не представляет собой простое присвоение нового значения. Метод `Increment` из рассмотренного выше примера подпадает под оба этих случая.

Пишите код методов с расчетом на экземпляр, равный nil

Когда чуть выше мы говорили об использовании экземпляров указательного типа, у вас мог возникнуть вопрос о том, что произойдет в том случае, если мы вызовем метод для экземпляра, равного `nil`. В большинстве других языков вы

получите при этом сообщение об ошибке. (Язык Objective-C позволяет вам вызывать метод для экземпляра, равного `nil`, но не производит при этом никаких действий.)

Язык Go ведет себя в таком случае немного иначе. Он действительно пытается вызвать метод. Если это метод, использующий приемник значений, вы получите панику (о том, что это такое, мы поговорим в разделе «Функции `panic` и `recover`» на с. 216) из-за отсутствия значения в том месте, куда указывает указатель. Если это метод, использующий приемник указателей, то он сможет работать, когда его код будет написан с учетом того, что экземпляр может быть равным значению `nil`.

На самом деле, если брать в расчет, что экземпляр может быть равным `nil`, можно упростить код. Например, вероятность того, что приемником может быть значение `nil`, с успехом используется в следующей реализации двоичного дерева:

```
type IntTree struct {
    val      int
    left, right *IntTree
}

func (it *IntTree) Insert(val int) *IntTree {
    if it == nil {
        return &IntTree{val: val}
    }
    if val < it.val {
        it.left = it.left.Insert(val)
    } else if val > it.val {
        it.right = it.right.Insert(val)
    }
    return it
}

func (it *IntTree) Contains(val int) bool {
    switch {
    case it == nil:
        return false
    case val < it.val:
        return it.left.Contains(val)
    case val > it.val:
        return it.right.Contains(val)
    default:
        return true
    }
}
```

Ниже показано, как можно использовать это дерево. Вы можете запустить этот код в онлайн-песочнице (<https://oreil.ly/-F2i->):

```
func main() {
    var it *IntTree
    it = it.Insert(5)
    it = it.Insert(3)
    it = it.Insert(10)
    it = it.Insert(2)
    fmt.Println(it.Contains(2)) // выводит true
    fmt.Println(it.Contains(12)) // выводит false
}
```



Метод `Contains` не меняет значение `*IntTree`, несмотря на то что объявлен как метод с приемником указателей. Это является примером применения упомянутого выше правила в отношении поддержки приемника, равного `nil`. Метод с приемником значений не может выполнять проверку на равенство значению `nil` и, как уже говорилось, выдает панику в случае его вызова для приемника, равного `nil`.

Это очень здорово, что язык Go позволяет вызывать методы для приемника, равного `nil`, и иногда эта возможность может быть очень полезной, как в нашем примере с узлами дерева. Однако в большинстве случаев это не несет какой-либо пользы. Приемник указателей действует точно так же, как параметр функций указательного типа: это копия указателя, которая передается в метод. Как и в случае передачи функции параметра, равного `nil`, изменение этой копии указателя не приведет к изменению оригинала. Это означает, что вы не можете написать метод с приемником указателей, который бы принимал указатель, равный `nil`, и делал исходный указатель не равным `nil`. Если ваш метод использует приемник указателей и не работает, когда приемник равен `nil`, выполняйте проверку на равенство значению `nil` и возвращайте ошибку (об ошибках мы поговорим в главе 8).

Методы тоже являются функциями

Методы в Go настолько мало отличаются от функций, что вы можете использовать метод вместо функции везде, где используется переменная или параметр функционального типа.

Сначала определим простейший структурный тип:

```
type Adder struct {
    start int
}

func (a Adder) AddTo(val int) int {
    return a.start + val
}
```

После этого мы можем в обычной манере создать экземпляр данного типа и вызвать его метод:

```
myAdder := Adder{start: 10}
fmt.Println(myAdder.AddTo(5))    // выводит 15
```

Мы также можем присвоить метод переменной и передать его в качестве параметра типа `func(int)int`. Это называют *значением метода*.

```
f1 := myAdder.AddTo
fmt.Println(f1(10))              // выводит 20
```

Значение метода имеет определенное сходство с замыканием, поскольку может обращаться к значениям полей того экземпляра, на основе которого оно было создано.

Вы также можете создать функцию непосредственно на основе самого типа. Это называют *выражением метода*.

```
f2 := Adder.AddTo
fmt.Println(f2(myAdder, 15))    // выводит 25
```

В случае выражения метода в качестве первого параметра указывается приемник метода; в данном случае сигнатура функции выглядит как `func(Adder, int) int`.

Значения метода и выражения метода не представляют собой просто интересный исключительный случай. Мы рассмотрим один из способов их использования, когда будем говорить о внедрении зависимостей в разделе «Неявные интерфейсы облегчают внедрение зависимостей» на с. 195.

Функции или методы?

Поскольку метод можно использовать в качестве функции, возникает вопрос: когда следует использовать функцию, а когда — метод?

Ключевым фактором здесь является зависимость функции от других данных. Как я уже неоднократно упоминал, состояние на уровне пакета должно быть фактически неизменным. Во всех тех случаях, когда логика зависит от значений, которые настраиваются на этапе запуска или изменяются во время выполнения программы, сохраняйте эти значения в структуре и реализуйте логику в виде метода. В тех случаях, когда логика зависит только от входных параметров, следует использовать функцию.

Типы, пакеты, модули, тестирование и внедрение зависимостей являются тесно связанными концепциями. О внедрении зависимостей мы подробно поговорим

чуть позже в этой главе. Пакеты и модули мы подробно обсудим в главе 9, а тестирование — в главе 13.

Объявление типа не является наследованием

Помимо объявления типа на основе одного из встроенных типов языка Go или литерала структуры, вы также можете объявлять пользовательский тип на основе другого пользовательского типа:

```
type HighScore Score
type Employee Person
```

Хотя существует много концепций, которые можно считать «объектно-ориентированными», *наследование* занимает среди них центральное место. Данный принцип подразумевает доступность в *дочернем* типе состояния и методов *родительского* типа с возможностью замены значений дочернего типа значениями родительского типа. (Для тех читателей, которые хорошо знакомы с теорией вычислительных систем, замечу: я понимаю, что подтипизация не является наследованием. Но, поскольку в большинстве языков программирования наследование используется для реализации подтипизации, в литературе, рассчитанной на массового читателя, эти понятия часто несут один и тот же смысл.)

Хотя объявление типа на основе другого типа выглядит как наследование, оно им не является. Сходство здесь состоит лишь в том, что два типа обладают одним и тем же базовым типом. Между этими типами нет никакой иерархии. В языках с наследованием дочерний экземпляр может быть использован в любом из тех мест, где используется родительский экземпляр. Дочерний экземпляр также обладает всеми методами и структурами данных родительского экземпляра. В Go дело обстоит иначе. Вы не можете присвоить экземпляр типа `HighScore` переменной типа `Score` и наоборот без преобразования типа, равно как не можете присвоить эти экземпляры переменной типа `int` без преобразования типа. Кроме того, методы, определенные для типа `Score`, не определены для типа `HighScore`:

```
// вы можете присвоить значения нетипизированным константам
var i int = 300
var s Score = 100
var hs HighScore = 200
hs = s           // ошибка компиляции!
s = i           // ошибка компиляции!
s = Score(i)    // ok
hs = HighScore(s) // ok
```

В случае пользовательских типов, базовым типом которых являются встроенные типы, пользовательский тип можно использовать в сочетании с операторами для

этих типов. Как показано в представленном выше примере, такие типы также могут представлять собой литералы или константы с присвоенными значениями, совместимыми с базовым типом.



Преобразование типа в тип с таким же базовым типом оставляет неизменным его место размещения в памяти, но при этом связывает с ним другие методы.

Типы являются исполняемой документацией

Хорошо известно, что для хранения набора связанных данных следует объявлять структурный тип, но в случае пользовательских типов уже совсем не так ясно, когда следует объявлять пользовательский тип на основе одного из встроенных типов, а когда на основе другого пользовательского типа. Краткий ответ на этот вопрос сводится к тому, что типы являются документацией. Они делают код более понятным, предоставляя имена для концепций и описывая, какие данные должны использоваться в том или ином месте. Если методу будет передаваться параметр типа `Percentage`, а не типа `int`, ваш код будет более понятным для других программистов и вероятность того, что они передадут этому методу некорректное значение, будет гораздо ниже.

Та же логика справедлива и в случае объявления пользовательского типа на основе другого пользовательского типа. Если над одинаковыми базовыми данными требуется выполнять разные наборы операций, создайте два типа. При этом объявление одного типа на основе другого позволяет в определенной мере избежать повторения и ясно показывает, что эти два типа взаимосвязаны.

Йота (иногда) используется для создания перечислений

Во многих языках программирования есть концепция перечислений, позволяющая указать, что тип может иметь только ограниченный набор значений. В Go нет перечисляемых типов. Вместо них в этом языке присутствует такая вещь, как *йота* (*iota*), с помощью которой вы можете присвоить ряд инкрементно возрастающих значений набору констант.

При использовании *iota* рекомендуется сначала на основе типа `int` определить тип, который будет служить для представления всех допустимых значений:

```
type MailCategory int
```

Затем нужно определить набор значений для этого типа с помощью блока `const`:

```
const (
    Uncategorized MailCategory = iota
    Personal
    Spam
    Social
    Advertisements
)
```



Концепция йоты взята из языка программирования APL (что расшифровывается как A Programming Language — «язык программирования»). Язык APL известен тем, что настолько сильно полагался на использование собственной нотации, что даже требовал использования компьютеров со специальной клавиатурой. Например, вот как на этом языке выглядела программа для поиска всех простых чисел вплоть до значения переменной R : $(\sim R \in R^{\circ} \times R) / R \leftarrow 1 \downarrow R$.

В том, что такой сфокусированный на читабельности язык, как Go, заимствовал концепцию из языка, отличающегося чрезмерной краткостью, можно увидеть некую иронию, однако это лишний раз показывает, почему мы должны знать много разных языков программирования: любой из них может стать источником вдохновения.

Мы указали тип для первой константы в блоке `const` и присвоили ей значение `iota`. Во всех последующих строках уже не указывается тип и не присваивается значение. Когда компилятор языка Go видит этот код, он повторяет тип и операцию присваивания для всех последующих констант в блоке, каждый раз инкрементно увеличивая значение `iota`. Это означает, что он присваивает `0` первой константе (`Uncategorized`), `1` — второй константе (`Personal`) и т. д. Если мы решим использовать еще один блок `const`, значение `iota` будет снова сброшено в `0`.

Вот лучший из тех советов по использованию `iota`, которые мне попадались на глаза.

Не используйте йоту для определения констант, если их значения явно определены (в другом месте). Например, если вы реализуете некоторые части спецификации и в спецификации указано, какие значения следует присваивать каждой константе, вы должны явно записать значения констант. Используйте йоту только для «внутреннего использования», то есть там, где обращение к константам выполняется по имени, а не по значению. Так вы сможете оптимально использовать преимущества йоты, вставляя новые константы в любой момент времени и в любом месте списка, не рискуя что-либо нарушить.

Дэнни ван Хоймен (Danny van Heumen)
(<https://oreil.ly/3MKwn>)

Важно понимать, что в Go ничто не мешает вам добавить в определенный вами тип дополнительные значения. Кроме того, если вы вставите новый идентификатор в середину своего списка литералов, все последующие константы будут перенумерованы. Это внесет в ваше приложение трудноуловимую ошибку, если значения этих констант будут использоваться в другой системе или в базе данных. В силу этих двух ограничений использовать перечисления на основе `iota` имеет смысл лишь в том случае, когда нужно просто отличать друг от друга некоторый ряд значений, и неважно, чему именно они будут равны. Если фактическое значение константы играет важную роль, его следует указать явным образом.



Поскольку константам можно присваивать литеральные выражения, вы можете встретить примеры кода, в которых предлагается использовать `iota` следующим образом:

```
type BitField int

const (
    Field1 BitField = 1 << iota // присваивается 1
    Field2                      // присваивается 2
    Field3                      // присваивается 4
    Field4                      // присваивается 8
)
```

Каким бы умным и продвинутым ни казалось это решение, будьте крайне внимательны и документируйте свои действия, если решили использовать этот паттерн. Как уже упоминалось, использование йоты для создания констант в том случае, когда значение констант играет важную роль, повышает вероятность возникновения ошибок. Вы же не хотите, чтобы программист, который будет сопровождать ваш код в дальнейшем, нарушил его работу, вставив новую константу в середину списка.

Помните о том, что нумерация `iota` начинается с нуля. Если набор констант используется для представления различных состояний конфигурации, нулевое значение будет совсем не лишним, как мы уже видели в случае типа `MailCategory`. При поступлении электронного письма оно изначально не относится ни к одной из категорий, что делает вполне логичным использование нулевого значения для константы *Uncategorized*. Если для константы невозможно выбрать имеющее смысл значение по умолчанию, обычно рекомендуется в качестве первого значения йоты в блоке констант задать идентификатор `_` или константу, показывающую, что значение является некорректным. Это позволяет легко выявлять те случаи, когда не удастся произвести инициализацию переменной должным образом.

Используйте встраивание для реализации композиции

Совет о том, что в ходе разработки программного обеспечения следует отдавать предпочтение объектной композиции, а не наследованию классов, был изложен еще в 1994 году в книге «Паттерны проектирования» от «банды четырех»¹. Язык Go не позволяет использовать наследование, но поощряет повторное использование кода, предоставляя встроенную поддержку для композиции и повышения типа:

```
type Employee struct {
    Name      string
    ID         string
}

func (e Employee) Description() string {
    return fmt.Sprintf("%s (%s)", e.Name, e.ID)
}

type Manager struct {
    Employee
    Reports []Employee
}

func (m Manager) FindNewEmployees() []Employee {
    // выполнение бизнес-логики
}
```

Обратите внимание, что структура `Manager` содержит поле типа `Employee`, но этому полю не присваивается какое-либо имя. Это делает поле `Employee` *встроенным полем*. Любые поля или методы, объявленные во встроенном поле, *повышаются* до содержащей его структуры и могут быть вызваны непосредственно в ней. Это позволяет нам написать следующий код:

```
m := Manager{
    Employee: Employee{
        Name:      "Bob Bobson",
        ID:         "12345",
    },
    Reports: []Employee{},
}
fmt.Println(m.ID)           // выводит 12345
fmt.Println(m.Description()) // выводит "Bob Bobson" (12345)
```

¹ Гамма Э., Хелм Р., Джонсон Р., Влиссидес Д. Паттерны объектно-ориентированного проектирования. — СПб.: Питер, 2020.



Вы можете встроить в структуру любой тип, а не только другую структуру. Это позволяет повышать методы встроеного типа до содержащей его структуры.

Если вмещающая структура имеет поля или методы с таким же именем, как у встроеного поля, то для обращения к этим затененным полям и методам следует использовать тип встроеного поля. Например, допустим, у нас есть типы, определенные следующим образом:

```
type Inner struct {
    X int
}

type Outer struct {
    Inner
    X int
}
```

В таком случае можно обратиться к полю X структуры Inner, явно указав имя структуры Inner:

```
o := Outer{
    Inner: Inner{
        X: 10,
    },
    X: 20,
}
fmt.Println(o.X)           // выводит 20
fmt.Println(o.Inner.X)    // выводит 10
```

Встраивание не является наследованием

Поддержкой встраивания обладают очень немногие языки программирования (насколько мне известно, ни один из других популярных языков не поддерживает встраивание). Многие из тех разработчиков, которые уже знакомы с концепцией наследования (присутствующей во многих языках), пытаются использовать встраивание, понимая его как наследование. Это может закончиться весьма плачевно. Вы не можете присвоить переменную типа `Manager` переменной типа `Employee`. Если вам нужно обратиться к полю `Employee` в структуре `Manager`, необходимо сделать это явным образом. Попробуйте запустить в онлайн-песочнице следующий код (<https://oreil.ly/vBI7o>):

```
var eFail Employee = m           // ошибка компиляции!
var eOK Employee = m.Employee   // ok!
```

Вы получите следующее сообщение об ошибке:

```
cannot use m (type Manager) as type Employee in assignment
```

Кроме того, Go не обеспечивает *динамическую диспетчеризацию* конкретных типов. Методы встроенного поля не знают о том, что они являются встроенными. Если метод встроенного поля будет вызывать другой метод этого встроенного поля и вмещающая структура будет обладать методом с таким же именем, то метод встроенного поля не будет вызывать метод вмещающей структуры. Эту особенность поведения показывает следующий пример кода, который вы можете запустить в онлайн-песочнице (<https://oreil.ly/yN6bV>):

```
type Inner struct {
    A int
}

func (i Inner) IntPrinter(val int) string {
    return fmt.Sprintf("Inner: %d", val)
}

func (i Inner) Double() string {
    return i.IntPrinter(i.A * 2)
}

type Outer struct {
    Inner
    S string
}

func (o Outer) IntPrinter(val int) string {
    return fmt.Sprintf("Outer: %d", val)
}

func main() {
    o := Outer{
        Inner: Inner{
            A: 10,
        },
        S: "Hello",
    }
    fmt.Println(o.Double())
}
```

Запустив этот код, вы получите следующий результат:

```
Inner: 20
```

Хотя встраивание одного конкретного типа в другой не позволяет вам использовать внешний тип в качестве внутреннего типа, методы встроенного поля добавляются в набор методов вмещающей структуры. Это означает, что бла-

годаря встраиванию можно обеспечить реализацию интерфейса вмещающей структурой.

Общее представление об интерфейсах

Хотя язык Go больше славится своей моделью конкурентности (о которой мы поговорим в главе 10), действительно яркой особенностью этого языка являются неявные интерфейсы — единственный абстрактный тип в Go. Посмотрим, что делает этот элемент языка настолько замечательным.

Сначала кратко коснемся того, как производится объявление интерфейсов. Здесь, по сути, нет ничего сложного. Как и в случае других пользовательских типов, для этого нужно воспользоваться ключевым словом `type`.

Вот, например, как выглядит определение интерфейса `Stringer` из пакета `fmt`:

```
type Stringer interface {  
    String() string  
}
```

В объявлении интерфейса после имени интерфейсного типа записывается литерал интерфейса. Он содержит список методов, которые должен реализовать конкретный тип, чтобы соответствовать интерфейсу. Определяемые интерфейсом методы называют «набором методов» интерфейса.

Как и другие типы, интерфейсы могут быть объявлены в любом блоке.

Имя интерфейса обычно оканчивается на `er`. Выше уже упоминался интерфейс `fmt.Stringer`, однако существует и много других таких имен: например, `io.Reader`, `io.Closer`, `io.ReadCloser`, `json.Marshaler`, `http.Handler`.

Интерфейсы обеспечивают типобезопасную утиную типизацию

Все сказанное до сих пор практически ничем не отличается от принципов работы интерфейсов в других языках. Что делает интерфейсы языка Go особенными, так это то, что они реализуются *неявным образом*. Конкретный тип не объявляет о том, что он реализует интерфейс. Если набор методов конкретного типа содержит все методы из набора методов интерфейса, то этот конкретный тип реализует интерфейс. Это означает, что этот конкретный тип может быть присвоен переменной или полю, в качестве типа которого указан данный интерфейс.

Такое неявное поведение делает интерфейсы самым интересным элементом системы типов языка Go, способным обеспечить типобезопасность в сочетании с низкой связанностью, объединяя возможности и статических, и динамических языков.

Чтобы понять, почему так, вспомним о том, зачем вообще в языках присутствует такая вещь, как интерфейсы. Выше упоминалось о том, что в книге *«Паттерны проектирования»* рекомендовалось отдавать предпочтение композиции, а не наследованию. Еще один совет из этой книги звучит так: «Программируйте на уровне интерфейса, а не на уровне реализации». В таком случае вы будете зависеть только от поведения, а не от реализации, что позволит при необходимости заменить одну реализацию на другую. Это делает возможным дальнейшее совершенствование кода по мере неизбежного изменения требований.

В языках с динамической типизацией, таких как Python, Ruby и JavaScript, нет интерфейсов. Вместо интерфейсов в этих языках используется так называемая утиная типизация, в основе которой лежит следующий принцип: «Если что-то ходит и крикает как утка, то это утка». Эта концепция подразумевает, что вы можете передать экземпляр типа в качестве параметра в функцию при условии, что у этого типа есть ожидаемый функцией метод:

```
class Logic:
    def process(self, data):
        # бизнес-логика

def program(logic):
    # получение данных
    logic.process(data)

logicToUse = Logic()
program(logicToUse)
```

Хотя такой подход на первый взгляд кажется немного странным, утиная типизация с успехом используется при создании крупных систем. Но если вы программируете на языке со статической типизацией, это может выглядеть для вас как полнейшая неразбериха. Ведь без явного указания типа трудно понять, какой именно функциональности следует ожидать. По мере того как к работе над проектом будут подключаться новые разработчики, а старые будут забывать, что происходит в том или ином месте программы, им придется просматривать весь код, чтобы выяснить, какие именно зависимости в нем присутствуют.

Java-разработчики используют другой паттерн. Они определяют интерфейс и создают реализацию этого интерфейса, но ссылаются на него только в клиентском коде:

```
public interface Logic {
    String process(String data);
}

public class LogicImpl implements Logic {
    public String process(String data) {
        // бизнес-логика
    }
}

public class Client {
    private final Logic logic;
    // этот тип является интерфейсом, а не реализацией

    public Client(Logic logic) {
        this.logic = logic;
    }

    public void program() {
        // получение данных
        this.logic(data);
    }
}

public static void main(String[] args) {
    Logic logic = new LogicImpl();
    Client client = new Client(logic);
    client.program();
}
```

Взглянув на явные интерфейсы языка Java, разработчики, использующие динамические языки, могут задаться вопросом: как будет производиться рефакторинг кода с течением времени при наличии таких явных зависимостей? Ведь для обеспечения перехода на использование новой реализации от другого поставщика придется переписать код таким образом, чтобы он зависел от нового интерфейса.

Разработчики языка Go посчитали верным и первый, и второй подход. Если ожидается, что приложение будет расти и изменяться со временем, то потребуется гибкость в плане изменения реализации. Но, чтобы ваш код был понятен тем разработчикам, которые с течением времени будут подключаться к работе над вашим кодом, вы также должны указать, от чего зависит ваш код. Именно здесь нам могут пригодиться неявные интерфейсы. В Go используется сочетание двух описанных выше подходов:

```
type LogicProvider struct {}

func (lp LogicProvider) Process(data string) string {
    // бизнес-логика
}
```

```

type Logic interface {
    Process(data string) string
}

type Client struct{
    L Logic
}

func(c Client) Program() {
    // получение данных
    c.L.Process(data)
}

main() {
    c := Client{
        L: LogicProvider{},
    }
    c.Program()
}

```

В этом Go-коде присутствует интерфейс, но об этом знает только вызывающая сторона (**Client**): в объявлении структуры **LogicProvider** нет никаких явных указаний на то, что она соответствует интерфейсу. Это позволяет одновременно и обеспечить возможность перехода в будущем на использование логики от нового поставщика, и предоставить исполняемую документацию, чтобы гарантировать, что передаваемый клиенту тип будет всегда отвечать его требованиям.



Интерфейсы указывают, что требуется вызывающей стороне. Клиентский код определяет интерфейс, чтобы указать, какая функциональность ему нужна.

Сказанное выше совсем не означает, что в Go не допускается совместное использование интерфейсов. Выше уже упоминались несколько интерфейсов из стандартной библиотеки, которые используются для ввода-вывода. Наличие стандартного интерфейса дает вам мощные возможности: так, используя в своем коде интерфейсы **io.Reader** и **io.Writer**, вы сможете обеспечить корректное чтение и запись вне зависимости от того, с чем именно вы работаете: с файлом на локальном диске или с данными в памяти.

Кроме того, наличие стандартных интерфейсов делает возможным использование *паттерна «Декоратор»*. В Go часто используются фабричные функции, которые принимают экземпляр интерфейса и возвращают другой тип, который реализует тот же интерфейс. Например, допустим, что у вас есть функция со следующим определением:

```
func process(r io.Reader) error
```

В таком случае для обработки данных из файла можно использовать следующий код:

```
r, err := os.Open(fileName)
if err != nil {
    return err
}
defer r.Close()
return process(r)
```

Экземпляр `os.File`, возвращаемый функцией `os.Open`, соответствует интерфейсу `io.Reader` и может быть использован в любом коде, читающем данные. Если же вы имеете дело со сжатым файлом в формате GZIP, то можете обернуть интерфейс `io.Reader` еще одним интерфейсом `io.Reader`:

```
r, err := os.Open(fileName)
if err != nil {
    return err
}
defer r.Close()
gz, err = gzip.NewReader(r)
if err != nil {
    return err
}
defer gz.Close()
return process(gz)
```

Теперь тот же код, который производил чтение из несжатого файла, вместо этого производит чтение из сжатого файла.



Если в стандартной библиотеке есть интерфейс, определяющий то, что нужно вашему коду, используйте его!

Ничто не мешает типу, который соответствует интерфейсу, определить дополнительные методы, помимо методов, входящих в интерфейс. Эти методы могут использоваться лишь в определенной части клиентского кода. Так, например, тип `io.File` одновременно соответствует и интерфейсу `io.Reader`, и интерфейсу `io.Writer`. Если вашему коду требуется только чтение из файла, используйте для обращения к экземпляру файла интерфейс `io.Reader`, не принимая во внимание другие методы.

Встраивание и интерфейсы

Подобно встраиванию типа в структуру также можно встроить интерфейс в интерфейс. Так, например, интерфейс `io.ReadCloser` включает в себя интерфейсы `io.Reader` и `io.Closer`:

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



Подобно встраиванию конкретного типа в структуру также можно встроить интерфейс в структуру. В каких случаях это может потребоваться, будет рассказано в разделе «Заглушки в Go» на с. 348.

Принимайте интерфейсы, возвращайте структуры

Опытные Go-разработчики часто рекомендуют принимать интерфейсы и возвращать структуры. Это означает, что выполняемая функциями бизнес-логика должна вызываться посредством интерфейсов, а результаты функций должны представлять собой конкретные типы. Как мы уже выяснили, функции должны принимать интерфейсы по причине того, что они делают код более гибким и явно объявляют, какая именно функциональность используется в функции.

Но при создании API, возвращающего интерфейсы, вы теряете одно из главных преимуществ неявных интерфейсов: низкую связанность. Вы ограничиваете себя теми сторонними интерфейсами, от которых зависит ваш клиентский код, потому что ваш код будет всегда зависеть от модуля, содержащего эти интерфейсы, а также от всех зависимостей этого модуля и т. д. (О модулях и зависимостях мы поговорим в главе 9.) Это будет ограничивать гибкость кода в дальнейшем. Для снижения связанности вам придется написать еще один интерфейс и преобразовать один интерфейс в другой, используя преобразование типов. Когда код зависит от конкретных экземпляров, это может породить зависимости, но вы можете свести данный эффект к минимуму,

используя в своем приложении слой внедрения зависимостей. Внедрение зависимостей будет подробно рассмотрено в разделе «Неявные интерфейсы облегчают внедрение зависимостей» на с. 195.

Еще одним аргументом против возвращения интерфейсов является управление версиями. При возвращении конкретного типа вы можете добавлять новые методы и поля без нарушения работы существующего кода. Однако то же самое уже не будет справедливым в случае возвращения интерфейса. При добавлении нового метода в интерфейс необходимо обновить все существующие реализации этого интерфейса: в противном случае это нарушит работу вашего кода. Если же вы решите внести в API изменения без сохранения обратной совместимости, вам потребуется увеличить основной номер версии.

Вместо того чтобы создавать одну фабричную функцию, возвращающую различные экземпляры интерфейса в зависимости от входных параметров, старайтесь создавать отдельные фабричные функции для каждого конкретного типа. Однако в некоторых случаях (как, например, при создании парсера, способного возвращать один или несколько различных видов лексем) это неизбежно, и придется возвращать интерфейс.

Исключением из этого правила являются ошибки. Как мы увидим в главе 8, в Go функции и методы могут объявлять возвращаемый параметр интерфейсного типа `error`. При этом очень высока вероятность того, что будут возвращаться разные реализации этого интерфейса, и поэтому вы должны использовать интерфейс для обработки всех возможных вариантов, поскольку интерфейсы являются единственным абстрактным типом в Go.

У этого паттерна есть один потенциальный недостаток. Как упоминалось в разделе «Уменьшение нагрузки на сборщик мусора» на с. 159, уменьшение объема памяти, выделяемой в куче, ведет к повышению производительности по причине снижения объема работы, выполняемой сборщиком мусора. Возвращение структуры позволяет обойтись без выделения памяти в куче, и это очень хорошо. Однако при вызове функции с параметрами интерфейсного типа для каждого из этих параметров память выделяется в куче. По мере эксплуатации программы вам нужно будет найти оптимальный баланс между степенью абстракции и производительностью. Прежде всего, старайтесь сделать свой код читабельным и легко поддающимся сопровождению. Если ваша программа работает слишком медленно и профилирование показывает, что причиной низкой производительности является выделение памяти в куче для параметра интерфейсного типа, перепишите функцию таким образом, чтобы она использовала параметр конкретного типа. В случае, когда в функцию передается несколько реализаций интерфейса, требуется создание нескольких функций с повторяющейся логикой.

Интерфейсы и значение `nil`

При обсуждении указателей в главе 6 было сказано, что нулевым значением для указательных типов является значение `nil`. Значение `nil` также используется для представления нулевого экземпляра интерфейса, но это не так легко, как для конкретных типов.

Интерфейс считается равным `nil` в том случае, когда значению `nil` равны и тип, и значение. Так, следующий код выводит значение `true` в первых двух строках и значение `false` в последней строке:

```
var s *string
fmt.Println(s == nil) // выводит true
var i interface{}
fmt.Println(i == nil) // выводит true
i = s
fmt.Println(i == nil) // выводит false
```

Вы можете запустить этот код в онлайн-песочнице (<https://oreil.ly/NBPbC>).

В среде выполнения языка Go интерфейсы реализуются как два указателя, один из которых указывает на используемый тип, а другой — на используемое значение. Если тип не равен `nil`, то и интерфейс не равен `nil`. (Поскольку у вас не может быть переменной без типа, если указатель на значение не равен `nil`, то указатель на тип никогда не будет равен `nil`.)

В случае интерфейса значение `nil` указывает, можно или нет вызывать для него методы. Поскольку, как упоминалось ранее, методы можно вызывать для равных `nil` экземпляров конкретного типа, в данном случае вполне логично то, что вы можете вызывать методы для переменной интерфейсного типа, которой был присвоен равный `nil` экземпляр конкретного типа. Если интерфейс равен `nil`, то вызов для него любых методов приведет к панике (подробнее о паниках будет рассказано в разделе «Функции `panic` и `recover`» на с. 216). Если интерфейс не равен `nil`, то вы можете вызывать для него методы. (Однако имейте в виду, что ваш код может выдать панику и в этом случае, если значение `nil` будет присвоено типу, методы которого не рассчитаны на обработку значения `nil`.)

Поскольку экземпляр интерфейса не равен `nil`, если его тип не равен `nil`, то очень сложно определить, равно ли `nil` связанное с интерфейсом значение. Чтобы выяснить это, придется воспользоваться рефлексией (о том, как это делается, будет рассказано в подразделе «Используйте рефлексии для проверки значения интерфейса на равенство значению `nil`» на с. 370).

Пустой интерфейс ничего не сообщает

Иногда в языке со статической типизацией требуется определенным образом сообщить о том, что переменная может содержать значение любого типа. В Go для этой цели предназначен пустой интерфейс `interface{}`:

```
var i interface{}
i = 20
i = "hello"
i = struct {
    FirstName string
    LastName  string
} {"Fred", "Fredson"}
```

Следует отметить, что синтаксис пустого интерфейса, `interface{}`, не представляет собой какой-то особый случай. Пустой интерфейсный тип просто сообщает, что переменная может содержать любое значение, тип которого реализует ноль или более методов. И так оказалось, что под это определение подходят все существующие в Go типы. Поскольку пустой интерфейс ничего не сообщает о том значении, которое он представляет, ему можно найти не так уж много вариантов применения. В частности, он широко используется в качестве заглушки для данных с неопределенной схемой размещения, считываемых из такого внешнего источника, как файл формата JSON:

```
// одна пара фигурных скобок используется для типа interface{},
// вторая пара служит для создания экземпляра карты
data := map[string]interface{}{}
contents, err := ioutil.ReadFile("testdata/sample.json")
if err != nil {
    return err
}
defer contents.Close()
json.Unmarshal(contents, &data)
// содержимое переменной contents теперь находится в карте data
```

Из-за того что в Go пока нет пользовательских обобщенных типов, пустой интерфейс также может использоваться для сохранения значений в пользовательской структуре данных. Если вам нужна некоторая другая структура данных, помимо срезов, массивов и карт, способная принимать данные нескольких типов, то для размещения значений этой структуры следует использовать поле типа `interface{}`. Попробуйте запустить в онлайн-песочнице следующий код (<https://oreil.ly/SBisO>):

```
type LinkedList struct {
    Value interface{}
    Next  *LinkedList
}
```

```
func (ll *LinkedList) Insert(pos int, val interface{}) *LinkedList {
    if ll == nil || pos == 0 {
        return &LinkedList{
            Value: val,
            Next: ll,
        }
    }
    ll.Next = ll.Next.Insert(pos-1, val)
    return ll
}
```



Данный код нельзя назвать эффективной реализацией вставки для связанного списка; я привожу его здесь лишь в силу его компактности. Пожалуйста, не используйте его в реальных программах.

Когда функция принимает пустой интерфейс, она, как правило, использует рефлексию (о которой мы поговорим в главе 14) либо для заполнения, либо для чтения значений. Так, в приведенном ранее примере тип `interface{}` объявлен в качестве типа второго параметра функции `json.Unmarshal`.

Поскольку такие ситуации случаются редко, по возможности не используйте пустой интерфейс. Как уже говорилось, Go — язык со строгой типизацией, и попытки обойти эту его особенность противоречат идиоматическому подходу.

В том случае, когда требуется размещать значение в пустом интерфейсе, возникает вопрос о том, как производить чтение этого значения. Чтобы это сделать, мы должны познакомиться с утверждениями типа и переключателями типа.

Утверждения типа и переключатели типа

В Go существует два способа выполнения проверки, не обладает ли переменная интерфейсного типа некоторым конкретным типом и не реализует ли конкретный тип другой интерфейс. Начнем с рассмотрения *утверждений типа*. Утверждение типа определяет конкретный тип, который реализовал интерфейс, или другой интерфейс, который также реализуется базовым конкретным типом интерфейса. Попробуйте запустить в онлайн-песочнице следующий код (https://oreil.ly/_nUSw):

```
type MyInt int
```

```
func main() {  
    var i interface{}  
    var mine MyInt = 20  
    i = mine  
    i2 := i.(MyInt)  
    fmt.Println(i2 + 1)  
}
```

Здесь переменная `i2` обладает типом `MyInt`.

Вы можете спросить: что произойдет при некорректном использовании утверждения типа? В таком случае код выдаст панику. Попробуйте запустить пример такого случая в онлайн-песочнице (https://oreil.ly/qoXu_):

```
i2 := i.(string)  
fmt.Println(i2)
```

Этот код выдает панику:

```
panic: interface conversion: interface {} is main.MyInt, not string
```

Как мы уже знаем, Go очень щепетильно относится к конкретным типам. Даже если два типа имеют общий базовый тип, утверждение типа должно соответствовать типу базового значения. Следующий код тоже выдаст панику. Попробуйте запустить его в онлайн-песочнице (<https://oreil.ly/YUaka>):

```
i2 := i.(int)  
fmt.Println(i2 + 1)
```

Очевидно, что сбой в работе программы будет нежелательным поведением. Чтобы не допустить этого, можно воспользоваться идиомой «запятая-ok», как это делалось в подразделе «Идиома “запятая-ok”» на с. 79 для проверки наличия нулевого значения:

```
i2, ok := i.(int)  
if !ok {  
    return fmt.Errorf("unexpected type for %v", i)  
}  
fmt.Println(i2 + 1)
```

Булева переменная `ok` устанавливается в `true` в случае успешного преобразования типа. В противном случае переменной `ok` присваивается значение `false`, а второй переменной (переменной `i2`) — соответствующее нулевое значение. Далее внутри оператора `if` обрабатывается случай получения неожиданного типа, но тут следует отметить, что в соответствии с идиоматическим подходом в Go код для обработки ошибок должен быть отделен отступом. Подробнее об обработке ошибок будет рассказано в главе 8.



Между операциями утверждения типа и преобразования типа имеется существенная разница. Операции преобразования типа могут применяться и к конкретным типам, и к интерфейсам и проверяются на этапе компиляции. Операции утверждения типа могут применяться только к интерфейсным типам и проверяются на этапе выполнения программы. По причине того, что проверка проводится на этапе выполнения, они могут вызвать сбой программы. Операции преобразования типа меняют тип, а операции утверждения типа его раскрывают.

Даже если вы абсолютно уверены в корректности используемого вами утверждения типа, используйте его в сочетании с соответствующей версией идиомы «запятая-ок». Ведь никто не знает, как этот код будут повторно использовать другие разработчики (или вы сами спустя полгода). Рано или поздно ваши утверждения типа могут стать некорректными и вызвать сбой на этапе выполнения программы.

Когда в качестве интерфейса может выступать один из нескольких возможных типов, вместо этого следует использовать *переключатель типа*:

```
func doThings(i interface{}) {
    switch j := i.(type) {
    case nil:
        // переменная i равна nil, переменная j обладает типом interface{}
    case int:
        // переменная j обладает типом int
    case MyInt:
        // переменная j обладает типом MyInt
    case io.Reader:
        // переменная j обладает типом io.Reader
    case string:
        // переменная j обладает типом string
    case bool, rune:
        // переменная i содержит булево значение или руну,
        // поэтому переменная j обладает типом interface{}
    default:
        // неизвестно, что содержит переменная i, поэтому переменная j
        // обладает типом interface{}
    }
}
```



Поскольку переключатель типа служит для получения новой переменной на основе существующей переменной, идиоматический подход сводится к тому, чтобы присвоить переключающую переменную переменной с таким же именем (`i := i.(type)`). Это один из тех немногих случаев, где затенение будет вполне уместным. В приведенном выше примере затенение не используется, чтобы сохранить читабельность комментариев.

Переключатель типа выглядит во многом так же, как обычный оператор `switch`, с которым мы познакомились в разделе «Оператор `switch`» на с. 107. Вместо булевой операции здесь записывается переменная интерфейсного типа, точка и ключевое слово `type` в скобках: `.(type)`. Обычно при этом проверяемая переменная присваивается другой переменной, область видимости которой ограничивается пределами оператора `switch`.

Тип новой переменной определяется тем, какая из ветвей оператора `switch` дает совпадение. Для одной из ветвей можно использовать значение `nil`, чтобы проверять интерфейс на отсутствие связанного с ним типа. Если для отдельной ветви указывается сразу несколько типов, то новая переменная будет обладать типом `interface{}`. Как и в случае обычного оператора `switch`, можно использовать ветвь `default`, которая выбирается при отсутствии совпадений с указанными типами. В противном случае новая переменная будет обладать типом, указанным в давшей совпадение ветви.



Если неизвестен базовый тип, следует использовать рефлексии. Подробнее о рефлексии будет рассказано в главе 14.

Используйте утверждения типа и переключатели типа как можно реже

Извлечение конкретной реализации из интерфейсной переменной кажется удобной возможностью, но описанные выше приемы необходимо использовать как можно реже. В большинстве случаев параметр или возвращаемое значение следует обрабатывать именно как указанный тип, а не что-то другое. В противном случае API функции неточно указывает, какие типы ему понадобятся для выполнения его задачи. Если требуется какой-то другой тип, это должно быть указано.

В то же время в некоторых случаях использование утверждений типа и переключателей типа будет вполне уместным. Утверждения типа, в частности, широко используются для проверки в отношении того, не реализует ли стоящий за интерфейсом конкретный тип также некоторый другой интерфейс. Это позволяет указывать опциональные интерфейсы. Например, в стандартной библиотеке этот прием используется для более эффективного создания копий при вызове функции `io.Copy`. Эта функция принимает два параметра, относящихся к типам `io.Writer` и `io.Reader`, и вызывает функцию `io.CopyBuffer` для выполнения своей работы. Если аналогичным образом параметр типа `io.Writer`

реализует интерфейс `io.WriterTo` или параметр типа `io.Reader` реализует интерфейс `io.ReaderFrom`, то функция может не выполнить значительную часть своей работы:

```
// Функция copyBuffer содержит реальную реализацию функций Copy и CopyBuffer
// Если параметр buf равен nil, выделяется память для буфера
func copyBuffer(dst Writer, src Reader, buf []byte) (written int64, err error) {
    // Если у отправителя есть метод WriteTo, копирование проводится
    // с его помощью, что позволяет обойтись без выделения памяти
    if wt, ok := src.(WriterTo); ok {
        return wt.WriteTo(dst)
    }
    // Сходным образом, если получатель обладает методом ReadFrom,
    // копирование производится с его помощью
    if rt, ok := dst.(ReaderFrom); ok {
        return rt.ReadFrom(src)
    } // продолжение функции...
}
```

Еще одной областью применения опциональных интерфейсов является доработка API. В главе 12 мы подробно поговорим о контексте. Контекст — это параметр, который передается функциям и, помимо прочего, обеспечивает стандартный способ управления отменой. Он присутствует в Go, начиная с версии 1.7, и это означает, что его не поддерживает более старый код, в том числе старые драйверы баз данных.

В Go 1.8 в пакете `database/sql/driver` были определены новые, учитывающие контекст аналоги существовавших интерфейсов. Например, интерфейс `StmtExecContext` определяет метод `ExecContext`, который является учитывающим контекст аналогом метода `Exec` интерфейса `Stmt`. Когда коду для работы с базами данных из стандартной библиотеки передается реализация интерфейса `Stmt`, он сначала проверяет, не реализует ли эта реализация также интерфейс `StmtExecContext`. Если это так, вызывается метод `ExecContext`. Если нет, стандартная библиотека языка Go предлагает запасной вариант реализации поддержки отмены, который представлен в более новом коде:

```
func ctxDriverStmtExec(ctx context.Context, si driver.Stmt,
    nvargs []driver.NamedValue) (driver.Result, error) {
    if siCtx, is := si.(driver.StmtExecContext); is {
        return siCtx.ExecContext(ctx, nvargs)
    }
    // здесь находится резервный код
}
```

У этого подхода с использованием дополнительного интерфейса есть один недостаток. Как мы видели ранее, реализации интерфейсов часто используют паттерн «Декоратор» для обертывания других реализаций того же интерфейса

с целью создания нескольких слоев поведения. Беда в том, что если одна из обернутых реализаций реализует дополнительный интерфейс, вы не сможете обнаружить это с помощью утверждения типа или переключателя типа. Например, стандартная библиотека включает в себя пакет `bufio`, который предоставляет буферизованный считыватель. Вы можете буферизовать любую другую реализацию интерфейса `io.Reader`, передав ее функции `bufio.NewReader` и используя возвращаемое значение типа `*bufio.Reader`. Если передаваемая реализация интерфейса `io.Reader` также реализует интерфейс `io.ReaderFrom`, то обертывание его в буферизованный считыватель не позволит производить оптимизацию.

То же самое наблюдается и в случае обработки ошибок. Как упоминалось ранее, ошибки реализуют интерфейс `error`. В них можно включать дополнительную информацию путем обертывания одних ошибок в другие. При этом переключатель типа или утверждение типа не может выявить обернутые ошибки или произвести переключение на их основе. Если вам нужно по-разному обрабатывать различные конкретные реализации возвращаемой ошибки, используйте функции `errors.Is` и `errors.As` для проверки на наличие обернутой ошибки и доступа к ней.

Отличить друг от друга реализации интерфейса, требующие различной обработки, можно с помощью переключателей типа. Их особенно удобно использовать в том случае, когда в качестве интерфейса может предоставляться лишь ограниченный набор допустимых типов. При этом не забывайте добавлять в переключатель типа ветвь `default` для обработки реализаций, еще неизвестных на момент разработки. Это избавит вас от проблем, если вы забудете обновить переключатели типа при добавлении новых реализаций интерфейса:

```
func walkTree(t *TreeNode) (int, error) {
    switch val := t.val.(type) {
    case nil:
        return 0, errors.New("invalid expression")
    case number:
        // мы знаем, что t.val обладает типом number, поэтому возвращаем
        // значение типа int
        return int(val), nil
    case operator:
        // мы знаем, что t.val обладает типом operator, поэтому находим
        // значения левого и правого дочерних узлов, а затем вызываем
        // метод process() типа operator, чтобы вернуть результат обработки
        // их значений
        left, err := walkTree(t.lchild)
        if err != nil {
            return 0, err
        }
        right, err := walkTree(t.rchild)
```

```

    if err != nil {
        return 0, err
    }
    return val.process(left, right), nil
default:
    // если определен новый тип узла дерева, но функция walkTree
не обновлена
    // для его обработки, данный тип будет выявлен этой ветвью
    return 0, errors.New("unknown node type")
}
}

```

Полную версию этой реализации можно найти в онлайн-песочнице (<https://oreil.ly/jDhqM>).



Чтобы еще больше защитить себя от неожиданных реализаций интерфейса, можно сделать неэкспортируемыми интерфейс и как минимум один метод. Экспортируемый интерфейс можно встроить в структуру в другом пакете, при этом такая структура будет реализовывать интерфейс. Подробнее о пакетах и экспорте идентификаторов будет рассказано в главе 9.

Функциональные типы — ключ к интерфейсам

Заканчивая разговор об объявлениях типа, следует отметить еще один важный момент. Наряду с такой легко укладываемой в голову возможностью, как снабжение методами целочисленных типов и строк, Go также позволяет снабжать методами и *любые* пользовательские типы, включая пользовательские функциональные типы. Может показаться, что это какой-то исключительный случай, представляющий интерес лишь с научной точки зрения, но в действительности это очень полезная возможность, позволяющая функциям реализовывать интерфейсы. Наиболее широко она используется при создании HTTP-обработчиков. HTTP-обработчик служит для обработки HTTP-запросов и определяется интерфейсом:

```

type Handler interface {
    ServeHTTP(http.ResponseWriter, *http.Request)
}

```

После преобразования типа в тип `http.HandlerFunc` любая функция с сигнатурой `func(http.ResponseWriter, *http.Request)` может быть использована как `http.Handler`:

```

type HandlerFunc func(http.ResponseWriter, *http.Request)

func (f HandlerFunc) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    f(w, r)
}

```

Это позволяет вам реализовывать HTTP-обработчики с помощью функций, методов или замыканий, использующих точно такой же путь выполнения кода, что и в случае других типов, соответствующих интерфейсу `http.Handler`.

Функции в Go являются концепциями первого класса и поэтому часто передаются функциям в качестве параметров. В то же время Go поощряет использование небольших интерфейсов, и интерфейс, обладающий только одним методом, может заменить параметр функционального типа. Возникает следующий вопрос: когда в качестве входного параметра функции или метода следует использовать функциональный тип, а когда — интерфейс?

Если речь идет о случае, когда одна функция может зависеть от многих других функций или другого состояния, которое не указано в ее входных параметрах, используйте интерфейсный параметр и определите функциональный тип для установления связи между функцией и интерфейсом. Именно так сделано в пакете `http`, где `Handler` может быть лишь точкой входа для той цепочки вызовов, которую вам нужно настроить. Однако если речь идет о простой функции (например, `sort.Slice`), лучше используйте параметр функционального типа.

Неявные интерфейсы облегчают внедрение зависимостей

Каждый, кто имеет хотя бы небольшой опыт в программировании, знает, что со временем приложения приходится корректировать. Чтобы упростить обеспечение низкой связанности, был разработан ряд методов, один из которых называется *внедрением зависимостей*. В его основе лежит идея о том, что ваш код должен явно указывать, какая функциональность ему необходима для выполнения его задачи. Эта идея не настолько нова, как можно было бы подумать: еще в 1996 году Роберт Мартин (Robert Martin) написал статью под названием «Принцип инверсии зависимостей» (<https://oreil.ly/6HVob>).

Как это ни удивительно, но одно из преимуществ неявных интерфейсов языка Go состоит в том, что они делают внедрение зависимостей очень удобным способом обеспечения низкой связанности кода. Если в других языках для внедрения зависимостей часто приходится использовать большие и сложные фреймворки, в Go вы можете легко реализовать внедрение зависимостей без использования каких-либо дополнительных библиотек. Разберем простой пример и посмотрим, как можно использовать неявные интерфейсы для создания приложений с применением внедрения зависимостей.

Чтобы лучше понять концепцию внедрения зависимостей и посмотреть, как она реализуется в Go, создадим простейшее веб-приложение. (Подробнее о встроенной поддержке HTTP-сервера в Go будет рассказано в подразделе «Сервер» на с. 304; здесь мы лишь кратко коснемся этой темы.) Для начала напишем небольшую сервисную функцию журналирования:

```
func LogOutput(message string) {
    fmt.Println(message)
}
```

Нашему приложению также потребуется хранилище данных. Определим в качестве такового простую структуру:

```
type SimpleDataStore struct {
    userData map[string]string
}

func (sds SimpleDataStore) UserNameForID(userID string) (string, bool) {
    name, ok := sds.userData[userID]
    return name, ok
}
```

Определим также фабричную функцию для создания экземпляра структуры `SimpleDataStore`:

```
func NewSimpleDataStore() SimpleDataStore {
    return SimpleDataStore{
        userData: map[string]string{
            "1": "Fred",
            "2": "Mary",
            "3": "Pat",
        },
    }
}
```

Теперь напишем бизнес-логику, которая будет находить пользователя и говорить ему слова приветствия или прощания. Поскольку эта логика будет обрабатывать определенные данные, ей потребуется хранилище данных. Нам также нужно, чтобы эта бизнес-логика заносила в журнал сведения о времени ее вызова, поэтому она также будет зависеть от функции журналирования. В то же время нам не нужно, чтобы зависимость от функции `LogOutput` или структуры `SimpleDataStore` носила обязательный характер, поскольку со временем нам, возможно, потребуется использовать другое средство журналирования или хранилище данных. Что нам нужно сделать, так это определить, в чем нуждается наша бизнес-логика, с помощью интерфейсов:

```
type DataStore interface {
    UserNameForID(userID string) (string, bool)
}
```

```
}  
  
type Logger interface {  
    Log(message string)  
}  
}
```

Чтобы сделать функцию `LogOutput` соответствующей представленному выше интерфейсу, определим функциональный тип с требуемым методом:

```
type LoggerAdapter func(message string)  
  
func (lg LoggerAdapter) Log(message string) {  
    lg(message)  
}  
}
```

Теперь типы `LoggerAdapter` и `SimpleDataStore` соответствуют тем интерфейсам, которые требуются нашей бизнес-логике, но ни первый, ни второй тип не знает об этом.

Закончив с определением зависимостей, мы можем перейти к реализации бизнес-логики:

```
type SimpleLogic struct {  
    l Logger  
    ds DataStore  
}  
  
func (sl SimpleLogic) SayHello(userID string) (string, error) {  
    sl.l.Log("in SayHello for " + userID)  
    name, ok := sl.ds.UserNameForID(userID)  
    if !ok {  
        return "", errors.New("unknown user")  
    }  
    return "Hello, " + name, nil  
}  
  
func (sl SimpleLogic) SayGoodbye(userID string) (string, error) {  
    sl.l.Log("in SayGoodbye for " + userID)  
    name, ok := sl.ds.UserNameForID(userID)  
    if !ok {  
        return "", errors.New("unknown user")  
    }  
    return "Goodbye, " + name, nil  
}  
}
```

Здесь мы имеем структуру с двумя полями: `Logger` и `DataStore`. Поскольку внутри структуры `SimpleLogic` нет никаких ссылок на конкретные типы, у нас нет зависимостей от конкретных типов. Мы свободно можем со временем заменить используемую реализацию новой от совершенно другого поставщика, потому что поставщик никак не связан с нашим интерфейсом. Это существенно отличается от использования явных интерфейсов в таких языках, как Java. Хотя

в Java можно использовать интерфейс для отделения реализации от интерфейса, явные интерфейсы при этом привязывают клиента к поставщику. Так, замена зависимости в Java (и других языках с явными интерфейсами) оказывается намного более сложным процессом по сравнению с тем, как это делается в Go.

Чтобы получить экземпляр структуры `SimpleLogic`, необходимо вызвать фабричную функцию, которая принимает интерфейсы и возвращает структуру:

```
func NewSimpleLogic(l Logger, ds DataStore) SimpleLogic {
    return SimpleLogic{
        l:    l,
        ds: ds,
    }
}
```



Поля структуры `SimpleLogic` являются неэкспортируемыми. Это означает, что они доступны только внутри того пакета, в котором определена структура `SimpleLogic`. В Go нельзя сделать поля неизменяемыми, ограничив доступ к этим полям, но мы можем существенно снизить вероятность их случайной модификации. Подробнее об экспортируемых и неэкспортируемых идентификаторах будет рассказано в главе 9.

Теперь мы можем заняться нашим API. У нас будет только одна конечная точка, `/hello`, которая будет говорить слова приветствия пользователю с указанным идентификатором. (Пожалуйста, не используйте параметры запроса для передачи данных аутентификации в реальных приложениях; здесь это делается лишь в целях демонстрации.) Поскольку наш контроллер нуждается в бизнес-логике, выдающей слова приветствия, мы определяем соответствующий интерфейс:

```
type Logic interface {
    SayHello(userID string) (string, error)
}
```

Этот метод доступен в структуре `SimpleLogic`, но здесь опять же конкретный тип не знает о существовании интерфейса. Более того, второй метод структуры `SimpleLogic`, `SayGoodbye`, не указан в этом интерфейсе, потому что для нашего контроллера не имеет значения, есть такой метод или нет. Поскольку этим интерфейсом владеет клиентский код, его набор методов настраивается в соответствии с потребностями клиентского кода:

```
type Controller struct {
    l    Logger
    logic Logic
}

func (c Controller) SayHello(w http.ResponseWriter, r *http.Request) {
    c.l.Log("In SayHello")
}
```

```
userID := r.URL.Query().Get("user_id")
message, err := c.logic.SayHello(userID)
if err != nil {
    w.WriteHeader(http.StatusBadRequest)
    w.Write([]byte(err.Error()))
    return
}
w.Write([]byte(message))
}
```

Подобно тому как мы использовали фабричные функции для других типов, напомним такую функцию и для типа `Controller`:

```
func NewController(l Logger, logic Logic) Controller {
    return Controller{
        l:      l,
        logic:  logic,
    }
}
```

Здесь мы опять же принимаем интерфейсы и возвращаем структуры.

Наконец, подключим все наши компоненты внутри функции `main` и запустим наш сервер:

```
func main() {
    l := LoggerAdapter(LogOutput)
    ds := NewSimpleDataStore()
    logic := NewSimpleLogic(l, ds)
    c := NewController(l, logic)
    http.HandleFunc("/hello", c.SayHello)
    http.ListenAndServe(":8080", nil)
}
```

Функция `main` является здесь единственной частью кода, которой известно, что в действительности представляют собой все используемые конкретные типы. Если нам потребуется заменить какие-либо реализации, изменения потребуются внести только в этой функции. Такая проверка зависимостей извне с помощью внедрения зависимостей уменьшает объем изменений, необходимых для доработки кода с течением времени.

Паттерн внедрения зависимостей также позволяет облегчить процесс тестирования. В этом нет ничего удивительного, поскольку написание модульных тестов по сути представляет собой повторное использование кода в другой среде, где на входные и выходные данные накладываются определенные ограничения для проверки функциональности. Например, мы можем проверить в тесте выходные данные журналирования, внедрив тип, который регистрирует выходные данные и соответствует интерфейсу `Logger`. О том, как это делается, будет подробно рассказано в главе 13.



Строка `http.HandleFunc("/hello", c.SayHello)` демонстрирует две уже упоминавшиеся ранее возможности.

Во-первых, метод `Sayhello` используется здесь в качестве функции.

Во-вторых, функция `http.HandleFunc` принимает функцию и преобразует ее в функциональный тип `http.HandlerFunc`, который объявляет метод, соответствующий интерфейсу `http.Handler`, то есть типу, используемому в Go для представления обработчика запросов. Таким образом, мы взяли метод одного типа и изящно преобразовали его в другой тип, обладающий собственным методом.

Утилита Wire

Если написание кода для внедрения зависимостей вручную кажется вам слишком большой работой, вы можете использовать Wire (https://oreil.ly/Akwt_), утилиту для внедрения зависимостей от компании Google. Она создает объявления конкретных типов, которые мы писали сами внутри функции `main`, путем автоматической генерации кода.

Go нельзя назвать объектно-ориентированным языком (и это здорово!)

Теперь, когда мы уже знаем, как выглядит идиоматический подход к использованию типов в Go, вы можете видеть, что Go сложно отнести к какой-либо определенной категории языков. Очевидно, что его нельзя назвать исключительно процедурным языком. В то же время отсутствие в Go перегрузки методов, наследования, да и, собственно, объектов, не позволяет назвать его и объектно-ориентированным языком. Несмотря на наличие функциональных типов и замыканий, Go нельзя назвать и функциональным языком. Попытки втиснуть Go в одну из этих категорий приведут к созданию неидиоматического кода.

Если Go и можно отнести к какой-то категории, то это категория *практичных* языков. Он заимствует концепции из многих источников, ставя главной целью создание простого и читабельного кода, который могли бы поддерживать большие команды разработчиков на протяжении многих лет.

Резюме

В этой главе были рассмотрены типы, методы, интерфейсы и рекомендуемые способы их использования. В следующей главе вы узнаете, как следует подходить к использованию одного из самых противоречивых элементов языка Go — ошибок.

Ошибки

Разработчикам, которые переходят на Go с других языков, сложнее всего дается обработка ошибок. Для тех, кто привык к исключениям, используемый в Go подход кажется анахронизмом. Но в основе этой концепции лежат надежные принципы программной разработки. В этой главе вы узнаете, как следует работать с ошибками в Go. Мы также рассмотрим функции `panic` и `recover` — систему обработки ошибок, которая останавливает выполнение программы.

Как обрабатывать ошибки: основы

Как уже кратко упоминалось в главе 5, обработка ошибок в Go сводится к возвращению значения типа `error` в качестве последнего возвращаемого значения функции. Это общепринятое соглашение, которое соблюдается настолько строго, что его никогда не следует нарушать. Когда функция выполняется ожидаемым образом, в качестве параметра ошибки возвращается значение `nil`. Если что-то идет не так, вместо этого возвращается значение ошибки. После этого вызывающая функция проверяет возвращаемое значение ошибки путем сравнения его со значением `nil` и обрабатывает ошибку или возвращает некоторую собственную ошибку. Вот как выглядит такой код:

```
func calcRemainderAndMod(numerator, denominator int) (int, int, error) {
    if denominator == 0 {
        return 0, 0, errors.New("denominator is 0")
    }
    return numerator / denominator, numerator % denominator, nil
}
```

Новая ошибка создается из строки путем вызова функции `New` из пакета `errors`. Сообщения об ошибке не должны начинаться с большой буквы и заканчиваться знаком пунктуации или символом новой строки. В большинстве случаев при возвращении ненулевой ошибки остальным возвращаемым значениям

следует присвоить соответствующие нулевые значения. Исключением из этого правила является возвращение сигнальных ошибок, о которых мы поговорим чуть позже.

В отличие от языков с исключениями в Go нет специальных конструкций для выявления тех случаев, когда возвращается ошибка. Всякий раз, когда функция возвращает значения, используйте оператор `if`, чтобы проверить, не является ли параметр ошибки ненулевым:

```
func main() {
    numerator := 20
    denominator := 3
    remainder, mod, err := calcRemainderAndMod(numerator, denominator)
    if err != nil {
        fmt.Println(err)
        os.Exit(1)
    }
    fmt.Println(remainder, mod)
}
```

Тип `error` представляет собой встроенный интерфейс, который определяет метод:

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

Любой тип, который реализует этот интерфейс, считается ошибкой. Мы возвращаем значение `nil` из функции при отсутствии ошибок, потому что значение `nil` является нулевым значением для интерфейсного типа.

Есть две веские причины, по которым в Go принято возвращать ошибки, а не выбрасывать исключения. Во-первых, исключения добавляют как минимум один дополнительный путь выполнения кода. Эти пути выполнения кода иногда бывают неясными, особенно в тех языках, в которых функции не содержат объявление о возможности исключения. В результате вы можете получить код, который будет неожиданно прекращать работу в случае неправильной обработки исключений или, что еще хуже, будет продолжать работу после некорректного выполнения инициализации, модификации или сохранения данных.

Вторая причина носит не столь явный характер, но показывает, насколько хорошо различные элементы языка Go согласуются друг с другом. Компилятор языка Go требует, чтобы все объявленные переменные были прочитаны. В силу того что ошибки являются возвращаемыми значениями, разработчикам приходится либо проверять наличие состояний ошибки и обрабатывать их, либо явно указывать, что ошибки будут игнорироваться, используя в качестве возвращаемого значения ошибки символ подчеркивания (`_`).



Как упоминалось в главе 5, хотя вы не имеете права игнорировать *некоторые* из возвращаемых значений функции, вы можете пропустить *все* возвращаемые значения. Но, не обращая внимания на все возвращаемые значения, можно не заметить ошибку. В большинстве случаев игнорирование возвращаемых значений будет крайне неудачной идеей. Пожалуйста, не делайте так! Исключение могут составлять лишь такие случаи, как использование функции `fmt.Println`.

Если использовать исключения, можно сократить размер кода, однако более короткий код далеко не всегда является более понятным и легким в сопровождении. Как мы уже видели, идиоматический подход в Go поощряет написание ясного кода, даже если такой код занимает больше строк.

Еще одним важным моментом является расположение кода в Go. Если код обработки ошибок снабжается отступом внутри операторов `if`, то код бизнес-логики записывается без отступов. Это позволяет сразу же увидеть, какой код «находится на правильном пути», а какой служит для обработки исключительных ситуаций.

Используйте строки в случае простых ошибок

Стандартная библиотека языка Go предоставляет два способа создания ошибки из строки. Первый способ сводится к использованию функции `errors.New`, которая принимает параметр типа `string` и возвращает значение типа `error`. Эта строка возвращается при вызове метода `Error` для возвращаемого экземпляра ошибки. Если вы передадите ошибку функции `fmt.Println`, она вызовет метод `Error` автоматически:

```
func doubleEven(i int) (int, error) {
    if i % 2 != 0 {
        return 0, errors.New("only even numbers are processed")
    }
    return i * 2, nil
}
```

Второй способ создания ошибки сводится к использованию функции `fmt.Errorf`, которая позволяет задействовать все глаголы форматирования функции `fmt.Printf`. Как и при использовании функции `errors.New`, строка ошибки возвращается при вызове метода `Error` для возвращаемого экземпляра ошибки.

```
func doubleEven(i int) (int, error) {
    if i % 2 != 0 {
        return 0, fmt.Errorf("%d isn't an even number", i)
    }
    return i * 2, nil
}
```

Сигнальные ошибки

В некоторых случаях ошибка должна сигнализировать о том, что обработка не может продолжаться из-за проблемы с текущим состоянием. В своем блоге под заголовком «Не просто выявляйте ошибки, а обрабатывайте их без нарушения работоспособности» (<https://oreil.ly/TiJnS>) Дейв Чейни (Dave Cheney), активный член Go-сообщества на протяжении многих лет, предложил называть такие ошибки *сигнальными ошибками* (*sentinel errors*).

Это название происходит от существующей в программировании практики использования определенного значения для обозначения невозможности выполнения какой-либо дальнейшей обработки. Так же и в Go мы используем определенные значения для обозначения ошибки.

Дейв Чейни (<https://oreil.ly/3fMAI>)

Сигнальные ошибки относятся к числу тех немногих разновидностей переменных, которые могут объявляться на уровне пакета. В соответствии с общепринятым соглашением их имена начинаются на `Err` (`io.EOF` является важным исключением). Они должны обрабатываться как данные, доступные только для чтения. Хотя компилятор языка Go не требует этого в принудительном порядке, модификация сигнальной ошибки считается программной ошибкой.

Сигнальные ошибки обычно используются в качестве указания, что вы не можете запустить или продолжить обработку. Так, например, в стандартной библиотеке есть пакет для обработки ZIP-файлов, `archive/zip`. В этом пакете определены несколько сигнальных ошибок, включая ошибку `ErrFormat`, которая возвращается, когда переданные данные не являются ZIP-файлом. Попробуйте запустить в онлайн-песочнице следующий код (<https://oreil.ly/DaW-s>):

```
func main() {
    data := []byte("This is not a zip file")
    notAZipFile := bytes.NewReader(data)
    _, err := zip.NewReader(notAZipFile, int64(len(data)))
    if err == zip.ErrFormat {
        fmt.Println("Told you so")
    }
}
```

Еще одним примером сигнальной ошибки из стандартной библиотеки является ошибка `rsa.ErrMessageTooLong` из пакета `crypto/rsa`. Она сообщает о том, что сообщение не может быть зашифровано, потому что является слишком длинным для предоставленного открытого ключа. При обсуждении контекста в главе 12 вы познакомитесь еще с одной распространенной сигнальной ошибкой: `context.Canceled`.

ИСПОЛЬЗОВАНИЕ КОНСТАНТ В КАЧЕСТВЕ СИГНАЛЬНЫХ ОШИБОК

В своем блог-посте под заголовком «Константные ошибки» (<https://oreil.ly/1AnVg>) Дейв Чейни выдвинул идею о том, что константы удобно использовать в качестве сигнальных ошибок. Имея в своем пакете тип следующего вида (о создании пакетов будет рассказано в главе 9):

```
package consterr

type Sentinel string

func(s Sentinel) Error() string {
    return string(s)
}
```

вы можете использовать его следующим образом:

```
package mypkg

const (
    ErrFoo = consterr.Sentinel("foo error")
    ErrBar = consterr.Sentinel("bar error")
)
```

Хотя это выглядит как вызов функции, на самом деле мы приводим строковый литерал к типу, реализующему интерфейс `error`. Значения констант `ErrFoo` и `ErrBar` будет невозможно изменить. На первый взгляд это кажется хорошим решением.

Однако такой подход не считается идиоматическим. Если использовать один и тот же тип для создания константных ошибок в различных пакетах, то две ошибки будут равны друг другу, если будут идентичны их строки с сообщением об ошибке. Они также будут равны, если их строковый литерал имеет то же значение. Однако ошибка, созданная с помощью функции `errors.New`, равна только самой себе или переменным, которым будет явно присвоено ее значение. В абсолютном большинстве случаев вам не будет нужно, чтобы ошибки в разных пакетах были одинаковыми; иначе зачем вам объявлять две разные ошибки? (Этого можно избежать, создав непубличный тип ошибки в каждом пакете, но в таком случае придется написать много шаблонного кода.)

Паттерн сигнальных ошибок является еще одним примером философии языка Go. Поскольку сигнальные ошибки случаются редко, их можно обрабатывать, руководствуясь общепринятым соглашением, а не правилами языка. Да, они представляют собой публичные переменные, объявленные

на уровне пакета. Это делает их изменяемыми, но вероятность того, что кто-то случайно переназначит публичную переменную в пакете, крайне низка. В общем, это исключительный случай, который обрабатывается с помощью других элементов и паттернов. Философия языка Go сводится к тому, что лучше оставить язык простым, больше полагаясь на разработчиков и инструменты, чем добавлять в него дополнительные возможности.

Перед тем как определять сигнальную ошибку, убедитесь в том, что она вам необходима. После того как вы ее определите, она станет составной частью вашего публичного API и вы будете обязаны обеспечить ее доступность во всех будущих обратно совместимых релизах. Гораздо лучше повторно использовать одну из ошибок, определенных в стандартной библиотеке, или задать тип ошибки, включающий в себя информацию о том, какое состояние привело к возвращению ошибки (о том, как это делается, будет рассказано в следующем разделе). Но если в ошибке указано, что в приложении достигнуто определенное состояние, при котором невозможно выполнение дальнейшей обработки и при этом вам не нужно использовать контекстную информацию для объяснения состояния ошибки, то сигнальная ошибка будет правильным выбором.

Вы можете спросить: как выполняется проверка на наличие сигнальной ошибки. Используйте оператор `==`, как показано в предыдущем примере кода, при вызове функции, в документации которой явно указано, что она возвращает сигнальную ошибку. В одном из следующих разделов вы узнаете, как проверяется наличие сигнальных ошибок в других ситуациях.

До сих пор все рассмотренные ошибки представляли собой строки. Однако в Go в ошибки можно включать и некоторую дополнительную информацию. Посмотрим, как это делается.

Ошибки являются значениями

Поскольку тип `error` представляет собой интерфейс, вы можете определить собственные ошибки с дополнительной информацией для целей журналирования или обработки ошибок. Например, иногда в ошибку требуется включить код состояния, чтобы указать, какое сообщение выдать пользователю. Это позволяет вам обойтись без сравнения строк (содержимое которых может измениться) при определении причин ошибки. Посмотрим, как это выглядит на практике. Сначала следует определить собственное перечисление для представления кодов состояния:

```
type Status int

const (
    InvalidLogin Status = iota + 1
    NotFound
)
```

Затем необходимо задать структуру `StatusErr`, которая будет содержать это значение:

```
type StatusErr struct {
    Status    Status
    Message string
}

func (se StatusErr) Error() string {
    return se.Message
}
```

Теперь мы можем использовать структуру `StatusErr` для предоставления более подробных сведений о том, что пошло не так:

```
func LoginAndGetData(uid, pwd, file string) ([]byte, error) {
    err := login(uid, pwd)
    if err != nil {
        return nil, StatusErr{
            Status:    InvalidLogin,
            Message:   fmt.Sprintf("invalid credentials for user %s", uid),
        }
    }
    data, err := getData(file)
    if err != nil {
        return nil, StatusErr{
            Status:    NotFound,
            Message:   fmt.Sprintf("file %s not found", file),
        }
    }
    return data, nil
}
```

Даже если вы определяете собственные пользовательские типы ошибок, всегда используйте тип `error` в качестве типа возвращаемой ошибки. Это позволяет возвращать из функции разные типы ошибок, а вызывающая сторона при этом не будет зависеть от конкретного типа ошибок.

При использовании собственного типа ошибки следует позаботиться о том, чтобы функция никогда не возвращала неинициализированный экземпляр. Это значит, что вы не должны объявлять переменную как тип вашей пользовательской ошибки и после этого возвращать эту переменную. Посмотрим, что

произойдет, если вы так поступите. Попробуйте запустить в онлайн-песочнице следующий код (<https://oreil.ly/5QJVN>):

```
func GenerateError(flag bool) error {
    var genErr StatusErr
    if flag {
        genErr = StatusErr{
            Status: NotFound,
        }
    }
    return genErr
}

func main() {
    err := GenerateError(true)
    fmt.Println(err != nil)
    err = GenerateError(false)
    fmt.Println(err != nil)
}
```

После запуска вы получите следующий результат:

```
true
true
```

Здесь нет проблемы выбора указательного или значимого типа: если бы переменная `genErr` была объявлена как переменная типа `*StatusErr`, мы получили бы тот же результат. Переменная `err` не равна здесь `nil` по той причине, что тип `error` представляет собой интерфейс. Как упоминалось в разделе «Интерфейсы и значение `nil`» на с. 186, чтобы интерфейс считался равным `nil`, значению `nil` должны быть равны и базовый тип, и базовое значение. Вне зависимости от того, будет ли переменная `genErr` указателем или нет, базовый тип этого интерфейса не будет равен `nil`.

Решить эту проблему можно двумя способами. Наиболее распространенный подход заключается в том, чтобы явным образом возвращать значение `nil` в качестве значения ошибки при успешном выполнении функции:

```
func GenerateError(flag bool) error {
    if flag {
        return StatusErr{
            Status: NotFound,
        }
    }
    return nil
}
```

Преимущество такого подхода состоит в том, что вам не нужно просматривать предыдущий код, чтобы убедиться, что переменная ошибки в операторе `return` определена правильно.

Второй подход сводится к тому, чтобы убедиться, что любая локальная переменная с ошибкой имеет тип `error`:

```
func GenerateError(flag bool) error {
    var genErr error
    if flag {
        genErr = StatusErr{
            Status: NotFound,
        }
    }
    return genErr
}
```



При использовании пользовательских ошибок никогда не определяйте переменную с типом вашей пользовательской ошибки. Либо явно возвращайте значение `nil` при отсутствии ошибок, либо определите переменную типа `error`.

Как упоминалось в разделе «Используйте утверждения типа и переключатели типа как можно реже» на с. 191, для доступа к полям и методам пользовательской ошибки не следует применять утверждения типа и переключатели типа. Вместо этого используйте функцию `errors.As`, о которой мы поговорим в разделе «Функции `Is` и `As`» на с. 212.

Обертывание ошибок

Когда ошибка передается в коде в обратном направлении, часто в нее требуется внести дополнительный контекст. Этим контекстом может быть имя функции, в которой произошла ошибка, или сведения о том, какую операцию она пыталась при этом выполнить. Если при добавлении дополнительной информации исходная ошибка сохраняется, это называют *обертыванием* ошибки. Последовательность обернутых ошибок называется *цепью ошибок*.

В стандартной библиотеке языка Go есть функция, позволяющая обертывать ошибки, и мы уже знакомы с ней. Функция `fmt.Errorf` позволяет использовать специальный глагол, `%w`, для создания ошибки, форматированная строка которой будет содержать форматированную строку другой ошибки и которая также будет содержать исходную ошибку. Согласно общепринятому соглашению в конце форматированной строки ошибки следует записать `:%w`, чтобы обернуть ошибку, переданную в качестве последнего параметра функции `fmt.Errorf`.

Стандартная библиотека также предоставляет функцию для извлечения обернутых ошибок: это функция `Unwrap` из пакета `errors`. Она принимает на вход ошибку и возвращает обернутую ошибку, если такая существует. При отсутствии

обернутой ошибки она возвращает `nil`. Обертывание ошибок с помощью функции `fmt.Errorf` и извлечение обернутых ошибок с помощью функции `errors.Unwrap` демонстрирует представленный ниже пример кода. Вы можете запустить его в онлайн-песочнице (<https://oreil.ly/HxdHz>):

```
func fileChecker(name string) error {
    f, err := os.Open(name)
    if err != nil {
        return fmt.Errorf("in fileChecker: %w", err)
    }
    f.Close()
    return nil
}

func main() {
    err := fileChecker("not_here.txt")
    if err != nil {
        fmt.Println(err)
        if wrappedErr := errors.Unwrap(err); wrappedErr != nil {
            fmt.Println(wrappedErr)
        }
    }
}
```

Запустив эту программу, вы получите следующий результат:

```
in fileChecker: open not_here.txt: no such file or directory
open not_here.txt: no such file or directory
```



Функция `errors.Unwrap` обычно не вызывается напрямую. Вместо этого нужно воспользоваться функциями `errors.Is` и `errors.As` для поиска конкретной обернутой ошибки. Об этих двух функциях мы поговорим в следующем разделе.

Если вы хотите обернуть ошибку в свой пользовательский тип, ваш тип ошибки должен реализовывать метод `Unwrap`. Этот метод не принимает параметров и возвращает значение типа `error`. Чтобы посмотреть, как это работает, обновим приводившееся ранее определение ошибки:

```
type StatusErr struct {
    Status Status
    Message string
    Err error
    In some cases, expecting}
```

```
func (se StatusErr) Error() string {
    return se.Message
}

func (se StatusError) Unwrap() error {
    return se.err
}
```

Теперь мы можем использовать структуру `StatusErr` для обертывания базовых ошибок:

```
func LoginAndGetData(uid, pwd, file string) ([]byte, error) {
    err := login(uid,pwd)
    if err != nil {
        return nil, StatusErr {
            Status: InvalidLogin,
            Message: fmt.Sprintf("invalid credentials for user %s",uid),
            Err: err,
        }
    }
    data, err := getData(file)
    if err != nil {
        return nil, StatusErr {
            Status: NotFound,
            Message: fmt.Sprintf("file %s not found",file),
            Err: err,
        }
    }
    return data, nil
}
```

Не все ошибки нуждаются в обертке. Иногда библиотека возвращает ошибку, которая означает невозможность выполнения дальнейшей обработки, но сообщение об ошибке при этом содержит детали, которые не требуются в других частях программы. В таком случае лучше создать совершенно новую ошибку и возвращать ее, не прибегая к обертыванию. Старайтесь всегда возвращать то, что требуется в конкретном случае.



Если вы хотите создать новую ошибку, которая содержала бы сообщение другой ошибки, не обертывая ее при этом, создайте ошибку с помощью функции `fmt.Errorf`, используя глагол `%v` вместо глагола `%w`:

```
err := internalFunction()
if err != nil {
    return fmt.Errorf("internal failure: %v", err)
}
```

Функции Is и As

Представляя собой удобный способ предоставления дополнительной информации об ошибке, обертывание ошибок в то же время создает и определенные проблемы. Вы не можете проверять наличие обернутой сигнальной ошибки с помощью оператора `==`, равно как и производить сопоставление с обернутой пользовательской ошибкой, используя утверждение типа или переключатель типа. Для решения этой проблемы в Go существует две функции из пакета `errors`: функции `Is` и `As`.

Чтобы проверить, не совпадает ли возвращаемая ошибка или какая-либо из обернутых в нее ошибок с определенным экземпляром сигнальной ошибки, следует использовать функцию `errors.Is`. Эта функция принимает два параметра: проверяемую ошибку и экземпляр, с которым ее нужно сравнить. Функция `errors.Is` возвращает значение `true`, если в цепи ошибок присутствует ошибка, совпадающая с указанной сигнальной ошибкой. Напишем небольшую программу и посмотрим, как использование функции `errors.Is` выглядит на практике. Вы можете запустить этот код в онлайн-песочнице (https://oreil.ly/5_6rI):

```
func fileChecker(name string) error {
    f, err := os.Open(name)
    if err != nil {
        return fmt.Errorf("in fileChecker: %w", err)
    }
    f.Close()
    return nil
}

func main() {
    err := fileChecker("not_here.txt")
    if err != nil {
        if errors.Is(err, os.ErrNotExist) {
            fmt.Println("That file doesn't exist")
        }
    }
}
```

Запустив этот код, вы получите следующий результат:

```
That file doesn't exist
```

По умолчанию функция `errors.Is` сравнивает каждую из обернутых ошибок с указанной ошибкой с помощью оператора `==`. Если это не подходит для определенного вами типа ошибки (например, если ваша ошибка является несравнимым типом), реализуйте метод `Is` для своей ошибки:

```

type MyErr struct {
    Codes []int
}

func (me MyErr) Error() string {
    return fmt.Sprintf("codes: %v", me.Codes)
}

func (me MyErr) Is(target error) bool {
    if me2, ok := target.(MyErr); ok {
        return reflect.DeepEqual(me, me2)
    }
    return false
}

```

(О функции `reflect.DeepEqual` уже упоминалось в главе 3. Она сравнивает все что угодно, в том числе и срезы.)

С помощью собственного метода `Is` также можно выявлять частичное соответствие с экземпляром ошибки. Иногда требуется сравнить ошибки по маске, указав фильтрующий экземпляр, выявляющий совпадение по некоторым полям. Определим новый тип ошибки, `ResourceErr`:

```

type ResourceErr struct {
    Resource string
    Code     int
}

func (re ResourceErr) Error() string {
    return fmt.Sprintf("%s: %d", re.Resource, re.Code)
}

```

Если требуется, чтобы два экземпляра ошибки `ResourceErr` считались совпадающими при совпадении хотя бы одного из полей, это можно сделать, определив пользовательский метод `Is`:

```

func (re ResourceErr) Is(target error) bool {
    if other, ok := target.(ResourceErr); ok {
        ignoreResource := other.Resource == ""
        ignoreCode := other.Code == 0
        matchResource := other.Resource == re.Resource
        matchCode := other.Code == re.Code
        return matchResource && matchCode ||
            matchResource && ignoreCode ||
            ignoreResource && matchCode
    }
    return false
}

```

Теперь мы, например, можем найти все ошибки, которые имеют отношение к базе данных, безотносительно к их коду:

```
if errors.Is(err, ResourceErr{Resource: "Database"}) {
    fmt.Println("The database is broken:", err)
    // обработка кодов
}
```

Вы можете запустить этот код в онлайн-песочнице (https://oreil.ly/Mz_Op).

Функция `errors.As` позволяет проверить, не совпадает ли возвращаемая ошибка (или какая-либо из обернутых в нее ошибок) с определенным типом. Эта функция принимает два параметра: проверяемую ошибку и указатель на переменную искомого типа. Она возвращает значение `true` при наличии в цепи ошибок совпадающей ошибки, которая затем присваивается второму параметру. При отсутствии совпадений в цепи ошибок возвращается значение `false`. Попробуем эту функцию на ошибке `MyErr`:

```
err := AFunctionThatReturnsAnError()
var myErr MyErr
if errors.As(err, &myErr) {
    fmt.Println(myErr.Code)
}
```

Обратите внимание, что мы используем здесь ключевое слово `var` для объявления переменной конкретного типа, равной нулевому значению, а затем передаем функции `errors.As` указатель на эту переменную.

В качестве второго параметра функции `errors.As` не обязательно передавать указатель на переменную, относящуюся к некоторому типу ошибки. Вы можете передать указатель на интерфейс, чтобы найти ошибку, которая соответствует интерфейсу:

```
err := AFunctionThatReturnsAnError()
var coder interface {
    Code() int
}
if errors.As(err, &coder) {
    fmt.Println(coder.Code())
}
```

Хотя мы используем здесь анонимный интерфейс, допускается использование любого интерфейсного типа.



Функция `errors.As` выдаст панику, если в качестве второго параметра вы передадите ей не указатель на ошибку или указатель на интерфейс, а что-то иное.

Точно так же, как вы переопределяете стандартное сравнение функции `errors.Is` с помощью метода `Is`, вы можете переопределить и стандартный способ сравнения функции `errors.As`, определив для своей ошибки метод `As`. Реализация метода `As` является нетривиальной задачей и требует применения рефлексии (о которой мы подробно поговорим в главе 14). Прибегать к этому следует только в исключительных случаях: например, когда требуется выявлять ошибку одного типа и возвращать ошибку другого типа.



Используйте функцию `errors.Is`, когда требуется выявить определенный экземпляр или определенные значения. Если же требуется выявить определенный тип, добавьте функцию `errors.As`.

Обертывание ошибок с помощью оператора defer

В некоторых случаях требуется обернуть в одно и то же сообщение несколько ошибок:

```
func DoSomeThings(val1 int, val2 string) (string, error) {
    val3, err := doThing1(val1)
    if err != nil {
        return "", fmt.Errorf("in DoSomeThings: %w", err)
    }
    val4, err := doThing2(val2)
    if err != nil {
        return "", fmt.Errorf("in DoSomeThings: %w", err)
    }
    result, err := doThing3(val3, val4)
    if err != nil {
        return "", fmt.Errorf("in DoSomeThings: %w", err)
    }
    return result, nil
}
```

Такой код можно упростить с помощью оператора `defer`:

```
func DoSomeThings(val1 int, val2 string) (_ string, err error) {
    defer func() {
        if err != nil {
            err = fmt.Errorf("in DoSomeThings: %w", err)
        }
    }()
    val3, err := doThing1(val1)
    if err != nil {
        return "", err
    }
    val4, err := doThing2(val2)
```

```
    if err != nil {  
        return "", err  
    }  
    return doThing3(val3, val4)  
}
```

Необходимо присвоить имена возвращаемым значениям, чтобы в отложенной функции была возможность обращаться к переменной `err`. Дополняя именем одно из возвращаемых значений, необходимо дать имена и всем остальным возвращаемым значениям, и мы используем здесь символ подчеркивания для возвращаемого строкового параметра, поскольку не присваиваем ему значение явным образом.

В заключительной части `defer` мы проверяем, не была ли возвращена ошибка. Если была, то мы присваиваем переменной ошибки новую ошибку, которая обортывает исходную ошибку в сообщение, указывающее, какая функция обнаружила ошибку.

Этот шаблон хорошо подходит для того случая, когда требуется обернуть каждую ошибку в одно и то же сообщение. Если требуется настроить обортывающую ошибку для предоставления более подробной контекстной информации о не-исправности, то в каждый вызов функции `fmt.Errorf` следует поместить и конкретизированное, и обобщенное сообщение.

Функции `panic` и `recover`

В предыдущих главах я уже вскользь упоминал паники, не вдаваясь в подробное рассмотрение того, что они собой представляют. В Go паника выдается всякий раз, когда среда выполнения языка Go оказывается неспособной определить, что следует делать дальше. Причиной этому может быть программная ошибка (например, попытка чтения данных после окончания фрагмента) или проблема со средой (например, закончилась память). После выдачи паники немедленно прекращается выполнение текущей функции и запускаются все отложенные функции, привязанные к текущей функции. После выполнения этих отложенных функций запускаются отложенные функции, привязанные к вызывающей функции, и так далее, пока не будет достигнута функция `main`. После этого производится выход из программы с выводом сообщения и трассировкой стека.

Если в ваших программах возникают необратимые ситуации, вы можете создать собственные паники. Встроенная функция `panic` принимает один параметр, который может обладать любым типом, но обычно представляет собой строку.

Создадим простейшую программу, которая будет выдавать панику, и попробуем запустить ее в онлайн-песочнице (<https://oreil.ly/yCBib>):

```
func doPanic(msg string) {
    panic(msg)
}

func main() {
    doPanic(os.Args[0])
}
```

Запустив этот код, вы получите следующий результат:

```
panic: /tmpfs/play

goroutine 1 [running]:
main.doPanic(...)
    /tmp/sandbox567884271/prog.go:6
main.main()
    /tmp/sandbox567884271/prog.go:10 +0x5f
```

Как вы можете видеть, функция `panic` выводит свое сообщение, за которым следует трассировка стека.

Go предоставляет возможность перехватить панику, чтобы обеспечить более мягкое прекращение работы или вообще обойтись без него. Для этого встроенная функция `recover` вызывается внутри оператора `defer`, чтобы проверить, была ли паника. При наличии паники возвращается присвоенное ей значение. После выполнения функции `recover` работа продолжается в обычной манере. Как это выглядит на практике, показывает представленный ниже пример кода. Попробуйте запустить его в онлайн-песочнице (<https://oreil.ly/f5Ybe>):

```
func div60(i int) {
    defer func() {
        if v := recover(); v != nil {
            fmt.Println(v)
        }
    }()
    fmt.Println(60 / i)
}

func main() {
    for _, val := range []int{1, 2, 0, 6} {
        div60(val)
    }
}
```

Существует конкретный паттерн использования функции `recover`. В операторе `defer` регистрируется функция для обработки возможной паники. Мы

вызываем функцию `recover` внутри оператора `if` и проверяем, не возвратила ли она ненулевое значение. Вызов функции `recover` должен производиться внутри оператора `defer`, поскольку при генерировании паники выполняются только отложенные функции.

Запустив этот код, вы получите следующий результат:

```
60
30
runtime error: integer divide by zero
10
```

Хотя использование функций `panic` и `recover` во многом напоминает обработку исключений в других языках, они не предназначены для такого использования. Закрепите паники за фатальными ситуациями и используйте функцию `recover` для обработки этих ситуаций без потери работоспособности. Когда программа выдает панику, следует быть крайне осмотрительными в отношении попыток дальнейшего выполнения программы, поскольку продолжать работу после возникновения паники требуется лишь в очень редких случаях. Если паника была вызвана тем, что компьютер исчерпал определенный ресурс, такой как память или дисковое пространство, то безопаснее всего будет передать ПО для мониторинга сведения о ситуации с помощью функции `recover` и прекратить выполнение программы вызовом `os.Exit(1)`. Если паника была вызвана программной ошибкой, то можно попытаться продолжить выполнение программы, однако это, скорее всего, приведет к повторному появлению той же проблемы. В представленном выше примере кода идиоматический подход сводится к тому, чтобы проверять, не производится ли деление на ноль, и возвращать ошибку в том случае, если функции был передан ноль.

Полагаться на функции `panic` и `recover` не стоит по той причине, что функция `recover` не дает нам информации о том, *что* могло дать сбой. Она лишь гарантирует, что в случае сбоя мы сможем вывести сообщение и продолжить работу. Согласно идиоматическому подходу Go следует отдавать предпочтение коду, четко определяющему возможные сбойные состояния, а не более короткому коду, который обрабатывает эти состояния, ничего при этом не сообщая.

В то же время существует ситуация, в которой использование функции `recover` будет вполне уместным. Если вы создаете библиотеку для сторонних потребителей, не позволяйте паникам выходить за пределы вашего публичного API. Если в публичной функции может возникнуть паника, она должна использовать функцию `recover` для преобразования паники в ошибку, а затем вернуть ее, позволив вызывающему коду обрабатывать ее по своему усмотрению.



Хотя встроенный HTTP-сервер языка Go может восстанавливаться после возникновения паники в обработчиках, Дэвид Саймондс (David Symonds) указал в своем комментарии на GitHub (<https://oreil.ly/BGOmg>), что сейчас команда разработчиков языка Go считает это ошибкой.

Извлечение трассировки стека из ошибки

Одним из поводов использования функций `panic` и `recover` для Go-разработчиков является то, что это позволяет получать трассировку стека в том случае, когда что-то идет не так. По умолчанию язык Go не предоставляет трассировку стека. Как было показано ранее, вы можете создавать стек вызовов вручную, используя обертывание ошибок, однако для этой цели можно использовать и сторонние библиотеки с типами ошибок, автоматически генерирующими такие стеки (о том, как можно встроить в свою программу сторонний код, будет рассказано в главе 9). Наиболее известная из этих сторонних библиотек (<https://github.com/pkg/errors>) предоставляет функции для обертывания ошибок с трассировкой стека.

По умолчанию трассировка стека не выводится на экран. Если вы хотите вывести трассировку стека, используйте функцию `fmt.Printf` с глаголом подробного вывода `%+v`. Более подробную информацию можно найти в документации (<https://oreil.ly/mBQRA>).



При включении в ошибку трассировки стека на выходе можно увидеть полный путь к файлу на том компьютере, где была скомпилирована программа. Если вы не хотите раскрывать информацию об этом пути, при компиляции используйте код с флагом `-trimpath`. В таком случае вместо полного пути будет указано имя пакета.

Резюме

В этой главе вы узнали, как производится обработка ошибок в Go, что они собой представляют и как можно определить и проанализировать собственные ошибки. Вы также познакомились с функциями `panic` и `recover`. В следующей главе мы поговорим о том, что собой представляют пакеты и модули, как можно использовать в своих программах сторонний код и как можно опубликовать свой код, чтобы его могли использовать другие пользователи.

ГЛАВА 9

Модули, пакеты и операции импорта

Большинство современных языков программирования обладает определенной системой организации кода в пространстве имен и библиотек, и Go не исключение. Как мы уже видели при рассмотрении других элементов этого языка, Go дополняет эту далеко не новую идею рядом новых подходов. В этой главе вы узнаете об организации кода с помощью пакетов и модулей, о том, как их импортировать и как работать со сторонними библиотеками и создавать собственные.

Репозитории, модули и пакеты

Управление библиотеками в Go основано на трех концепциях: *репозиториях*, *модулях* и *пакетах*. Что такое репозиторий, знает любой разработчик. Это то место в системе контроля версий, в котором хранится исходный код проекта. Модуль представляет собой корень размещенной в репозитории Go-библиотеки или приложения. Каждый модуль включает в себя один пакет или более, что придает модулю определенную структуру.



Вы можете разместить в одном репозитории сразу несколько модулей, но так делать не стоит. Содержимое каждого модуля версионруется как единое целое. Поэтому хранение двух модулей в одном репозитории будет означать отслеживание в одном репозитории отдельных версий для двух разных проектов.

Для того чтобы можно было использовать код пакетов, не входящих в стандартную библиотеку, каждый проект должен быть объявлен как модуль, обладающий глобально уникальным идентификатором. В этом плане Go ничем не отличается

от других языков. Так, в языке Java используются глобально уникальные объявления пакетов вида `com.companyname.projectname.library`.

В Go в качестве идентификатора обычно используется путь к тому репозиторию, в котором находится модуль. Так, например, модуль Proteus, написанный мной для упрощения доступа к реляционным базам данных в Go, находится на сайте GitHub по адресу <https://github.com/jonbodner/proteus> и, соответственно, обладает путем модуля `github.com/jonbodner/proteus`.

Файл go.mod

Набор исходного кода на языке Go становится модулем в том случае, если в его корневом каталоге имеется корректный файл `go.mod`. Вместо того чтобы создавать этот файл вручную, для этой цели следует использовать подкоманды команды для управления модулями `go mod`. Команда `go mod init ПУТЬ_МОДУЛЯ` создает файл `go.mod`, который делает корнем модуля текущий каталог. `ПУТЬ_МОДУЛЯ` здесь представляет собой глобально уникальное имя, идентифицирующее ваш модуль. Этот путь модуля чувствителен к регистру. Во избежание путаницы не используйте в нем буквы верхнего регистра.

Кратко коснемся того, что содержит файл `go.mod`:

```
module github.com/learning-go-book/money

go 1.15

require (
    github.com/learning-go-book/formatter v0.0.0-20200921021027-5abc380940ae
    github.com/shopspring/decimal v1.2.0
)
```

Каждый файл `go.mod` начинается с объявления модуля, которое включает в себя ключевое слово `module` и уникальный путь модуля. Далее указывается минимальная совместимая версия языка Go. Наконец, в секции `require` перечислены модули, от которых зависит данный модуль, и необходимая минимальная версия для каждого из них. О том, что означают эти версии, мы подробно поговорим в подразделе «Импортирование стороннего кода» на с. 235. Если ваш модуль не зависит от каких-либо других модулей, секцию `require` следует опустить.

Существует еще две опциональные секции. Секция `replace` позволяет переопределить расположение зависимого модуля, а секция `exclude` запрещает использование конкретной версии модуля.

Компиляция пакетов

Теперь, когда мы уже знаем, как можно превратить в модуль каталог с файлами исходного кода, пришла пора заняться организацией кода с помощью пакетов. Сначала вы узнаете, как работает оператор `import`, а затем мы поговорим о создании и организации пакетов, после чего коснемся некоторых плюсов и минусов пакетов языка Go.

Операции импорта и экспорта

Мы уже использовали оператор `import` языка Go, не касаясь того, что именно он делает и чем отличается от других языков. В Go оператор `import` позволяет получить доступ к экспортируемым константам, переменным, функциям и типам другого пакета. Обратиться к экспортируемым идентификаторам пакета (где под *идентификатором* понимается имя переменной, константы, типа, функции, метода или поля структуры) из другого текущего пакета невозможно, не используя оператор `import`.

Это порождает следующий вопрос: как в языке Go можно экспортировать идентификатор? Вместо того чтобы применять для этого специальное ключевое слово, язык Go использует *капитализацию* для выделения идентификаторов уровня пакета, доступных за пределами того пакета, в котором они объявляются. Когда идентификатор начинается с буквы верхнего регистра, он является *экспортируемым*. И наоборот, когда идентификатор начинается с буквы нижнего регистра или символа подчеркивания, доступ к нему может производиться только из того пакета, в котором он был объявлен.

Все, что вы экспортируете, является частью API вашего пакета. Прежде чем экспортировать идентификатор, убедитесь в том, что вы хотите предоставить клиентам доступ к нему. Документируйте все экспортируемые идентификаторы и следите за тем, чтобы они оставались обратно совместимыми при внесении изменений, если только вы не планируете намеренно внести значительные изменения в версию модуля (подробнее об этом будет рассказано в разделе «Версионирование своего модуля» на с. 246).

Создание и использование пакета

Создание пакета в Go не представляет больших сложностей. Посмотрим, как это делается, на примере небольшой программы. Код этой программы можно найти на сайте GitHub (<https://oreil.ly/WE7RN>). Каталог `package_example` содержит два дополнительных каталога, `math` и `formatter`. Каталог `math` содержит файл `math.go` со следующим кодом:

```
package math

func Double(a int) int {
    return a * 2
}
```

Первая строка в этом файле содержит *спецификатор пакета*, который состоит из ключевого слова `package` и имени пакета. Первая непустая и незакомментированная строка любого файла исходного кода на языке Go содержит спецификатор пакета.

Каталог `formatter` содержит файл `formatter.go` со следующим кодом:

```
package print

import "fmt"

func Format(num int) string {
    return fmt.Sprintf("The number is %d", num)
}
```

Обратите внимание, что в спецификаторе пакета мы указали, что пакет обладает именем `print`, но следует помнить, что он находится в каталоге `formatter`. Чуть позже мы поговорим об этом подробнее.

Наконец, корневой каталог содержит файл `main.go` со следующим кодом:

```
package main

import (
    "fmt"

    "github.com/learning-go-book/package_example/formatter"
    "github.com/learning-go-book/package_example/math"
)

func main() {
    num := math.Double(2)
    output := print.Format(num)
    fmt.Println(output)
}
```

В начале этой программы стоит уже знакомая нам строка. Все примеры кода, которые приводились до этой главы, начинались со строки `package main`. Чуть позже мы подробнее разберемся с тем, что это значит.

Далее идет секция операций импорта. Мы импортируем три пакета, первым из которых является пакет `fmt` из стандартной библиотеки. Этот пакет мы уже импортировали в предыдущих главах. Следующие две операции импорта относятся к пакетам внутри нашей программы. При импортировании пакетов,

которые не входят в состав стандартной библиотеки, необходимо указывать *путь импорта*. Чтобы получить путь импорта, нужно объединить путь модуля и путь к пакету внутри модуля.

Если, импортировав пакет, вы не воспользуетесь ни одним из его экспортируемых идентификаторов, то вам будет выдана ошибка компиляции. Такой подход гарантирует, что в любой двоичный файл, создаваемый компилятором языка Go, будет включаться только тот код, который действительно используется в программе.



Вы можете использовать относительный путь для импорта зависимого пакета в пределах того же модуля, но так делать не стоит. Абсолютные пути импорта ясно показывают, что именно вы импортируете, и это упрощает рефакторинг кода. Вам придется исправлять операции импорта с относительными ссылками, когда содержащий их файл будет перемещен в другой пакет, а если этот файл будет перемещен в другой модуль, вам придется сделать ссылки импорта абсолютными.

Запустив эту программу, вы получите следующий результат:

```
$ go run main.go
The number is 4
```

В функции `main` мы вызываем функцию `Double` из пакета `math`, записывая имя пакета перед именем функции. Мы уже видели примеры таких вызовов в предыдущих главах, когда использовали функции из стандартной библиотеки. Мы также вызываем здесь функцию `Format` из пакета `print`. Возможно, вас удивляет, откуда взялся этот пакет `print`, если мы импортируем пакет `github.com/learning-go-book/package_example/formatter`.

Каждый Go-файл, расположенный в определенном каталоге, должен обладать идентичным спецификатором пакета. (Из этого правила есть лишь одно небольшое исключение, которое мы обсудим в подразделе «Тестирование своего публичного API» на с. 335.) Однако здесь мы импортируем пакет `print`, используя путь импорта `github.com/learning-go-book/package_example/formatter`. Это объясняется тем, что *имя пакета определяется его спецификатором, а не путем импорта*.

Как правило, имя пакета совпадает с именем содержащего его каталога, поскольку в противном случае будет трудно выяснить, как называется пакет. Однако бывают ситуации, когда необходимо использовать имя пакета, которое отличается от имени каталога.

С первым из таких случаев мы имеем дело с самого начала и даже не подозреваем об этом. Для объявления пакета в качестве входной точки Go-приложения

используется специальное имя пакета `main`. Чтобы исключить возможность создания в таком случае сбивающих с толку операторов импорта, в Go не допускается импорт пакета `main`.

Другие причины для того, чтобы имя пакета не совпадало с именем каталога, возникают гораздо реже. Если в имени каталога присутствует символ, который недопустим для идентификаторов языка Go, то для пакета следует выбрать какое-то другое имя. Однако лучше вообще исключить вероятность такой ситуации, никогда не создавая каталог с именем, которое нельзя использовать в качестве идентификатора.

Последним случаем является создание каталога с именем, не совпадающим с именем пакета, для поддержки управления версиями при помощи каталогов. Мы обсудим этот случай подробнее в разделе «Версионирование своего модуля» на с. 246.

Имена пакетов находятся в *блоке файла*. Если один и тот же пакет используется в двух разных файлах одного пакета, этот пакет нужно импортировать в обоих файлах.

Именованние пакетов

Тот факт, что имя пакета является составной частью имени, используемого для обращения к определенным в пакете функциям и типам, несет с собой ряд последствий. Первое последствие состоит в том, что имена пакетов должны носить описательный характер. Вместо такого имени, как `util`, пакету лучше дать имя, описывающее предоставляемую им функциональность. Например, допустим, что у вас есть две вспомогательные функции, одна из которых служит для извлечения всех имен из строки, а вторая — для их приведения к необходимому формату. В таком случае не стоит создавать две функции в пакете `util` с именами `ExtractNames` и `FormatNames`. Если вы это сделаете, то при каждом вызове этих функций будут использованы имена `util.ExtractNames` и `util.FormatNames`, в которых префикс `util` не дает никакой информации о том, что делают функции.

Будет лучше определить одну функцию как функцию `Names` в пакете `extract`, а вторую — как функцию `Names` в пакете `format`. В том, что эти функции будут обладать одинаковым именем, нет ничего страшного, поскольку разные имена пакетов никогда не позволят принять одну функцию за другую. Для вызова первой функции будет использоваться имя `extract.Names`, а для вызова второй функции — `format.Names`.

Следует также не допускать того, чтобы имя пакета повторялось в именах функций и типов, определенных в этом пакете. Не стоит давать функции имя

`ExtractNames`, если она определяется в пакете `extract`. Исключением из этого правила является тот случай, когда имя идентификатора совпадает с именем пакета. Например, в пакете `sort` стандартной библиотеки есть функция `Sort`, а в пакете `context` — интерфейс `Context`.

Как следует подходить к организации кода модуля

В Go не существует какого-либо официального способа организации пакетов внутри модуля, но за годы использования этого языка появилось несколько подходов к организации кода, каждый из которых делает упор на обеспечение понятности и легкости в сопровождении. В случае небольшого модуля весь код следует разместить в одном пакете. Если от вашего модуля пока не зависят какие-либо другие модули, организацию кода вполне можно отложить на потом.

По мере роста вашего проекта вам потребуется навести в нем определенный порядок, чтобы сделать код более читабельным. Если ваш модуль состоит из одного или нескольких приложений, создайте каталог с именем `cmd` в корневом каталоге вашего модуля. Внутри каталога `cmd` сформируйте по одному каталогу для каждого двоичного файла, получаемого на основе вашего модуля. Например, ваш модуль может включать в себя веб-приложение и инструмент командной строки для анализа информации, содержащейся в базе данных этого веб-приложения. В каждом из этих каталогов используйте `main` в качестве имени пакета.

Если в корневом каталоге вашего модуля содержится множество файлов, предназначенных для управления тестированием и развертыванием вашего проекта (например, скрипты командной оболочки, конфигурационные файлы системы непрерывной интеграции или файлы `Dockerfile`), поместите весь свой Go-код (кроме пакетов `main`, расположенных в каталоге `cmd`) в пакеты внутри каталога с именем `pkg`.

Внутри каталога `pkg` организуйте код таким образом, чтобы зависимости между пакетами были сведены к минимуму. Один из распространенных подходов сводится к тому, чтобы разбить код на части в соответствии с выполняемыми задачами. Например, если вы используете Go для разработки сайта интернет-магазина, то в одном пакете можно разместить код, который служит для поддержки работы с клиентами, а в другом пакете — код для управления товарными запасами. Такой подход сводит к минимуму зависимости между пакетами, в результате чего вам будет легче в дальнейшем произвести рефакторинг кода с преобразованием одиночного веб-приложения в несколько микросервисов.

Хороший обзор рекомендаций в отношении структуры Go-проектов представила Кэт Зиен (Kat Zien) в своем докладе на конференции GopherCon 2018 (<https://oreil.ly/0zHY4>).

Переопределение имени пакета

В некоторых случаях вам потребуется импортировать два пакета с одинаковыми именами. Например, в стандартной библиотеке есть два пакета для генерирования случайных чисел, один из которых обеспечивает криптостойкость (`crypto/rand`), а другой — нет (`math/rand`). Простой генератор будет вполне уместен в том случае, когда случайные числа генерируются не для целей шифрования, а для предоставления непредсказуемого начального значения. Распространенный подход сводится к тому, чтобы предоставлять простому генератору случайных чисел начальное значение, сгенерированное криптографическим генератором. В Go оба соответствующих пакета обладают одинаковым именем (`rand`). В случае их импорта дайте одному из пакетов альтернативное имя в пределах текущего файла. Вы можете запустить соответствующий пример кода в онлайн-песочнице (<https://oreil.ly/YVwkm>). Прежде всего здесь следует обратить внимание на секцию операций импорта:

```
import (  
    crand "crypto/rand"  
    "encoding/binary"  
    "fmt"  
    "math/rand"  
)
```



В качестве имени пакета также можно использовать точку (.) и символ подчеркивания (_). При использовании точки (.) все экспортируемые идентификаторы импортируемого пакета переносятся в пространство имен текущего пакета, в результате чего вы можете обращаться к ним, не используя префикс. Так делать не рекомендуется, поскольку такой подход делает исходный код менее понятным и вы уже не сможете сразу понять, где был определен тот или иной идентификатор: в текущем или импортированном пакете.

О том, что происходит при использовании в качестве имени пакета символа подчеркивания (_), мы поговорим, когда будем обсуждать функцию `init` в разделе «По возможности не используйте функцию `init`» на с. 231.

Мы импортируем пакет `crypto/rand` как пакет `crand`. Это переопределяет имя `rand`, объявленное в этом пакете. После этого мы импортируем пакет `math/rand` обычным образом. Если вы посмотрите на код функции `seedRand`, то увидите, что при обращении к идентификаторам пакета `math/rand` используется префикс `rand`, а в случае пакета `crypto/rand` — префикс `crand`:

```
func seedRand() *rand.Rand {  
    var b [8]byte  
    _, err := crand.Read(b[:])  
    if err != nil {
```

```
    panic("cannot seed with cryptographic random number generator")
}
r := rand.New(rand.NewSource(int64(binary.LittleEndian.Uint64(b[:]))))
return r
}
```

Как упоминалось в подразделе «Затенение переменных» на с. 89, имена пакетов могут быть затенены. Объявление переменных, типов или функций с тем же именем, что и у пакета, делает этот пакет недоступным в блоке с таким объявлением. Если такого совпадения нельзя избежать (как, например, в том случае, когда имя вновь импортируемого пакета совпадает с именем имеющегося идентификатора), переопределите имя пакета во избежание конфликта имен.

Комментарии пакета и *godoc*

У языка Go есть собственный формат для написания комментариев, автоматически преобразуемых в документацию. Этот формат называется *godoc* и отличается исключительной простотой. Комментарии *godoc* не предполагают использования каких-либо специальных символов. Нужно лишь придерживаться общепринятых правил.

- Комментарий следует размещать непосредственно перед документируемым элементом, не оставляя пустых строк между комментарием и объявлением этого элемента.
- В начале комментария должны стоять два символа косой черты (*//*), а за ними — имя элемента.
- Комментарии можно разбивать на несколько абзацев с помощью пустого комментария.
- Для придания комментариям определенного формата можно дополнять строки отступами.

Комментарии, расположенные перед объявлением пакета, представляют собой комментарии уровня пакета. Если комментарии к вашему пакету занимают большой объем (как, например, в случае подробной документации по возможностям форматирования, предоставляемым пакетом *fmt*), то, согласно общепринятому соглашению, эти комментарии следует разместить внутри пакета в файле с именем *doc.go*.

Посмотрим, как должен выглядеть хорошо прокомментированный файл. Прежде всего, здесь присутствует комментарий уровня пакета, показанный в примере 9.1.

Пример 9.1. Комментарий уровня пакета

```
// Пакет money предоставляет различные утилиты с целью облегчить
// управление денежными средствами
```

Далее мы размещаем комментарий к экспортируемой структуре (пример 9.2). Обратите внимание, что он начинается с имени структуры.

Пример 9.2. Комментарий к экспортируемой структуре

```
// Money содержит информацию о размере денежной суммы и о том,
// в какой валюте она исчисляется
type Money struct {
    Value decimal.Decimal
    Currency string
}
```

Наконец, у нас здесь имеется комментарий к функции (пример 9.3).

Пример 9.3. Хорошо прокомментированная функция

```
// Convert преобразует размер суммы из одной валюты в другую.
//
// Эта функция принимает два параметра: экземпляр структуры Money,
// содержащий преобразуемый размер суммы, и строку с названием той валюты,
// в которую преобразуется денежная сумма.
// Convert возвращает сумму в указанной валюте или ошибку в том случае,
// если валюта будет неизвестной или в нее нельзя будет преобразовать
// денежную сумму.
// При возвращении ошибки экземпляр структуры Money устанавливается
// равным нулевому значению.
//
// Поддерживаются следующие валюты:
//     USD — доллар США
//     CAD — канадский доллар
//     EUR — евро
//     INR — индийская рупия
//
// Более подробные сведения о курсах обмена валют можно найти
// по адресу https://www.investopedia.com/terms/e/exchangerate.asp
func Convert(from Money, to string) (Money, error) {
    // ...
}
```

В «комплект поставки» языка Go входит инструмент командной строки `go doc`, который позволяет просматривать комментарии `godoc`. Команда `go doc ИМЯ_ПАКЕТА` отображает комментарии `godoc` для указанного пакета и список имеющихся в нем идентификаторов. С помощью команды `go doc ИМЯ_ПАКЕТА.ИМЯ_ИДЕНТИФИКАТОРА` можно отобразить документацию для определенного идентификатора в пакете.



Не забывайте снабжать свой код надлежащей документацией. Комментарии должны быть снабжены хотя бы все экспортируемые идентификаторы. Такие инструменты статического анализа, как `golangci-lint`, могут выполнять проверку наличия комментариев у всех экспортируемых идентификаторов.

Пакет `internal`

В некоторых случаях требуется сделать функцию, тип или константу общими для пакетов модуля, не делая их при этом частью вашего API. В Go это можно сделать путем использования специального имени пакета, `internal`.

При создании пакета с именем `internal` экспортируемые идентификаторы этого пакета и его подпакетов будут доступны только для других пакетов того же уровня и его родительского пакета. Посмотрим, как это выглядит на практике. Код этого примера можно найти на сайте GitHub (<https://oreil.ly/ksyAO>). Соответствующее дерево каталогов показано на рис. 9.1.



Рис. 9.1. Дерево файлов для примера использования пакета `internal`

Мы определили простую функцию в файле `internal.go` в пакете `internal`:

```
func Doubler(a int) int {
    return a * 2
}
```

Эту функцию можно вызывать из файла `foo.go` в пакете `foo` и из файла `sibling.go` в пакете `sibling`.

Имейте в виду, что попытка использовать внутреннюю функцию из файла `bar.go` в пакете `bar` или из файла `example.go` в корневом пакете приведет к ошибке при компиляции:

```
$ go build ./...
package github.com/learning-go-book/internal_example
example.go:3:8: use of internal package
github.com/learning-go-book/internal_example/foo/internal not allowed

package github.com/learning-go-book/internal_example/bar
bar/bar.go:3:8: use of internal package
github.com/learning-go-book/internal_example/foo/internal not allowed
```

По возможности не используйте функцию `init`

При чтении Go-кода обычно вполне понятно, какие методы и функции вызываются и когда именно. Отсутствие в Go переопределения методов и перегрузки функций отчасти обусловлено тем, что это позволяет быстрее понять, какой код выполняется в том или ином месте. В то же время вы можете настроить состояние пакета с помощью функции `init` без явного вызова чего-либо. Если вы определите функцию `init` без входных параметров и возвращаемых значений, она будет выполняться при первом обращении к данному пакету из другого пакета. Поскольку функции `init` не обладают входными параметрами и возвращаемыми значениями, они могут лишь производить некоторые побочные эффекты путем взаимодействия с функциями и переменными уровня пакета.

Еще одна уникальная особенность функции `init` состоит в том, что вы можете определить несколько функций `init` в одном пакете или даже в одном файле пакета. В документации определена четкая последовательность выполнения нескольких функций `init`, определенных в одном пакете, но вместо того, чтобы ее запоминать, лучше просто никогда не используйте эту возможность.

Некоторые пакеты, например пакеты драйверов баз данных, содержат функции `init` для регистрации драйвера базы данных. При этом не применяется ни один из определенных в пакете идентификаторов, но, как уже упоминалось, в Go нельзя импортировать пакет и затем нигде его не использовать. Чтобы обойти эту проблему, Go предлагает *пустой импорт* — оператор импорта, в котором в качестве имени пакета указан символ подчеркивания (`_`). Подобно тому как символ подчеркивания используется для пропуска неиспользуемых возвращаемых значений функции, пустой импорт запускает определенную в пакете функцию `init`, но не позволяет обращаться к экспортируемым идентификаторам пакета:

```
import (  
    "database/sql"  
  
    _ "github.com/lib/pq"  
)
```

Этот паттерн считается устаревшим, потому что при его использовании неясно, какая именно операция регистрации выполняется. Гарантия совместимости стандартной библиотеки в Go означает, что мы можем использовать ее для регистрации драйверов баз данных и форматов изображений, но в случае собственного паттерна регистрации необходимо явно регистрировать свои подключаемые модули.

Сегодня функции `init` используются главным образом для инициализации переменных уровня пакета, которые не могут быть настроены с помощью одной операции присваивания. Как мы знаем, определение изменяемого состояния на уровне пакета будет плохой идеей, поскольку это усложняет анализ существующих в приложении потоков данных. Это означает, что любые переменные уровня пакета, настраиваемые с помощью функций `init`, должны быть *фактически неизменными*. Хотя Go не предоставляет возможности гарантировать неизменность таких переменных, вы должны проследить за тем, чтобы ваш код не менял их значение. Если у вас есть переменные уровня пакета, значение которых требуется менять во время работы программы, посмотрите, нельзя ли произвести рефакторинг таким образом, чтобы это состояние было перемещено в структуру, которая инициализируется и возвращается функцией в пакете.

Нужно сделать еще пару оговорок в отношении использования функции `init`. Хотя Go позволяет определять несколько функций `init` для одного пакета, всегда используйте только одну такую функцию. Если ваша функция `init` производит загрузку файлов или обращение к сети, задокументируйте это поведение, чтобы эти операции ввода-вывода не стали неожиданностью для пользователей, уделяющих особое внимание безопасности.

Циклические зависимости

Помимо прочего, создатели языка Go ставили себе целью создание быстрого компилятора и простого в понимании исходного кода. Поэтому Go не допускает наличия *циклических зависимостей* между пакетами. Это значит, что если пакет А напрямую или косвенно импортирует пакет В, то пакет В не может напрямую или косвенно импортировать пакет А. Рассмотрим небольшой пример кода, чтобы лучше понять эту концепцию. Код этого примера можно найти на сайте GitHub (<https://oreil.ly/CXsyd>). В данном случае проект включает в себя

два подкаталога, `pet` и `person`. В файле `pet.go` в пакете `pet` импортируется пакет `github.com/learning-go-book/circular_dependency_example/person`:

```
var owners = map[string]person.Person{
    "Bob":  {"Bob", 30, "Fluffy"},
    "Julia": {"Julia", 40, "Rex"},
}
```

В то же время в файле `person.go` в пакете `person` импортируется пакет `github.com/learning-go-book/circular_dependency_example/pet`:

```
var pets = map[string]pet.Pet{
    "Fluffy": {"Fluffy", "Cat", "Bob"},
    "Rex":    {"Rex", "Dog", "Julia"},
}
```

При попытке скомпилировать этот проект будет выведено следующее сообщение об ошибке:

```
$ go build
package github.com/learning-go-book/circular_dependency_example
    imports github.com/learning-go-book/circular_dependency_example/person
    imports github.com/learning-go-book/circular_dependency_example/pet
    imports github.com/learning-go-book/circular_dependency_example/person:
        import cycle not allowed
```

Если у вас имеется циклическая зависимость, то вы можете выбрать один из нескольких вариантов решения этой проблемы. В некоторых случаях причиной этой проблемы является слишком мелкое разделение кода на пакеты. Если два пакета зависят друг от друга, то вполне вероятно, что их следует объединить в один пакет. В данном случае мы можем решить проблему, объединив в один пакет пакеты `person` и `pet`.

Если у вас есть веские основания для того, чтобы использовать два отдельных пакета, то, возможно, стоит переместить из одного пакета в другой или в новый пакет только те элементы, которые порождают циклическую зависимость.

Переименование и реорганизация API без потери работоспособности

По мере использования модуля вы можете обнаружить, что его API требует определенных изменений. Возможно, вы захотите переименовать часть экспортируемых идентификаторов или переместить их в другой пакет в пределах того же модуля. Чтобы не нарушать обратную совместимость при внесении

этих изменений, не удаляйте исходные идентификаторы, а просто дополните их альтернативными именами.

Для функции или метода это можно сделать достаточно легко. Нужно просто определить функцию или метод, вызывающие исходную версию. В случае константы объявляется новая константа того же типа и с тем же значением, но с другим именем.

Когда требуется переименовать или переместить экспортируемый тип, следует использовать *псевдоним*. Выражаясь простым языком, псевдоним — это новое имя для типа. В главе 7 было показано, как с помощью ключевого слова `type` можно объявить новый тип на основе уже существующего. Это ключевое слово также можно использовать для объявления псевдонима. Допустим, что у нас есть тип `Foo`:

```
type Foo struct {
    x int
    S string
}

func (f Foo) Hello() string {
    return "hello"
}

func (f Foo) goodbye() string {
    return "goodbye"
}
```

Чтобы пользователи могли обращаться к типу `Foo`, используя имя `Bar`, достаточно сделать следующее:

```
type Bar = Foo
```

Чтобы создать псевдоним, нужно записать ключевое слово `type`, имя, которое будет служить псевдонимом, знак равенства и имя исходного типа. Псевдоним обладает теми же полями и методами, что и исходный тип.

Кроме того, псевдоним может быть присвоен переменной исходного типа без преобразования типа:

```
func MakeBar() Bar {
    bar := Bar{
        x: 20,
        S: "Hello",
    }
    var f Foo = bar
    fmt.Println(f.Hello())
    return bar
}
```

Важно помнить, что псевдоним представляет собой просто другое имя типа. Если вы хотите добавить новые методы или изменить поля дополненной псевдонимом структуры, эти изменения следует произвести в исходном типе.

Псевдоним можно присвоить типу, определенному в этом же или другом пакете, и даже типу из другого модуля. При снабжении псевдонимом типа, определенного в другом пакете, псевдоним нельзя использовать для обращения к неэкспортируемым методам и полям исходного типа. Это вполне логично, поскольку псевдонимы существуют для того, чтобы сделать возможным постепенное изменение API пакетов, которые включают в себя только экспортируемые идентификаторы пакетов. Это ограничение можно обойти, вызывая в исходном пакете типа код, производящий манипуляции над неэкспортируемыми полями и методами.

Альтернативными именами можно снабжать два вида экспортируемых идентификаторов: переменные уровня пакета и поля структур. После выбора имени для экспортируемого поля структуры ему уже нельзя будет присвоить альтернативное имя.

Работа с модулями

Теперь, когда вы уже знаете, как работать с пакетами в пределах одного модуля, пришла пора узнать, как производится интеграция с другими модулями и определенными в них пакетами. После этого мы поговорим о публикации и версионировании собственных модулей, а также о централизованных сервисах языка Go: `pkg.go.dev`, прокси-сервере модулей и сводной базе данных.

Импортирование стороннего кода

До сих пор мы импортировали только такие пакеты стандартной библиотеки, как `fmt`, `errors`, `os` и `math`. Точно такая же система импорта используется в Go и для интеграции сторонних пакетов. В отличие от многих других компилируемых языков, Go компилирует в один двоичный файл весь код приложения, включая и ваш собственный, и сторонний код. Так же как мы импортировали пакет из нашего собственного проекта, при импортировании стороннего пакета нужно указать его расположение в репозитории исходного кода.

Рассмотрим соответствующий пример. В главе 2 было упомянуто, что в случае, когда требуется точное представление десятичных чисел, не следует использовать числа с плавающей запятой. Хорошим решением в таком случае будет использование модуля `decimal` из библиотеки ShopSpring (<https://github.com/shopspring/decimal>). Мы также рассмотрим простую библиотеку форматирования

(<https://github.com/learning-go-book/money>), которую я написал для этой книги. Мы применим эти два модуля в небольшой программе, где точно рассчитывается цена товара с учетом налога и выводится результат в требуемом формате. Файл `main.go` будет содержать следующий код:

```
package main

import (
    "fmt"
    "log"
    "os"

    "github.com/learning-go-book/formatter"
    "github.com/shopspring/decimal"
)

func main() {
    if len(os.Args) < 3 {
        fmt.Println("Need two parameters: amount and percent")
        os.Exit(1)
    }
    amount, err := decimal.NewFromString(os.Args[1])
    if err != nil {
        log.Fatal(err)
    }
    percent, err := decimal.NewFromString(os.Args[2])
    if err != nil {
        log.Fatal(err)
    }
    percent = percent.Div(decimal.NewFromInt(100))
    total := amount.Add(amount.Mul(percent)).Round(2)
    fmt.Println(formatter.Space(80, os.Args[1], os.Args[2],
        total.StringFixed(2)))
}
```

Импорт стороннего кода здесь производят ссылки `github.com/learning-go-book/formatter` и `github.com/shopspring/decimal`. Обратите внимание, что эти ссылки указывают расположение пакета в репозитории. Выполнив импорт, мы можем обращаться к экспортируемым идентификаторам этих пакетов точно так же, как это делается при импорте любого другого пакета.

Перед тем как скомпилировать это приложение, посмотрите, что содержит файл `go.mod`. Его содержимое должно быть таким:

```
module github.com/learning-go-book/money

go 1.15
```

После запуска компиляции на экран будет выведено следующее:

```
$ go build
go: finding module for package github.com/shopspring/decimal
go: finding module for package github.com/learning-go-book/formatter
go: found github.com/learning-go-book/formatter in
    github.com/learning-go-book/formatter v0.0.0-20200921021027-5abc380940ae
go: found github.com/shopspring/decimal in github.com/shopspring/decimal v1.2.0
```

Наличие ссылки на расположение пакета в каталоге исходного кода позволяет команде `go build` найти и скачать модуль этого пакета. Если вы посмотрите, что теперь содержит файл `go.mod`, то увидите следующее:

```
module github.com/learning-go-book/money

go 1.15

require (
    github.com/learning-go-book/formatter v0.0.0-20200921021027-5abc380940ae
    github.com/shopspring/decimal v1.2.0
)
```

Секция `require` файла `go.mod` содержит список модулей, импортируемых вами в свой модуль. После имени модуля здесь стоит номер версии. Поскольку у модуля `formatter` нет тега версии, для него была сгенерирована *псевдоверсия*.

Одновременно с этим был создан файл `go.sum`, содержащий следующее:

```
github.com/google/go-cmp v0.5.2/go.mod h1:v8dTdLbMG2kIc/vJv1+f65V22db...
github.com/learning-go-book/formatter v0.0.0-20200921021027-5abc38094...
github.com/learning-go-book/formatter v0.0.0-20200921021027-5abc38094...
github.com/shopspring/decimal v1.2.0 h1:abSATXmQEYyShuxI4/vyW3tV1MrKA...
github.com/shopspring/decimal v1.2.0/go.mod h1:DKyhrW/HYNuLGq1+MJL6WC...
golang.org/x/xerrors v0.0.0-20191204190536-9bdfabe68543/go.mod h1:I/5...
```

При каждом выполнении команды `go`, запрашивающей определенные зависимости (например, `go run`, `go build`, `go test` или даже `go list`), производится скачивание в кэш тех импортируемых пакетов, которые пока отсутствуют в файле `go.mod`. При этом в файл `go.mod` автоматически добавляются путь и версия модуля, содержащего нужный пакет. В файл `go.sum` добавляются две записи, одна из которых содержит путь, версию и хеш модуля, а вторая — хеш файла `go.mod` для этого модуля. Для чего используются эти хеши, будет рассказано далее в разделе «Прокси-серверы модулей» на с. 248.

Убедимся в том, что наш код работает, передав ему некоторые аргументы:

```
$ ./money 99.99 7.25
99.99          7.25                      107.24
```



Данная программа была зарегистрирована без файла `go.sum` и с неполным файлом `go.mod` — так я хочу показать, что происходит при заполнении этих файлов. Фиксируя свои проекты в системе контроля версий, всегда включайте в их состав обновленные файлы `go.mod` и `go.sum`. Это позволяет точно указать, какие версии зависимостей используются в вашем коде.

Работа с версиями

Посмотрим, как система модулей языка Go использует версии. Я написал простой модуль (<https://github.com/learning-go-book/simpletax>), который мы применим в еще одной программе для расчета налога (<https://oreil.ly/gjxYL>). В данном случае в файле `main.go` производится импорт следующих сторонних модулей:

```
"github.com/learning-go-book/simpletax"
"github.com/shopspring/decimal"
```

Как и в предыдущем случае, данная программа была зарегистрирована без обновленных файлов `go.mod` и `go.sum`, чтобы вы могли видеть, как заполняются эти файлы. Запустив компиляцию этой программы, мы увидим на экране следующее:

```
$ go build
go: finding module for package github.com/learning-go-book/simpletax
go: finding module for package github.com/shopspring/decimal
go: downloading github.com/learning-go-book/simpletax v1.1.0
go: found github.com/learning-go-book/simpletax in
    github.com/learning-go-book/simpletax v1.1.0
go: found github.com/shopspring/decimal in github.com/shopspring/decimal v1.2.0
```

Теперь файл `go.mod` содержит следующее:

```
module region_tax

go 1.15

require (
    github.com/learning-go-book/simpletax v1.1.0
    github.com/shopspring/decimal v1.2.0
)
```

Мы также теперь располагаем файлом `go.sum` с хешами для наших зависимостей. Запустим наш код и посмотрим, насколько хорошо он работает:

```
$ ./region_tax 99.99 12345
unknown zip: 12345
```

Похоже, мы получили неверный результат, причиной которого может быть ошибка в последней версии используемой библиотеки. При добавлении в проект

зависимости Go по умолчанию выбирает ее последнюю версию. Однако один из плюсов использования системы контроля версий состоит в том, что вы можете указать более раннюю версию модуля. Сначала посмотрим, какие версии модуля доступны, используя команду `go list`:

```
$ go list -m -versions github.com/learning-go-book/simpletax
github.com/learning-go-book/simpletax v1.0.0 v1.1.0
```

По умолчанию команда `go list` выводит список пакетов, используемых в вашем проекте. С флагом `-m` она выводит вместо этого список модулей, а с флагом `-versions` — список доступных версий указанного модуля. В этом случае мы видим, что нам доступны две версии: `v1.0.0` и `v1.1.0`. Откатимся к версии `v1.0.0` и посмотрим, не решит ли это нашу проблему. Это можно сделать с помощью команды `go get`:

```
$ go get github.com/learning-go-book/simpletax@v1.0.0
```

Команда `go get` позволяет работать с модулями, обновляя версию используемых зависимостей. Если мы взглянем на содержимое файла `go.mod`, то увидим, что теперь в нем указана другая версия:

```
module region_tax

go 1.15

require (
    github.com/learning-go-book/simpletax v1.0.0
    github.com/shopspring/decimal v1.2.0
)
```

Файл `go.sum` теперь содержит обе версии модуля `simpletax`:

```
github.com/learning-go-book/simpletax v1.0.0 h1:iH+7ADkdyrSqrMR2GzuwS...
github.com/learning-go-book/simpletax v1.0.0/go.mod h1:/YqHwHy95m0M4Q...
github.com/learning-go-book/simpletax v1.1.0 h1:Z/6s1ydS/vjblI6PFuEn...
github.com/learning-go-book/simpletax v1.1.0/go.mod h1:/YqHwHy95m0M4Q...
```

В этом нет ничего страшного: когда вы изменяете версию модуля или даже удаляете модуль из своего проекта, соответствующая запись может по-прежнему оставаться в файле `go.sum`. Это не вызывает никаких проблем.

Еще раз скомпилировав и запустив этот код, мы увидим, что ошибка была исправлена:

```
$ go build
$ ./region_tax 99.99 12345
107.99
```



Иногда в файле `go.mod` присутствуют зависимости с пометкой `// indirect`. Это зависимости, которые не объявляются в вашем проекте напрямую. Они могут быть добавлены в файл `go.mod` по нескольким причинам. В частности, это может объясняться тем, что ваш проект зависит от более старого модуля, у которого нет файла `go.mod`, или от модуля, файл `go.mod` которого является некорректным и не указывает некоторые из имеющихся зависимостей. При выполнении компиляции с использованием модулей в `go.mod` должны быть перечислены все имеющиеся зависимости. Поскольку эти объявления зависимостей нужно где-то разместить, производится модификация вашего файла `go.mod`.

Наличие косвенных объявлений также может объясняться тем, что прямая зависимость корректно определяет косвенную зависимость, но указывает при этом более старую ее версию вместо версии, установленной в вашем проекте. Такое возможно в том случае, когда вы обновляете косвенную зависимость напрямую с помощью команды `go get` или уменьшаете версию зависимости.

СЕМАНТИЧЕСКОЕ ВЕРСИОНИРОВАНИЕ

Практика снабжения программ номерами версий существует с незапамятных времен, но при этом не существует единого подхода к тому, что следует понимать под номером версии. В Go модули снабжаются номерами версий в соответствии с правилами *семантического версионирования* (semantic versioning, SemVer). Накладывая требование по использованию семантического версионирования для модулей, Go упрощает код для управления модулями, вместе с тем гарантируя, что пользователи модуля будут понимать, чего следует ожидать от нового релиза.

Если вы еще не знаете, что такое семантическое версионирование, то можете ознакомиться с полной спецификацией этого подхода по адресу <https://semver.org>. В кратком виде суть данного подхода сводится к тому, что номер версии составляется из трех частей: *основной версии*, *второстепенной версии* и версии *патча*. Эти версии записываются как *основная_версия.второстепенная_версия.версия_патча* (`major.minor.patch`) и начинаются с буквы `v`. Номер версии патча инкрементируется при исправлении программной ошибки, номер второстепенной версии инкрементируется (со сбросом в 0 версии патча) при добавлении новой обратно совместимой возможности, а номер основной версии инкрементируется (со сбросом в 0 второстепенной версии и версии патча) при внесении изменений, нарушающих обратную совместимость.

Выбор минимальной версии

В определенный момент может оказаться так, что ваш проект будет зависеть от двух или более модулей и все они будут зависеть от одного и того же модуля. В таком случае несколько модулей часто объявляют, что они зависят от разных номеров второстепенной версии и версии патча этого общего модуля. Как же Go разрешает эту проблему?

В таком случае система модулей руководствуется принципом *выбора минимальной версии*. Это означает, что вы всегда получаете наименьшую версию зависимости, которая в файлах `go.mod` всех ваших зависимостей объявлена как совместимая. Допустим, что ваш модуль напрямую зависит от модулей А, Б и В, каждый из которых, в свою очередь, зависит от модуля Г. В файлах `go.mod` этих модулей объявляется, что модуль А зависит от версии v1.1.0 модуля Г, модуль Б — от версии v1.2.0, а модуль В — от версии v1.2.3. Go произведет импорт модуля Г только один раз, выбрав версию v1.2.3, поскольку это наиболее свежая из указанных версий.

Однако, как иногда случается, вы можете обнаружить, что модуль А работает в сочетании с версией v1.1.0 модуля Г, но отказывается работать в сочетании с версией v1.2.3. Что же следует делать в таком случае? Go дает на это следующий ответ: вы должны связаться с разработчиками модуля и попросить их устранить имеющиеся несовместимости. Правило в отношении совместимости импорта гласит, что все второстепенные версии и версии патча модуля должны быть обратно совместимыми. Несоблюдение этого правила считается программной ошибкой. В нашем примере исправления нужно внести либо в модуль Г, поскольку он не обеспечивает обратную совместимость, либо в модуль А, поскольку он делает неверное допущение в отношении поведения модуля Г.

Это не самый приятный, но зато очень честный ответ. Некоторые системы компиляции, такие как `rpm`, в таких случаях используют несколько версий одного и того же пакета. Это может внести дополнительный ряд программных ошибок, особенно если вы используете состояние на уровне пакета. Кроме того, это ведет к увеличению размера приложения. В общем, надо сказать, что некоторые вещи лучше решать с помощью сообщества, а не путем изменения кода.

Обновление до совместимых версий

А что можно сказать насчет того случая, когда вам нужно напрямую обновить определенную зависимость? Допустим, что после того, как мы написали исходную версию своей программы, появилось еще три версии модуля `simpletax`. Первая версия исправила проблемы, которые были в исходном релизе v1.1.0. Представляя собой патч для исправления ошибок, не несущий никакой новой

функциональности, она получила номер версии v1.1.1. Во второй версии имеющаяся функциональность была дополнена некоторой новой функцией. Соответственно, она получила номер версии v1.2.0. Наконец, третья версия исправила ошибку, обнаруженную в версии v1.2.0, и соответственно получила номер версии v1.2.1.

Чтобы произвести обновление до версии патча, исправляющей ошибки текущей второстепенной версии, нужно воспользоваться командой `go get -u=patch github.com/learning-go-book/simpletax`. Поскольку ранее мы откатились до версии v1.0.0, эта команда не поменяет ее на другую версию, потому что у данного модуля нет версии патча с такой же второстепенной версией.

Если бы мы произвели обновление до версии v1.1.0 с помощью команды `go get github.com/learning-go-book/simpletax@v1.1.0` и уже после этого выполнили команду `go get -u=patch github.com/learning-go-book/simpletax`, то версия модуля была бы обновлена до версии v1.1.1.

Наконец, с помощью команды `go get -u github.com/learning-go-book/simpletax` можно получить самую свежую версию модуля `simpletax`. То есть будет произведено обновление до версии v1.2.1.

Обновление до несовместимых версий

Вернемся к нашей программе. Допустим, мы решили выйти на канадский рынок, и, к счастью, у модуля `simpletax` есть версия, позволяющая рассчитывать налоги и для США, и для Канады. Однако эта версия обладает несколько иным API по сравнению с предыдущей версией, и поэтому ей присвоен номер версии v2.0.0.

Во избежание несовместимости для Go-модулей установлено правило *семантического версионирования импорта*, которое сводится к следующему.

- Номер основной версии модуля следует увеличивать инкрементным образом.
- Для всех номеров основной версии, кроме 0 и 1, путь к модулю должен оканчиваться на vN , где N — номер основной версии.

Требование по изменению пути обусловлено тем, что путь импорта однозначно идентифицирует пакет, и, по определению, несовместимые версии пакета представляют собой другой пакет. Использование разных путей означает, что вы можете импортировать две несовместимые версии пакета в разные части своей программы, и это позволяет вам производить обновление без потери работоспособности.

Посмотрим, какие изменения потребуется внести в случае нашей программы. Прежде всего нам нужно изменить ссылку импорта для модуля `simpletax`:

```
"github.com/learning-go-book/simpletax/v2"
```

Таким образом, теперь мы ссылаемся в импорте на версию v2 этого модуля.

Далее мы изменим код функции `main` так, как показано ниже:

```
func main() {
    amount, err := decimal.NewFromString(os.Args[1])
    if err != nil {
        log.Fatal(err)
    }
    zip := os.Args[2]
    country := os.Args[3]
    percent, err := simpletax.ForCountryPostalCode(country, zip)
    if err != nil {
        log.Fatal(err)
    }
    total := amount.Add(amount.Mul(percent)).Round(2)
    fmt.Println(total)
}
```

Теперь мы считываем из командной строки третий параметр, представляющий собой код страны, и вызываем другую функцию в пакете `simpletax`. При выполнении команды `go build` наши зависимости будут автоматически обновлены:

```
$ go build
go: finding module for package github.com/learning-go-book/simpletax/v2
go: downloading github.com/learning-go-book/simpletax/v2 v2.0.0
go: found github.com/learning-go-book/simpletax/v2 in
    github.com/learning-go-book/simpletax/v2 v2.0.0
```

Выполнив эту программу, мы увидим, что ее результат уже выглядит по-другому:

```
$ ./region_tax 99.99 M4B1B4 CA
112.99
$ ./region_tax 99.99 12345 US
107.99
```

Взглянув на содержимое файла `go.mod`, мы убедимся, что в нем теперь указана новая версия модуля `simpletax`:

```
module region_tax

go 1.15

require (
    github.com/learning-go-book/simpletax v1.0.0 // indirect
    github.com/learning-go-book/simpletax/v2 v2.0.0
    github.com/shopspring/decimal v1.2.0
)
```

Также было обновлено и содержимое файла `go.sum`:

```
github.com/learning-go-book/simpletax v1.0.0 h1:iH+7ADkdyrSqrMR2GzuWS...
github.com/learning-go-book/simpletax v1.0.0/go.mod h1:/YqHwHy95m0M4Q...
github.com/learning-go-book/simpletax v1.1.0 h1:Z/6s1ydS/vjb1I6PFuDEn...
github.com/learning-go-book/simpletax v1.1.0/go.mod h1:/YqHwHy95m0M4Q...
github.com/learning-go-book/simpletax/v2 v2.0.0 h1:cZURCo1tEqdw/cJygg...
github.com/learning-go-book/simpletax/v2 v2.0.0/go.mod h1:DVMa7zPtIFG...
github.com/shopspring/decimal v1.2.0 h1:abSATxmQEYyShuxI4/vyW3tV1MrKA...
github.com/shopspring/decimal v1.2.0/go.mod h1:DKyhrW/HYNuLGq1+MJL6WC...
```

Несмотря на то что старые версии модуля `simpletax` уже не используются, они по-прежнему указаны в этом файле. Хотя в этом и нет ничего страшного, в Go имеется команда для удаления ссылок на неиспользуемые версии:

```
go mod tidy
```

После выполнения этой команды в файлах `go.mod` и `go.sum` останутся только ссылки на версию `v2.0.0`.

Вендоринг

Чтобы гарантировать, что модуль всегда будет компилироваться с использованием одних и тех же зависимостей, некоторые организации предпочитают сохранять копии используемых зависимостей внутри своего модуля. Такой подход называется *вендорингом*. Его можно активизировать с помощью команды `go mod vendor`, которая создает на верхнем уровне модуля каталог `vendor`, содержащий все его зависимости.

После добавления новых зависимостей в файле `go.mod` или обновления версии существующих зависимостей с помощью команды `go get` необходимо еще раз выполнить команду `go mod vendor`, чтобы обновить содержимое каталога `vendor`. Если вы забудете это сделать, то команды `go build`, `go run` и `go test` будут отказываться работать, выдавая сообщение об ошибке.

Выбор в отношении того, следует ли производить вендоринг зависимостей, оставляется на усмотрение вашей организации. Хотя более старые системы управления зависимостями для языка Go требовали, чтобы вы использовали вендоринг, после появления в Go системы модулей и прокси-серверов (подробности см. в разделе «Прокси-серверы модулей» на с. 248) эта практика стала уже не столь популярной. Преимущество вендоринга заключается в том, что вы точно знаете, какой сторонний код будет использован в вашем проекте. Его недостатком является значительное возрастание размера вашего проекта в системе контроля версий.

Сайт pkg.go.dev

Хотя не существует единого централизованного репозитория для Go-модулей, разработана отдельная служба, собирающая в одном месте документацию по Go-модулям. Разработчики языка Go создали сайт `pkg.go.dev` (<https://pkg.go.dev>), который автоматически индексирует Go-проекты с открытым исходным кодом. Для каждого модуля, представленного в каталоге пакетов, сайт выдает godoc-документацию, сведения об используемой лицензии, файл README, сведения о зависимостях модуля и о том, какие проекты с открытым исходным кодом, в свою очередь, зависят от него. На рис. 9.2 показано, какую информацию выдает сайт `pkg.go.dev` в случае нашего модуля `simpletax`.

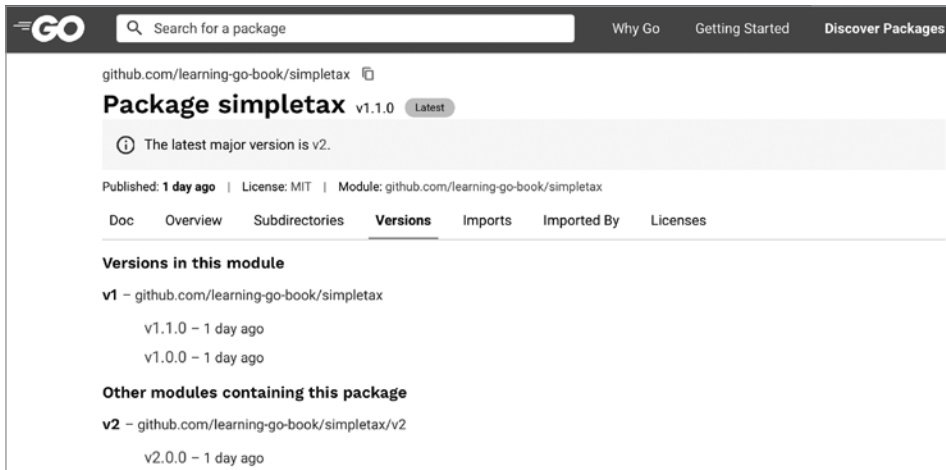


Рис. 9.2. Используйте сайт `pkg.go.dev` для поиска сторонних модулей и получения информации о них

Дополнительная информация

Мы рассмотрели здесь наиболее важные аспекты работы с модулями в Go. Более подробную информацию о модулях можно найти в вики-энциклопедии языка Go (<https://oreil.ly/LfIUj>).

Публикация своего модуля

Чтобы сделать свой модуль доступным для других пользователей, нужно просто разместить его в системе контроля версий. Это справедливо и в случае выпуска проекта в виде свободно распространяемого кода с помощью такой

общедоступной системы контроля версий, как GitHub, и в случае его выпуска в виде закрытого кода, хранимого внутри организации. Поскольку Go-программы компилируются из исходного кода и идентифицируются с помощью пути к репозиторию, вам не нужно явным образом загружать свой модуль в централизованный библиотечный репозиторий, как это делается при использовании таких систем, как Maven Central или npm. Не забудьте только зарегистрировать свои файлы `go.mod` и `go.sum`.

При выпуске модуля с открытым исходным кодом вы должны разместить в корне своего репозитория файл `LICENSE`, в котором будет указано, под какой лицензией для ПО с открытым исходным кодом вы выпускаете свой код. Подробную информацию о различных видах свободных лицензий можно найти на сайте It's FOSS (<https://oreil.ly/KVlrd>).

Если не вдаваться в подробности, то все свободные лицензии можно разделить на две основные категории: разрешительные (которые позволяют пользователям вашего кода сделать свой код закрытым) и неразрешительные (которые требуют, чтобы пользователи вашего кода сделали свой код открытым). Хотя выбор подходящей лицензии всегда остается за вами, Go-сообщество чаще отдает предпочтение таким разрешительным лицензиям, как BSD, MIT и Apache. Поскольку Go при компиляции всегда включает сторонний код непосредственно в состав приложения, в случае использования неразрешительной лицензии, такой как GPL, пользователям вашего кода тоже придется выпускать свой код как свободно распространяемое ПО, что является неприемлемым для многих организаций.

И еще один, последний совет: не выпускайте ПО под собственной лицензией. У большинства пользователей при этом возникнут сомнения относительно корректности вашей лицензии с правовой точки зрения и того, какие притязания вы можете выдвинуть в отношении их проекта.

Версионирование своего модуля

Вне зависимости от того, является ли ваш модуль общедоступным или закрытым, вы должны обеспечить его надлежащее версионирование, чтобы он мог корректно взаимодействовать с системой модулей языка Go. Это не представляет труда в случае добавления функциональности или исправления ошибок. Нужно лишь сохранить изменения в репозитории исходного кода и применить тег версии в соответствии с правилами семантического версионирования, которые были изложены во врезке «Семантическое версионирование» на с. 240.

Ситуация усложняется, когда вы доходите до точки, в которой требуется нарушить обратную совместимость. Как мы видели, когда импортировали версию 2 модуля `simpletax`, в случае обратно несовместимого изменения требуется использовать другой путь импорта. Для этого требуется выполнить несколько шагов.

Сначала нужно выбрать способ сохранения новой версии. Go поддерживает два способа создания отличающихся путей импорта.

- Создайте в своем модуле подкаталог с именем `vN`, где `N` — основная версия модуля. Например, если вы создаете версию 2 своего модуля, то нужно создать подкаталог `v2`. Скопируйте в этот подкаталог свой код, включая файлы `README` и `LICENSE`.
- Создайте ветвь в своей системе контроля версий. В эту ветвь можно поместить либо старый, либо новый код. Назовите ее `vN`, если вы решили поместить в нее новый код, и `vN-1`, если планируете поместить в нее старый код. Например, если вы создаете версию 2 своего модуля и хотите поместить в ветвь системы контроля версий код версии 1, эту ветвь следует назвать `v1`.

После того как вы определитесь со способом сохранения нового кода, нужно будет изменить путь импорта в том коде, который вы разместили в подкаталоге или в ветви системы контроля версий. Путь модуля в вашем файле `go.mod` должен заканчиваться на `/vN`, так же как и ссылки импорта внутри модуля. Внесение этих исправлений по всему коду может быть утомительным, поэтому Марван Сулайман (Marwan Sulaiman) создал инструмент, позволяющий делать это автоматически (<https://oreil.ly/BeOAr>). После того как эти пути будут исправлены, можно приступить к реализации изменений.



В принципе, вы можете просто изменить файл `go.mod` и операторы импорта, дополнить новым тегом версии свою основную ветвь и не беспокоиться о создании подкаталога или ветви в системе контроля версий. Однако такой подход считается плохой практикой, поскольку в таком случае не будет работать код, созданный с использованием более старых версий языка Go или сторонних менеджеров зависимостей.

Когда вы будете готовы опубликовать свой новый код, добавьте в свой репозиторий тег вида `vN.0.0`. Если вы используете систему подкаталогов или размещаете новый код в основной ветви, поставьте новый тег на основную ветвь. Если вы размещаете новый код в другой ветви, то следует пометить тегом эту ветвь.

Более подробные сведения об обновлении кода до несовместимой версии можно найти в посте «Go-модули: версия v2 и дальше» (<https://oreil.ly/E-3Qo>) в блоге Go Blog.

Прокси-серверы модулей

Вместо того чтобы использовать один централизованный репозиторий для библиотек, Go использует комбинированную модель. Каждый Go-модуль хранится в некотором репозитории исходного кода, таком как GitHub или GitLab. Но по умолчанию команда `go get` не извлекает код непосредственно из репозитория исходного кода. Вместо этого она отправляет запросы на *прокси-сервер*, поддерживаемый компанией Google (<https://proxy.golang.org>). На этом сервере хранятся копии каждой версии практически всех общедоступных Go-модулей. Если модуля или версии модуля нет на прокси-сервере, то команда `go get` скачивает этот модуль из репозитория модуля, сохраняет копию на сервере и возвращает модуль.

Наряду с прокси-сервером компания Google также поддерживает *сводную базу данных* с информацией о каждой версии каждого модуля. Это включает в себя записи из файла `go.sum` для каждой версии модуля и подписанное и закодированное описание дерева, содержащего эти записи. Подобно тому как прокси-сервер исключает вероятность отсутствия в интернете модуля или версии модуля, сводная база данных исключает вероятность использования модифицированной версии модуля. Изменения могут быть внесены со злым умыслом (как, например, в случае, когда злоумышленник похищает модуль и вносит в него вредоносный код) или по невнимательности (как, например, в случае, когда сопровождающий модуль специалист исправляет ошибку или добавляет новую функцию и повторно использует текущий тег версии). В любом случае вам нужно использовать немодифицированную версию модуля, поскольку иначе будет скомпилирован отличающийся двоичный файл и неизвестно, как это скажется на вашем приложении.

Каждый раз, когда вы скачиваете модуль с помощью команд `go build`, `go test` или `go get`, эти инструменты языка Go вычисляют хеш модуля и сверяют его с хешем, сохраненным для этой версии модуля в сводной базе данных. Если эти хеши не совпадают, модуль не устанавливается.

Указание прокси-сервера

Некоторые разработчики возражают против того, чтобы отправлять в компанию Google запросы на получение сторонних библиотек. Существует несколько вариантов решения этой проблемы.

- Если вы не возражаете против использования общедоступного прокси-сервера, но не хотите использовать сервер компании Google, то можете переключиться на сервер GoCenter (поддерживаемый компанией JFrog), присвоив переменной среды `GOPROXY` значение <https://gocenter.io,direct>.

- Вы можете вообще отключить использование прокси-сервера, присвоив переменной среды `GOPROXY` значение `direct`. При этом модули будут скачиваться непосредственно из своих репозиториев, но, если вам потребуется версия, которая была удалена из репозитория, вы не получите к ней доступ.
- Можете использовать свой собственный прокси-сервер. Так, корпоративные версии репозиториев `Artifactory` и `Sonatype` предлагают встроенную поддержку прокси-сервера для языка `Go`. Проект `Athens Project` (<https://docs.gomods.io>) предлагает прокси-сервер с открытым исходным кодом. Установите один из этих продуктов в своей сети, а затем присвойте его URL-адрес переменной среды `GOPROXY`.

Закрытые репозитории

Большинство организаций хранит свой код в закрытых репозиториях. Когда требуется использовать закрытый модуль в другом Go-проекте, вы не можете запросить его у прокси-сервера компании Google. В таком случае прокси-сервер компании Google откатится к запасной схеме и обратится к закрытому репозиторию напрямую, но вы, вероятно, не захотите раскрывать внешним сервисам имена закрытых серверов и репозиториев.

Если вы используете собственный прокси-сервер или отключили использование прокси-сервера, то эта ситуация не будет для вас проблемой. Использование закрытого прокси-сервера несет за собой и ряд других преимуществ. Прежде всего это повышает скорость скачивания сторонних модулей, поскольку они кэшируются в сети вашей компании. Если для доступа к вашим закрытым репозиториям требуется аутентификация, то использование закрытого прокси-сервера позволяет вам не беспокоиться о возможном раскрытии аутентификационной информации в вашем конвейере непрерывной интеграции и непрерывного развертывания. Закрытый прокси-сервер настраивается для доступа к вашим закрытым репозиториям с прохождением аутентификации (см. документацию по настройке аутентификации для сервера `Athens` (<https://oreil.ly/Nl4hv>)), в то время как доступ к самому закрытому прокси-серверу выполняется без аутентификации.

При использовании общедоступного прокси-сервера вы можете присвоить переменной среды `GOPRIVATE` список ваших закрытых репозиториев, разделенный запятыми. Например, допустим, что вы присвоили этой переменной следующий список:

```
GOPRIVATE=*.example.com,company.com/repo
```

В таком случае вы сможете напрямую скачивать любой модуль из репозитория, который расположен в любом поддомене домена `example.com` или URL-адрес которого начинается с `company.com/repo`.

Резюме

В этой главе вы изучили, как в Go следует подходить к организации кода и взаимодействовать с экосистемой исходного кода. Вы узнали о том, как работают модули, как производить организацию кода в виде пакетов, как использовать сторонние модули и как публиковать собственные модули. В следующей главе мы займемся исследованием одной из самых важных составляющих языка Go: средств конкурентного программирования.

Конкурентность в Go

Под конкурентностью в теории вычислительных машин понимается разбиение одного процесса на несколько независимых составляющих и определение для них безопасного способа совместного использования данных. В большинстве языков конкурентность обеспечивается с помощью библиотеки, которая использует потоки операционной системы, осуществляющие совместную работу с данными путем установки блокировок. В отличие от этого, модель конкурентности языка Go, которая по праву считается его наиболее важной особенностью, основана на теории взаимодействующих последовательных процессов (Communicating Sequential Processes, CSP). Этот стиль конкурентности был впервые описан в 1978 году в статье Тони Хоара (Tony Hoare) (<https://oreil.ly/x1IVG>), который в свое время разработал алгоритм быстрой сортировки (Quicksort). Паттерны конкурентного программирования, реализованные на основе теории CSP, такие же мощные, как и стандартные, но при этом менее сложные для понимания.

В этой главе мы кратко рассмотрим основные средства обеспечения конкурентности в Go: горутины, каналы и ключевое слово `select`. После этого мы рассмотрим ряд паттернов конкурентного программирования, широко используемых в Go, и вы узнаете, в каких случаях будет лучше использовать более низкоуровневые методы.

Когда следует использовать конкурентность

Наверное, стоит начать со следующего предупреждения: применяйте конкурентность лишь в том случае, когда она делает вашу программу лучше. Когда неопытные Go-разработчики начинают экспериментировать с конкурентностью, они обычно проходят через следующие этапы.

1. Это *потрясающе*! Теперь я все буду помещать в горутины!
2. Моя программа не стала работать быстрее. Наверное, стоит снабдить каналы буферами.
3. Мои каналы блокируют друг друга, и я получаю взаимоблокировки. Теперь я буду использовать буферизованные каналы с *действительно большими* буферами.
4. Мои каналы по-прежнему блокируют друг друга. Теперь я буду использовать мьютексы.
5. С меня хватит! Обойдемся без конкурентности!

Конкурентность привлекает разработчиков тем, что, по идее, конкурентные программы должны работать быстрее. К сожалению, это не всегда так. Дополнительное использование конкурентности не всегда ведет к повышению производительности и часто делает код менее понятным. Важно понимать, что *конкурентность* — это не *параллелизм*. Конкурентность — это инструмент, позволяющий лучше структурировать решаемую задачу. Будет или нет конкурентный код выполняться параллельно (одновременно), зависит от используемого аппаратного обеспечения и от того, позволяет ли это алгоритм. В 1967 году Джин Амдал (Gene Amdahl), один из первопроходцев в области вычислительной техники, вывел закон Амдала, который показывает, насколько параллельная обработка может повысить производительность в зависимости от того, какой объем работы должен выполняться последовательно. Подробнее о законе Амдала можно прочитать в книге Клея Брешерса (Clay Breshears) «Искусство конкурентности» (<https://oreil.ly/HaZQ8>). Для понимания материала данной книги достаточно знать, что дополнительное использование конкурентности не всегда ведет к повышению скорости.

Если не вдаваться в подробности, то процесс выполнения любой программы можно разделить на три этапа: получение данных, их преобразование и вывод результата. Ответ на вопрос, следует или нет использовать в программе конкурентность, зависит от того, как движутся данные в программе по мере выполнения этих этапов. Иногда два этапа могут выполняться конкурентно, потому что результаты одного этапа не требуются для выполнения другого этапа, а иногда два этапа должны выполняться последовательно, потому что результаты одного этапа необходимы для выполнения другого этапа. Применяйте конкурентность, когда требуется объединить результаты нескольких операций, которые могут выполняться независимо друг от друга.

Также важно отметить, что конкурентность не стоит использовать в тех случаях, когда выполнение процессов не занимает много времени. Помните о том, что конкурентность несет за собой определенные издержки. Резидентные реализации широко используемых алгоритмов часто выполняются настолько быстро, что

накладные расходы на передачу значений посредством конкурентности перевешивают любую потенциальную экономию за счет параллельного выполнения конкурентного кода. Именно поэтому конкурентность часто используется при выполнении операций ввода-вывода: операции чтения или записи на диск или в сеть осуществляются в тысячи раз медленнее, чем любые резидентные процессы, кроме самых сложных. Если вы не уверены в том, что конкурентность повысит производительность, сначала реализуйте код, используя последовательный подход, а затем напишите сравнительный тест, позволяющий оценить уровень производительности, обеспечиваемый в случае конкурентной реализации. (О том, как следует проводить сравнительное тестирование своего кода, будет подробно рассказано в разделе «Сравнительные тесты» на с. 343.)

Рассмотрим пример. Допустим, мы создаем веб-сервис, вызывающий три других веб-сервиса. Мы отправляем данные двум сервисам, получаем результаты этих двух вызовов и отправляем их третьему сервису, который возвращает окончательный результат. Весь этот процесс должен занимать не более 50 миллисекунд; в противном случае следует вернуть сообщение об ошибке. Этот случай хорошо подходит для использования конкурентности, поскольку здесь есть части кода, которые должны выполнять ввод-вывод и могут выполняться без взаимодействия друг с другом, часть кода, в которой результаты сводятся воедино, и ограничение в отношении времени выполнения кода. Как можно реализовать этот код, будет показано в конце этой главы.

Горутины

Горутины являются ключевой концепцией в модели конкурентности языка Go. Чтобы лучше понять, что это такое, определим пару терминов. Прежде всего разберемся, что такое *процесс*. Процесс — это экземпляр программы, выполняемой операционной системой компьютера. Операционная система связывает с процессом такие ресурсы, как память, и следит за тем, чтобы другие процессы не могли их использовать. Процесс состоит из одного или нескольких *поток*ов. Поток — это единица выполнения, на работу которой операционная система дает некоторое время. Потоки одного процесса совместно используют его ресурсы. Центральный процессор может выполнять инструкции из одного или нескольких потоков одновременно, в зависимости от количества имеющихся у него ядер. Одной из задач операционной системы является планирование выполнения потоков процессором таким образом, чтобы был выполнен каждый процесс (и каждый его поток).

Горутины представляют собой легковесные процессы, которыми распоряжается среда выполнения языка Go. При запуске Go-программы среда выполнения языка Go создает для ее выполнения несколько потоков и запускает одну горутину.

Все создаваемые программой горутины, включая самую первую, автоматически закрепляются за этими потоками планировщиком среды выполнения языка Go, подобно тому как операционная система планирует выполнение потоков центральным процессором. Хотя это и выглядит как лишняя работа, поскольку нижележащая операционная система уже включает в себя планировщик, осуществляющий управление потоками и процессами, такой подход несет с собой несколько преимуществ.

- Создание горутины занимает меньше времени, чем создание потока, поскольку при этом не создается системный ресурс.
- Исходный размер стека горутин меньше размера стека потоков и может быть увеличен по мере необходимости. Это делает горутины более эффективными в плане использования памяти.
- Переключение между горутинами занимает меньше времени, чем переключение между потоками, поскольку осуществляется полностью внутри процесса, что исключает необходимость выполнения сравнительно медленных системных вызовов.
- Планировщик может оптимизировать свои решения, поскольку он является составной частью Go-процесса. Взаимодействуя с сетевым опрашивателем, планировщик выявляет те случаи, когда следует отменить выполнение горутины, чтобы она не блокировала ввод-вывод. Он также взаимодействует со сборщиком мусора и следит за тем, чтобы работа была равномерно распределена между потоками операционной системы, выделенными для вашего Go-процесса.

Эти преимущества позволяют Go-программам создавать сотни, тысячи и даже десятки тысяч одновременных горутин. Если же вы попытаетесь запустить тысячи потоков в языке со встроенной поддержкой работы с потоками, то ваша программа станет до невозможности медленной.



Если вы хотите узнать подробнее, как планировщик делает свою работу, прослушайте доклад по этой теме, который представила Кавья Джоши (Kavya Joshi) на конференции GopherCon 2018 (<https://oreil.ly/879mk>).

Горутина запускается путем размещения ключевого слова **go** перед вызовом функции. Этой функции можно передать параметры для инициализации ее состояния, как это делается в случае любой другой функции, но ее возвращаемые значения будут игнорироваться.

В качестве горутин можно запустить любую функцию. В этом Go отличается от языка JavaScript, где функция может работать асинхронно только в том случае,

если это будет указано в ее определении с помощью ключевого слова `async`. В то же время горутины принято запускать с помощью замыкания, обертывающего бизнес-логику. При этом замыкание берет на себя выполнение всей работы по обеспечению конкурентности. Так, в частности, замыкание считывает значения из каналов и передает их бизнес-логике, которая не знает о том, что она выполняется в горутине. После этого результат функции записывается обратно в другой канал. (О каналах поговорим в следующем разделе.) Такое разделение обязанностей делает ваш код модульным и легко тестируемым, исключая конкурентность из ваших API:

```
func process(val int) int {
    // выполнение действий над переменной val
}

func runThingConcurrently(in <-chan int, out chan<- int) {
    go func() {
        for val := range in {
            result := process(val)
            out <- result
        }
    }()
}
```

Каналы

Горутины общаются друг с другом посредством *каналов*. Подобно срезам и картам, каналы представляют собой встроенный тип, экземпляры которого создаются с помощью функции `make`:

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

Как и карты, каналы представляют собой ссылочный тип. Когда вы передаете канал функции, ей в действительности передается указатель на канал. И так же, как для карт и срезов, нулевым значением для каналов является значение `nil`.

Чтение, запись и буферизация

Для взаимодействия с каналом нужно воспользоваться оператором `<-`. Для чтения из канала нужно разместить оператор `<-` слева от переменной канала, а для записи в канал — справа от нее.

```
a := <-ch // считывает значение из канала ch и присваивает его переменной a
ch <- b   // записывает значение переменной b в канал ch
```

Каждое записанное в канал значение может быть считано только один раз. Если несколько горутин производят чтение из одного и того же канала, то записанное в него значение считает только одна из этих горутин.

Горутин редко производит чтение и запись в один и тот же канал. Если значение канала присваивается переменной или полю либо передается в качестве параметра функции, поставьте стрелку перед ключевым словом `chan` (`ch <- chan int`), чтобы указать, что горутин производит только *чтение* из канала. Чтобы указать, что горутин только *записывает* в канал, поставьте стрелку после ключевого слова `chan` (`ch chan <- int`). При таком подходе компилятор языка Go сможет гарантировать, что функция будет производить только чтение или запись в канал.

По умолчанию каналы являются *небуферизованными*. После каждой операции записи в открытый небуферизованный канал производящая запись горутин делает паузу до тех пор, пока другая горутин не произведет чтение из этого канала. И точно так же после каждой операции чтения из открытого небуферизованного канала производящая чтение горутин делает паузу до тех пор, пока другая горутин не произведет запись в этот канал. Это означает, что для чтения или записи в небуферизованный канал требуются как минимум две параллельно работающие горутин.

Go также позволяет использовать и *буферизованные* каналы, которые буферизуют без блокировки некоторое ограниченное количество операций записи. Если буфер заполнится еще до выполнения каких-либо операций чтения из канала, то следующая операция записи в этот канал приостановит записывающую горутин до тех пор, пока не будет произведено чтение из канала. К такой же блокировке, как в случае записи в канал с заполненным буфером, приводит и попытка чтения из канала с пустым буфером.

Чтобы создать буферизованный канал, нужно указать емкость буфера при создании канала:

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

Встроенные функции `len` и `cap` позволяют получить информацию о буферизованном канале. С помощью функции `len` можно узнать текущее количество значений в буфере, с помощью функции `cap` — максимальный размер буфера, или его «емкость». Емкость буфера нельзя изменить.



При передаче небуферизованного канала функции `len` и `cap` возвращают 0. Это вполне логично, поскольку по определению у небуферизованного канала нет буфера, в котором можно было бы разместить значения.

В большинстве случаев следует использовать небуферизованные каналы. Когда именно могут оказаться полезными буферизованные каналы, будет рассказано в разделе «Когда следует использовать буферизованные и небуферизованные каналы» на с. 268.

Цикл `for-range` и каналы

Для чтения из канала также можно использовать цикл `for-range`:

```
for v := range ch {  
    fmt.Println(v)  
}
```

В отличие от других вариантов использования цикла `for-range` в данном случае для канала объявляется только одна переменная, которая представляет содержащиеся в нем значения. Выполнение цикла продолжается до тех пор, пока канал не будет закрыт или не будет встречен оператор `break` либо `return`.

Заккрытие канала

После завершения записи в канал его следует закрыть с помощью встроенной функции `close`:

```
close(ch)
```

После закрытия канала любые попытки произвести запись в канал или закрыть его снова приведут к панике. Однако, что интересно, попытки чтения из закрытого канала всегда завершаются успехом. Если канал является буферизованным и в нем еще есть непрочитанные значения, то они будут возвращены в последовательном порядке. Если канал является небуферизованным или буферизованным, но уже без каких-либо значений, то будет возвращено нулевое значение для используемого каналом типа.

Здесь мы сталкиваемся с практически такой же проблемой, как в случае работы с картами: как при чтении из канала можно отличить нулевое значение, которое было записано в канал, от нулевого значения, возвращаемого, потому что канал уже закрыт? Поскольку создатели языка Go постарались сделать его предельно единообразным, данная проблема решается так же, как и в случае карт, а именно путем использования идиомы «запятая-ок» для проверки, закрыт канал или нет:

```
v, ok := <-ch
```

Если переменной `ok` будет присвоено значение `true`, значит, канал открыт. Если переменной `ok` будет присвоено значение `false`, значит, канал закрыт.



Производя чтение из канала, который может оказаться закрытым, всегда используйте идиому «запятая-ok», чтобы убедиться в том, что канал еще открыт.

Задача закрытия канала возлагается на ту горутину, которая производит запись в канал. Помните о том, что закрывать канал нужно только в том случае, если определенная горутина ожидает закрытия канала (как, например, горутина, производящая чтение из канала с помощью цикла `for-range`). Поскольку каналы представляют собой лишь еще одну разновидность переменных, среда выполнения языка Go может выявлять уже неиспользуемые каналы и удалять их путем сборки мусора.

Каналы являются одной из тех двух вещей, которые выгодно отличают модель конкурентности языка Go от других языков. Они позволяют представить код в виде последовательности этапов и делают ясно выраженными зависимости от данных, что облегчает понимание концепции конкурентности. В других языках для обмена данными между потоками используется глобальное общее состояние. Это изменяемое общее состояние усложняет процесс прохождения данных через программу, из-за чего, в свою очередь, трудно определить, являются ли два потока в действительности независимыми друг от друга.

Различия в поведении каналов

Каналы могут находиться во множестве разных состояний, каждое из которых обладает разным поведением в случае чтения, записи или закрытия. Эти различия в поведении каналов представлены в табл. 10.1.

Ситуаций, в которых Go-программы вынуждены выдавать панику, следует по возможности избегать. Как уже упоминалось, стандартный подход сводится к тому, чтобы сделать записывающую горутину ответственной за закрытие канала после того, как будут записаны все имеющиеся данные. Ситуация усложняется, когда несколько горутин производят запись в один и тот же канал, поскольку двукратный вызов функции `close` для одного и того же канала вызывает панику. Кроме того, если вы закроете канал в одной горутине, то попытка произвести в него запись в другой горутине тоже приведет к панике. Эта проблема решается путем использования типа `sync.WaitGroup`. Как это можно сделать, будет показано в подразделе «Использование типа `WaitGroup`» на с. 272.

Опасность могут представлять и каналы, равные `nil`, однако существуют ситуации, в которых они будут полезны. Подробнее о таких каналах будет рассказано в подразделе «Отключение ветвей оператора `select`» на с. 270.

Таблица 10.1. Различия в поведении каналов

	Небуферизованный, открытый	Небуферизованный, закрытый	Буферизованный, открытый	Буферизованный, закрытый	Равный nil
Чтение	Приостанавливает выполнение до тех пор, пока не будет произведена запись	Возвращает нулевое значение (используйте идиому «запятая-ок», чтобы проверить, закрыт ли канал)	Приостанавливает выполнение, если буфер пуст	Возвращает имеющееся в буфере значение. Если буфер пуст, возвращает нулевое значение (используйте идиому «запятая-ок», чтобы проверить, закрыт ли канал)	Бесконечное зависание
Запись	Приостанавливает выполнение до тех пор, пока не будет произведено чтение	ПАНИКА	Приостанавливает выполнение, если буфер заполнен	ПАНИКА	Бесконечное зависание
Закрытие	Работает	ПАНИКА	Работает, в буфере еще имеются значения	ПАНИКА	ПАНИКА

Оператор select

Оператор `select` является еще одной составляющей, которая выгодно отличает модель конкурентности языка Go от других языков. Это управляющая конструкция, предусмотренная в Go для конкурентности и позволяющая изящно решить часто возникающий вопрос о том, какую из двух конкурентно выполняемых операций следует выполнять первой. При этом нельзя сделать какую-либо операцию более приоритетной, чтобы избежать проблемы зависания процесса из-за недостатка ресурсов.

Оператор `select` позволяет горутине произвести чтение или запись в один из нескольких каналов и во многом напоминает пустой оператор `switch`.

```
select {
case v := <-ch:
    fmt.Println(v)
case v := <-ch2:
    fmt.Println(v)
```

```
case ch3 <- x:
    fmt.Println("wrote", x)
case <-ch4:
    fmt.Println("got value on ch4, but ignored it")
}
```

Каждая ветвь `case` оператора `select` производит чтение или запись в канал. Если указанная в ветви `case` операция чтения или записи может быть выполнена, то она выполняется вместе с телом этой ветви `case`. Как и в случае оператора `switch`, каждая ветвь `case` оператора `select` создает собственный блок.

Что же происходит, когда выполняются операции чтения или записи в канал, указанные в нескольких ветвях? В таком случае оператор `select` просто выбирает случайным образом одну из тех ветвей, которые могут быть выполнены, не обращая внимания на порядок их выполнения. В этом оператор `select` сильно отличается от оператора `switch`, который всегда выбирает первую ветвь `case`, дающую в результате значение `true`. Это также полностью исключает вероятность зависания процессов из-за недостатка ресурсов, поскольку все ветви обладают одинаковым приоритетом и проверяются в одно и то же время.

Еще одно преимущество случайного выбора ветвей `case` в операторе `select` — исключение вероятности установки блокировок в несогласованном порядке, что является одной из наиболее частых причин взаимоблокировок. Если две горутины осуществляют доступ к некоторым двум каналам, то в обеих горутинах доступ должен производиться в одинаковом порядке. В противном случае произойдет *взаимоблокировка*, то есть ситуация, при которой ни одна из горутин не может продолжить выполнение, поскольку ожидает, что другая горутина выполнит определенные действия. Если каждая горутина в Go-приложении окажется в состоянии взаимоблокировки, то среда выполнения языка Go прервет выполнение программы (пример 10.1).

Пример 10.1. Взаимоблокировка горутин

```
func main() {
    ch1 := make(chan int)
    ch2 := make(chan int)
    go func() {
        v := 1
        ch1 <- v
        v2 := <-ch2
        fmt.Println(v, v2)
    }()
    v := 2
    ch2 <- v
    v2 := <-ch1
    fmt.Println(v, v2)
}
```

Если вы попытаете выполнить эту программу в онлайн-песочнице (<https://oreil.ly/trOam>), то получите следующее сообщение об ошибке:

```
fatal error: all goroutines are asleep - deadlock!
```

Функция `main` выполняется в горутине, которая запускается средой выполнения языка Go при запуске программы. Запущенная нами вторая горутина не может продолжить выполнение, пока не будет считано значение из канала `ch1`, а основная горутина не может продолжить выполнение, пока не будет считано значение из канала `ch2`.

Однако если мы обернем операции доступа к каналам в основной горутине в оператор `select`, вероятность взаимной блокировки будет исключена (пример 10.2).

Пример 10.2. Исключение взаимоблокировок с помощью оператора `select`

```
func main() {
    ch1 := make(chan int)
    ch2 := make(chan int)
    go func() {
        v := 1
        ch1 <- v
        v2 := <-ch2
        fmt.Println(v, v2)
    }()
    v := 2
    var v2 int
    select {
    case ch2 <- v:
    case v2 = <-ch1:
    }
    fmt.Println(v, v2)
}
```

Запустив эту программу в онлайн-песочнице (<https://oreil.ly/SdQX5>), вы получите следующий результат:

```
2 1
```

Здесь исключена вероятность взаимоблокировки, поскольку оператор `select` проверяет возможность выполнения каждой ветви. Запущенная нами вторая горутина записала в канал `ch1` значение 1, поэтому в основной горутине может быть успешно выполнено чтение из канала `ch1` в переменную `v2`.

Поскольку оператор `select` обеспечивает обмен данными с несколькими каналами, его часто встраивают в цикл `for`:

```
for {
    select {
```

```

    case <-done:
        return
    case v := <-ch:
        fmt.Println(v)
}
}

```

В силу широкого использования такого подхода эту комбинацию часто называют циклом **for-select**. При использовании цикла **for-select** нужно позаботиться о том, чтобы в определенный момент производился выход из цикла. Один из возможных подходов к этому будет показан в подразделе «Паттерн на основе канала **done**» на с. 266.

Как и оператор **switch**, оператор **select** может обладать ветвью **default**. И точно так же, как в операторе **switch**, эта ветвь выбирается, когда не может быть выполнена ни одна из операций чтения или записи в канал, указанных в ветвях **case**. Чтобы реализовать неблокирующую операцию чтения или записи в канал, используйте оператор **select** с ветвью **default**. При отсутствии значения в канале **ch** следующий код не будет дожидаться того момента, когда из него можно будет произвести чтение, а сразу же перейдет к выполнению тела ветви **default**:

```

select {
case v := <-ch:
    fmt.Println("read from ch:", v)
default:
    fmt.Println("no value written to ch")
}

```

О том, как может использоваться оператор **select** с ветвью **default**, будет рассказано в подразделе «Противодавление» на с. 269.



В большинстве случаев не стоит использовать ветвь **default** внутри цикла **for-select**. При таком подходе ветвь **default** будет срабатывать на каждой итерации цикла, не выявившей ветвей **case** с выполнимыми операциями чтения или записи. В результате цикл **for** будет работать непрерывно, повышая расход времени процессора.

Принципы и паттерны конкурентного программирования

Теперь, когда мы уже знаем, какие инструменты предусмотрены в Go для обеспечения конкурентности, рассмотрим некоторые принципы и паттерны конкурентного программирования.

Следите за тем, чтобы конкурентности не было в ваших API

Конкурентность является одной из деталей реализации, которые должны быть по возможности скрыты в API. Это позволяет вносить изменения в работу кода, не изменяя то, как он вызывается.

На практике это означает, что при указании типов, функций и методов API никогда не следует экспонировать каналы или мьютексы (о которых мы поговорим в разделе «Когда вместо каналов следует использовать мьютексы» на с. 279). Если вы экспонируете каналы, то тем самым возлагаете на пользователей вашего API ответственность за управление каналами. Это значит, что пользователям придется беспокоиться о том, является ли канал буферизованным, закрытым или равным `nil`. Они также могут вызвать взаимоблокировку, попытавшись выполнить доступ к каналам или мьютексам в неожиданном порядке.



Сказанное выше не означает, что вы не должны использовать каналы в качестве параметров функции или полей структуры. Это означает, что они не должны быть экспортируемыми.

У этого правила есть исключения. Каналы могут входить в состав API, если речь идет о библиотеке, содержащей вспомогательную функцию для работы с конкурентностью (такую как функция `time.After`, с которой вы познакомитесь в подразделе «Как можно установить тайм-аут для кода» на с. 271).

Горутины, циклы `for` и изменяющиеся переменные

Обычно замыкание, используемое для запуска горутины, не принимает параметры, а извлекает значения из той среды, в которой оно было объявлено. Однако такой подход не будет работать, в частности, в том случае, когда горутина попытается получить индекс или значение цикла `for`. Так, следующий код содержит трудноуловимую ошибку:

```
func main() {
    a := []int{2, 4, 6, 8, 10}
    ch := make(chan int, len(a))
    for _, v := range a {
        go func() {
            ch <- v * 2
        }()
    }
    for i := 0; i < len(a); i++ {
        fmt.Println(<-ch)
    }
}
```

Мы запускаем здесь по одной горутине для каждого значения в срезе `a`. По идее, каждой горутине должно передаваться разное значение, но, запустив этот код, мы увидим совсем другой результат:

```
20
20
20
20
20
```

Все горютины записали в канал `ch` значение `20` по той причине, что замыкание каждой горютины получило одно и то же значение. В цикле `for` переменные для индекса и значения повторно используются на каждой итерации. На последней итерации переменной `v` присваивается значение `10`, и именно его видят горютины в момент их запуска. Эта проблема свойственна не только циклам `for`: каждый раз, когда горютина использует переменную, значение которой может изменяться, это значение следует передавать горутине в качестве параметра. Это можно сделать двумя способами. Первый способ сводится к тому, чтобы затенить переменную внутри цикла:

```
for _, v := range a {
    v := v
    go func() {
        ch <- v * 2
    }()
}
```

Если же вы хотите обойтись без затенения и сделать движение данных более очевидным, воспользуйтесь альтернативным способом, при котором значение передается горутине в качестве параметра:

```
for _, v := range a {
    go func(val int) {
        ch <- val * 2
    }(v)
}
```



Всякий раз, когда горютина использует переменную, значение которой может изменяться, передавайте горутине текущее значение этой переменной в качестве параметра.

Всегда закрывайте горютины

При запуске каждой горютины необходимо убедиться в том, что рано или поздно она будет закрыта. В отличие от переменных в случае горютин среда выполне-

ния языка Go не может проверить, будет ли еще использоваться в программе та или иная горутин. Если горутин не будет закрыта, планировщик продолжит периодически предоставлять ей время, которое она не будет использовать, что негативно скажется на производительности программы. Эта проблема называется *утечкой горутин*.

То, что закрытие горутин не является гарантированным, не всегда очевидно. Например, допустим, что вы используете горутин в качестве генератора:

```
func countTo(max int) <-chan int {
    ch := make(chan int)
    go func() {
        for i := 0; i < max; i++ {
            ch <- i
        }
        close(ch)
    }()
    return ch
}

func main() {
    for i := range countTo(10) {
        fmt.Println(i)
    }
}
```



Учтите, что это лишь демонстрационный пример: в реальном коде не стоит использовать горутин для генерирования списка чисел. Это слишком простая операция, в которой использование горутин идет вразрез с одним из принципов, изложенных в разделе «Когда следует использовать конкурентность» на с. 251.

В общем случае выход из горутин следует производить после того, как будут использованы все значения. Однако если мы выйдем из цикла слишком рано, то горутин заблокируется и будет бесконечно ждать, когда из канала будет считано еще одно значение:

```
func main() {
    for i := range countTo(10) {
        if i > 5 {
            break
        }
        fmt.Println(i)
    }
}
```

Паттерн на основе канала `done`

Паттерн на основе канала `done` позволяет уведомить горутину о том, что пора прекратить обработку. Этот сигнал о необходимости выхода из горутин передается посредством канала. В качестве примера рассмотрим случай, когда нам нужно передать одни и те же данные нескольким функциям и получить результат только от самой быстрой функции:

```
func searchData(s string, searchers []func(string) []string) []string {
    done := make(chan struct{})
    result := make(chan []string)
    for _, searcher := range searchers {
        go func(searcher func(string) []string) {
            select {
            case result <- searcher(s):
            case <-done:
            }
        }(searcher)
    }
    r := <-result
    close(done)
    return r
}
```

В этой функции мы объявляем канал с именем `done`, который содержит данные типа `struct{}`. В качестве типа канала используется пустая структура, поскольку это значение не играет большой роли; мы не будем производить запись в этот канал, а только закроем его. Мы запускаем по одной горутине для каждой переданной нам функции поиска. Операторы `select` внутри этих рабочих горутин ожидают того момента, когда будет выполнена либо операция записи в канал `result` (при возвращении значения функцией `searcher`), либо операция чтения из канала `done`. Как вы помните, чтение из открытого канала приостанавливается до появления в нем данных, а чтение из закрытого канала всегда возвращает нулевое значение для используемого каналом типа. Это означает, что выполнение ветви, отвечающей за чтение из канала `done`, будет приостановлено до закрытия этого канала. Внутри функции `searchData` мы считываем первое значение, записанное в канал `result`, а затем закрываем канал `done`. Тем самым мы сообщаем горутинам, что они должны закрыться, и исключаем возможность утечки горутин.

Иногда закрытие горутин требуется выполнять в зависимости от результата, выдаваемого одной из предыдущих функций в стеке вызовов. В разделе «Отмена» на с. 314 будет показано, как можно уведомлять одну или несколько горутин о том, что пришла пора закрыться, используя для этого контекст.

Прекращение выполнения горутины с помощью функции отмены

Паттерн на основе канала `done` также можно использовать для реализации паттерна, с которым мы познакомились в главе 5: возвращение функции отмены вместе с каналом. Вернемся к представленному ранее примеру с функцией `countTo` и посмотрим, как это выглядит на практике. Функцию отмены нужно вызвать после выполнения цикла `for`:

```
func countTo(max int) (<-chan int, func()) {
    ch := make(chan int)
    done := make(chan struct{})
    cancel := func() {
        close(done)
    }
    go func() {
        for i := 0; i < max; i++ {
            select {
            case <-done:
                return
            default:
                ch <- i
            }
        }
        close(ch)
    }()
    return ch, cancel
}

func main() {
    ch, cancel := countTo(10)
    for i := range ch {
        if i > 5 {
            break
        }
        fmt.Println(i)
    }
    cancel()
}
```

Функция `countTo` создает два канала: канал для возвращения данных и сигнальный канал `done`. Вместо того чтобы возвращать канал `done` напрямую, мы создаем замыкание, которое закрывает канал `done`, и возвращаем это замыкание. Такая отмена с помощью замыкания позволяет нам выполнять дополнительную работу по «уборке» в тех случаях, когда это необходимо.

Когда следует использовать буферизованные и небуферизованные каналы

Один из наиболее сложных в освоении навыков в случае работы с конкурентностью в Go — умение правильно принять решение об использовании буферизованного канала. По умолчанию в Go используются небуферизованные каналы, принцип действия которых достаточно прост: одна горутина производит запись и ожидает, пока другая горутина «подхватит» результат ее работы, подобно эстафетной палочке. Разобраться с буферизованными каналами уже гораздо труднее. Вы должны выбрать размер буфера, поскольку буферизованный канал не может иметь буфер неограниченного размера. Для правильного использования буферизованного канала также нужно предусмотреть обработку того случая, когда буфер заполняется и записывающая горутина блокируется до тех пор, пока считывающая горутина не выполнит чтение. Так к чему же сводится правильное использование буферизованного канала?

Хотя этому сложно дать точное определение, в одном предложении это можно выразить так: буферизованные каналы следует использовать в том случае, когда вы знаете количество запущенных горутин и хотите ограничить количество горутин, которые еще будут запущены, или хотите ограничить объем работы, стоящей в очереди на выполнение.

Буферизованные каналы отлично работают, когда нужно либо собрать данные из некоторого набора запущенных горутин, либо ограничить конкурентное использование. Их также можно использовать для управления объемом работы, поставленной системой в очередь на выполнение, чтобы не допустить снижения производительности и перегрузки ваших сервисов. Пара примеров использования буферизованных каналов представлена ниже.

В первом примере производится обработка первых десяти результатов канала. Для этого мы запускаем десять горутин, каждая из которых записывает свой результат в буферизованный канал:

```
func processChannel(ch chan int) []int {
    const conc = 10
    results := make(chan int, conc)
    for i := 0; i < conc; i++ {
        go func() {
            v := <- ch
            results <- process(v)
        }()
    }
    var out []int
    for i := 0; i < conc; i++ {
        out = append(out, <-results)
    }
}
```

```
    return out
}
```

Мы точно знаем количество запущенных горутин, и нам нужно, чтобы каждая горутина закрылась после завершения своей работы. Это значит, что мы можем создать буферизованный канал, содержащий по одной ячейке для каждой запущенной горутин, и позволить каждой горутине записать свои данные в этот канал без блокировки. После этого мы можем обойти в цикле буферизованный канал и считать значения в том же порядке, в каком они были записаны. Считав из канала все значения, мы возвращаем результаты, зная, что у нас нет какой-либо утечки горутин.

Противодавление

С помощью буферизованного канала также можно реализовать такой прием, как *противодавление*. Хотя это не столь очевидно, система работает в целом лучше, когда ее компоненты ограничивают объем выполняемой ими работы. Мы можем ограничить количество одновременных запросов в системе, используя буферизованный канал и оператор `select`:

```
type PressureGauge struct {
    ch chan struct{}
}

func New(limit int) *PressureGauge {
    ch := make(chan struct{}, limit)
    for i := 0; i < limit; i++ {
        ch <- struct{}{}
    }
    return &PressureGauge{
        ch: ch,
    }
}

func (pg *PressureGauge) Process(f func()) error {
    select {
    case <-pg.ch:
        f()
        pg.ch <- struct{}{}
        return nil
    default:
        return errors.New("no more capacity")
    }
}
```

В этом примере кода мы создаем структуру, содержащую буферизованный канал с несколькими токенами и выполняемую функцию. Каждый раз, когда

горутине требуется использовать функцию, она вызывает функцию `Process`. Оператор `select` пытается считать токен из канала. Если это возможно, то выполняется функция и токен возвращается. Если он не может считать токен, то выполняется ветвь `default` и вместо токена возвращается ошибка. Следующий небольшой пример показывает, как этот код можно использовать в сочетании со встроенным HTTP-сервером (об использовании которого будет подробно рассказано в подразделе «Сервер» на с. 304):

```
func doThingThatShouldBeLimited() string {
    time.Sleep(2 * time.Second)
    return "done"
}

func main() {
    pg := New(10)
    http.HandleFunc("/request", func(w http.ResponseWriter, r *http.Request) {
        err := pg.Process(func() {
            w.Write([]byte(doThingThatShouldBeLimited()))
        })
        if err != nil {
            w.WriteHeader(http.StatusTooManyRequests)
            w.Write([]byte("Too many requests"))
        }
    })
    http.ListenAndServe(":8080", nil)
}
```

Отключение ветвей оператора `select`

Если вам нужно объединить данные, получаемые из нескольких параллельных источников, то с этой задачей прекрасно справляется оператор `select`. Однако при этом нужно проследить за тем, чтобы надлежащим образом обрабатывались закрытые каналы. Если какая-либо из ветвей оператора `select` будет выполнять чтение из закрытого канала, то она всегда будет успешно выполняться, возвращая нулевое значение. При каждом выборе такой ветви должны осуществляться проверка корректности значения и выход из ветви. В противном случае периодическое выполнение таких операций чтения может привести к большим затратам времени на чтение ненужных значений.

В такой ситуации нам может помочь то, что выглядит как ошибка, а именно чтение из канала, равного `nil`. Как мы уже видели ранее, чтение или запись в канал, равный `nil`, вызывает бесконечное зависание. В этом нет ничего хорошего, когда это происходит из-за программной ошибки, однако вы можете использовать канал, равный `nil`, для отключения ветвей `case` в операторе `select`. Обнаружив, что канал уже закрыт, присвойте переменной канала значение `nil`. После этого

связанная с этим каналом ветвь уже не будет выполняться, поскольку чтение из канала, равного `nil`, никогда не возвращает значение:

```
// in и in2 — это каналы для данных, done — это канал done.
for {
    select {
        case v, ok := <-in:
            if !ok {
                in = nil // Эта ветвь больше не будет успешно выполняться!
                continue
            }
            // Обработка значения переменной v, считанного из канала in
        case v, ok := <-in2:
            if !ok {
                in2 = nil // Эта ветвь больше не будет успешно выполняться!
                continue
            }
            // Обработка значения переменной v, считанного из канала in2
        case <-done:
            return
    }
}
```

Как можно установить тайм-аут для кода

В большинстве случаев интерактивные программы должны возвращать ответ в течение определенного промежутка времени. Помимо прочего, конкурентность в Go позволяет нам управлять количеством времени, отводимым для выполнения запроса (или определенной части запроса). В то время как в других языках эта функциональность реализуется путем введения дополнительных элементов языка или дается обещание реализовать ее в перспективе, идиома тайм-аута языка Go показывает, как можно создавать сложные функциональные возможности путем комбинации существующих элементов языка. Вот как это выглядит:

```
func timeLimit() (int, error) {
    var result int
    var err error
    done := make(chan struct{})
    go func() {
        result, err = doSomeWork()
        close(done)
    }()
    select {
        case <-done:
            return result, err
        case <-time.After(2 * time.Second):
            return 0, errors.New("work timed out")
    }
}
```

Всякий раз, когда в Go требуется ограничить длительность выполнения операции, используется некоторая вариация этого паттерна. Оператор `select` здесь производит выбор между двумя ветвями `case`. В первой ветви `case` используется уже знакомый нам паттерн на основе канала `done`. В замыкании горутины мы присваиваем значения переменным `result` и `err` и закрываем канал `done`. Если канал `done` успевает закрыться в отведенное время, то чтение из канала `done` выполняется успешно и значения возвращаются.

Второй канал возвращается функцией `After` из пакета `time`. Он содержит значение, которое записывается в него по истечении промежутка времени, заданного как значение типа `time.Duration`. (Подробнее о пакете `time` будет рассказано в разделе «Пакет `time`» на с. 292.) Если значение этого канала будет считано до того, как функция `doSomeWork` завершит свою работу, функция `timeLimit` возвратит ошибку превышения тайм-аута.



Если выход из функции `timeLimit` будет выполнен до того, как горутина завершит обработку, то горутина продолжит свою работу. Мы просто ничего не будем делать с тем результатом, который она в итоге возвратит. Если вам нужно, чтобы горутина прекращала свою работу, когда вам уже не нужно ждать ее завершения, используйте контекстную отмену, о которой мы поговорим в разделе «Отмена» на с. 314.

Использование типа `WaitGroup`

Иногда требуется, чтобы одна горутина ждала, пока завершат свою работу несколько других горутин. Когда вы ждете завершения одной горутины, можно использовать описанный ранее паттерн на основе канала `done`. Но когда вы ждете завершения нескольких горутин, необходимо использовать тип `WaitGroup` из пакета `sync` стандартной библиотеки. Рассмотрим следующий простой пример, который вы можете запустить в онлайн-песочнице (<https://oreil.ly/hg7IF>):

```
func main() {
    var wg sync.WaitGroup
    wg.Add(3)
    go func() {
        defer wg.Done()
        doThing1()
    }()
    go func() {
        defer wg.Done()
        doThing2()
    }()
    go func() {
        defer wg.Done()
        doThing3()
    }()
}
```



```
    wg.Wait()  
}
```

Тип `sync.WaitGroup` не нужно инициализировать: достаточно лишь объявить его, поскольку мы используем его нулевое значение. У типа `sync.WaitGroup` есть три метода: метод `Add`, который инкрементирует счетчик горутин, ожидающих завершения, метод `Done`, который декрементирует счетчик и вызывается горутинкой в момент завершения ее работы, и метод `Wait`, который приостанавливает выполнение своей горутинки до тех пор, пока счетчик не станет равным нулю. Метод `Add` обычно вызывается только один раз, при этом ему передается количество запускаемых горутин. Метод `Done` вызывается внутри горутинки. Чтобы он вызывался даже в том случае, когда горутинка выдает панику, при его вызове мы используем оператор `defer`.

Возможно, вы заметили, что мы не передаем тип `sync.WaitGroup` явным образом. На это есть две причины. Первая причина состоит в том, что нам нужно, чтобы во всех тех местах, где используется тип `sync.WaitGroup`, применялся один и тот же его экземпляр. Если мы передадим тип `sync.WaitGroup` в функцию горутинки, не используя при этом указатель, то эта функция получит *копию* и метод `Done` не сможет декрементировать исходный экземпляр типа `sync.WaitGroup`. Перехватывая тип `sync.WaitGroup` с помощью замыкания, мы гарантируем, что каждая горутинка будет использовать один и тот же его экземпляр.

Второй причиной являются принципы проектирования. Если вы помните, конкурентность рекомендуется исключать из API. Как мы уже видели в случае с каналами, стандартный подход сводится к тому, чтобы запустить горутину с помощью замыкания, которое оборачивает бизнес-логику. Замыкание при этом управляет всем, что связано конкурентностью, а основная функция реализует алгоритм.

Теперь рассмотрим более реалистичный пример. Как уже упоминалось ранее, в том случае, когда несколько горутин производят запись в один и тот же канал, необходимо проследить за тем, чтобы этот канал закрывался только один раз, и тип `sync.WaitGroup` прекрасно справляется с этой задачей. Посмотрим, как его можно использовать в функции, которая производит конкурентную обработку значений канала, заносит результаты в один срез и возвращает этот срез:

```
func processAndGather(in <-chan int, processor func(int) int, num int) []int {  
    out := make(chan int, num)  
    var wg sync.WaitGroup  
    wg.Add(num)  
    for i := 0; i < num; i++ {  
        go func() {  
            defer wg.Done()  
            for v := range in {  
                out <- processor(v)  
            }  
        }()  
    }  
    wg.Wait()  
    return out
```

```

    }
    }()
}
go func() {
    wg.Wait()
    close(out)
}()
var result []int
for v := range out {
    result = append(result, v)
}
return result
}

```

В данном примере мы запускаем следящую горутину, которая дожидается завершения всех обрабатывающих горутин. После их завершения следящая горутина вызывает функцию `close` для выходного канала. Выход из цикла `for-range` по содержимому канала выполняется после того, как канал `out` закрывается и его буфер становится пустым. Наконец, наша функция возвращает обработанные значения.

Каким бы удобным ни был этот подход, тип `WaitGroup` не следует считать основным средством согласования горутин. Его следует использовать только в тех случаях, когда требуется произвести определенную «уборку» после закрытия всех рабочих горутин (например, закрыть канал, в который они записывали данные).

ПАКЕТЫ GOLANG.ORG/X И ТИП ERRGROUP

Разработчики языка Go поддерживают ряд утилит, дополняющих возможности стандартной библиотеки, которые собирательно называют пакетами `golang.org/x`. Помимо прочего, они содержат тип `ErrGroup`, созданный на основе типа `WaitGroup` для получения группы горутин, прекращающих обработку в том случае, когда одна из них возвращает ошибку. Более подробную информацию об этом типе можно найти в документации (https://oreil.ly/_EVsK).

Однократное выполнение кода

Как уже говорилось в подразделе «По возможности не используйте функцию `init`» на с. 231, для инициализации фактически неизменного состояния на уровне пакета следует использовать функцию `init`. Однако в некоторых случаях требуется произвести так называемую *отложенную загрузку*, то есть один раз вызвать некоторый код инициализации уже после запуска программы. Обычно

это объясняется тем, что инициализация выполняется сравнительно медленно или, возможно, требуется не при каждом запуске программы. В пакете `sync` есть удобный тип `Once`, который может предоставить эту функциональность. Как он работает, показывает следующий небольшой пример:

```
type SlowComplicatedParser interface {
    Parse(string) string
}

var parser SlowComplicatedParser
var once sync.Once

func Parse(dataToParse string) string {
    once.Do(func() {
        parser = initParser()
    })
    return parser.Parse(dataToParse)
}

func initParser() SlowComplicatedParser {
    // здесь выполняются различные операции настройки и загрузки
}
```

Мы объявляем здесь две переменные уровня пакета: переменную `parser`, которая относится к типу `SlowComplicatedParser`, и переменную `once`, относящуюся к типу `sync.Once`. Как и в случае типа `sync.WaitGroup`, нам не нужно настраивать экземпляр типа `sync.Once` (благодаря тому что мы *делаем полезным нулевое значение*). Еще одно сходство с типом `sync.WaitGroup` состоит в том, что мы должны проследить за тем, чтобы не создавалась копия экземпляра `sync.Once`, поскольку каждая копия обладает собственным состоянием, указывающим, использовалась она уже или нет. Объявление экземпляра типа `sync.Once` внутри функции обычно является плохой идеей, поскольку при каждом вызове функции будет создаваться новый экземпляр, который не будет помнить о предыдущих вызовах.

В данном примере нам нужно позаботиться о том, чтобы переменная `parser` была инициализирована только один раз. Для этого мы присваиваем значение переменной `parser` внутри замыкания, которое передается методу `Do` переменной `once`. Если функция `Parse` будет вызвана несколько раз, то метод `once.Do` не будет выполнять замыкание повторно.

Собираем инструменты для обеспечения конкурентности

Вернемся к примеру, который приводился в начале этой главы. Допустим, у нас есть функция, которая вызывает три веб-сервиса. Мы отправляем данные двум сервисам, получаем результаты этих двух вызовов и отправляем их третьему сервису, который возвращает окончательный результат. Весь этот процесс

должен занимать не более 50 миллисекунд; в противном случае возвращается сообщение об ошибке.

Сначала рассмотрим вызываемую нами функцию:

```
func GatherAndProcess(ctx context.Context, data Input) (COut, error) {
    ctx, cancel := context.WithTimeout(ctx, 50*time.Millisecond)
    defer cancel()
    p := processor{
        outA: make(chan AOut, 1),
        outB: make(chan BOut, 1),
        inC:  make(chan CIn, 1),
        outC: make(chan COut, 1),
        errs: make(chan error, 2),
    }
    p.launch(ctx, data)
    inputC, err := p.waitForAB(ctx)
    if err != nil {
        return COut{}, err
    }
    p.inC <- inputC
    out, err := p.waitForC(ctx)
    return out, err
}
```

Первое, что мы здесь делаем, это определяем контекст с тайм-аутом в 50 миллисекунд. В тех случаях, когда доступен контекст, используйте его поддержку таймера, вместо того чтобы вызывать функцию `time.After`. Одним из преимуществ использования таймера контекста является то, что это позволяет учитывать тайм-аут, установленный функцией, вызвавшей текущую функцию. Мы поговорим о контексте в главе 12 и подробно обсудим тайм-ауты в разделе «Таймеры» на с. 318. Пока вам достаточно просто знать, что по истечении тайм-аута производится отмена контекста. Вызов метода `Done` в контексте возвращает канал, выводящий значение при отмене контекста, выполняемой либо путем установки тайм-аута, либо с помощью явного вызова метода отмены.

После создания контекста мы используем оператор `defer`, чтобы обеспечить гарантированный вызов функции контекста `cancel`. Как будет рассказано подробнее в разделе «Отмена» на с. 314, эту функцию необходимо вызывать во избежание утечки ресурсов.

После этого мы заполняем экземпляр структуры `processor` набором каналов, которые будем использовать для обмена данными с горутинами. Здесь используются буферизованные каналы, чтобы после записи в них горутин могли закрываться, не дожидаясь выполнения чтения. (У канала `errs` размер буфера равен двум, поскольку в него потенциально могут быть записаны две ошибки.)

Структура `processor` выглядит следующим образом:

```
type processor struct {
    outA chan AOut
    outB chan BOut
    outC chan COut
    inC  chan CIn
    errs chan error
}
```

Затем мы вызываем метод `launch` в структуре `processor`, чтобы запустить три горутин, которые производят вызов функций `getResultA`, `getResultB` и `getResultC`:

```
func (p *processor) launch(ctx context.Context, data Input) {
    go func() {
        aOut, err := getResultA(ctx, data.A)
        if err != nil {
            p.errs <- err
            return
        }
        p.outA <- aOut
    }()
    go func() {
        bOut, err := getResultB(ctx, data.B)
        if err != nil {
            p.errs <- err
            return
        }
        p.outB <- bOut
    }()
    go func() {
        select {
        case <-ctx.Done():
            return
        case inputC := <-p.inC:
            cOut, err := getResultC(ctx, inputC)
            if err != nil {
                p.errs <- err
                return
            }
            p.outC <- cOut
        }
    }()
}
```

Горутин для вызова функций `getResultA` и `getResultB` не представляют собой ничего сложного. Они просто вызывают соответствующие функции. Если функция возвращает ошибку, они записывают эту ошибку в канал `p.errs`. Если функция возвращает корректное значение, записывают его в соответствующий

канал (канал `p.outA` для функции `getResultA` и канал `p.outB` для функции `getResultB`).

Поскольку функция `getResultC` вызывается лишь в том случае, когда функции `getResultA` и `getResultB` успешно выполняются в течение 50 миллисекунд, третья горютина выглядит немного сложнее. Она содержит оператор `select` с двумя ветвями. Первая ветвь срабатывает в случае отмены контекста. Вторая ветвь срабатывает при наличии данных, необходимых для вызова функции `getResultC`. Если данные доступны, вызывается эта функция, и здесь работает та же логика, что и в первых двух горютинах.

После запуска этих горютин мы вызываем метод `waitForAB` в структуре `processor`:

```
func (p *processor) waitForAB(ctx context.Context) (CIn, error) {
    var inputC CIn
    count := 0
    for count < 2 {
        select {
            case a := <-p.outA:
                inputC.A = a
                count++
            case b := <-p.outB:
                inputC.B = b
                count++
            case err := <-p.errs:
                return CIn{}, err
            case <-ctx.Done():
                return CIn{}, ctx.Err()
        }
    }
    return inputC, nil
}
```

Здесь мы используем цикл `for-select` для заполнения экземпляра `inputC` типа `CIn`, передаваемого в качестве входного параметра функции `getResultC`. Оператор `select` содержит четыре ветви. Первые две ветви производят чтение из каналов, в которые делают запись наши первые две горютины, и заполняют поля экземпляра `inputC`. Если выполняются обе эти ветви, мы выходим из цикла `for-select` и возвращаем значение экземпляра `inputC` и ошибку, равную `nil`.

Следующий две ветви служит для обработки состояния ошибки. Если была записана ошибка в канал `p.errs`, то мы возвращаем эту ошибку. В случае отмены контекста мы возвращаем ошибку, которая сообщает об отмене запроса.

Далее в функции `GatherAndProcess` мы выполняем стандартную проверку ошибки на равенство значению `nil`. Если все в порядке, то мы записываем значение переменной `inputC` в канал `p.inC` и вызываем метод `waitForC` в структуре `processor`:

```
func (p *processor) waitForC(ctx context.Context) (COut, error) {
    select {
    case out := <-p.outC:
        return out, nil
    case err := <-p.errs:
        return COut{}, err
    case <-ctx.Done():
        return COut{}, ctx.Err()
    }
}
```

Этот метод содержит только один оператор `select`. В случае успешного выполнения функции `getResultC` мы считываем ее результат из канала `p.outC` и возвращаем его. Если функция `getResultC` возвратила ошибку, мы считываем эту ошибку из канала `p.errs` и возвращаем ее. Наконец, при отмене контекста мы возвращаем ошибку, которая сообщает об этом. После выполнения метода `waitForC` функция `GatherAndProcess` возвращает результат вызывающей стороне.

Этот код можно упростить, если вы уверены в том, что разработчик функции `getResultC` все сделал как надо. Поскольку мы передаем контекст в функцию `getResultC`, эту функцию можно определить так, чтобы она учитывала тайм-аут и возвращала ошибку в случае его истечения. Тогда функцию `getResultC` можно будет вызвать непосредственно из функции `GatherAndProcess`. Это позволит нам убрать каналы `inC` и `outC` из структуры `processor`, одну горутину из метода `launch` и весь метод `waitForC`. Общий принцип сводится к тому, чтобы использовать конкурентность только в том объеме, который является минимально необходимым для получения корректно работающей программы.

Структурируя код с помощью горутин, каналов и операторов `select`, мы выделяем отдельные шаги, позволяем независимым частям программы выполняться в любом порядке и обеспечиваем четко выраженный обмен данными между зависимыми частями программы. Это также позволяет нам исключить вероятность зависания какой-либо части программы и обеспечить надлежащую обработку тайм-аутов, устанавливаемых и внутри текущей функции, и внутри предыдущих функций в стеке вызовов. Если вы еще не уверены в том, что это лучший подход к реализации конкурентности, попробуйте реализовать конкурентность, используя какой-либо другой язык. Вы будете удивлены тому, насколько это сложно.

Когда вместо каналов следует использовать мьютексы

Если вам приходилось координировать доступ к данным в рамках нескольких потоков в других языках программирования, то вы уже, наверное, сталкивались

с такой вещью, как *мьютекс* (mutex, от mutual exclusion — «взаимное исключение»). Мьютекс накладывает ограничение на конкурентное выполнение определенного кода или на доступ к совместно используемым данным. Эта защищаемая часть программы называется *критической секцией*.

У разработчиков языка Go были веские основания для того, чтобы вместо мьютексов использовать для управления конкурентностью каналы и оператор `select`. Главной проблемой мьютексов является то, что они затрудняют понимание движения данных внутри программы. Когда значение передается из одной горутины в другую посредством ряда каналов, вполне очевидно, как движутся данные. Доступ к значению всегда выполняется только в одной горутине. Но при использовании мьютекса для защиты значения невозможно определить, какой процесс производит доступ к значению в данный момент, поскольку его совместно используют все конкурентные процессы. Это усложняет понимание порядка выполнения обработки. Эту философию отражает существующая в Go-сообществе поговорка: «Делитесь памятью путем общения, а не общайтесь, делаясь памятью».

В то же время иногда более понятный код можно получить, используя мьютекс, и для этого в стандартную библиотеку языка Go была включена поддержка мьютексов. Наиболее распространенным примером такой ситуации является тот случай, когда горутины считывают или записывают совместно используемое значение, но не обрабатывают его. В качестве примера рассмотрим размещаемую в памяти таблицу результатов многопользовательской игры. Сначала посмотрим, как это можно реализовать с помощью каналов. Вот как при этом будет выглядеть запускаемая в качестве горутины функция для управления таблицей результатов:

```
func scoreboardManager(in <-chan func(map[string]int), done <-chan struct{}) {
    scoreboard := map[string]int{}
    for {
        select {
        case <-done:
            return
        case f := <-in:
            f(scoreboard)
        }
    }
}
```

Эта функция объявляет карту, а затем прослушивает два канала, в одном из которых передается функция для чтения или модификации карты, а во втором — сигнал о завершении работы. Создадим тип с методом для записи значения в карту:

```
type ChannelScoreboardManager chan func(map[string]int)
```



```
func NewChannelScoreboardManager() (ChannelScoreboardManager, func()) {
    ch := make(ChannelScoreboardManager)
    done := make(chan struct{})
    go scoreboardManager(ch, done)
    return ch, func() {
        close(done)
    }
}

func (csm ChannelScoreboardManager) Update(name string, val int) {
    csm <- func(m map[string]int) {
        m[name] = val
    }
}
```

Метод `Update` очень прост: он передает функцию для записи значения в карту. Но что насчет чтения из таблицы результатов? В этом случае нужно возвращать значение. Это означает, что мы должны использовать паттерн на основе канала `done`, чтобы сообщать о необходимости завершить работу после возвращения функции в функцию `scoreboardManager`:

```
func (csm ChannelScoreboardManager) Read(name string) (int, bool) {
    var out int
    var ok bool
    done := make(chan struct{})
    csm <- func(m map[string]int) {
        out, ok = m[name]
        close(done)
    }
    <-done
    return out, ok
}
```

Этот код работает, но он слишком громоздкий и не позволяет одновременно использовать несколько считывателей. В таком случае лучше использовать мьютекс. Стандартная библиотека предлагает две реализации мьютекса, и они обе включены в пакет `sync`. Первой реализацией является тип `Mutex`, который обладает методами `Lock` и `Unlock`. Вызов метода `Lock` приостанавливает выполнение текущей горутин, если критическая секция в данный момент занята другой горутин. Если критическая секция свободна, текущая горутина устанавливает блокировку и выполняет код в критической секции. Вызов метода `Unlock` в экземпляре типа `Mutex` завершает использование критической секции.

Второй реализацией мьютекса является тип `RWMutex`, который позволяет устанавливать блокировки и на чтение, и на запись. При этом критическую секцию не могут одновременно использовать несколько записывателей, однако блокировки на чтение могут использоваться совместно, что делает возможным одновременный доступ к критической секции нескольких считывателей.

С помощью методов `Lock` и `Unlock` осуществляется управление блокировкой на запись, а с помощью методов `RLock` и `RUnlock` — управление блокировкой на чтение.

Всякий раз, когда вы устанавливаете блокировку мьютекса, нужно позаботиться о том, чтобы эта блокировка высвобождалась. Используйте оператор `defer` для вызова метода `Unlock` непосредственно после вызова метода `Lock` или `RLock`:

```
type MutexScoreboardManager struct {
    l      sync.RWMutex
    scoreboard map[string]int
}

func NewMutexScoreboardManager() *MutexScoreboardManager {
    return &MutexScoreboardManager{
        scoreboard: map[string]int{},
    }
}

func (msm *MutexScoreboardManager) Update(name string, val int) {
    msm.l.Lock()
    defer msm.l.Unlock()
    msm.scoreboard[name] = val
}

func (msm *MutexScoreboardManager) Read(name string) (int, bool) {
    msm.l.RLock()
    defer msm.l.RUnlock()
    val, ok := msm.scoreboard[name]
    return val, ok
}
```

Теперь вы уже знаете, как может выглядеть реализация с использованием мьютексов, но перед тем, как их использовать, всегда нужно тщательно рассматривать возможные альтернативы. Упростить для вас выбор между каналами и мьютексами может схема принятия этого решения, изложенная в замечательной книге Кэтрин Кокс-Будай (Katherine Cox-Buday) «Конкурентность в Go» (<https://oreil.ly/G7bpu>).

- Если вам нужно координировать горутин или отслеживать значение по мере его преобразования с помощью нескольких горутин, используйте каналы.
- Если вам нужно обеспечить совместный доступ к полю структуры, используйте мьютексы.
- Если вы выявили критическую проблему производительности при использовании каналов (о том, как производится оценка производительности, будет рассказано в разделе «Сравнительные тесты» на с. 343) и не можете

найти других способов решения этой проблемы, используйте вместо каналов мьютексы.

Использование мьютексов вполне уместно в рассмотренном выше примере, поскольку таблица результатов представляет собой поле структуры и не подвергается каким-либо преобразованиям. Мьютексы хорошо подходят здесь только лишь потому, что данные размещаются в памяти. Когда данные размещаются в таких внешних хранилищах, как HTTP-сервер или база данных, не используйте мьютексы для защиты доступа к системе.

Для обеспечения работы мьютексов приходится выполнять достаточно большой объем работы. Так, например, во избежание зависания программы каждой операции установки блокировки нужно поставить в соответствие операцию снятия блокировки. В нашем примере и установка, и снятие блокировки выполняются внутри одного и того же метода. Еще одна проблема заключается в том, что мьютексы в Go не являются *повторно входимыми*. Если горутина попытается установить одну и ту же блокировку дважды, это приведет к зависанию из-за того, что она будет ждать высвобождения установленной ею блокировки. В этом Go отличается от таких языков, как Java, в которых блокировки являются повторно входимыми.

SYNC.MAP — ЭТО НЕ ТА КАРТА, КОТОРАЯ ВАМ НУЖНА

Просматривая содержимое пакета `sync`, вы можете обнаружить тип с именем `Map`. Это совместимая с конкурентностью версия встроенного типа `map`. Из-за особенностей его реализации использование типа `sync.Map` будет уместным лишь в следующих случаях:

- когда нужно обеспечить совместное использование карты, при котором пары «ключ — значение» заносятся в карту один раз и считываются многократно;
- когда несколько горутин совместно используют карту, но не обращаются к ключам и значениям друг друга.

Кроме того, поскольку в Go пока нет обобщенных типов, тип `sync.Map` использует тип `interface{}` в качестве типа ключей и значений, что не позволяет компилятору проследить за тем, чтобы использовались надлежащие типы данных.

По причине этих ограничений в тех редких случаях, когда требуется обеспечить совместное использование карты несколькими горутинами, лучше применяйте встроенный тип `map`, защищенный с помощью типа `sync.RWMutex`.

Тот факт, что блокировки в Go не являются повторно входимыми, затрудняет установку блокировки внутри функции, вызывающей себя рекурсивным образом. В таком случае следует снять блокировку до вызова рекурсивной функции. В целом следует по возможности не вызывать функции при установленной блокировке, поскольку неизвестно, какие блокировки будут в них установлены. Если ваша функция вызовет другую функцию, которая попытается установить ту же блокировку мьютекса, это приведет к зависанию горутин.

Как и типы `sync.WaitGroup` и `sync.Once`, мьютексы никогда не следует копировать. Если мьютекс передается в функцию или используется как поле структуры, это следует делать посредством указателя. Если мьютекс будет скопирован, вы не сможете обеспечить совместное использование его блокировки.



Никогда не пытайтесь производить доступ к переменной из нескольких горутин, не установив предварительно мьютекс для этой переменной. Это может привести к периодическому возникновению трудно отслеживаемых ошибок. О том, как можно выявлять такие проблемы, будет рассказано в разделе «Выявление проблем конкурентности с помощью детектора состояний гонки» на с. 358.

Атомарные операции — скорее всего, они вам не понадобятся

Наряду с мьютексами, Go предлагает вам альтернативный способ обеспечения согласованности данных в рамках нескольких потоков. Пакет `sync/atomic` позволяет вам использовать встроенные в современные процессоры наборы операций для *атомарных переменных* — операций сложения, обмена, загрузки, сохранения и сравнения с обменом для значений, которые могут поместиться в одном регистре.

Если вам нужно выжать всю возможную производительность и вы являетесь экспертом по написанию конкурентного кода, то вас порадует наличие в Go поддержки атомарных операций. Всем остальным читателям я советую использовать для реализации необходимой конкурентности только горутин и мьютексы.

Где можно найти более подробную информацию о конкурентности

Мы рассмотрели здесь несколько простых паттернов конкурентного программирования, но существует и много других. На самом деле о том, как следует подходить к реализации различных паттернов конкурентного программирования

в Go, можно написать целую книгу, и, к счастью, Кэтрин Кокс-Будай уже это сделала. Когда мы говорили о выборе между мьютексами или каналами, я уже упоминал о ее книге «Конкурентность в Go», в которой превосходно освещается все, что имеет отношение к обеспечению конкурентности в Go. Если вы хотите получить более подробную информацию, обратитесь к этой книге.

Резюме

В этой главе мы рассмотрели конкурентность и узнали, почему используемый в Go подход проще по сравнению с более традиционными способами реализации конкурентности. Мы также разобрались с вопросом о том, когда следует использовать конкурентность, и изучили ряд принципов и паттернов реализации конкурентности. В следующей главе мы проведем общий обзор стандартной библиотеки языка Go, которая предоставляет вам «полный комплект» современных средств программирования.

Стандартная библиотека

Одной из самых приятных особенностей разработки на языке Go является наличие обширной стандартной библиотеки. Подобно языку Python, Go стремится предоставить «полный комплект» средств, необходимых вам для создания приложения. Поскольку это сравнительно новый язык, включенная в его «комплект поставки» библиотека создана с учетом проблем, характерных для современной среды программирования.

Мы не можем здесь рассмотреть абсолютно все пакеты стандартной библиотеки, и, к счастью, нам и не нужно этого делать, поскольку существует много отличных источников информации о стандартной библиотеке, начиная с официальной документации (<https://golang.org/pkg>). Вместо этого мы сосредоточимся на нескольких наиболее важных пакетах и посмотрим, как в их дизайне и способах использования находят свое выражение принципы написания идиоматического Go-кода. Некоторые пакеты библиотеки (`errors`, `sync`, `context`, `testing`, `reflect` и `unsafe`) рассматриваются в посвященных им отдельных главах. В этой главе мы рассмотрим встроенную в Go поддержку ввода-вывода, времени, формата JSON и протокола HTTP.

Пакет `io` и его друзья

Чтобы выполнять какую-то полезную работу, программа должна считывать и записывать данные. Основные принципы работы с вводом-выводом в Go нашли свое выражение в пакете `io`. В частности, в этом пакете определены два интерфейса, которые, пожалуй, занимают в Go второе и третье место по частоте использования: `io.Reader` и `io.Writer`.



Какой же интерфейс является лидером по частоте использования? Это интерфейс `fmt`, с которым мы уже познакомились в главе 8.

Интерфейсы `io.Reader` и `io.Writer` определяют по одному методу:

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

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

Метод `Write` интерфейса `io.Writer` принимает срез байтов, записываемых в реализацию интерфейса, и возвращает количество записываемых байтов и ошибку в случае возникновения проблем. Метод `Read` интерфейса `io.Reader` более интересен. Вместо возвращения данных посредством возвращаемого параметра, принимаемый срез передается в реализацию и модифицируется. В срез записывается до `len(p)` байтов. Возвращает метод `Read` количество записанных байтов. Это может показаться немного странным. Возможно, вы ожидали, что он будет выглядеть следующим образом:

```
type NotHowReaderIsDefined interface {
    Read() (p []byte, err error)
}
```

Для того чтобы интерфейс `io.Reader` был определен именно таким образом, есть достаточно веские основания. Чтобы понять это, напомним функцию, демонстрирующую типичный способ использования интерфейса `io.Reader`:

```
func countLetters(r io.Reader) (map[string]int, error) {
    buf := make([]byte, 2048)
    out := map[string]int{}
    for {
        n, err := r.Read(buf)
        for _, b := range buf[:n] {
            if (b >= 'A' && b <= 'Z') || (b >= 'a' && b <= 'z') {
                out[string(b)]++
            }
        }
        if err == io.EOF {
            return out, nil
        }
        if err != nil {
            return nil, err
        }
    }
}
```

Здесь следует отметить три момента. Прежде всего мы создаем буфер один раз и затем многократно его используем в каждом вызове метода `r.Read`. Это позволяет нам только один раз выделить память для чтения потенциально

большого объема данных. Если бы метод `Read` возвращал срез байтов `[]byte`, то нам потребовалось бы выделять память для каждого вызова. При этом каждый раз память выделялась бы в куче, что порождало бы большой объем работы для сборщика мусора.

Чтобы еще больше сократить количество операций выделения памяти, мы могли бы создавать пул буферов в момент запуска программы. После этого мы могли бы брать буфер из пула при запуске функции и возвращать его обратно после завершения ее работы. Передавая срез интерфейсу `io.Reader`, разработчик может держать под контролем выделение памяти.

Второй важный момент здесь состоит в том, что, используя возвращаемое методом `r.Read` значение `n`, мы узнаем, сколько байтов было записано в буфер, и производим обход части среза `buf`, чтобы обработать считанные данные.

Наконец, сигналом о завершении чтения из переменной `r` для нас служит возвращение методом `r.Read` ошибки `io.EOF`. Эта ошибка выделяется из общего ряда тем, что фактически не является ошибкой, а просто сообщает о том, что в экземпляре интерфейса `io.Reader` больше нет непрочитанных данных. При возвращении ошибки `io.EOF` мы прекращаем обработку и возвращаем результат.

Метод `Read` интерфейса `io.Reader` обладает необычной особенностью. Как правило, когда функция или метод возвращает ошибку, мы проверяем, чему она равна, и уже потом приступаем к обработке других возвращаемых значений. В случае метода `Read` мы поступаем наоборот, потому что ошибка может быть вызвана окончанием потока данных или некоторым неожиданным состоянием уже после возвращения байтов.



При неожиданном окончании данных в экземпляре интерфейса `io.Reader` возвращается другая сигнальная ошибка, `io.ErrUnexpectedEOF`. Обратите внимание, что ее имя начинается на `Err`, и это говорит о том, что мы столкнулись с неожиданным состоянием.

Интерфейсы `io.Reader` и `io.Writer` являются очень простыми, и поэтому существует множество разных способов их реализации. Интерфейс `io.Reader`, в частности, можно реализовать на основе строки, используя функцию `strings.NewReader`:

```
s := "The quick brown fox jumped over the lazy dog"
sr := strings.NewReader(s)
counts, err := countLetters(sr)
if err != nil {
    return err
}
fmt.Println(counts)
```


Как упоминалось в разделе «Интерфейсы обеспечивают типобезопасную утиную типизацию» на с. 179, реализации интерфейсов `io.Reader` и `io.Writer` часто состыковываются в цепочку с использованием паттерна «Декоратор». Поскольку функция `countLetters` зависит от интерфейса `io.Reader`, мы можем использовать в точности такую же функцию `countLetters` и для подсчета букв английского алфавита в сжатом файле формата `gzip`. Прежде всего напишем функцию, которая будет возвращать экземпляр типа `*gzip.Reader` для заданного имени файла:

```
func buildGZipReader(fileName string) (*gzip.Reader, func(), error) {
    r, err := os.Open(fileName)
    if err != nil {
        return nil, nil, err
    }
    gr, err := gzip.NewReader(r)
    if err != nil {
        return nil, nil, err
    }
    return gr, func() {
        gr.Close()
        r.Close()
    }, nil
}
```

Эта функция демонстрирует надлежащий способ обертывания типов, реализующих интерфейс `io.Reader`. Мы создаем экземпляр типа `*os.File` (который соответствует интерфейсу `io.Reader`) и, убедившись в его корректности, передаем его в функцию `gzip.NewReader`, которая возвращает экземпляр типа `*gzip.Reader`. Если возвращается корректный экземпляр типа `*gzip.Reader`, мы возвращаем его и закрывающее замыкание, которое обеспечивает надлежащее высвобождение ресурсов.

Поскольку тип `*gzip.Reader` реализует интерфейс `io.Reader`, мы можем использовать его в сочетании с функцией `countLetters`, подобно тому как ранее использовали тип `*strings.Reader`:

```
r, closer, err := buildGZipReader("my_data.txt.gz")
if err != nil {
    return err
}
defer closer()
counts, err := countLetters(r)
if err != nil {
    return err
}
fmt.Println(counts)
```

Поскольку мы располагаем стандартными интерфейсами для чтения и записи, в пакет `io` включена и стандартная функция для копирования из экземпляра

интерфейса `io.Reader` в экземпляр интерфейса `io.Writer`, `io.Copy`. В этом пакете есть и другие стандартные функции для расширения функциональности экземпляров интерфейсов `io.Reader` и `io.Writer`.

- `io.MultiReader` — функция возвращает экземпляр интерфейса `io.Reader`, выполняющий последовательное чтение нескольких экземпляров интерфейса `io.Reader`.
- `io.LimitReader` — эта функция возвращает экземпляр интерфейса `io.Reader`, считывающий не более указанного количества байтов из предоставленного экземпляра интерфейса `io.Reader`.
- `io.MultiWriter` — функция возвращает экземпляр интерфейса `io.Writer`, выполняющий запись сразу в несколько экземпляров интерфейса `io.Writer`.

Другие пакеты стандартной библиотеки предоставляют собственные типы и функции для работы с интерфейсами `io.Reader` и `io.Writer`. Мы уже рассмотрели некоторые из них, но существует и много других. Они предназначены для работы с алгоритмами сжатия, архивами, шифрованием, буферами, срезами байтов и строками.

В пакете `io` есть и другие интерфейсы с одним методом, в частности `io.Closer` и `io.Seeker`:

```
type Closer interface {
    Close() error
}

type Seeker interface {
    Seek(offset int64, whence int) (int64, error)
}
```

Интерфейс `io.Closer` реализуется такими типами, как `os.File`, которым нужно высвобождать ресурсы после завершения чтения или записи. Обычно метод `Close` вызывается с помощью оператора `defer`:

```
f, err := os.Open(fileName)
if err != nil {
    return nil, err
}
defer f.Close()
// использование экземпляра f
```



Не используйте оператор `defer`, когда ресурс открывается в цикле, поскольку он вызывается лишь в момент закрытия функции. Вместо этого вызывайте метод `Close` в конце каждой итерации цикла. В случае возникновения ошибок, ведущих к выходу из цикла, также следует вызывать метод `Close`.

Интерфейс `io.Seeker` используется для произвольного доступа к ресурсу. В качестве параметра `whence` могут использоваться константы `io.SeekStart`, `io.SeekCurrent` и `io.SeekEnd`. Это было бы более очевидно, если бы параметр `whence` относился к пользовательскому типу, но по странному недосмотру в дизайне он относится к типу `int`.

В пакете `io` также определены интерфейсы, позволяющие использовать различные комбинации этих четырех интерфейсов: `io.ReadCloser`, `io.ReadSeeker`, `io.ReadWriteCloser`, `io.ReadWriteSeeker`, `io.ReadWriter`, `io.WriteCloser` и `io.WriteSeeker`. С помощью этих интерфейсов вы можете указать, что ваши функции будут делать с данными. Например, вместо того, чтобы просто передавать в качестве параметра тип `os.File`, используйте интерфейсы, указывающие, что именно функция будет делать с параметром. Тем самым вы не только сделаете свои функции более универсальными, но и понятнее выразите свои намерения. Совместимость с этими интерфейсами следует обеспечить и в том случае, когда вы пишете собственные источники и приемники данных. При создании собственных интерфейсов старайтесь делать их столь же простыми и несвязанными, как интерфейсы пакета `io`, которые служат примером того, какие мощные возможности дают вам простые абстракции.

Пакет `ioutil` предоставляет ряд простых утилит для выполнения таких вещей, как считывание целиком реализаций интерфейса `io.Reader` в байтовые срезы, чтение и запись файлов и работа с временными файлами. Функции `ioutil.ReadAll`, `ioutil.ReadFile` и `ioutil.WriteFile` прекрасно подойдут для работы с небольшими источниками данных, но в случае более крупных источников данных все же лучше использовать типы `Reader`, `Writer` и `Scanner` из пакета `bufio`.

Одна из наиболее удачных функций пакета `ioutil` демонстрирует паттерн добавления метода в тип языка Go. Если у вас есть тип, который реализует интерфейс `io.Reader`, но не реализует интерфейс `io.Closer` (как, например, тип `strings.Reader`), и вам нужно передать его в функцию, которая ожидает экземпляр интерфейса `io.ReadCloser`, передайте свой экземпляр интерфейса `io.Reader` в функцию `ioutil.NopCloser`, чтобы получить тип, реализующий интерфейс `io.ReadCloser`. Если вы взглянете на реализацию этой функции, то увидите, что она очень простая:

```
type nopCloser struct {
    io.Reader
}

func (nopCloser) Close() error { return nil }

func NopCloser(r io.Reader) io.ReadCloser {
    return nopCloser{r}
}
```

Всякий раз, когда нужно снабдить определенный тип методами так, чтобы он соответствовал некоторому интерфейсу, используйте этот паттерн встроенного типа.



Функция `ioutil.NopCloser` нарушает общее правило, согласно которому вы не должны возвращать интерфейс из функции, но это простой адаптер интерфейса, который гарантированно останется неизменным, поскольку является составной частью стандартной библиотеки.

Пакет `time`

Как и в большинстве других языков, стандартная библиотека Go включает в себя поддержку работы со временем, которая предсказуемо находится в пакете `time`. Двумя основными типами для представления времени являются типы `time.Duration` и `time.Time`.

Для представления промежутка времени используется тип `time.Duration`, основанный на типе `int64`. Минимально возможный промежуток времени при этом равен одной наносекунде, но в пакете `time` также определены константы типа `time.Duration` для представления наносекунды, микросекунды, миллисекунды, секунды, минуты и часа. Например, 2 часа 30 минут можно представить следующим образом:

```
d := 2 * time.Hour + 30 * time.Minute // переменная d
                                     // относится к типу time.Duration
```

Эти константы делают использование типа `time.Duration` и более читабельным, и более типобезопасным, представляя собой пример надлежащего использования типизированных констант.

В Go определен продуманный строковый формат представления времени в виде последовательного ряда чисел, который можно преобразовать в значение типа `time.Duration` с помощью функции `time.ParseDuration`. Вот как он описывается в документации по стандартной библиотеке.

Строка длительности — это опционально снабжаемая знаком последовательность десятичных чисел с опциональной дробной частью, каждое из которых снабжается суффиксом, обозначающим единицы измерения, следующим образом: 300ms, -1.5h или 2h45m. Допустимыми единицами измерения являются ns, us (или μs), ms, s, m, h.

*Документация по стандартной библиотеке
языка Go (<https://oreil.ly/wmZdy>)*

Для типа `time.Duration` определены несколько методов. Он соответствует интерфейсу `fmt.Stringer` и возвращает форматированную строку длительности посредством метода `String`. У него также есть методы для получения значения в виде количества часов, минут, секунд, миллисекунд, микросекунд или наносекунд. Методы `Truncate` и `Round` позволяют отбросить дробную часть или округлить значение типа `time.Duration` до целого количества указанных единиц измерения, выраженных как значение типа `time.Duration`.

Для представления определенного момента времени используется тип `time.Time`, дополненный временной зоной. Получить ссылку на текущий момент времени можно с помощью функции `time.Now`. Эта функция возвращает экземпляр типа `time.Time`, содержащий значение текущего локального времени.



Тот факт, что экземпляр типа `time.Time` содержит значение временной зоны, означает, что вы не должны использовать оператор `==` для проверки двух экземпляров типа `time.Time` на предмет того, не ссылаются ли они на один и тот же момент времени. Вместо этого используйте метод `Equal`, который делает поправку с учетом временной зоны.

Функция `time.Parse` выполняет преобразование из типа `string` в тип `time.Time`, а метод `Format` — преобразование из типа `time.Time` в тип `string`. Хотя обычно язык Go заимствует подходы, хорошо зарекомендовавшие себя в прошлом, в данном случае он использует собственный метод форматирования даты и времени (https://oreil.ly/yfm_V). Суть этого метода сводится к тому, чтобы определять собственный формат, используя для этого следующее значение даты и времени: January 2, 2006, 3:04:05PM MST (Mountain Standard Time) — 2 января 2006 года, 3:04:05 после полудня по горному стандартному времени.



Почему используется именно это значение даты и времени? Это объясняется тем, что каждый его элемент представляет собой число в последовательном ряду от 1 до 7, то есть 01/02 03:04:05PM '06 -0700 (зона MST на семь часов опережает зону UTC).

Так, например, следующий код:

```
t, err := time.Parse("2006-02-01 15:04:05 -0700", "2016-13-03 00:00:00 +0000")
if err != nil {
    return err
}
fmt.Println(t.Format("January 2, 2006 at 3:04:05PM MST"))
```

выдаст такой результат:

March 13, 2016 at 12:00:00AM UTC

Хотя используемое для форматирования значение даты и времени, по идее, должно быть легко запоминаемым, мне никак не удастся его запомнить, и поэтому при каждом его использовании мне приходится уточнять, что именно оно собой представляет. К счастью, для наиболее часто используемых форматов даты и времени в пакете `time` определены специальные константы.

Как и у типа `time.Duration`, у типа `time.Time` есть методы для извлечения различных составляющих значения, включая `Day`, `Month`, `Year`, `Hour`, `Minute`, `Second`, `Weekday`, `Clock` (этот метод возвращает время, указанное в экземпляре типа `time.Time`, в виде набора значений типа `int`, представляющих количество часов, минут и секунд) и `Date` (этот метод возвращает набор значений типа `int`, представляющих год, месяц и день). Сравнить один экземпляр типа `time.Time` с другим можно с помощью методов `After`, `Before` и `Equal`.

Метод `Sub` возвращает экземпляр типа `time.Duration`, представляющий время, прошедшее между двумя моментами времени `time.Time`, метод `Add` возвращает момент времени `time.Time`, продвинутый вперед на длительность `time.Duration`, а метод `AddDate` — момент времени `time.Time`, продвинутый вперед на указанное количество лет, месяцев и дней. Как и у типа `time.Duration`, у типа `time.Time` также есть методы `Truncate` и `Round`. Все эти методы используют приемник значений и поэтому не изменяют экземпляр типа `time.Time`.

Монотонное время

В большинстве операционных систем отслеживаются две разновидности времени: *системное время*, представляющее собой текущее время, и *монотонное время*, представляющее собой время, прошедшее с момента запуска компьютера. Это делается по причине того, что продвижение системного времени вперед может происходить неравномерно. Переход на летнее время, ежегодно вводимые секунды координации и синхронизация с использованием сетевого протокола синхронизации времени могут приводить к неожиданному смещению системного времени вперед или назад. Это может вызвать проблемы при установке таймера или подсчете количества прошедшего времени.

Во избежание таких проблем Go использует монотонное время для отслеживания времени всякий раз, когда вы устанавливаете таймер или создаете экземпляр типа `time.Time` с помощью функции `time.Now`. Это делается неявным образом: таймеры используют монотонное время автоматически. Метод `Sub` использует монотонное время для расчета длительности `time.Duration`, если оно указано в обоих экземплярах типа `time.Time`. В противном случае (если один или оба экземпляра не были созданы с помощью функции `time.Now`) метод `Sub` использует для расчета длительности `time.Duration` разновидность времени, указанную в этих экземплярах.



Если вам интересно, к каким проблемам может привести неправильное обращение с монотонным временем, прочитайте в блоге компании Cloudflare статью (<https://oreil.ly/IxS2D>), в которой подробно описывается ошибка, возникающая из-за отсутствия поддержки монотонного времени в более ранних версиях языка Go.

Таймеры и тайм-ауты

Как упоминалось в подразделе «Как можно установить тайм-аут для кода» на с. 271, пакет `time` включает в себя функции, которые возвращают каналы, выдающие выходные значения по истечении указанного времени. Функция `time.After` возвращает канал, выдающий значение только один раз, в то время как функция `time.Tick` возвращает канал, выдающий новое значение многократно с указанным интервалом `time.Duration`. Эти функции используются в сочетании со средствами конкурентного программирования языка Go для реализации тайм-аутов или многократного выполнения задач. Вы также можете произвести вызов некоторой функции по истечении указанной длительности `time.Duration`, используя функцию `time.AfterFunc`. Функцию `time.Tick` следует использовать лишь в очень простых программах, поскольку используемый ею экземпляр типа `time.Ticker` невозможно остановить (и, соответственно, он не может быть удален сборщиком мусора). В более сложных программах используйте вместо нее функцию `time.NewTicker`, возвращающую тип `*time.Ticker` с каналом, который вы можете прослушивать, и методы для сброса и остановки тикера.

Пакет `encoding/json`

REST-подобные API предусматривают использование формата JSON в качестве стандартного способа обмена данными между сервисами, и стандартная библиотека языка Go включает себя поддержку для преобразования типов данных этого языка в формат JSON и обратно. Процесс преобразования из типа данных языка Go называется *маршализацией*, а процесс преобразования в тип данных языка Go — *демаршализацией*.

Используйте теги структур для добавления метаданных

Допустим, мы разрабатываем систему для управления заказами и нам нужно выполнять чтение и запись следующего формата JSON:

```
{
  "id": "12345",
  "date_ordered": "2020-05-01T13:01:02Z",
```

```
"customer_id": "3",
"items": [{ "id": "xyz123", "name": "Thing 1" }, { "id": "abc789", "name": "Thing 2" } ]
}
```

Мы должны определить типы, соответствующие этим данным:

```
type Order struct {
    ID          string          `json:"id"`
    DateOrdered time.Time      `json:"date_ordered"`
    CustomerID   string          `json:"customer_id"`
    Items        []Item           `json:"items"`
}

type Item struct {
    ID   string `json:"id"`
    Name string `json:"name"`
}
```

Для определения правил обработки формата JSON здесь используются *теги структур* — строки, записываемые после полей структуры. Несмотря на то что эти строки заключаются в обратные апострофы, они не могут занимать больше одной строки. Теги структур представляют собой одну или несколько пар «тег — значение», которые записываются в виде *имяТега*: «*значениеТега*» и отделяются друг от друга пробелами. Поскольку они представляют собой обычные строки, компилятор не может проверять их на предмет корректности формата, однако это может делать команда `go vet`. Обратите также внимание, что все эти поля являются экспортируемыми. Как и любой другой пакет, пакет `encoding/json` не может обращаться к неэкспортируемым полям структуры в другом пакете.

В случае обработки формата JSON с помощью тега `json` указывается имя поля формата JSON, ассоциируемого с полем структуры. Если тег `json` не будет предоставлен, то по умолчанию будет предполагаться, что имя поля объекта JSON совпадает с именем поля структуры языка Go. Несмотря на такое поведение по умолчанию, рекомендуется явно указывать имена JSON-полей с помощью тегов структур, даже когда они совпадают с именами полей структуры.



При демаршализации из формата JSON в поля структуры без тега `json` сопоставление имен будет производиться без учета регистра. При маршализации из полей структуры без тега `json` в формат JSON имена JSON-полей будут всегда начинаться с буквы верхнего регистра, поскольку эти поля являются экспортируемыми.

Если при маршализации или демаршализации нужно проигнорировать определенное поле, поставьте вместо его имени дефис (-). Если поле необходимо исключить из выходных данных в том случае, если оно является пустым, добавьте после его имени слово `omitempty`.



К сожалению, определение «пустое» здесь не совсем совпадает с нулевым значением, как можно было бы ожидать. Если срезы и карты нулевой длины считаются пустыми, то структура с нулевым значением не считается таковой.

Теги структур позволяют контролировать поведение программы с помощью метаданных. В таких языках, как Java, поощряется снабжение различных элементов программы аннотациями, которые указывают, *как* их следует обрабатывать, без явного определения того, *что* будет делать эта обработка. Хотя декларативное программирование и позволяет получать более лаконичные программы, автоматическая обработка метаданных затрудняет понимание поведения программы. Каждый, кому приходилось работать над большим Java-проектом с аннотациями, знает, какое паническое чувство тебя охватывает, когда что-то идет не так и ты не понимаешь, какой код обрабатывает ту или иную аннотацию и какие изменения при этом производятся. В Go отдается предпочтение ясности, а не краткости. Теги структур никогда не обрабатываются автоматически; их обработка производится при передаче экземпляра структуры в функцию.

Демаршализация и маршализация

Функция `Unmarshal` из пакета `encoding/json` используется для преобразования среза байтов в структуру. Если у нас есть строка `data`, то преобразовать ее в структурный тип `Order` можно следующим образом:

```
var o Order
err := json.Unmarshal([]byte(data), &o)
if err != nil {
    return err
}
```

Функция `json.Unmarshal` заполняет данными входной параметр, как это делается в случае реализаций интерфейса `io.Reader`. Такой подход используется по двум причинам. Во-первых, как и в случае реализаций интерфейса `io.Reader`, это делает возможным эффективное многократное использование одной и той же структуры, позволяя вам контролировать использование памяти. Во-вторых, другого способа это сделать просто не существует. Поскольку в Go пока нет обобщенных типов, вы не можете каким-либо образом указать, экземпляр какого типа следует создавать для сохранения считываемых байтов. И даже когда в Go появятся обобщенные типы, данный подход будет выглядеть предпочтительнее в плане использования памяти.

С помощью функции `Marshal` из пакета `encoding/json` мы можем записать экземпляр типа `Order` в виде JSON-объекта, помещенного в срез байтов:

```
out, err := json.Marshal(o)
```

Здесь вы можете спросить, каким образом производится обработка тегов структур. Возможно, вас также интересует, каким образом функциям `json.Marshal` и `json.Unmarshal` удастся выполнять чтение и запись структур любого типа? Ведь все другие методы, которые использовали до этого, работают с типами, известными в момент компиляции программы (при этом заранее регистрируются даже типы, указываемые в переключателе типов). Ответ на оба этих вопроса звучит так: с помощью рефлексии. Подробнее о рефлексии будет рассказано в главе 14.

JSON, считыватели и записыватели

Функции `json.Marshal` и `json.Unmarshal` работают со срезами байтов. Как мы только что видели, большинство источников и приемников данных в Go реализует интерфейсы `io.Reader` и `io.Writer`. Мы могли бы копировать все содержимое экземпляра интерфейса `io.Reader` в байтовый срез с помощью функции `ioutil.ReadAll`, чтобы его можно было считывать с помощью функции `json.Unmarshal`, однако этот подход неэффективен. Мы могли бы также записывать данные в расположенный в памяти байтовый срез с помощью функции `json.Marshal`, а затем записывать этот байтовый срез в сеть или на диск, но будет лучше, если мы будем записывать данные непосредственно в экземпляр интерфейса `io.Writer`.

В пакете `encoding/json` есть два типа, которые можно использовать в таких ситуациях. Типы `json.Decoder` и `json.Encoder` позволяют производить чтение и запись в экземпляр любого типа, удовлетворяющего соответственно интерфейсу `io.Reader` или интерфейсу `io.Writer`. Кратко ознакомимся с тем, как он работает.

Изначально наши данные будут находиться в переменной `toFile`, реализующей простую структуру:

```
type Person struct {
    Name string `json:"name"`
    Age  int    `json:"age"`
}
toFile := Person {
    Name: "Fred",
    Age:  40,
}
```

Поскольку тип `os.File` реализует и интерфейс `io.Reader`, и интерфейс `io.Writer`, мы можем использовать его для демонстрации работы с типами `json.Decoder` и `json.Encoder`. Сначала мы должны записать переменную `toFile` во временный файл. Для этого мы передаем временный файл в функцию `json.NewEncoder`,

которая возвращает экземпляр типа `json.Encoder` для временного файла, и передаем переменную `toFile` в метод `Encode`:

```
tmpFile, err := ioutil.TempFile(os.TempDir(), "sample-")
if err != nil {
    panic(err)
}
defer os.Remove(tmpFile.Name())
err = json.NewEncoder(tmpFile).Encode(toFile)
if err != nil {
    panic(err)
}
err = tmpFile.Close()
if err != nil {
    panic(err)
}
```

Разобравшись с записью переменной `toFile`, мы можем заняться считыванием формата JSON. Для этого мы передаем ссылку на временный файл в функцию `json.NewDecoder` и вызываем в возвращаемом экземпляре типа `json.Decoder` метод `Decode`, передавая ему переменную типа `Person`:

```
tmpFile2, err := os.Open(tmpFile.Name())
if err != nil {
    panic(err)
}
var fromFile Person
err = json.NewDecoder(tmpFile2).Decode(&fromFile)
if err != nil {
    panic(err)
}
err = tmpFile2.Close()
if err != nil {
    panic(err)
}
fmt.Printf("%+v\n", fromFile)
```

Полную версию этого примера можно найти в онлайн-песочнице (<https://oreil.ly/HU8Ie>).

Кодирование и декодирование JSON-поточков

А что делать в том случае, когда требуется считывать или записывать в формате JSON сразу несколько структур? В таком случае нам придут на помощь все те же типы `json.Decoder` и `json.Encoder`.

Допустим, что у вас есть следующие данные:

```
{"name": "Fred", "age": 40}
```

```

{"name": "Mary", "age": 21}
{"name": "Pat", "age": 30}

```

В данном примере мы будем предполагать, что они сохранены в строковой переменной `data`, но, в принципе, они могут находиться и в файле или даже во входящем HTTP-запросе (о том, как работают HTTP-серверы, мы поговорим чуть позже).

Мы будем сохранять эти данные в переменной `t` по одному JSON-объекту за раз.

Как и раньше, мы инициализируем экземпляр типа `json.Decoder` источником данных, но на этот раз вызываем в экземпляре типа `json.Decoder` метод `More`, используя его результат в качестве условия цикла `for`. Это позволяет нам производить чтение данных по одному JSON-объекту за раз:

```

dec := json.NewDecoder(strings.NewReader(data))
var t struct { Name string `json:"name"` Age int `json:"age"` }
for dec.More() {
    err := dec.Decode(&t)
    if err != nil {
        panic(err)
    }
    // обработка переменной t
}

```

Запись нескольких значений с помощью типа `json.Encoder` выполняется так же, как в случае записи с помощью этого типа одного значения. Хотя в данном примере мы производим запись в экземпляр типа `bytes.Buffer`, для этой цели можно использовать любой тип, удовлетворяющий интерфейсу `io.Writer`:

```

var b bytes.Buffer
enc := json.NewEncoder(&b)
for _, input := range allInputs {
    t := process(input)
    err = enc.Encode(t)
    if err != nil {
        panic(err)
    }
}
out := b.String()

```

Вы можете запустить этот код в онлайн-песочнице (<https://oreil.ly/gOAnt>).

Хотя в этом примере поток данных содержит несколько JSON-объектов, которые не обернуты в массив, с помощью типа `json.Decoder` также можно производить чтение одного объекта из массива без загрузки в память всего массива. Это позволяет существенно повысить производительность и уменьшить потребление памяти. Соответствующий пример можно найти в документации языка Go (https://oreil.ly/_LTZQ).

Парсинг пользовательского формата JSON

В большинстве случаев вам будет достаточно встроенной функциональности, но иногда ее все же потребуется переопределить. По умолчанию тип `time.Time` поддерживает JSON-поля в формате RFC 339, но иногда приходится работать и с другими форматами времени. Это можно сделать путем создания нового типа, реализующего интерфейсы `json.Marshaler` и `json.Unmarshaler`:

```
type RFC822ZTime struct {
    time.Time
}

func (rt RFC822ZTime) MarshalJSON() ([]byte, error) {
    out := rt.Time.Format(time.RFC822Z)
    return []byte(`"` + out + `"`), nil
}

func (rt *RFC822ZTime) UnmarshalJSON(b []byte) error {
    if string(b) == "null" {
        return nil
    }
    t, err := time.Parse(`"` + time.RFC822Z + `"` , string(b))
    if err != nil {
        return err
    }
    *rt = RFC822ZTime{t}
    return nil
}
```

Мы встроили в новую структуру `RFC822ZTime` экземпляр типа `time.Time`, чтобы по-прежнему иметь доступ к другим методам типа `time.Time`. Как упоминалось в подразделе «Приемники указателей и приемники значений» на с. 166, метод для чтения значения времени объявляется в приемнике значений, а метод для модификации значения времени — в приемнике указателей.

После этого мы можем изменить тип поля `DateOrdered` и работать со временем в формате RFC 822:

```
type Order struct {
    ID          string      `json:"id"`
    DateOrdered RFC822ZTime `json:"date_ordered"`
    CustomerID  string      `json:"customer_id"`
    Items       []Item      `json:"items"`
}
```

Вы можете запустить этот код в онлайн-песочнице (https://oreil.ly/I_cSY).

Минусом этого подхода с философской точки зрения является то, что мы позволили формату обрабатываемых нами JSON-данных изменить типы полей нашей структуры данных. Это является недостатком подхода, используемого

в пакете `encoding/json`. Мы могли бы реализовать интерфейсы `json.Marshaler` и `json.Unmarshaler` в типе `Order`, но при этом нам пришлось бы написать код для обработки всех полей, в том числе и тех, которые не нуждаются в нестандартной поддержке. Формат тегов структур не позволяет нам каким-либо образом указать, с помощью какой функции следует производить парсинг определенного поля. Поэтому нам остается лишь один вариант — создание пользовательского типа для поля.

Чтобы сократить объем кода, определяющего то, как будут выглядеть ваши JSON-данные, определите две структуры. Одну из них используйте для преобразования в формат JSON и обратно, а другую — для обработки данных. Считывайте JSON-данные в JSON-совместимый тип, а затем копируйте их во второй тип. В случае записи данных в JSON формате делайте то же самое в обратном направлении. Такой подход порождает некоторое дублирование, но зато позволяет сделать бизнес-логику независимой от используемых протоколов проводной связи.

Для преобразования данных из формата языка Go в формат JSON и обратно можно передавать функциям `json.Marshal` и `json.Unmarshal` тип `map[string]interface{}`, однако оставьте эту возможность для того случая, когда будете заниматься экспериментами, и используйте вместо этого конкретный тип, который ясно показывает, что вы обрабатываете. Типы используются в Go не без причины: они документируют ожидаемые данные и их типы.

Хотя JSON-кодировщик является наиболее часто используемым кодировщиком в стандартной библиотеке, в Go встроена поддержка и других форматов, в частности XML и Base64. Если вам нужно кодировать формат данных, который не поддерживается стандартной библиотекой или каким-либо сторонним модулем, вы можете реализовать кодировщик для этого формата сами. О том, как можно реализовать собственный кодировщик, будет рассказано в подразделе «Используйте рефлексию для создания маршализатора данных» на с. 371.



Стандартная библиотека включает в себя пакет `encoding/gob`, который позволяет использовать собственный двоичный формат представления данных языка Go, чем-то напоминающий сериализацию языка Java. Подобно тому как сериализация в Java служит в качестве протокола проводной связи для технологий Enterprise Java Beans и Java RMI, протокол `gob` рассчитан на использование в качестве формата проводной связи для применяемой в Go реализации удаленного вызова процедур (RPC, remote procedure call), включенной в пакет `net/rpc`. Не используйте ни пакет `encoding/gob`, ни пакет `net/rpc`. Если вам нужно произвести удаленный вызов метода из Go-кода, используйте стандартный протокол наподобие GRPC (<https://grpc.io>), чтобы не быть привязанными к конкретному языку. Даже если вы очень любите язык Go, чтобы ваши сервисы были полезными, вы должны обеспечить возможность их вызова для разработчиков, использующих другие языки.

Пакет net/http

Каждый язык поставляется со стандартной библиотекой, но ожидания относительно того, что должна включать в себя стандартная библиотека, со временем менялись. Поскольку язык Go появился на свет совсем недавно — в начале 2010-х годов, его стандартная библиотека включает в себя то, что в других языках относилось к сфере ответственности сторонних разработчиков: клиент и сервер стандарта HTTP/2, пригодные для использования в реальных приложениях.

Клиент

В пакете net/http определен тип `Client`, позволяющий отправлять HTTP-запросы и получать HTTP-ответы. В этом пакете также можно найти дефолтный экземпляр клиента (уместно названный `DefaultClient`), но его не стоит использовать в реальных приложениях, потому что он по умолчанию не обладает тайм-аутом. Вместо этого лучше создайте собственный экземпляр. Вам нужно создать только один экземпляр типа `http.Client` для всей своей программы, поскольку он сможет должным образом обработать даже большое количество одновременных запросов в рамках нескольких горутин:

```
client := &http.Client{
    Timeout: 30 * time.Second,
}
```

Когда требуется выполнить запрос, нужно создать экземпляр типа `*http.Request` с помощью функции `http.NewRequestWithContext`, передав ей контекст, метод и тот URL-адрес, к которому вам нужно подключиться. В случае запроса PUT, POST или PATCH необходимо задать тело запроса с помощью последнего параметра типа `io.Reader`. Если тела нет, следует указать значение `nil`:

```
req, err := http.NewRequestWithContext(context.Background(),
    http.MethodGet, "https://jsonplaceholder.typicode.com/todos/1", nil)
if err != nil {
    panic(err)
}
```



О том, что такое контекст, мы поговорим в главе 12.

После создания экземпляра типа `*http.Request` можно установить необходимые заголовки посредством поля `Header` этого экземпляра. Затем вызовите метод `Do` в экземпляре типа `http.Client`, передав ему свой запрос (экземпляр

типа `http.Request`), и вы получите результат запроса в виде экземпляра типа `http.Response`:

```
req.Header.Add("X-My-Client", "Learning Go")
res, err := client.Do(req)
if err != nil {
    panic(err)
}
```

Экземпляр ответа содержит несколько полей с информацией, выданной в ответ на запрос. Поле `StatusCode` содержит числовой код состояния ответа, поле `Status` — текст кода состояния, поле `Header` — заголовки ответа, а поле `Body` типа `io.ReadCloser` — возвращаемый контент. Это позволяет нам использовать этот тип в сочетании с типом `json.Decoder` для обработки ответов API REST-full:

```
defer res.Body.Close()
if res.StatusCode != http.StatusOK {
    panic(fmt.Sprintf("unexpected status: got %v", res.Status))
}
fmt.Println(res.Header.Get("Content-Type"))
var data struct {
    UserID    int    `json:"userId"`
    ID        int    `json:"id"`
    Title     string `json:"title"`
    Completed bool   `json:"completed"`
}
err = json.NewDecoder(res.Body).Decode(&data)
if err != nil {
    panic(err)
}
fmt.Printf("%+v\n", data)
```



В пакете `net/http` есть функции для выполнения запросов GET, HEAD и POST. Я не рекомендую вам использовать эти функции, поскольку они не позволяют задать тайм-аут для запроса.

Сервер

Работа с HTTP-сервером в Go строится на основе типа `http.Server` и интерфейса `http.Handler`. Подобно тому как тип `http.Client` производит отправку HTTP-запросов, тип `http.Server` отвечает за прослушивание трафика на предмет HTTP-запросов. Это высокопроизводительный сервер стандарта HTTP/2 с поддержкой протокола TLS.

Обработку поступающего на сервер запроса осуществляет реализация интерфейса `http.Handler`, указанная в поле `Handler`. Этот интерфейс определяет один метод:

```
type Handler interface {
    ServeHTTP(http.ResponseWriter, *http.Request)
}
```

Тип `*http.Request` уже нам знаком: он уже использовался ранее для отправки запроса на HTTP-сервер. `http.ResponseWriter` — это интерфейс с тремя методами:

```
type ResponseWriter interface {
    Header() http.Header
    Write([]byte) (int, error)
    WriteHeader(statusCode int)
}
```

Эти методы следует вызывать в определенном порядке. Сначала вы должны вызвать метод `Header`, чтобы получить экземпляр типа `http.Header` и установить нужные вам заголовки ответа. Если вам не нужно устанавливать заголовки, то этот метод можно не вызывать. Затем следует вызвать метод `WriteHeader`, передав ему код состояния HTTP вашего ответа. (Все коды состояния определены в виде констант в пакете `net/http`. Здесь было бы уместно использовать пользовательский тип, но разработчики языка поступили иначе: все константы кода состояния представляют собой нетипизированные целые числа.) Если требуется отправить ответ с кодом состояния 200, можно обойтись без метода `WriteHeader`. Наконец, вы должны вызвать метод `Write`, чтобы определить тело ответа. Вот как будет выглядеть простейший обработчик запросов:

```
type HelloHandler struct{}

func (hh HelloHandler) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    w.Write([]byte("Hello!\n"))
}
```

Создание экземпляра типа `http.Server` производится точно так же, как и в случае любой другой структуры:

```
s := http.Server{
    Addr:      ":8080",
    ReadTimeout: 30 * time.Second,
    WriteTimeout: 90 * time.Second,
    IdleTimeout: 120 * time.Second,
    Handler:   HelloHandler{},
}
err := s.ListenAndServe()
if err != nil {
    if err != http.ErrServerClosed {
```

```
        panic(err)
    }
}
```

В поле `Addr` нужно указать, трафик какого хоста и на каком порте должен прослушивать сервер. Если вы не укажете хост и порт, то сервер будет по умолчанию прослушивать трафик всех хостов на стандартном для HTTP-соединений порте 80. Затем нужно задать тайм-ауты для чтения, записи и режима ожидания, используя для этого значения типа `time.Duration`. Обязательно задайте эти тайм-ауты, чтобы должным образом обрабатывать запросы от вредоносных или некорректно работающих HTTP-клиентов, поскольку по умолчанию они вообще не используются. Наконец, в поле `Handler` нужно указать, какой обработчик (экземпляр типа `http.Handler`) следует применять для вашего сервера.

Поскольку от сервера, способного обрабатывать лишь один запрос, будет мало проку, стандартная библиотека языка Go также включает в себя тип маршрутизатора запросов, `*http.ServeMux`. Экземпляр этого типа можно создать с помощью функции `http.NewServeMux`. Поскольку этот тип соответствует интерфейсу `http.Handler`, его экземпляр можно присвоить полю `Handler` типа `http.Server`. Тип `*http.ServeMux` также обладает двумя методами для диспетчеризации запросов. Первый метод называется `Handle` и принимает два параметра: путь и обработчик (экземпляр интерфейса `http.Handler`). В случае совпадения путей производится вызов обработчика.

Вы могли бы создать реализации интерфейса `http.Handler`, но более распространенный подход сводится к вызову метода `HandleFunc` в экземпляре типа `*http.ServeMux`:

```
mux.HandleFunc("/hello", func(w http.ResponseWriter, r *http.Request) {
    w.Write([]byte("Hello!\n"))
})
```

Этот метод принимает функцию или замыкание и преобразует его в тип `http.HandlerFunc`. Мы уже рассматривали этот тип в разделе «Функциональные типы — ключ к интерфейсам» на с. 194. В случае простых обработчиков можно будет обойтись замыканием. В случае более сложных обработчиков, которые зависят от другой бизнес-логики, используйте метод в экземпляре структуры, как было показано в разделе «Неявные интерфейсы облегчают внедрение зависимостей» на с. 195.

Поскольку экземпляр типа `*http.ServeMux` перенаправляет запросы экземплярам интерфейса `http.Handler` и поскольку тип `*http.ServeMux` реализует интерфейс `http.Handler`, вы можете создать экземпляры типа `*http.ServeMux` для нескольких родственных запросов и зарегистрировать их в родительском экземпляре типа `*http.ServeMux`:

```

person := http.NewServeMux()
person.HandleFunc("/greet", func(w http.ResponseWriter, r *http.Request) {
    w.Write([]byte("greetings!\n"))
})
dog := http.NewServeMux()
dog.HandleFunc("/greet", func(w http.ResponseWriter, r *http.Request) {
    w.Write([]byte("good puppy!\n"))
})
mux := http.NewServeMux()
mux.Handle("/person/", http.StripPrefix("/person", person))
mux.Handle("/dog/", http.StripPrefix("/dog", dog))

```

В этом примере запрос адреса `/person/greet` обрабатывается обработчиками, привязанными к переменной `person`, а запросы адреса `/dog/greet` — обработчиками, привязанными к переменной `dog`. При регистрации переменных `person` и `dog` в переменной `mux` мы используем вспомогательную функцию `http.StripPrefix`, чтобы удалить часть пути, обрабатываемую переменной `mux`.



Функции пакетного уровня `http.Handle`, `http.HandleFunc`, `http.ListenAndServe` и `http.ListenAndServeTLS` позволяют работать с объявленным на уровне пакета экземпляром типа `*http.ServeMux` с именем `http.DefaultServeMux`. Не используйте эти функции где-либо еще, кроме простейших тестовых программ. Функции `http.ListenAndServe` и `http.ListenAndServeTLS` создают экземпляр типа `http.Server`, не позволяя вам задать такие свойства сервера, как тайм-ауты. Кроме того, сторонние библиотеки могут зарегистрировать в экземпляре `http.DefaultServeMux` собственные обработчики, что невозможно будет выяснить, не просканировав все имеющиеся зависимости (и прямые, и косвенные). Чтобы сохранить контроль над своим приложением, обойдитесь без совместно используемого состояния.

Промежуточный слой

HTTP-серверу часто приходится выполнять для нескольких обработчиков некоторый общий набор таких действий, как проверка авторизации пользователя, расчет времени выполнения запроса или проверка заголовка запроса. Go позволяет решать эти общие задачи с помощью паттерна *промежуточного слоя*. Вместо специального типа этот паттерн использует функцию, которая принимает экземпляр интерфейса `http.Handler` и возвращает экземпляр интерфейса `http.Handler`. Обычно возвращаемый экземпляр интерфейса `http.Handler` представляет собой замыкание, которое преобразуется в тип `http.HandlerFunc`. В качестве примера ниже представлены два промежуточных генератора, один из которых рассчитывает время выполнения запросов, а второй реализует один из самых неудачных способов контроля доступа:

```

func RequestTimer(h http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {

```

```

        start := time.Now()
        h.ServeHTTP(w, r)
        end := time.Now()
        log.Printf("request time for %s: %v", r.URL.Path, end.Sub(start))
    })
}

var securityMsg = []byte("You didn't give the secret password\n")

func TerribleSecurityProvider(password string) func(http.Handler) http.Handler {
    return func(h http.Handler) http.Handler {
        return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
            if r.Header.Get("X-Secret-Password") != password {
                w.WriteHeader(http.StatusUnauthorized)
                w.Write(securityMsg)
                return
            }
            h.ServeHTTP(w, r)
        })
    }
}

```

Эти две реализации промежуточного слоя показывают, чем он занимается. Сначала мы выполняем определенную настройку или проверку. В случае неудачного завершения проверки мы записываем результат в промежуточный слой (обычно с кодом ошибки) и выходим из функции. В случае успешного прохождения проверки мы вызываем метод `ServeHTTP` обработчика. После его выполнения мы выполняем операции по высвобождению ресурсов.

Функция `TerribleSecurityProvider` показывает, как можно создать настраиваемый промежуточный слой. Мы передаем функции параметры настройки (в данном случае пароль), и она возвращает нам промежуточный слой, использующий эти параметры настройки. Постарайтесь не запутаться: эта функция возвращает замыкание, которое, в свою очередь, тоже возвращает замыкание.



Возможно, вас интересует, каким образом можно передавать значения через слои промежуточного слоя. Это можно делать с помощью контекста, о котором мы поговорим в главе 12.

Мы можем добавить промежуточные слои к своим обработчикам, выстроив их в цепочку:

```

terribleSecurity := TerribleSecurityProvider("GOPHER")

mux.Handle("/hello", terribleSecurity(RequestTimer(
    http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {

```

```

        w.Write([]byte("Hello!\n"))
    })))

```

Мы получаем промежуточный слой, возвращаемый функцией `TerribleSecurityProvider`, а затем обортываем свой обработчик несколькими вызовами функций. При этом сначала вызывается замыкание `terribleSecurity`, затем — функция `RequestTimer`, и уже после этого — реальный обработчик запросов.

Поскольку тип `*http.ServeMux` реализует интерфейс `http.Handler`, вы можете применить набор промежуточных слоев ко всем обработчикам, зарегистрированным в одном маршрутизаторе запросов:

```

terribleSecurity := TerribleSecurityProvider("GOPHER")
wrappedMux := terribleSecurity(RequestTimer(mux))
s := http.Server{
    Addr:    ":8080",
    Handler: wrappedMux,
}

```

Используйте идиоматические сторонние модули для расширения возможностей сервера

Тот факт, что Go предлагает вам сервер, пригодный для использования в реальных приложениях, совсем не означает, что вы не должны использовать сторонние модули для расширения его функциональности. Если вы не хотите использовать цепочки функций в качестве промежуточного слоя, то можете воспользоваться сторонним модулем под названием `alice` (https://oreil.ly/_cS1w), который позволяет использовать следующий синтаксис:

```

helloHandler := func(w http.ResponseWriter, r *http.Request) {
    w.Write([]byte("Hello!\n"))
}
chain := alice.New(terribleSecurity, RequestTimer).ThenFunc(helloHandler)
mux.Handle("/hello", chain)

```

Наиболее слабым элементом встроенной поддержки протокола HTTP является маршрутизатор запросов `*http.ServeMux`. Он не позволяет использовать разные обработчики, в зависимости от HTTP-глагола или заголовка, или использовать переменные в URL-адресе. Также, по правде говоря, вложение экземпляров типа `*http.ServeMux` друг в друга выглядит достаточно неуклюже. Существует очень много проектов, призванных заменить этот тип, но наиболее популярными из них являются проекты `gorilla mux` (<https://oreil.ly/CrQ4i>) и `chi` (<https://oreil.ly/twYcG>). Оба этих пакета являются идиоматическими, потому что в них используются экземпляры типов `http.Handler` и `http.HandlerFunc`, что хорошо согласуется с исповедуемым в Go принципом создания библиотек, совместимых со стандартной библиотекой. Каждый пакет также использует идиоматические про-

межуточные слои и предоставляет опциональные реализации промежуточных слоев для распространенных задач.

Резюме

В этой главе вы познакомились с рядом наиболее часто используемых пакетов стандартной библиотеки и посмотрели, как в них реализуются те рекомендуемые практики, которые вам следует имитировать в своем коде. Мы также коснулись таких принципов рациональной программной разработки, как изменение некоторых решений с учетом имеющегося опыта разработки и сохранение обратной совместимости, чтобы обеспечить прочный фундамент для создаваемых приложений.

В следующей главе мы рассмотрим контекст — пакет и паттерн для передачи состояния и таймеров из одной части Go-кода в другую.

Контекст

Серверы должны быть способны обрабатывать метаданные, относящиеся к отдельному запросу. Эти метаданные можно разделить на две основные категории: метаданные, необходимые для корректной обработки запроса, и метаданные, указывающие, когда следует прекратить обработку запроса. Например, иногда HTTP-серверу требуется идентификатор отслеживания для идентификации цепочки запросов, проходящей через некоторый ряд микросервисов. Иногда серверу также требуется установить таймер, прекращающий выполнение запросов к другим микросервисам в том случае, если они выполняются слишком долго. Для сохранения такой информации во многих языках используются *локальные переменные потока*, которые ассоциируют данные с конкретным потоком выполнения операционной системы. Этот подход не работает в Go, потому что у горутин нет уникальных идентификаторов, с помощью которых можно было бы находить те или иные значения. Что еще важнее, применение локальных переменных потока напоминает магию: значения поступают в одно место и затем вдруг появляются в другом месте.

В Go проблема с метаданными запросов решается с помощью конструкции, называемой *контекстом*. Посмотрим, как выглядит корректный подход к ее использованию.

Что такое контекст

Контекст — это не какой-то дополнительный элемент языка, а просто экземпляр, который соответствует интерфейсу `Context`, определенному в пакете `context`. Как вы помните, идиоматический подход в Go поощряет явную передачу данных в виде параметров функции. То же самое справедливо и в случае контекста, который представляет собой просто еще один параметр функции. Наряду с соглашением о том, что ошибка должна быть последним возвращаемым значением функции, в Go имеется и соглашение о том, что контекст должен передаваться

в программе явным образом в виде первого параметра функции. Обычно этому параметру контекста дают имя `ctx`:

```
func logic(ctx context.Context, info string) (string, error) {
    // здесь производятся определенные действия
    return "", nil
}
```

Помимо интерфейса `Context`, пакет `context` также содержит несколько фабричных функций для создания и обертывания контекстов. Когда у вас еще нет контекста, как, например, в точке входа в консольную программу, необходимо создать пустой исходный контекст с помощью функции `context.Background`. Эта функция возвращает переменную типа `context.Context`. (Это исключение из общепринятого подхода, согласно которому функция должна возвращать конкретный тип.)

Пустой контекст является отправной точкой; при этом каждое последующее добавление метаданных в контекст осуществляется путем *обертывания* существующего контекста с помощью одной из фабричных функций пакета `context`:

```
ctx := context.Background()
result, err := logic(ctx, "a string")
```



В пакете `context` есть еще одна функция, создающая пустой экземпляр типа `context.Context`. Это функция `context.TODO`, которая предназначена для временного использования на этапе разработки. Если вы еще не знаете, откуда будет поступать контекст и как он будет применяться, используйте функцию `context.TODO` в качестве временной заглушки. Однако эта функция не должна использоваться в окончательном коде приложения.

При создании HTTP-сервера следует использовать несколько иной паттерн получения и передачи контекста через промежуточные слои до обработчика (экземпляра типа `http.Handler`), расположенного на верхнем уровне иерархии. К сожалению, контекст был добавлен в API языка Go намного позже создания пакета `net/http`, и в силу обязательства по обеспечению совместимости в интерфейс `http.Handler` уже нельзя было добавить параметр `context.Context`.

Однако обязательство по обеспечению совместимости не запрещает добавление новых методов в существующие типы, что и сделали разработчики языка Go. У типа `http.Request` есть два метода для работы с контекстом.

- Метод `Context` возвращает ассоциированный с запросом экземпляр типа `context.Context`.

- Метод `WithContext` принимает экземпляр типа `context.Context` и возвращает новый экземпляр типа `http.Request`, содержащий состояние старого запроса, дополненное предоставленным экземпляром типа `context.Context`.

Общий паттерн выглядит следующим образом:

```
func Middleware(handler http.Handler) http.Handler {
    return http.HandlerFunc(func(rw http.ResponseWriter, req *http.Request) {
        ctx := req.Context()
        // обертываем контекст — как это делается, мы увидим чуть позже!
        req = req.WithContext(ctx)
        handler.ServeHTTP(rw, req)
    })
}
```

В промежуточном слое мы прежде всего извлекаем существующий контекст из запроса с помощью метода `Context`. После занесения значений в контекст мы создаем новый запрос на основе старого запроса и незаполненного контекста, используя метод `WithContext`. Наконец, мы вызываем свой обработчик и передаем ему новый запрос и имеющийся экземпляр типа `http.ResponseWriter`.

Внутри обработчика мы извлекаем контекст из запроса с помощью метода `Context` и вызываем свою бизнес-логику, передавая ей контекст в качестве первого параметра, как уже делалось ранее:

```
func handler(rw http.ResponseWriter, req *http.Request) {
    ctx := req.Context()
    err := req.ParseForm()
    if err != nil {
        rw.WriteHeader(http.StatusInternalServerError)
        rw.Write([]byte(err.Error()))
        return
    }
    data := req.FormValue("data")
    result, err := logic(ctx, data)
    if err != nil {
        rw.WriteHeader(http.StatusInternalServerError)
        rw.Write([]byte(err.Error()))
        return
    }
    rw.Write([]byte(result))
}
```

Метод `WithContext` используется еще в одном случае: когда нужно вызвать из своего приложения некоторый другой HTTP-сервис. Как и в случае передачи контекста через промежуточные слои, при этом нужно снабдить исходящий запрос контекстом с помощью метода `WithContext`:

```
type ServiceCaller struct {
    client *http.Client
}
```

```

}

func (sc ServiceCaller) callAnotherService(ctx context.Context, data string)
    (string, error) {
    req, err := http.NewRequest(http.MethodGet,
        "http://example.com?data="+data, nil)
    if err != nil {
        return "", err
    }
    req = req.WithContext(ctx)
    resp, err := sc.client.Do(req)
    if err != nil {
        return "", err
    }
    defer resp.Body.Close()
    if resp.StatusCode != http.StatusOK {
        return "", fmt.Errorf("Unexpected status code %d",
            resp.StatusCode)
    }
    // оставшиеся действия по обработке ответа
    id, err := processResponse(resp.Body)
    return id, err
}

```

Теперь, уже зная, как можно получить и передать контекст, посмотрим, как можно сделать его полезным. Начнем мы с операции отмены.

Отмена

Допустим, что у вас есть запрос, который порождает несколько горутин, каждая из которых вызывает определенный HTTP-сервис. Если один из вызываемых сервисов возвратит ошибку, не позволяющую вернуть корректный результат, в продолжении дальнейшей обработки внутри остальных горутин не будет никакого смысла. В таком случае в Go производится отмена горутин, реализуемая посредством контекста.

Отменяемый контекст можно создать с помощью функции `context.WithCancel`, которая принимает экземпляр типа `context.Context` и возвращает экземпляры типов `context.Context` и `context.CancelFunc`. При этом возвращаемый экземпляр типа `context.Context` представляет собой не тот же контекст, который был передан в функцию, а *дочерний* контекст, который обертывает переданный в функцию *родительский* экземпляр типа `context.Context`. Экземпляр типа `context.CancelFunc` представляет собой функцию, которая *отменяет* контекст, сообщая о том, что нужно прекратить обработку, всему тому коду, который осуществляет прослушивание на предмет возможной отмены.



Мы еще не раз увидим этот паттерн обертывания. Контекст при этом рассматривается как неизменяемый экземпляр. Каждое последующее добавление информации в контекст осуществляется путем обертывания имеющегося родительского контекста в дочерний контекст. Это позволяет нам использовать контексты для передачи информации в более глубокие слои кода. Контекст никогда не применяется для передачи информации из более глубоких в более высокие слои.

Посмотрим, как это выглядит на практике. Поскольку этот код производит настройку сервера, его нельзя запустить в онлайн-песочнице, однако вы можете скачать его по адресу <https://oreil.ly/qQy5c>. Сначала произведем настройку двух серверов в файле с именем `servers.go`:

```
func slowServer() *httptest.Server {
    s := httptest.NewServer(http.HandlerFunc(func(w http.ResponseWriter,
        r *http.Request) {
        time.Sleep(2 * time.Second)
        w.Write([]byte("Slow response"))
    }))
    return s
}

func fastServer() *httptest.Server {
    s := httptest.NewServer(http.HandlerFunc(func(w http.ResponseWriter,
        r *http.Request) {
        if r.URL.Query().Get("error") == "true" {
            w.Write([]byte("error"))
            return
        }
        w.Write([]byte("ok"))
    }))
    return s
}
```

При вызове этих функций происходит запуск серверов. Первый сервер в течение двух секунд находится в режиме ожидания, после чего возвращает сообщение `Slow response` (Медленная реакция). Второй сервер выполняет проверку в отношении того, не равен ли параметр запроса типа `error` значению `true`. Если это так, он возвращает сообщение `error` (ошибка). В противном случае он возвращает сообщение `ok`.



Мы используем тип `httptest.Server`, который облегчает написание модульных тестов для кода, использующего удаленные серверы. Его удобно использовать в данном случае, поскольку и клиент, и сервер находятся здесь в пределах одной программы. Подробнее о типе `httptest.Server` будет рассказано в разделе «Пакет `httptest`» на с. 354.

Теперь мы напишем клиентскую часть кода, разместив ее в файле с именем `client.go`:

```
var client = http.Client{}

func callBoth(ctx context.Context, errVal string, slowURL string,
             fastURL string) {
    ctx, cancel := context.WithCancel(ctx)
    defer cancel()
    var wg sync.WaitGroup
    wg.Add(2)
    go func() {
        defer wg.Done()
        err := callServer(ctx, "slow", slowURL)
        if err != nil {
            cancel()
        }
    }()
    go func() {
        defer wg.Done()
        err := callServer(ctx, "fast", fastURL+"?error="+errVal)
        if err != nil {
            cancel()
        }
    }()
    wg.Wait()
    fmt.Println("done with both")
}

func callServer(ctx context.Context, label string, url string) error {
    req, err := http.NewRequestWithContext(ctx, http.MethodGet, url, nil)
    if err != nil {
        fmt.Println(label, "request err:", err)
        return err
    }
    resp, err := client.Do(req)
    if err != nil {
        fmt.Println(label, "response err:", err)
        return err
    }
    data, err := ioutil.ReadAll(resp.Body)
    if err != nil {
        fmt.Println(label, "read err:", err)
        return err
    }
    result := string(data)
    if result != "" {
        fmt.Println(label, "result:", result)
    }
    if result == "error" {
        fmt.Println("cancelling from", label)
    }
}
```

```

        return errors.New("error happened")
    }
    return nil
}

```

В этом файле происходит самое интересное. Сначала функция `callBoth` создает на основе переданного ей контекста отменяемый контекст и функцию отмены. В соответствии с общепринятым соглашением мы дали этой функциональной переменной имя `cancel`. Важно помнить, что каждый раз, когда вы создаете отменяемый контекст, вы *должны* вызывать функцию отмены. При этом не произойдет ничего страшного, если она будет вызвана несколько раз: после первого вызова все следующие игнорируются. Чтобы эта функция была гарантированно вызвана в любом случае, мы вызываем ее в операторе `defer`. Затем мы запускаем две горутин, передаем отменяемый контекст, метку и URL-адрес в функцию `callServer` и ждем, пока обе функции завершат свое выполнение. Если один или оба вызова функции `callServer` возвращают ошибку, мы вызываем функцию `cancel`.

Функция `callServer` представляет собой простой клиент. Здесь мы создаем свои запросы с отменяемым контекстом и выполняем вызов. В случае возникновения ошибки или возвращения строки `error` мы возвращаем ошибку.

Вот как будет выглядеть расположенная в файле `main.go` основная функция, которая будет запускать нашу программу:

```

func main() {
    ss := slowServer()
    defer ss.Close()
    fs := fastServer()
    defer fs.Close()

    ctx := context.Background()
    callBoth(ctx, os.Args[1], ss.URL, fs.URL)
}

```

В функции `main` мы запускаем серверы и создаем контекст, после чего вызываем клиенты, передавая им контекст, первый аргумент программы и URL-адреса серверов.

При успешном выполнении мы увидим следующий результат:

```

$ make run-ok
go build
./context_cancel false
fast result: ok
slow result: Slow response
done with both

```

В случае возникновения ошибки результат будет выглядеть следующим образом:

```
$ make run-cancel
go build
./context_cancel true
fast result: error
cancelling from fast
slow response err: Get "http://127.0.0.1:38804": context canceled
done with both
```



Каждый раз, когда вы создаете контекст с привязанной к нему функцией отмены, вы *должны* вызвать эту функцию отмены после выполнения обработки и в случае успешного выполнения, и при возникновении ошибки. В противном случае в вашей программе будет происходить утечка ресурсов (памяти и горутин), что в конечном итоге приведет к замедлению или прекращению работы программы. Ошибки не будет, если вы вызовете функцию отмены несколько раз, поскольку все последующие вызовы после первого не будут производить никаких действий. Самый простой способ обеспечить гарантированный вызов функции отмены сводится к тому, чтобы вызывать ее с помощью оператора `defer` сразу после ее возвращения другой функцией.

Хотя в некоторых случаях и удобно использовать ручную отмену, это не единственный возможный подход. В следующем разделе будет показано, как можно производить автоматическую отмену с использованием тайм-аутов.

Таймеры

Одной из самых важных функций сервера является управление запросами. Начинающие программисты часто думают, что сервер должен принимать максимально возможное количество запросов и работать над ними в течение максимально возможного времени до тех пор, пока не сможет вернуть результат каждому клиенту.

Проблемой этого подхода является то, что он не поддается масштабированию. Сервер является совместно используемым ресурсом, и, как в случае любого совместно используемого ресурса, каждый пользователь хочет использовать его в максимально возможной степени, не сильно беспокоясь о том, что в нем могут нуждаться и другие пользователи. Совместно используемый ресурс должен сам обеспечить справедливое распределение времени между всеми имеющимися пользователями.

Для управления своей нагрузкой сервер обычно может сделать следующее:

- ограничить количество одновременных запросов;
- ограничить количество запросов в очереди на выполнение;

- ограничить время выполнения запроса;
- ограничить количество используемых запросом ресурсов (таких как память или дисковое пространство).

Go предоставляет инструменты для наложения первых трех ограничений. Мы уже видели, как можно наложить первые два ограничения, при обсуждении конкурентности в главе 10. Сервер может управлять объемом одновременной нагрузки, ограничивая количество запускаемых горутин. Управлять размером очереди на выполнение можно посредством буферизованных каналов.

Контекст позволяет вам управлять временем выполнения запроса. При создании приложения вы должны иметь представление о минимально необходимой производительности, то есть о том, насколько быстро должен выполняться запрос, чтобы пользователь оставался удовлетворенным работой приложения. Если вы знаете, каким должно быть максимальное время выполнения запроса, вы можете наложить это ограничение с помощью контекста.



Если вы хотите ограничить объем используемой запросом памяти или дискового пространства, вам придется написать собственный код для управления этим ресурсом. Обсуждение этой темы выходит за рамки этой книги.

Для создания контекста с ограничением по времени можно использовать одну из двух функций. Первой функцией является функция `context.WithTimeout`, которая принимает два параметра: существующий контекст и экземпляр типа `time.Duration`, представляющий промежуток времени, после которого будет производиться автоматическая отмена контекста. Возвращает эта функция контекст, автоматически запускающий отмену после указанного промежутка времени, и функцию отмены, вызываемую для немедленной отмены контекста.

Второй функцией является функция `context.WithDeadline`, которая принимает существующий контекст и экземпляр типа `time.Time`, указывающий, в какой момент времени будет производиться автоматическая отмена контекста. Подобно функции `context.WithTimeout`, она возвращает контекст, автоматически запускающий отмену в указанный момент времени, и функцию отмены.



Если вы передадите функции `context.WithDeadline` уже прошедший момент времени, то она создаст уже отмененный контекст.

Узнать, когда будет произведена автоматическая отмена контекста, можно, вызвав метод `Deadline` в экземпляре типа `context.Context`. Этот метод возвращает экземпляр типа `time.Time`, представляющий этот момент времени, и значение типа `bool`, указывающее, был ли установлен тайм-аут. Это напоминает использование идиомы «запятая-ок» для чтения карт или каналов.

При наложении ограничения на общую длительность выполнения запроса иногда нужно дополнительно разбить это время на несколько промежутков. Если ваш сервис вызывает некоторый другой сервис, часто требуется ограничить время выполнения сетевого вызова, зарезервировав время для остальной обработки или для других сетевых вызовов. В таком случае для управления длительностью выполнения отдельного вызова следует создать дочерний контекст, обертывающий родительский контекст, с помощью функции `context.WithTimeout` или `context.WithDeadline`.

При этом любой тайм-аут, установленный в дочернем контексте, будет ограничен тайм-аутом, установленным в родительском контексте. Таким образом, если длительность тайм-аута будет составлять две секунды в родительском контексте и три секунды в дочернем контексте, то по истечении двух секунд будет произведена отмена и родительского, и дочернего контекста.

В качестве примера рассмотрим следующую простую программу:

```
ctx := context.Background()
parent, cancel := context.WithTimeout(ctx, 2*time.Second)
defer cancel()
child, cancel2 := context.WithTimeout(parent, 3*time.Second)
defer cancel2()
start := time.Now()
<-child.Done()
end := time.Now()
fmt.Println(end.Sub(start))
```

В этом примере кода мы устанавливаем двухсекундный тайм-аут в родительском контексте и трехсекундный тайм-аут в дочернем контексте. Затем мы ждем отмены дочернего контекста, дожидаясь момента возвращения значения каналом, возвращенным методом `Done`, вызванным в дочернем экземпляре типа `context.Context`. Подробнее о методе `Done` мы поговорим в следующем разделе.

Выполнив эту программу в онлайн-песочнице (<https://oreil.ly/FS8h2>), вы получите следующий результат:

2s

Управление отменой контекста в собственном коде

В большинстве случаев вам не потребуется управлять тайм-аутами или отменой в собственном коде: время его выполнения просто не будет настолько большим, чтобы это было необходимо. Вы должны передавать контекст каждый раз, когда ваш код вызывает другой HTTP-сервис или базу данных, и эти библиотеки обеспечат надлежащее управление отменой посредством контекста.

Если вам все же необходимо, чтобы ваш код прерывал свое выполнение посредством отмены контекста, то для этого потребуется реализовать связанные с отменой проверки, используя возможности конкурентного программирования, рассмотренные в главе 10. Для управления отменой используются два метода типа `context.Context`.

Метод `Done` возвращает канал типа `struct{}`. (Применение этого типа в качестве типа возвращаемых значений объясняется тем, что пустая структура не использует память.) Этот канал закрывается при отмене контекста, запускаемой срабатыванием таймера или вызовом функции отмены. Как вы помните, закрытый канал всегда немедленно возвращает нулевое значение при попытке считать из него значение.



При вызове метода `Done` в неотменяемом контексте возвращается значение `nil`. Как упоминалось в главе 10, операция чтения из канала, равного `nil`, никогда не возвращает значение. Если это будет делаться не внутри ветви `case` оператора `select`, то ваша программа зависнет.

Метод `Err` возвращает значение `nil`, если контекст по-прежнему активен, если же контекст уже был отменен, он возвращает одну из двух сигнальных ошибок: `context.Canceled` и `context.DeadlineExceeded`. Первая ошибка возвращается в случае явной отмены, а вторая — в случае отмены, запущенной тайм-аутом.

Паттерн управления отменой контекста в собственном коде выглядит следующим образом:

```
func longRunningThingManager(ctx context.Context, data string) (string, error) {
    type wrapper struct {
        result string
        err     error
    }
    ch := make(chan wrapper, 1)
    go func() {
        // выполнение долго выполняющихся операций
        result, err := longRunningThing(ctx, data)
```

```
    ch <- wrapper{result, err}
  }()
  select {
  case data := <-ch:
    return data.result, data.err
  case <-ctx.Done():
    return "", ctx.Err()
  }
}
```

В своем коде мы должны занести в структуру данные, возвращаемые долго выполняющейся функцией, чтобы их можно было передать в канал. После этого мы создаем канал типа `wrapper` с размером буфера, равным 1. Используя буферизованный канал, мы делаем возможным выход из горутины даже в том случае, если буферизованное значение никогда не будет считано из-за отмены.

В горутине мы принимаем результат, возвращаемый долго выполняющейся функцией, и заносим его в буферизованный канал. Далее мы имеем оператор `select` с двумя ветвями. В первой ветви мы считываем и возвращаем данные, выданные долго выполняющейся функцией. Эта ветвь срабатывает в том случае, когда контекст не отменяется срабатыванием таймера или вызовом функции отмены. Вторая ветвь оператора `select` срабатывает в том случае, когда контекст отменяется. При этом мы возвращаем нулевое значение в качестве данных и извлекаемую из контекста ошибку, сообщая тем самым о причине выполнения отмены.

Это во многом напоминает паттерн, рассмотренный нами в главе 11, когда мы говорили о том, как можно наложить ограничение на время выполнения кода с помощью функции `time.After`. В целом здесь используется такой же код, но с тем отличием, что ограничение по времени (или условие отмены) определяется фабричными методами контекста.

Значения

Существует еще одна область применения контекста. Его также можно использовать для передачи связанных с отдельным запросом метаданных внутри программы.

В большинстве случаев следует отдавать предпочтение явной передаче данных посредством параметров. Как упоминалось ранее, идиоматический подход в Go поощряет использование более явных способов, что, помимо прочего, включает в себя и явную передачу данных. Если функция зависит от некоторых данных, вы должны четко понимать, откуда они поступают.

В то же время в некоторых случаях невозможно передавать данные явным образом. Наиболее распространенным примером такой ситуации является обработчик HTTP-запросов и ассоциированный с ним промежуточный слой. Как мы уже видели, любой обработчик HTTP-запросов имеет два параметра: один для запроса и один для ответа. Если вам нужно сделать значение доступным для обработчика в промежуточном слое, вы должны сохранить его в контексте. Примером такой ситуации может служить извлечение пользователя из веб-токена JSON (JWT, JSON Web Token) или создание для каждого запроса глобального уникального идентификатора, который передается в обработчик и бизнес-логику через несколько промежуточных слоев.

Наряду с фабричными методами для создания контекстов, отменяемых таймаутом или функцией отмены, в пакете `context` имеется и фабричный метод для занесения значений в контекст, `context.WithValue`. Он принимает три параметра: обертываемый контекст, ключ для извлечения значения и само значение. Возвращает он дочерний контекст с парой «ключ — значение». В качестве типа ключа и значения при этом используется пустой интерфейс (`interface{}`).

С помощью метода `Value` типа `context.Context` можно проверить, содержит ли значение контекст или один из его родительских контекстов. Этот метод принимает ключ и возвращает ассоциированное с ним значение. При этом в качестве типа ключа и возвращаемого значения опять же используется пустой интерфейс (`interface{}`). Если не удастся найти значение, соответствующее указанному ключу, данный метод возвращает значение `nil`. Чтобы привести возвращаемое значение к подходящему типу, используйте идиому «запятая-ок».



Если вы знакомы со структурами данных, то могли заметить, что поиск значений, сохраненных в цепочке контекстов, представляет собой линейный поиск. Это почти не сказывается на производительности, когда нужно найти лишь несколько значений, но приводит к серьезному ее снижению в том случае, если для каждого запроса в контексте будут сохраняться десятки значений. Однако если ваша программа создает цепочку контекстов с десятками значений, то она, вероятно, нуждается в некотором рефакторинге.

Хотя в контексте можно сохранять значение любого типа, существует идиоматический паттерн, позволяющий гарантировать уникальность ключа. Как и ключ карты, ключ сохраняемого в контексте значения должен относиться к сравнимому типу. Создайте для ключа новый неэкспортируемый тип на основе типа `int`:

```
type userKey int
```

Если в качестве типа ключа вы будете использовать строку или другой общедоступный тип, то в других пакетах можно будет создать идентичные ключи, что приведет к конфликтам. Это вызовет трудно поддающиеся отладке проблемы,

как, например, в случае, когда один пакет записывает в контекст данные, маскирующие данные, записанные другим пакетом, или считывает из контекста данные, записанные другим пакетом.

После объявления неэкспортируемого типа ключей следует объявить неэкспортируемую константу этого типа:

```
const key userKey = 1
```

По причине того, что и тип, и константа ключа будут неэкспортируемыми, никакой внешний код не сможет заносить данные в контекст внутри вашего пакета, что исключает возможность возникновения конфликта. Если вам нужно заносить в контекст несколько значений в своем пакете, определите для каждого значения разные ключи одного и того же типа, используя паттерн *iota*, рассмотренный нами в подразделе «Йота (иногда) используется для создания перечислений» на с. 173. *Iota* прекрасно подойдет для этого случая, поскольку мы используем здесь значение константы лишь для того, чтобы отличать друг от друга несколько ключей.

После этого определите API для размещения значения в контексте и чтения значения из контекста. Эти функции следует делать публичными лишь в том случае, если требуется предоставить возможность чтения и записи контекстных значений внешнему коду. Имя функции, создающей контекст со значением, должно начинаться с *ContextWith*. Имя функции, возвращающей значение из контекста, должно оканчиваться на *FromContext*. В нашем случае функции для записи и чтения из контекста информации о пользователе будут выглядеть следующим образом:

```
func ContextWithUser(ctx context.Context, user string) context.Context {
    return ctx.WithValue(key, user)
}

func UserFromContext(ctx context.Context) (string, bool) {
    user, ok := ctx.Value(key).(string)
    return user, ok
}
```

Теперь, уже располагая кодом для управления пользователями, посмотрим, как его можно использовать. Напишем промежуточный слой, извлекающий ID пользователя из файла cookie:

```
// в реальной реализации следует использовать подпись,
// чтобы исключить возможность подделки идентификатора пользователя
func extractUser(req *http.Request) (string, error) {
    userCookie, err := req.Cookie("user")
    if err != nil {
```

```
        return "", err
    }
    return userCookie.Value, nil
}

func Middleware(h http.Handler) http.Handler {
    return http.HandlerFunc(func(rw http.ResponseWriter, req *http.Request) {
        user, err := extractUser(req)
        if err != nil {
            rw.WriteHeader(http.StatusUnauthorized)
            return
        }
        ctx := req.Context()
        ctx = ContextWithUser(ctx, user)
        req = req.WithContext(ctx)
        h.ServeHTTP(rw, req)
    })
}
```

В промежуточном слое мы сначала получаем значение пользователя. Затем мы извлекаем контекст из запроса с помощью метода `Context` и создаем новый контекст со значением пользователя с помощью функции `ContextWithUser`. После этого мы создаем новый запрос на основе старого запроса и нового контекста, используя метод `WithContext`. Наконец, мы вызываем следующую функцию в нашей цепи обработчиков, передавая ей новый запрос и полученный в качестве параметра экземпляр типа `http.ResponseWriter`.

В большинстве случаев вы должны извлекать значение из контекста в своем обработчике запросов и явным образом передавать его в свою бизнес-логику. Функции языка Go позволяют использовать явные параметры для этой цели, и вы не должны использовать контекст для неявной передачи значений в обход API:

```
func (c Controller) handleRequest(rw http.ResponseWriter, req *http.Request) {
    ctx := req.Context()
    user, ok := identity.UserFromContext(ctx)
    if !ok {
        rw.WriteHeader(http.StatusInternalServerError)
        return
    }
    data := req.URL.Query().Get("data")
    result, err := c.Logic.businessLogic(ctx, user, data)
    if err != nil {
        rw.WriteHeader(http.StatusInternalServerError)
        rw.Write([]byte(err.Error()))
        return
    }
    rw.Write([]byte(result))
}
```

Наш обработчик получает контекст, вызывая метод `Context` в экземпляре запроса, извлекает пользователя из контекста с помощью функции `UserFromContext` и вызывает бизнес-логику.

В некоторых случаях все же будет лучше оставить значение в контексте. Одним из таких случаев является уже упоминавшийся выше случай использования глобального уникального идентификатора отслеживания. Эта информация используется для управления приложением и не является частью состояния бизнес-логики. Явная передача этих данных внутри программы потребует использования дополнительных параметров и сделает невозможной интеграцию со сторонними библиотеками, разработчики которых не знают о том, какую метаданную вы используете. Если вы оставите глобальный уникальный идентификатор отслеживания в контексте, то он будет незаметно передаваться сквозь бизнес-логику, которой не нужно что-либо знать об отслеживании, и будет доступен, когда вашей программе потребуется занести сообщение в журнал или подключиться к другому серверу.

Вот как выглядит простая реализация глобального уникального идентификатора (GUID) с поддержкой контекста, позволяющая производить отслеживание в рамках нескольких сервисов и создавать сообщения журнала, содержащие GUID-идентификатор:

```
package tracker

import (
    "context"
    "fmt"
    "net/http"
    "github.com/google/uuid"
)

type guidKey int

const key guidKey = 1

func contextWithGUID(ctx context.Context, guid string) context.Context {
    return ctx.WithValue(key, guid)
}

func guidFromContext(ctx context.Context) (string, bool) {
    g, ok := ctx.Value(key).(string)
    return g, ok
}

func Middleware(h http.Handler) http.Handler {
    return http.HandlerFunc(func(rw http.ResponseWriter, req *http.Request) {
        ctx := req.Context()
        if guid := req.Header.Get("X-GUID"); guid != "" {
```

```
        ctx = contextWithGUID(ctx, guid)
    } else {
        ctx = contextWithGUID(ctx, uuid.New().String())
    }
    req = req.WithContext(ctx)
    h.ServeHTTP(rw, req)
})
}

type Logger struct{}

func (Logger) Log(ctx context.Context, message string) {
    if guid, ok := guidFromContext(ctx); ok {
        message = fmt.Sprintf("GUID: %s - %s", guid, message)
    }
    // выполняем журналирование
    fmt.Println(message)
}

func Request(req *http.Request) *http.Request {
    ctx := req.Context()
    if guid, ok := guidFromContext(ctx); ok {
        req.Header.Add("X-GUID", guid)
    }
    return req
}
```

Функция `Middleware` либо извлекает GUID-идентификатор из входящего запроса, либо генерирует новый GUID-идентификатор. В обоих случаях она заносит GUID-идентификатор в контекст, создает новый запрос с обновленным контекстом и выполняет следующий вызов в цепочке вызовов.

Далее мы видим, как используется этот GUID-идентификатор. Структура `Logger` предоставляет универсальный метод журналирования, который принимает в качестве параметров контекст и строку. Если в контексте содержится GUID-идентификатор, он добавляется в начало сообщения журнала, которое затем выводится на экран. Функция `Request` используется в том случае, когда данный сервис производит вызов другого сервиса. Она принимает экземпляр типа `*http.Request`, добавляет заголовок с GUID-идентификатором при его наличии в контексте и возвращает экземпляр типа `*http.Request`.

Теперь, когда у нас уже есть этот пакет, мы можем использовать методы внедрения зависимостей, рассмотренные нами в разделе «Неявные интерфейсы облегчают внедрение зависимостей» на с. 195, для создания бизнес-логики, ничего не знающей об используемой информации отслеживания. Прежде всего мы должны объявить интерфейс для представления нашего диспетчера журналирования, функциональный тип для представления декоратора запросов и использующую эти типы структуру для бизнес-логики:

```

type Logger interface {
    Log(context.Context, string)
}

type RequestDecorator func(*http.Request) *http.Request

type BusinessLogic struct {
    RequestDecorator RequestDecorator
    Logger            Logger
    Remote            string
}

```

После этого мы реализуем нашу бизнес-логику:

```

func (bl BusinessLogic) businessLogic(
    ctx context.Context, user string, data string) (string, error) {
    bl.Logger.Log(ctx, "starting businessLogic for " + user + " with " + data)
    req, err := http.NewRequestWithContext(ctx,
        http.MethodGet, bl.Remote+"?query="+data, nil)
    if err != nil {
        bl.Logger.Log(ctx, "error building remote request:" + err)
        return "", err
    }
    req = bl.RequestDecorator(req)
    resp, err := http.DefaultClient.Do(req)
    // продолжение обработки
}

```

GUID-идентификатор передается диспетчеру журналирования и декоратору запросов так, чтобы бизнес-логика не знала о его наличии: то есть мы отделяем данные, необходимые для логики программы, от данных, необходимых для управления программой. О том, что мы ассоциируем эти данные, знает только код, выполняющий подключение зависимостей внутри функции `main`:

```

bl := BusinessLogic{
    RequestDecorator: tracker.Request,
    Logger:          tracker.Logger{},
    Remote:          "http://www.example.com/query",
}

```

Полный код пользовательского промежуточного слоя и трекера GUID-идентификаторов можно найти на сайте GitHub (<https://oreil.ly/oyhmp>).



Используйте контекст для передачи значений «сквозь» стандартные API. Копируйте значения из контекста в явные параметры, если они требуются для обработки бизнес-логики. Служебная системная информация может считываться из контекста напрямую.

Резюме

В этой главе вы узнали, как можно управлять метаданными запроса с помощью контекста. Теперь вы уже умеете устанавливать тайм-ауты, выполнять явную отмену, передавать значения с помощью контекста и знаете, когда следует делать все эти вещи. В следующей главе вы познакомитесь со встроенным фреймворком тестирования языка Go и узнаете, как его можно использовать для выявления ошибок и проблем производительности в своих программах.

Написание тестов

В последние два десятилетия повсеместное принятие на вооружение методов автоматизированного тестирования, вероятно, сказалось на качестве создаваемого кода сильнее, чем использование любой другой технологии программной разработки. Поскольку и язык Go, и его экосистема создавались с расчетом на улучшение качества программного обеспечения, не вызывает удивления тот факт, что поддержка тестирования была включена в его стандартную библиотеку. Go делает тестирование кода настолько простым, что у вас не остается никаких оправданий для того, чтобы его не делать. В этой главе мы научимся тестировать свой Go-код, выделять наборы модульных и интеграционных тестов, проверять степень покрытия кода, писать сравнительные тесты и проверять код на наличие проблем конкурентности с помощью детектора состояний гонки. Попутно мы научимся писать тестируемый код и узнаем, почему это улучшает качество кода.

Основы тестирования

Поддержка тестирования в Go включает в себя две составляющие: библиотеки и инструменты. Пакет `testing` стандартной библиотеки предоставляет типы и функции для написания тестов, а встроенная команда `go test` обеспечивает выполнение тестов и генерирование отчетов. В отличие от многих других языков, в Go тесты размещаются в том же каталоге и в том же пакете, где размещается окончательный код приложения. Такое расположение позволяет тестам использовать и тестировать неэкспортируемые функции и переменные. Чуть позже будет показано, как следует писать тесты, проверяющие только публичные API.



Полный код приводимых в этой главе примеров можно найти на сайте GitHub (<https://oreil.ly/txE4b>).

Напишем простую функцию и тест для проверки ее работоспособности. Определение функции будет находиться в файле `adder/adder.go`:

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

Соответствующий тест будет находиться в файле `adder/adder_test.go`:

```
func Test_addNumbers(t *testing.T) {
    result := addNumbers(2,3)
    if result != 5 {
        t.Error("incorrect result: expected 5, got", result)
    }
}
```

Каждый тест записывается в файле с именем, оканчивающимся на `_test.go`. Так, если вы пишете тесты для файла `foo.go`, разместите их в файле с именем `foo_test.go`.

Имена функций тестирования начинаются со слова `Test`; они принимают один параметр типа `*testing.T`, в качестве имени которого принято использовать букву `t`. Функции тестирования не возвращают каких-либо значений. Поскольку имена тестов должны не только начинаться со слова `Test`, но и документировать то, что вы тестируете, старайтесь выбирать для них имена, описывающие предмет тестирования. При написании модульных тестов для отдельных функций в качестве имени теста принято использовать слово `Test`, дополненное именем функции. При тестировании неэкспортируемых функций между словом `Test` и именем функции иногда ставят символ подчеркивания.

Также обратите внимание, что мы используем стандартный Go-код для того, чтобы вызвать тестируемый код и проверить, выдает ли он надлежащие ответы. В случае некорректного результата мы сообщаем об ошибке с помощью метода `t.Error`, который работает подобно функции `fmt.Print`. Чуть позже мы рассмотрим и ряд других методов для выдачи сообщений об ошибках.

Теперь, когда мы познакомились с библиотечной составляющей поддержки тестирования в Go, рассмотрим используемые для этой цели инструменты. Подобно тому как команда `go build` компилирует двоичный файл, а команда `go run` запускает файл на выполнение, команда `go test` запускает тесты, расположенные в текущем каталоге:

```
$ go test
--- FAIL: Test_addNumbers (0.00s)
    adder_test.go:8: incorrect result: expected 5, got 4
FAIL
exit status 1
FAIL    test_examples/adder    0.006s
```

Похоже, что наш код содержит ошибку. Еще раз взглянув на код функции `addNumbers`, можно заметить, что вместо того, чтобы складывать `x` и `y`, она складывает `x` и `x`. Исправим код и запустим тест еще раз, чтобы убедиться в отсутствии ошибки:

```
$ go test
PASS
ok      test_examples/adder    0.006s
```

Команда `go test` позволяет указать, какие именно пакеты следует тестировать. Задав в качестве имени пакета выражение `./...`, можно запустить тесты, расположенные в текущем каталоге и во всех вложенных в него каталогах. Для получения подробных результатов тестирования используйте эту команду с флагом `-v`.

Выдача сообщения о неудачном завершении теста

Сообщить о неудачном завершении теста можно с помощью нескольких методов типа `*testing.T`. Мы уже успели познакомиться с методом `Error`, который создает строку с описанием ошибки на основе разделенного запятыми списка значений.

Если же для выдачи сообщения вы предпочитаете использовать форматированную строку в стиле функции `Printf`, используйте вместо этого метод `Errorf`:

```
t.Errorf("incorrect result: expected %d, got %d", 5, result)
```

Хотя методы `Error` и `Errorf` помечают тест как непройденный, функция тестирования при этом продолжает работать. Если вам нужно, чтобы функция тестирования прекращала свою работу сразу после обнаружения ошибки, используйте методы `Fatal` и `Fatalf`. Метод `Fatal` работает так же, как метод `Error`, а метод `Fatalf` — так же, как метод `Errorf`. Отличие состоит лишь в том, что функция тестирования прекращает свою работу сразу после выдачи сообщения о неудачном завершении теста. Обратите внимание, что при этом не прекращается выполнение *всех* тестов: после выхода из текущей функции тестирования будет продолжено выполнение остальных функций тестирования.

Когда же следует использовать методы `Fatal/Fatalf`, а когда — методы `Error/Errorf`? Если отрицательный результат выполняемой в тесте проверки означает, что все дальнейшие проверки внутри той же функции тестирования тоже дадут отрицательный результат или приведут к панике, используйте метод `Fatal` или `Fatalf`. Если же вы тестируете несколько независимых элементов (например, проверяете корректность значений, присвоенных полям структуры), используйте метод `Error` или `Errorf`, чтобы сразу выводить информацию о максимально возможном количестве имеющихся проблем. Это упрощает исправление кода

при наличии большого количества проблем, поскольку вам не приходится многократно перезапускать тесты.

Подготовка и заключительная «уборка»

Иногда перед выполнением тестов требуется настроить некоторое общее состояние и удалить его по завершении тестирования. В таком случае для управления этим состоянием и запуска тестов следует использовать функцию `TestMain`:

```
var testTime time.Time

func TestMain(m *testing.M) {
    fmt.Println("Set up stuff for tests here")
    testTime = time.Now()
    exitVal := m.Run()
    fmt.Println("Clean up stuff after tests here")
    os.Exit(exitVal)
}

func TestFirst(t *testing.T) {
    fmt.Println("TestFirst uses stuff set up in TestMain", testTime)
}

func TestSecond(t *testing.T) {
    fmt.Println("TestSecond also uses stuff set up in TestMain", testTime)
}
```

И в функции `TestFirst`, и в функции `TestSecond` используется переменная пакетного уровня `testTime`. Функции `TestMain` передается параметр типа `*testing.M`. При выполнении команды `go test` для пакета с функцией `TestMain` вместо непосредственного вызова тестов будет вызываться эта функция. После настройки состояния вы должны запустить выполнение функций тестирования, вызвав метод `Run` в экземпляре типа `*testing.M`. Метод `Run` возвращает код завершения, который будет равен `0` в случае успешного прохождения всех тестов. Наконец, вы должны вызвать функцию `os.Exit`, передав ей код завершения, возвращенный методом `Run`.

Выполнив команду `go test`, мы получим следующий результат:

```
$ go test
Set up stuff for tests here
TestFirst uses stuff set up in TestMain 2020-09-01 21:42:36.231508 -0400 EDT
    m=+0.000244286
TestSecond also uses stuff set up in TestMain 2020-09-01 21:42:36.231508 -0400
    EDT m=+0.000244286
PASS
Clean up stuff after tests here
ok      test_examples/testmain 0.006s
```

Имейте в виду, что функция `TestMain` вызывается только один раз, а не до и после каждого отдельного теста. Кроме того, в каждом пакете можно определить только одну функцию `TestMain`.

Функция `TestMain` может быть полезной в следующих двух распространенных случаях:

- когда требуется настроить данные, размещенные в некотором внешнем репозитории, например в базе данных;
- когда тестируемый код зависит от переменных пакетного уровня, которые нужно инициализировать.

Как я уже упоминал ранее (и не устану напоминать вам снова и снова!), если это возможно, ваши программы не должны содержать переменных пакетного уровня. Наличие таких переменных затрудняет понимание того, как перемещаются данные внутри программы. Если вы используете функцию `TestMain` по этой причине, то, возможно, вам стоит произвести рефакторинг кода.

Для высвобождения временных ресурсов, выделенных для отдельного теста, следует вызвать метод `Cleanup` в экземпляре типа `*testing.T`. В качестве единственного параметра этот метод принимает функцию без входных параметров и возвращаемых значений, которая выполняется по завершении теста. Хотя в случае простых тестов вы можете добиться того же результата, используя оператор `defer`, метод `Cleanup` будет полезен в том случае, когда перед выполнением тестов нужно настраивать образцы данных с помощью вспомогательных функций, как это делается в примере 13.1. Этот метод можно вызывать многократно. Как и в случае оператора `defer`, многократные вызовы метода `Cleanup` обрабатываются в порядке, обратном порядку их добавления.

Пример 13.1. Использование метода `t.Cleanup`

```
// createfile – это вспомогательная функция, вызываемая из нескольких тестов
func createFile(t *testing.T) (string, error) {
    f, err := os.Create("tempFile")
    if err != nil {
        return "", err
    }
    // записываем данные в переменную f
    t.Cleanup(func() {
        os.Remove(f.Name())
    })
    return f.Name(), nil
}

func TestFileProcessing(t *testing.T) {
    fName, err := createFile(t)
    if err != nil {
```

```
    t.Fatal(err)
}
// выполняем тестирование, не беспокоясь о высвобождении ресурсов
}
```

Расположение образцов тестовых данных

При обходе дерева исходного кода команда `go test` использует каталог текущего пакета в качестве текущего рабочего каталога. Если вы хотите использовать образцы данных для тестирования определенных в пакете функций, создайте для своих файлов подкаталог с именем `testdata`. Go резервирует это имя каталога для размещения тестовых файлов. При чтении данных из каталога `testdata` всегда используйте относительные ссылки на файлы. Поскольку команда `go test` задействует в качестве текущего рабочего каталога каталог текущего пакета, каждый пакет обращается по относительному пути к собственному каталогу `testdata`.



Примером использования каталога `testdata` может служить пакет `text` (<https://oreil.ly/nHtrc>).

Кэширование результатов теста

В главе 9 уже говорилось о том, что Go кэширует скомпилированные пакеты и использует кэшированную копию в том случае, когда в пакет не вносятся никаких изменений. Подобно этому, Go кэширует результаты тестирования каждого пакета и берет их из кэша в случае успешного прохождения того же теста при отсутствии каких-либо изменений. При изменении любого файла пакета или содержимого каталога `testdata` тесты компилируются и выполняются заново. Если вам нужно, чтобы тесты выполнялись всегда, используйте команду `go test` с флагом `-count=1`.

Тестирование своего публичного API

Созданные нами тесты находятся в том же пакете, где расположен окончательный код приложения. Это позволяет нам тестировать и экспортируемые, и неэкспортируемые функции.

Если же вам нужно тестировать только публичный API своего пакета, то и для этого в Go имеется общепринятый способ, который сводится к следующему.

Исходный код теста по-прежнему сохраняется в том же каталоге, где расположен окончательный код приложения, но в качестве имени пакета при этом указывается имя *имяПакета_test*. Перепишем наш первый тестовый случай, используя на этот раз экспортируемую функцию. Если в пакете `adder` будет определена следующая функция:

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

то ее можно будет протестировать как публичный API, используя в пакете `adder` файл `adder_public_test.go`, содержащий следующий код:

```
package adder_test  
  
import (  
    "testing"  
    "test_examples/adder"  
)  
  
func TestAddNumbers(t *testing.T) {  
    result := adder.AddNumbers(2, 3)  
    if result != 5 {  
        t.Error("incorrect result: expected 5, got", result)  
    }  
}
```

Обратите внимание, что в качестве имени пакета в данном файле теста используется имя `adder_test`. При этом мы должны импортировать пакет `test_examples/adder`, несмотря на то что файлы находятся в том же каталоге. В соответствии с соглашением относительно именования тестов, имя функции тестирования включает в себя имя функции `AddNumbers`. Также обратите внимание, что при вызове этой функции используется имя пакета, `adder.AddNumbers`, поскольку мы вызываем здесь экспортируемую функцию другого пакета.

Подобно тому как вы можете вызывать экспортируемые функции внутри пакета, вы можете тестировать публичный API из теста, расположенного в том же пакете, что и ваш исходный код. Использование имени пакета с суффиксом `_test` позволяет вам превратить свой пакет в своего рода «черный ящик», заставляя вас взаимодействовать с ним только через экспортируемые функции, методы, типы, константы и переменные. Имейте также в виду, что в одном и том же каталоге исходного кода при этом могут присутствовать файлы тестов, использующие оба имени пакета.

Используйте модуль go-cmp для сравнения результатов тестов

Для полного сравнения двух экземпляров составного типа часто требуется довольно громоздкий код. Хотя для сравнения структур, карт и срезов можно использовать функцию `reflect.DeepEqual`, существует и более удачный способ. Компания Google выпустила сторонний модуль `go-cmp` (<https://github.com/google/go-cmp>), который выполняет сравнение за вас и возвращает подробное описание несовпадений. Посмотрим, как он работает, определив простую структуру и фабричную функцию для ее заполнения:

```
type Person struct {
    Name      string
    Age       int
    DateAdded time.Time
}

func CreatePerson(name string, age int) Person {
    return Person{
        Name:      name,
        Age:       age,
        DateAdded: time.Now(),
    }
}
```

В файле теста мы должны импортировать пакет `github.com/google/go-cmp/cmp`, а функция тестирования будет выглядеть следующим образом:

```
func TestCreatePerson(t *testing.T) {
    expected := Person{
        Name: "Dennis",
        Age:  37,
    }
    result := CreatePerson("Dennis", 37)
    if diff := cmp.Diff(expected, result); diff != "" {
        t.Error(diff)
    }
}
```

Функция `cmp.Diff` принимает в качестве параметров ожидаемый результат и результат, возвращаемый тестируемой функцией. Возвращает он строку, которая описывает имеющиеся несовпадения между входными параметрами. В случае полного совпадения возвращается пустая строка. Мы присваиваем результат функции `cmp.Diff` переменной `diff` и проверяем ее на равенство пустой строке. Если она не равна пустой строке, значит, произошла ошибка.

Скомпилировав и запустив этот тест, мы увидим, какой результат генерирует модуль `go-cmp` в случае несовпадения сравниваемых структур:

```
$ go test
--- FAIL: TestCreatePerson (0.00s)
    ch13_cmp_test.go:16:   ch13_cmp.Person{
                          Name:      "Dennis",
                          Age:       37,
    -   DateAdded: s"0001-01-01 00:00:00 +0000 UTC",
    +   DateAdded: s"2020-03-01 22:53:58.087229 -0500 EST m=+0.001242842",
      }

FAIL
FAIL    ch13_cmp    0.006s
```

Строки со знаками «-» и «+» указывают, какие поля содержат отличающиеся значения. В данном случае неудачное завершение проверки объясняется несовпадением дат. Это неразрешимая проблема, потому что мы не можем управлять тем, какую дату присвоит функция `CreatePerson`. В силу этого мы должны проигнорировать поле `DateAdded`. Это можно сделать, определив функцию-компаратор. Объявите эту функцию в тесте как локальную переменную:

```
comparer := cmp.Comparer(func(x, y Person) bool {
    return x.Name == y.Name && x.Age == y.Age
})
```

Передав эту функцию в функцию `cmp.Comparer`, мы получим пользовательский компаратор. Передаваемая функция должна принимать два параметра одинакового типа и возвращать булево значение. Она также должна быть симметричной (не зависящей от порядка параметров), детерминированной (всегда возвращающей один и тот же результат для тех же входных данных) и чистой (не модифицирующей свои параметры). В данном случае мы сравниваем поля `Name` и `Age`, игнорируя поле `DateAdded`.

Теперь мы должны изменить свой вызов функции `cmp.Diff`, включив в него компаратор (`comparer`):

```
if diff := cmp.Diff(expected, result, comparer); diff != "" {
    t.Error(diff)
}
```

Это лишь беглый обзор наиболее важных возможностей модуля `go-cmp`. В документации по этому модулю вы найдете более подробные сведения об управлении процессом сравнения и выходным форматом.

Табличные тесты

Обычно для проверки того, насколько правильно работает функция, требуется не один, а несколько тестовых случаев. Для проверки своей функции вы можете написать несколько функций тестирования или несколько тестов внутри одной и той же функции, однако используемая при этом логика тестирования будет в значительной мере повторяться. Каждый раз вы будете настраивать вспомогательные данные и функции, определять входные данные и проверять выходные данные путем сравнения их с ожидаемым результатом. Вместо того чтобы писать этот код снова и снова, вы можете воспользоваться паттерном «Табличные тесты». Рассмотрим пример. Допустим, у нас есть следующая функция в пакете `table`:

```
func DoMath(num1, num2 int, op string) (int, error) {
    switch op {
    case "+":
        return num1 + num2, nil
    case "-":
        return num1 - num2, nil
    case "*":
        return num1 * num2, nil
    case "/":
        if num2 == 0 {
            return 0, errors.New("division by zero")
        }
        return num1 / num2, nil
    default:
        return 0, fmt.Errorf("unknown operator %s", op)
    }
}
```

Чтобы протестировать эту функцию, мы должны проверить различные ветви, используя входные данные, возвращающие корректные результаты, и входные данные, которые вызывают ошибки. Мы могли бы сделать это так, как показано ниже, однако при этом используется много повторяющегося кода:

```
func TestDoMath(t *testing.T) {
    result, err := DoMath(2, 2, "+")
    if result != 4 {
        t.Error("Should have been 4, got", result)
    }
    if err != nil {
        t.Error("Should have been nil error, got", err)
    }
    result2, err2 := DoMath(2, 2, "-")
    if result2 != 0 {
        t.Error("Should have been 0, got", result2)
    }
    if err2 != nil {
```

```

        t.Error("Should have been nil error, got", err2)
    }
    // и так далее...
}

```

Заменяем все эти повторы табличным тестом. Сначала мы объявляем срез анонимных структур. Каждая структура будет содержать поля для имени теста, входных параметров и возвращаемых значений. Каждый элемент этого среза будет служить для представления отдельного теста:

```

data := []struct {
    name      string
    num1      int
    num2      int
    op        string
    expected  int
    errMsg    string
}{
    {"addition", 2, 2, "+", 4, ""},
    {"subtraction", 2, 2, "-", 0, ""},
    {"multiplication", 2, 2, "*", 4, ""},
    {"division", 2, 2, "/", 1, ""},
    {"bad_division", 2, 0, "/", 0, `division by zero`},
}

```

Затем мы производим обход тестовых случаев, содержащихся в срезе `data`, каждый раз вызывая метод `Run`. Именно эта строка и производит всю магию. Мы передаем методу `Run` два параметра: имя подтеста и функцию с одним параметром типа `*testing.T`. Внутри этой функции мы вызываем функцию `DoMath`, передавая ей поля текущего элемента среза `data`; тем самым мы многократно используем одну и ту же логику. Запустив эти тесты, вы увидите, что они не только успешно выполняются, но и позволяют снабдить каждый подтест именем путем использования флага `-v`:

```

for _, d := range data {
    t.Run(d.name, func(t *testing.T) {
        result, err := DoMath(d.num1, d.num2, d.op)
        if result != d.expected {
            t.Errorf("Expected %d, got %d", d.expected, result)
        }
        var errMsg string
        if err != nil {
            errMsg = err.Error()
        }
        if errMsg != d.errMsg {
            t.Errorf("Expected error message `%s`, got `%s`,",
                d.errMsg, errMsg)
        }
    })
}

```



Сравнение сообщений об ошибках является не очень надежным подходом, поскольку в отношении текста таких сообщений обычно не дается никаких гарантий совместимости. У нас не остается других вариантов в данном случае, потому что тестируемая функция создает ошибки с помощью функций `errors.New` и `fmt.Errorf`. В случае пользовательского типа ошибок для проверки корректности возвращаемых ошибок следует использовать функцию `errors.Is` или `errors.As`.

Теперь, когда мы уже умеем запускать большое количество тестов, поговорим о степени покрытия кода и узнаем, что именно проверяют наши тесты.

Проверка степени покрытия кода

Степень покрытия кода является очень полезным инструментом, когда нужно узнать, не упустили ли вы из виду какие-либо очевидные случаи. В то же время даже 100%-ное покрытие кода не гарантирует вам корректную работу кода при всех возможных входных данных. Сначала мы посмотрим, какие сведения о покрытии кода выводит команда `go test`, а затем узнаем, с какими ограничениями вам придется столкнуться, если вы будете полагаться только на эту информацию.

При выполнении команды `go test` с флагом `-cover` вычисляется информация о покрытии кода, которая затем включается в выводимые результаты теста. Используя второй флаг `-coverprofile`, вы можете сохранить информацию о покрытии кода в файл:

```
go test -v -cover -coverprofile=c.out
```

Если мы запустим наш тест с флагом покрытия кода, то, помимо прочего, он выведет строку с информацией о степени покрытия кода, которая в данном случае составляет 87,5 %. Это полезная информация, но было бы удобнее, если бы мы также знали, что было упущено из виду. Входящий в комплект поставки языка Go инструмент `cover` позволяет сгенерировать HTML-представление исходного кода, содержащее эту информацию:

```
go tool cover -html=c.out
```

После запуска этой команды ваш браузер отобразит страницу, как показано на рис. 13.1.

В левом верхнем углу расположен выпадающий список, в котором можно выбрать любой из протестированных файлов. Исходный код может быть окрашен в один из трех цветов. Серым цветом выделяется нетестируемый код, зеленым — код, который был покрыт тестами, а красным — код, который не был протестирован. (Если вы читаете печатную версию книги или не различаете красный

и зеленый цвета, то покрытые тестами строки кода будут более светлыми.) В данном случае мы видим, что тестами не покрыта ветвь `default`, выполняемая в том случае, когда нашей функции передается некорректный математический оператор. Добавим этот случай в наш срез тестовых случаев:

```
{"bad_op", 2, 2, "?", 0, `unknown operator ?`},
```

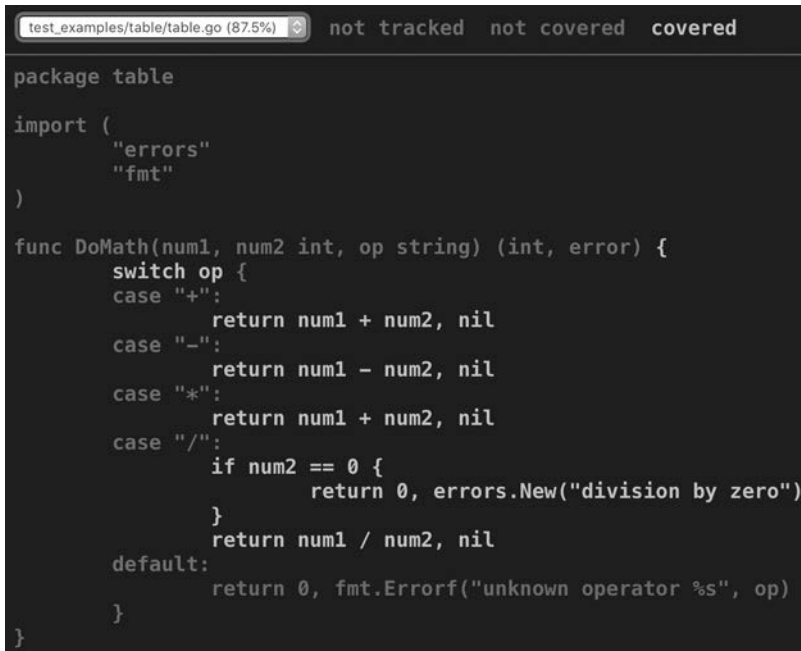


Рис. 13.1. Исходное покрытие кода

Еще раз выполнив команды `go test -v -cover -coverprofile=c.out` и `go tool cover -html=c.out`, мы увидим результат, представленный на рис. 13.2. Теперь последняя строка тоже покрыта тестами и мы имеем 100%-ное покрытие тестируемого кода.

Будучи весьма полезной, информация о покрытии кода все же не является достаточной. Так, несмотря на 100%-ную степень покрытия, наш код в действительности содержит ошибку. Вы заметили ее? Если нет, добавим еще один тестовый случай и выполним тесты еще раз:

```
{"another_mult", 2, 3, "*", 6, ""},
```

Вы увидите следующее сообщение об ошибке:

```
table_test.go:57: Expected 6, got 5
```

```
test_examples/table/table.go (100.0%) not tracked not covered covered

package table

import (
    "errors"
    "fmt"
)

func DoMath(num1, num2 int, op string) (int, error) {
    switch op {
    case "+":
        return num1 + num2, nil
    case "-":
        return num1 - num2, nil
    case "*":
        return num1 + num2, nil
    case "/":
        if num2 == 0 {
            return 0, errors.New("division by zero")
        }
        return num1 / num2, nil
    default:
        return 0, fmt.Errorf("unknown operator %s", op)
    }
}
```

Рис. 13.2. Окончательное покрытие кода

Это объясняется опечаткой в ветви `case` для операции умножения. Вместо операции умножения она производит операцию сложения. (Будьте крайне внимательны, когда используете копирование и вставку при написании кода!) Исправив код и еще раз выполнив команды `go test -v -cover -coverprofile=c.out` и `go tool cover -html=c.out`, мы увидим, что наш код снова успешно проходит проверку.



Сведения о покрытии кода — необходимая, но далеко не достаточная информация. Даже при 100%-ной степени покрытия в вашем коде еще могут оставаться ошибки!

Сравнительные тесты

Оценить код на предмет того, насколько быстро (или медленно) он работает, иногда удивительно сложно. Вместо того чтобы пытаться произвести такую оценку самостоятельно, вы должны использовать поддержку сравнительного тестирования, предоставляемую фреймворком тестирования языка Go. Рассмотрим

вариант использования этих возможностей на примере следующей функции, определенной в пакете `test_examples/bench`:

```
func FileLen(f string, bufsize int) (int, error) {
    file, err := os.Open(f)
    if err != nil {
        return 0, err
    }
    defer file.Close()
    count := 0
    for {
        buf := make([]byte, bufsize)
        num, err := file.Read(buf)
        count += num
        if err != nil {
            break
        }
    }
    return count, nil
}
```

Эта функция подсчитывает количество символов, содержащихся в файле. Она принимает два параметра: имя файла и размер буфера, используемого для чтения файла (зачем нужен второй параметр, я объясню чуть позже).

Перед тем как проверять скорость работы нашей библиотеки, мы должны убедиться в том, что она работает (а она работает). Это можно сделать с помощью следующего простого теста:

```
func TestFileLen(t *testing.T) {
    result, err := FileLen("testdata/data.txt", 1)
    if err != nil {
        t.Fatal(err)
    }
    if result != 65204 {
        t.Error("Expected 65204, got", result)
    }
}
```

Теперь мы можем посмотреть, сколько времени занимает выполнение нашей функции для подсчета длины файла. Мы должны выяснить, какой размер буфера следует использовать для чтения данных из файла.

В Go сравнительные тесты представляют собой функции, определяемые в файлах тестов, которые обладают именем, начинающимся со слова `Benchmark`, и принимают один параметр типа `*testing.B`. Этот тип обладает функциональностью типа `*testing.T`, дополненной поддержкой сравнительного тестирования. Сначала рассмотрим сравнительный тест, в котором используется буфер размером 1 байт:


```
var blackhole int

func BenchmarkFileLen1(b *testing.B) {
    for i := 0; i < b.N; i++ {
        result, err := FileLen("testdata/data.txt", 1)
        if err != nil {
            b.Fatal(err)
        }
        blackhole = result
    }
}
```

Здесь представляет интерес переменная пакетного уровня `blackhole`. Мы заносим в эту переменную результат функции `FileLen`, чтобы компилятор не мог проявить излишнюю сообразительность при проведении оптимизации, убрав вызов функции `FileLen`, что не позволило бы нам провести сравнительное тестирование.



Прежде чем запрыгивать в «кроличью нору» оптимизации, убедитесь в том, что вы нуждаетесь в оптимизации. Если ваша программа уже и так работает достаточно быстро, соответствуя требованиям в отношении времени реакции и используя допустимый объем памяти, то будет лучше потратить имеющееся у вас время на добавление функциональных возможностей и исправление ошибок. Что при этом следует понимать под словами «достаточно быстро» и «допустимый объем памяти», зависит от ваших бизнес-требований.

Каждый сравнительный тест в Go должен содержать цикл, производящий обход значений от 0 до `b.N`. При этом фреймворк тестирования многократно вызывает функцию сравнительного теста, каждый раз передавая ей все большее значение `N`, пока не получит точные данные о времени выполнения. Мы увидим это в следующем результате чуть ниже.

Для выполнения сравнительного теста нужно выполнить команду `go test` с флагом `-bench`. Этот флаг принимает регулярное выражение, указывающее, сравнительные тесты с каким именем необходимо выполнить. С помощью флага `-bench=.` можно запустить все имеющиеся сравнительные тесты. Второй флаг, `-benchmem`, позволяет включить в результат сравнительного теста информацию об объеме выделенной памяти. Все остальные тесты выполняются до запуска сравнительных тестов, поэтому сравнительное тестирование проводится только в случае успешного прохождения остальных проверок.

Вот какой результат выдал данный сравнительный тест на моей машине:

```
BenchmarkFileLen1-12 25 47201025 ns/op 65342 B/op 65208 allocs/op
```

При запуске сравнительного теста с выводом информации об объеме выделенной памяти выводимый результат включает в себя пять столбцов. Эти столбцы содержат следующую информацию.

- **BenchmarkFileLen1-12** — имя теста, дефис и значение **GOMAXPROCS** для данного сравнительного теста.
- **25** — сколько раз потребовалось выполнить тест для получения стабильного результата.
- **47201025 ns/op** — длительность однократного выполнения данного сравнительного теста в наносекундах (в одной секунде содержится 1 000 000 000 наносекунд).
- **65342 B/op** — количество байтов, выделяемых в ходе однократного выполнения данного сравнительного теста.
- **65208 allocs/op** — сколько раз потребовалось выделять байты в куче в ходе однократного выполнения данного сравнительного теста. Эта цифра никогда не превышает количество выделяемых байтов.

Теперь, получив результаты для буфера размером 1 байт, посмотрим, как будут выглядеть результаты при использовании других размеров буфера:

```
func BenchmarkFileLen(b *testing.B) {
    for _, v := range []int{1, 10, 100, 1000, 10000, 100000} {
        b.Run(fmt.Sprintf("FileLen-%d", v), func(b *testing.B) {
            for i := 0; i < b.N; i++ {
                result, err := FileLen("testdata/data.txt", v)
                if err != nil {
                    b.Fatal(err)
                }
                blackhole = result
            }
        })
    }
}
```

Подобно тому как мы запускали табличные тесты с помощью метода `t.Run`, мы запускаем здесь сравнительные тесты с помощью метода `b.Run`, передавая ему разные входные значения. На моей машине этот сравнительный тест выдал следующие результаты:

BenchmarkFileLen/FileLen-1-12 allocs/op	25	47828842 ns/op	65342 B/op	65208
BenchmarkFileLen/FileLen-10-12 allocs/op	230	5136839 ns/op	104488 B/op	6525
BenchmarkFileLen/FileLen-100-12 allocs/op	2246	509619 ns/op	73384 B/op	657

BenchmarkFileLen/FileLen-1000-12 allocs/op	16491	71281	ns/op	68744	B/op	70
BenchmarkFileLen/FileLen-10000-12 allocs/op	42468	26600	ns/op	82056	B/op	11
BenchmarkFileLen/FileLen-100000-12 allocs/op	36700	30473	ns/op	213128	B/op	5

Эти результаты не должны вызывать удивления: по мере увеличения размера буфера выполняется все меньше операций выделения памяти, и наш код работает все быстрее, пока размер буфера не становится большим, чем размер файла. После этого выполнение лишних операций выделения памяти снова замедляет работу программы. Если размер файлов будет примерно таким же, то наилучшую производительность обеспечит буфер размером 10 000 байт.

Однако, добавив одно изменение, мы сможем получить еще более хорошие цифры. Дело в том, что мы заново выделяем память для буфера при извлечении из файла каждого следующего набора байтов, хотя в этом нет необходимости. Вынесем операцию выделения памяти для байтового среза из цикла, разместив ее перед ним. Выполнив сравнительный тест еще раз, мы увидим, что выводимые результаты стали лучше:

BenchmarkFileLen/FileLen-1-12 allocs/op	25	46167597	ns/op	137	B/op	4
BenchmarkFileLen/FileLen-10-12 allocs/op	261	4592019	ns/op	152	B/op	4
BenchmarkFileLen/FileLen-100-12 allocs/op	2518	478838	ns/op	248	B/op	4
BenchmarkFileLen/FileLen-1000-12 allocs/op	20059	60150	ns/op	1160	B/op	4
BenchmarkFileLen/FileLen-10000-12 allocs/op	62992	19000	ns/op	10376	B/op	4
BenchmarkFileLen/FileLen-100000-12 allocs/op	51928	21275	ns/op	106632	B/op	4

Теперь мы всегда получаем одинаковое количество операций выделения памяти, а именно всего четыре операции выделения памяти при любом размере буфера. Что интересно, теперь мы можем подбирать нужное нам соотношение показателей. При наличии ограниченного объема памяти мы можем использовать буфер меньшего размера и таким образом уменьшить расход памяти за счет производительности.

Мы не будем в этой книге обсуждать профилировщик языка Go. В интернете есть много отличных источников информации по этой теме. Хорошей отправной точкой может стать блог-пост Джулии Эванс (Julia Evans) «Профилирование Go-программ с помощью pprof» (<https://oreil.ly/HHe9c>).

ПРОФИЛИРОВАНИЕ СВОЕГО GO-КОДА

Если сравнительное тестирование выявляет наличие проблемы с производительностью или расходом памяти, то следующим шагом должно стать выяснение причин этой проблемы. В «комплект поставки» языка Go входят инструменты профилирования, позволяющие собрать данные об использовании процессора и памяти работающей программой, а также инструменты для визуализации и интерпретации полученных данных. Вы даже можете экспонировать конечную точку веб-сервиса и удаленно собирать данные профилирования из работающего Go-сервиса.

Заглушки в Go

До сих пор мы писали тесты для функций, не зависящих от другого кода, однако такая ситуация является нетипичной, потому что обычно в коде полно зависимостей. Как мы видели в главе 7, абстрагировать вызовы функций в Go можно двумя способами: путем определения функционального типа и путем определения интерфейса. Эти абстракции помогают и в написании модульного окончательного кода, и в написании модульных тестов.



Когда код зависит от абстракций, легче писать модульные тесты!

Рассмотрим пример из пакета `test_examples/solver`. Мы определяем здесь тип `Processor`:

```
type Processor struct {  
    Solver MathSolver  
}
```

Данный тип обладает полем типа `MathSolver`:

```
type MathSolver interface {  
    Resolve(ctx context.Context, expression string) (float64, error)  
}
```

Чуть позже мы реализуем и протестируем тип `MathSolver`.

Тип `Processor` также обладает методом, который считывает выражение из экземпляра типа `io.Reader` и возвращает вычисленное значение:

```
func (p Processor) ProcessExpression(ctx context.Context, r io.Reader)
    (float64, error) {
    curExpression, err := readToNewLine(r)
    if err != nil {
        return 0, err
    }
    if len(curExpression) == 0 {
        return 0, errors.New("no expression to read")
    }
    answer, err := p.Solver.Resolve(ctx, curExpression)
    return answer, err
}
```

Напишем код для тестирования метода `ProcessExpression`. Чтобы написать этот тест, мы сначала должны создать простую реализацию метода `Resolve`:

```
type MathSolverStub struct{}

func (ms MathSolverStub) Resolve(ctx context.Context, expr string)
    (float64, error) {
    switch expr {
    case "2 + 2 * 10":
        return 22, nil
    case "( 2 + 2 ) * 10":
        return 40, nil
    case "( 2 + 2 * 10":
        return 0, errors.New("invalid expression: ( 2 + 2 * 10")
    }
    return 0, nil
}
```

Затем мы напишем модульный тест, использующий эту заглушку (в случае окончательного кода также следует проверить и сообщения об ошибках, но для краткости мы обойдемся здесь без них):

```
func TestProcessorProcessExpression(t *testing.T) {
    p := Processor{MathSolverStub{}}
    in := strings.NewReader(`2 + 2 * 10
( 2 + 2 ) * 10
( 2 + 2 * 10`)
    data := []float64{22, 40, 0, 0}
    for _, d := range data {
        result, err := p.ProcessExpression(context.Background(), in)
        if err != nil {
            t.Error(err)
        }
        if result != d {
            t.Errorf("Expected result %f, got %f", d, result)
        }
    }
}
```

Теперь мы можем запустить свой тест и убедиться в том, что все работает.

Интерфейсы в Go обычно определяют только один или два метода, но иногда бывает и по-другому. В некоторых случаях интерфейс может иметь и большее количество методов. Допустим, что у вас есть интерфейс, который выглядит следующим образом:

```
type Entities interface {
    GetUser(id string) (User, error)
    GetPets(userID string) ([]Pet, error)
    GetChildren(userID string) ([]Person, error)
    GetFriends(userID string) ([]Person, error)
    SaveUser(user User) error
}
```

Существует два паттерна тестирования кода, зависящего от больших интерфейсов. Первый паттерн сводится ко встраиванию интерфейса в структуру. Встраивание интерфейса в структуру обеспечивает автоматическое определение всех методов интерфейса в структуре, не предоставляя реализации этих методов. Поэтому вы должны реализовать те методы, которые вас интересуют в текущем тесте. Допустим, что у вас есть структура `Logic` с полем типа `Entities`:

```
type Logic struct {
    Entities Entities
}
```

Также допустим, что вам нужно протестировать следующий метод:

```
func (l Logic) GetPetNames(userID string) ([]string, error) {
    pets, err := l.Entities.GetPets(userID)
    if err != nil {
        return nil, err
    }
    out := make([]string, len(pets))
    for _, p := range pets {
        out = append(out, p.Name)
    }
    return out, nil
}
```

Этот метод использует только один из методов, объявленных в типе `Entities`, метод `GetPets`. Вместо того чтобы создавать заглушку, реализующую все методы типа `Entities`, только для того, чтобы протестировать метод `GetPets`, вы можете создать в качестве заглушки структуру, реализующую только тот метод, который требуется протестировать:

```
type GetPetNamesStub struct {
    Entities
}
```

```
func (ps GetPetNamesStub) GetPets(userID string) ([]Pet, error) {
    switch userID {
    case "1":
        return []Pet{{Name: "Bubbles"}}, nil
    case "2":
        return []Pet{{Name: "Stampy"}, {Name: "Snowball II"}}, nil
    default:
        return nil, fmt.Errorf("invalid id: %s", userID)
    }
}
```

Теперь мы можем написать свой модульный тест, внедрив заглушку в тип `Logic`:

```
func TestLogicGetPetNames(t *testing.T) {
    data := []struct {
        name      string
        userID     string
        petNames []string
    }{
        {"case1", "1", []string{"Bubbles"}},
        {"case2", "2", []string{"Stampy", "Snowball II"}},
        {"case3", "3", nil},
    }
    l := Logic{GetPetNamesStub{}}
    for _, d := range data {
        t.Run(d.name, func(t *testing.T) {
            petNames, err := l.GetPetNames(d.userID)
            if err != nil {
                t.Error(err)
            }
            if diff := cmp.Diff(d.petNames, petNames); diff != "" {
                t.Error(diff)
            }
        })
    }
}
```

Кстати, метод `GetPetNames` содержит ошибку. Вы заметили ее? Даже простые методы иногда содержат ошибки.



При встраивании интерфейса в структуру, используемую в качестве заглушки, не забудьте реализовать все те методы, которые вы будете вызывать в ходе тестирования! Ваши тесты выдадут панику, если в них будет вызываться нереализованный метод.

Этот подход хорошо работает в том случае, когда нужно реализовать только один или два метода интерфейса для отдельного теста. Однако его неудобно использовать в том случае, когда нужно вызывать один и тот же метод в разных тестах с использованием разных входных данных и получением разных результатов.

В таком случае нужно включить все возможные результаты для каждого теста в одну реализацию или создать отдельные реализации структуры для каждого теста. Такой подход быстро становится трудным в понимании и сопровождении. Более удачное решение сводится к тому, чтобы создать в качестве заглушки структуру, которая будет отображать вызовы методов на поля функционального типа. Для каждого метода, определенного в типе `Entities`, мы должны определить в структуре-заглушке поля функционального типа с такой же сигнатурой:

```
type EntitiesStub struct {
    getUser      func(id string) (User, error)
    getPets      func(userID string) ([]Pet, error)
    getChildren  func(userID string) ([]Person, error)
    getFriends   func(userID string) ([]Person, error)
    saveUser     func(user User) error
}
```

После этого следует обеспечить соответствие типа `EntitiesStub` интерфейсу `Entities` путем определения методов. В каждом методе мы будем вызывать соответствующее функциональное поле. Например:

```
func (es EntitiesStub) GetUser(id string) (User, error) {
    return es.getUser(id)
}

func (es EntitiesStub) GetPets(userID string) ([]Pet, error) {
    return es.getPets(userID)
}
```

После создания этой заглушки мы можем указывать разные реализации методов в разных тестовых случаях, используя для этого поля структуры с данными табличного теста:

```
func TestLogicGetPetNames(t *testing.T) {
    data := []struct {
        name      string
        getPets   func(userID string) ([]Pet, error)
        userID    string
        petNames  []string
        errMsg    string
    }{
        {"case1", func(userID string) ([]Pet, error) {
            return []Pet{{Name: "Bubbles"}}, nil
        }, "1", []string{"Bubbles"}, ""},
        {"case2", func(userID string) ([]Pet, error) {
            return nil, errors.New("invalid id: 3")
        }, "3", nil, "invalid id: 3"},
    }
    l := Logic{}
    for _, d := range data {
```



```

t.Run(d.name, func(t *testing.T) {
    l.Entities = EntitiesStub{getPets: d.getPets}
    petNames, err := l.GetPetNames(d.userID)
    if diff := cmp.Diff(petNames, d.petNames); diff != "" {
        t.Error(diff)
    }
    var errMsg string
    if err != nil {
        errMsg = err.Error()
    }
    if errMsg != d.errMsg {
        t.Errorf("Expected error `%s`, got `%s`", d.errMsg, errMsg)
    }
})
}
}

```

Мы добавляем поле функционального типа в одну из анонимных структур среза `data`. Для каждого тестового случая мы указываем функцию, возвращающую те же данные, которые возвратил бы метод `GetPets`. При таком подходе к созданию тестовых заглушек вы ясно видите, что будет возвращать заглушка для каждого тестового случая. При выполнении каждого теста создается новый экземпляр типа `EntitiesStub` с присвоением функционального поля `getPets`, определенного в тестовых данных, функциональному полю `getPets`, определенному в экземпляре `EntitiesStub`.

ИМИТАЦИИ И ЗАГЛУШКИ

Хотя под словами «имитация» (`mock`) и «заглушка» (`stub`) часто понимается одно и то же, в действительности это разные концепции. Мартин Фаулер (Martin Fowler), авторитетный эксперт во всем, что касается программной разработки, посвятил имитациям один из своих блог-постов (<https://oreil.ly/nDkF5>), где, помимо прочего, коснулся и вопроса разницы между имитацией и заглушкой. Если выразить это в двух словах, то заглушка возвращает готовое значение для определенных входных данных, а имитация позволяет убедиться в том, что некоторый набор вызовов производится в ожидаемом порядке с ожидаемыми входными данными.

В приведенных выше примерах использовались заглушки, возвращающие готовые значения для определенных данных. Вы также можете вручную написать собственные имитации или сгенерировать их, используя одну из сторонних библиотек. Наиболее популярными из них являются библиотека `gomock` от компании Google (https://oreil.ly/_EjoS) и библиотека `testify` от компании Stretchr (<https://oreil.ly/AfDGD>).

Пакет `httptest`

Когда тестируемая функция вызывает HTTP-сервис, написание тестов может вызывать определенные затруднения. Обычно при этом выполняют интеграционный тест, что подразумевает развертывание тестового экземпляра вызываемого функцией сервиса. В состав стандартной библиотеки языка Go входит пакет `net/http/httptest`, который упрощает создание заглушек, заменяющих HTTP-сервисы. Вернемся к нашему пакету `test_examples/solver` и предоставим реализацию типа `MathSolver`, вызывающую HTTP-сервис для вычисления выражений:

```
type RemoteSolver struct {
    MathServerURL string
    Client         *http.Client
}

func (rs RemoteSolver) Resolve(ctx context.Context, expression string)
    (float64, error) {
    req, err := http.NewRequestWithContext(ctx, http.MethodGet,
        rs.MathServerURL+"?expression="+url.QueryEscape(expression),
        nil)
    if err != nil {
        return 0, err
    }
    resp, err := rs.Client.Do(req)
    if err != nil {
        return 0, err
    }
    defer resp.Body.Close()
    contents, err := ioutil.ReadAll(resp.Body)
    if err != nil {
        return 0, err
    }
    if resp.StatusCode != http.StatusOK {
        return 0, errors.New(string(contents))
    }
    result, err := strconv.ParseFloat(string(contents), 64)
    if err != nil {
        return 0, err
    }
    return result, nil
}
```

Теперь посмотрим, как можно использовать библиотеку `httptest` для тестирования кода без развертывания сервера. Соответствующая функция тестирования `TestRemoteSolver_Resolve` определена в файле `test_examples/solver/remote_solver_test.go`; я затрону здесь только основные моменты. Прежде всего мы должны убедиться в том, что передаваемые в функцию данные поступают

на сервер. Для этого мы определяем в нашей функции тестирования тип `info` для размещения входных и выходных данных и переменную `io`, которая будет содержать текущие входные и выходные данные:

```
type info struct {
    expression string
    code        int
    body        string
}
var io info
```

Затем мы настраиваем свою имитацию удаленного сервера и используем ее для настройки экземпляра типа `RemoteSolver`:

```
server := httptest.NewServer(
    http.HandlerFunc(func(rw http.ResponseWriter, req *http.Request) {
        expression := req.URL.Query().Get("expression")
        if expression != io.expression {
            rw.WriteHeader(http.StatusBadRequest)
            rw.Write([]byte("invalid expression: " + io.expression))
            return
        }
        rw.WriteHeader(io.code)
        rw.Write([]byte(io.body))
    }))
defer server.Close()
rs := RemoteSolver{
    MathServerURL: server.URL,
    Client:        server.Client(),
}
```

Функция `httptest.NewServer` создает и запускает HTTP-сервер на случайно выбранном неиспользуемом порте. Для обработки запроса нужно предоставить реализацию интерфейса `http.Handler`. Поскольку это сервер, его необходимо закрывать после выполнения теста. Экземпляр `server` содержит URL-адрес и предварительно настроенный HTTP-клиент для взаимодействия с тестовым сервером. Мы передаем эти данные в экземпляр типа `RemoteSolver`.

Остальная часть функции работает так же, как рассмотренные нами ранее табличные тесты:

```
data := []struct {
    name  string
    io    info
    result float64
}{
    {"case1", info{"2 + 2 * 10", http.StatusOK, "22"}, 22},
    // остальные случаи
}
for _, d := range data {
```

```
t.Run(d.name, func(t *testing.T) {
    io = d.io
    result, err := rs.Resolve(context.Background(), d.io.expression)
    if result != d.result {
        t.Errorf("io `%f`, got `%f`", d.result, result)
    }
    var errMsg string
    if err != nil {
        errMsg = err.Error()
    }
    if errMsg != d.errMsg {
        t.Errorf("io error `%s`, got `%s`", d.errMsg, errMsg)
    }
})
}
```

Интересным моментом здесь является то, что переменную `io` перехватывают сразу два замыкания: замыкание сервера-заглушки и замыкание для выполнения каждого теста. В одном замыкании мы производим запись в нее, а в другом — чтение из нее. Хотя такой подход будет неуместен в окончательном коде приложения, его с успехом можно использовать в коде тестирования в пределах одной функции.

Интеграционные тесты и теги сборки

Несмотря на то что пакет `httptest` позволяет провести тестирование без использования внешних сервисов, вам все равно следует написать *интеграционные тесты* — автоматизированные тесты, производящие подключение к другим сервисам. Они позволяют вам убедиться в том, что вы имеете правильное представление об API используемого сервиса. Здесь важно правильно сгруппировать автоматизированные тесты, поскольку интеграционные тесты должны запускаться только при наличии необходимой вспомогательной среды. Кроме того, интеграционные тесты обычно выполняются реже, поскольку их выполнение, как правило, занимает больше времени, чем выполнение модульных тестов.

Компилятор языка Go позволяет вам указать, когда следует компилировать код, с помощью так называемых *тегов сборки*. Теги сборки определяются в первой строке файла с помощью магического комментария, начинающегося выражением `// +build`. Изначально теги сборки планировалось использовать для компилирования разного кода для разных платформ, но, как оказалось, их также удобно использовать и для разделения тестов на группы. Тесты, размещенные в файлах без тега сборки, выполняются всегда. Это модульные тесты, которые не имеют зависимостей от внешних ресурсов. Тесты, размещенные в файлах с тегом сборки, выполняются только при наличии вспомогательных ресурсов.

Посмотрим, как это будет выглядеть в случае нашего кода для математических вычислений. Скачайте реализацию сервера посредством приложения Docker, выполнив команду `docker pull jonbodner/math-server`, а затем запустите сервер локально на порте 8080 с помощью команды `docker run -p 8080:8080 jonbodner/math-server`.



Если у вас не установлено приложение Docker или если вы хотите скомпилировать этот код самостоятельно, вы можете найти его на сайте GitHub (<https://oreil.ly/yjMzc>).

ИСПОЛЬЗОВАНИЕ ФЛАГА -SHORT

Еще один подход сводится к тому, чтобы использовать команду `go test` с флагом `-short`. Если вам иногда нужно обходиться без выполнения долго работающих тестов, пометьте эти тесты путем размещения следующего кода в начале функции тестирования:

```
if testing.Short() {  
    t.Skip("skipping test in short mode.")  
}
```

В тех случаях, когда нужно выполнять только короткие тесты, используйте команду `go test` с флагом `-short`.

У данного подхода есть несколько недостатков. При использовании флага `-short` вы имеете только два уровня тестирования: выполнение коротких тестов и выполнение всех тестов. Теги сборки, с другой стороны, позволяют вам сгруппировать интеграционные тесты, указав, какой сервис требуется для их выполнения. Второй аргумент против использования флага `-short` для выделения интеграционных тестов носит философский характер. Если теги сборки определяют зависимость, то флаг `-short` лишь указывает, что не должны выполняться долго выполняющиеся тесты. Первое и второе — совершенно разные концепции. Наконец, на мой взгляд, в использовании флага `-short` нет логики. Короткие тесты нужно выполнять всегда, и было бы логичнее использовать флаг, позволяющий не *исключить*, а *включить* долго выполняющиеся тесты в число выполняемых тестов.

Мы должны написать интеграционный тест, чтобы убедиться в том, что наш метод `Resolve` должным образом взаимодействует с сервером математических вычислений. Соответствующая функция тестирования, `TestRemoteSolver_`

`ResolveIntegration`, определена в файле `test_examples/solver/remote_solver_integration_test.go`. Данный тест выглядит точно так же, как любой из табличных тестов, написанных нами ранее. Интерес представляет лишь первая строка этого файла, отделенная от объявления пакета пустой строкой:

```
// +build integration
```

Чтобы запустить этот интеграционный тест вместе с остальными тестами, нужно выполнить следующую команду:

```
$ go test -tags integration -v ./...
```

Выявление проблем конкурентности с помощью детектора состояний гонки

Даже при использовании встроенной поддержки конкурентности языка Go вы иногда будете допускать ошибки. Так, например, вы можете ненамеренно обратиться к одной и той же переменной в двух разных горутинах без установки блокировки. В терминах компьютерных технологий это называется *состоянием гонки*. Для облегчения поиска таких ошибок в «комплект поставки» языка Go включен *детектор состояний гонки*. Он не гарантирует выявление абсолютно всех имеющихся состояний гонки, но, когда он находит такое состояние, вы должны обеспечить в этом месте установку надлежащей блокировки.

Рассмотрим простой пример кода, сохраненный в файле `test_examples/race/race.go`:

```
func getCounter() int {
    var counter int
    var wg sync.WaitGroup
    wg.Add(5)
    for i := 0; i < 5; i++ {
        go func() {
            for i := 0; i < 1000; i++ {
                counter++
            }
            wg.Done()
        }()
    }
    wg.Wait()
    return counter
}
```

Этот код запускает пять горутин, ждет, пока каждая из них 1000 раз обновит общую переменную `counter`, а затем возвращает результат. В итоге эта пере-

менная должна содержать значение 5000, поэтому убедимся в этом, используя модульный тест, который определен в файле `test_examples/race/race_test.go`:

```
func TestGetCounter(t *testing.T) {
    counter := getCounter()
    if counter != 5000 {
        t.Error("unexpected counter:", counter)
    }
}
```

Выполнив команду `go test` несколько раз, вы увидите, что иногда этот тест выполняется успешно, но в большинстве случаев выдает отрицательный результат с сообщением об ошибке следующего вида:

```
unexpected counter: 3673
```

Причиной этой проблемы является наличие в коде состояния гонки. В столь простой программе мы легко можем понять, чем это вызвано: несколько горутин пытаются обновлять переменную `counter` одновременно, и часть этих обновлений теряется. В более сложных программах выявление таких состояний гонки дается труднее. Посмотрим, что в данном случае сделает детектор состояний гонки. Его можно активизировать, используя команду `go test` с флагом `-race`:

```
$ go test -race
=====
WARNING: DATA RACE
Read at 0x00c000128070 by goroutine 10:
  test_examples/race.getCounter.func1()
    test_examples/race/race.go:12 +0x45

Previous write at 0x00c000128070 by goroutine 8:
  test_examples/race.getCounter.func1()
    test_examples/race/race.go:12 +0x5b
```

Из этого сообщения можно понять, что источником наших проблем является строка `counter++`.



Некоторые разработчики пытаются устранить состояние гонки путем вставки в код периодов ожидания, чтобы разнести по времени операции доступа к переменной, используемой в нескольких горутинках. Это крайне неудачная идея. Это может устранить проблему в некоторых случаях, но ваш код по-прежнему останется некорректным и иногда будет давать сбой.

Флаг `-race` также можно использовать и при компиляции программ. При этом создается двоичный файл, который включает в себя детектор состояний гонки и выводит в консоль информацию о выявленных состояниях гонки. Это позволяет выявлять состояния гонки в коде, для которого вы не создали тесты.

Если детектор состояний гонки настолько полезен, то почему же он не всегда используется при тестировании и компиляции окончательного кода? Дело в том, что двоичный файл с активизированным детектором состояний гонки работает примерно в десять раз медленнее, чем обычный двоичный файл. Это не представляет проблемы в том случае, когда тестовый набор выполняется в течение нескольких секунд, но в случае больших тестовых наборов, выполнение которых и так занимает несколько минут, десятикратное замедление будет слишком заметным снижением производительности.

Резюме

В этой главе вы научились писать тесты и повышать качество кода, используя встроенную в Go поддержку тестирования, контроля степени покрытия кода, сравнительного тестирования и выявления состояний гонки. В следующей главе мы рассмотрим компоненты языка Go, позволяющие выйти за рамки установленных правил: пакеты `unsafe`, `reflect` и `cgo`.

Здесь водятся драконы: пакеты `reflect`, `unsafe` и `sgo`

За пределами изученного мира вас встречает пугающая неизвестность. На древних картах неизученные территории обозначались с помощью изображений драконов и львов. В предыдущих главах особо подчеркивался тот факт, что Go — безопасный язык, который использует типизированные переменные для четкого определения используемых данных и сборку мусора для управления памятью. Даже указатели становятся в этом языке «ручными», что исключает все те возможности их неправильного использования, которые имеются в C и C++.

Все это, конечно, верно, и в подавляющем большинстве случаев вы можете писать Go-код с уверенностью в том, что среда выполнения языка Go защитит вас от неожиданностей, но при этом в этом языке предусмотрены и обходные лазейки. Иногда Go-программам требуется «сделать вылазку» в области, отличающиеся меньшей степенью определенности. В этой главе мы поговорим о том, каким образом можно решить задачу в тех случаях, когда она не может быть решена с помощью стандартного Go-кода. Например, если тип данных невозможно определить на этапе компиляции, вы можете обеспечить взаимодействие с данными и даже их конструирование, используя средства поддержки рефлексии из пакета `reflect`. Если нужно воспользоваться теми преимуществами, которые дает вам схема размещения в памяти типов языка Go, вы можете использовать пакет `unsafe`. А если некоторая функциональность может быть обеспечена только с помощью библиотек, написанных на языке C, вы можете производить вызов C-кода с помощью пакета `sgo`.

Возможно, вас удивляет, почему я решил включить столь сложные темы в книгу, рассчитанную на тех, кто еще только знакомится с языком Go. На это есть две причины. Во-первых, при поиске решения задачи разработчики иногда находят и используют (путем копирования и вставки) не вполне понятные им способы решения. Будет лучше, если вы получите хотя бы общее представление о том,

какие проблемы могут возникать при использовании продвинутых приемов, до того, как вы примените их в своей кодовой базе. Во-вторых, использование этих инструментов может быть очень увлекательным процессом. Поскольку они позволяют сделать то, что обычно невозможно сделать в Go, часто интересно поиграться с ними и посмотреть, что это вам дает.

Рефлексия позволяет нам работать с типами на этапе выполнения

Одной из наиболее привлекательных особенностей языка Go является статическая типизация. В большинстве случаев объявление переменных, типов и функций в Go не представляет никаких проблем. Когда вам требуется тип, переменная или функция, вы просто определяете их следующим образом:

```
type Foo struct {  
    A int  
    B string  
}  
  
var x Foo  
  
func DoSomething(f Foo) {  
    fmt.Println(f.A, f.B)  
}
```

Мы используем типы для представления структур данных, о необходимости которых мы знаем на этапе написания программы. Поскольку типы являются неотъемлемой составляющей языка Go, компилятор использует их для проверки корректности создаваемого нами кода. Однако иногда использование только информации, известной на этапе компиляции, является ограничением. Иногда требуется работать с переменными на этапе выполнения, используя информацию, которой еще не было на момент написания программы. Например, вам нужно обеспечить занесение в переменную данных из файла или сетевого запроса либо создать функцию, способную работать с разными типами. В этих случаях вы должны использовать *рефлексию*. Рефлексия позволяет нам исследовать типы на этапе выполнения. Она также позволяет нам изучать, модифицировать и создавать переменные, функции и структуры на этапе выполнения.

Это подводит нас к вопросу о том, в каких случаях нужна эта функциональность. Общее представление об этом можно получить, взглянув на содержимое стандартной библиотеки Go. Области применения рефлексии можно разделить на следующие основные категории.

- Чтение и запись из базы данных. Пакет `database/sql` использует рефлексиию для того, чтобы отправлять записи в базы данных и, наоборот, производить чтение записей из баз данных.
- Встроенные библиотеки шаблонизации языка Go, `text/template` и `html/template`, используют рефлексиию для обработки значений, передаваемых шаблонам.
- Пакет `fmt` активно использует рефлексиию для определения типа предоставляемых параметров при вызове функции `fmt.Println` и других подобных функций.
- Пакет `errors` использует рефлексиию для реализации функций `errors.Is` и `errors.As`.
- Пакет `sort` использует рефлексиию для реализации функций для сортировки и проверки содержимого срезов любого типа: `sort.Slice`, `sort.SliceStable` и `sort.SliceIsSorted`.
- Последней основной областью применения рефлексии в стандартной библиотеке является маршализация данных в форматы JSON, XML и ряд других форматов, определенных в различных пакетах `encoding`, и демаршализация данных из этих форматов. Рефлексия используется для доступа к тегам структур (о которых мы поговорим чуть ниже), а также для чтения и записи соответствующих полей структур.

Общей особенностью этих примеров является то, что в большинстве этих случаев вы имеете дело с использованием или форматированием данных, которые импортируются в Go-программу или экспортируются из нее. Рефлексия часто используется на границе между программой и внешним миром.



Еще одной областью применения пакета `reflect` из стандартной библиотеки языка Go является тестирование. В разделе «Срезы» на с. 59 упоминалось о том, что пакет `reflect` содержит функцию `DeepEqual`. Эта функция включена в состав пакета `reflect` по той причине, что для выполнения своей работы она использует рефлексиию. Функция `reflect.DeepEqual` проверяет, являются ли два значения «глубоко равными» друг другу. При этом выполняется более тщательное сравнение, чем при использовании оператора `==`, и такой способ сравнения применяется в стандартной библиотеке для проверки результатов тестирования, а также для сравнения вещей, которые нельзя сравнить с помощью оператора `==`, таких как срезы и карты.

Хотя в большинстве случаев можно обойтись без использования функции `DeepEqual`, она прекрасно подойдет, когда нужно сравнить две карты, убедившись в идентичности их ключей и значений, или убедиться в идентичности двух срезов.

Типы, разновидности и значения

Теперь, когда мы уже знаем, что такое рефлексия и когда она может понадобиться, разберемся с тем, как все это работает. Типы и функции, реализующие рефлексия в Go, определены в пакете `reflect` стандартной библиотеки. Механизм действия рефлексии опирается на следующие три концепции: типы, разновидности типов и значения.

Сначала уясним, что такое типы. В контексте рефлексии тип является именно тем, что означает это слово. То есть он определяет, какими свойствами обладает переменная, какие значения она может содержать и как с ней можно взаимодействовать. При использовании рефлексии вы можете запросить у типа информацию об этих свойствах с помощью кода.

Типы и разновидности

Получить рефлексивное представление типа переменной можно с помощью функции `TypeOf` из пакета `reflect`:

```
vType := reflect.TypeOf(v)
```

Функция `reflect.TypeOf` возвращает значение типа `reflect.Type`, представляющее тип переданной в эту функцию переменной. Тип `reflect.Type` обладает рядом методов, возвращающих информацию о типе переменной. Мы не можем рассмотреть здесь все эти методы, но рассмотрим наиболее важные из них.

Для вас не должно быть сюрпризом, что метод `Name` возвращает имя типа. Рассмотрим небольшой пример.

```
var x int
xt := reflect.TypeOf(x)
fmt.Println(xt.Name())    // возвращает "int"
f := Foo{}
ft := reflect.TypeOf(f)
fmt.Println(ft.Name())    // возвращает "Foo"
xpt := reflect.TypeOf(&x)
fmt.Println(xpt.Name())   // возвращает пустую строку
```

Мы начинаем здесь с переменной `x` типа `int`; передаем ее в функцию `reflect.TypeOf` и получаем обратно экземпляр типа `reflect.Type`. В случае простых типов, таких как `int`, метод `Name()` возвращает имя типа; в данном случае это строка `"int"` для типа `int`. В случае структуры этот метод возвращает имя структуры. Некоторые типы, например срезы и указатели, не обладают именем; в таком случае метод `Name` возвращает пустую строку.

Метод `Kind` типа `reflect.Type` возвращает значение типа `reflect.Kind`, представляющее собой константу, которая указывает, на основе чего создан тип — на основе среза, карты, указателя, структуры, интерфейса, строки, массива, функции, типа `int` или какого-либо другого простого типа. При этом иногда сложно понять, в чем состоит разница между типом и разновидностью типа. Запомните следующее правило: если вы определяете структуру с именем `Foo`, то она обладает разновидностью `reflect.Struct` и типом `Foo`.

Разновидности типов играют очень важную роль. При использовании рефлексии следует помнить о том, что любой код из пакета `reflect` исходит из предположения, что вы знаете, что делаете. Некоторые из методов типа `reflect.Type` и других типов пакета `reflect` имеют смысл только для определенных разновидностей типов. Так, например, у типа `reflect.Type` есть метод `NumIn`. Если ваш экземпляр типа `reflect.Type` представляет функцию, то этот метод возвратит количество входных параметров, передаваемых этой функции. Если же ваш экземпляр типа `reflect.Type` не является функцией, то при вызове метода `NumIn` программа выдаст панику.



Обычно при вызове метода, бессмысленного для имеющейся разновидности типа, этот метод выдает панику. Поэтому не забывайте выяснять, какие методы будут работать, а какие — выдавать панику, используя разновидность рефлекслируемого типа.

Еще одним важным методом типа `reflect.Type` является метод `Elem`. Некоторые типы в Go содержат ссылки на другие типы, и метод `Elem` позволяет выяснить, что представляют собой эти вложенные типы. Например, вызовем функцию `reflect.TypeOf`, передав ей указатель на тип `int`:

```
var x int
xpt := reflect.TypeOf(&x)
fmt.Println(xpt.Name())           // возвращает пустую строку
fmt.Println(xpt.Kind())           // возвращает reflect.Ptr
fmt.Println(xpt.Elem().Name())    // возвращает "int"
fmt.Println(xpt.Elem().Kind())    // возвращает reflect.Int
```

В результате мы получим экземпляр типа `reflect.Type` с пустой строкой вместо имени и разновидностью `reflect.Ptr`, что подразумевает указатель. Когда передаваемый на вход экземпляр типа `reflect.Type` представляет указатель, метод `Elem` возвращает экземпляр типа `reflect.Type`, представляющий тот тип, на который указывает этот указатель. В данном случае метод `Name` возвращает строку `int`, а метод `Kind` — разновидность `reflect.Int`. Метод `Elem` также можно использовать для срезов, карт, каналов и массивов.

У типа `reflect.Type` также есть методы для применения рефлексии при работе со структурами. Метод `NumField` позволяет узнать, сколько полей содержит

структура, а метод `Field` — извлечь поле структуры по индексу. Вторым методом возвращает структуру каждого поля, как ее определяет тип `reflect.StructField`, что включает в себя имя, порядок, тип и имеющиеся в поле теги структур. Рассмотрим небольшой пример, который вы можете запустить в онлайн-песочнице (https://oreil.ly/Ynv_4):

```
type Foo struct {
    A int    `myTag:"value"`
    B string `myTag:"value2"`
}

var f Foo
ft := reflect.TypeOf(f)
for i := 0; i < ft.NumField(); i++ {
    curField := ft.Field(i)
    fmt.Println(curField.Name, curField.Type.Name(),
        curField.Tag.Get("myTag"))
}
```

Мы создаем экземпляр типа `Foo` и с помощью функции `reflect.TypeOf` получаем экземпляр типа `reflect.Type` для переменной `f`. Затем с помощью метода `NumField` настраиваем цикл `for` таким образом, чтобы он обошел индексы всех полей переменной `f`. Далее с помощью метода `Field` получаем структуру `reflect.StructField`, представляющую отдельное поле. После этого мы можем использовать поля структуры `reflect.StructField` для получения дополнительной информации о поле. Этот код выдает следующий результат:

```
A int value
B string value2
```

У типа `reflect.Type` есть еще много других методов, но все они используют тот же паттерн, позволяя нам получать определенную информацию о типе переменной. Более подробную информацию о типе `reflect.Type` стандартной библиотеки можно найти в соответствующей документации (<https://oreil.ly/p4AZ6>).

Значения

Рефлексию можно использовать не только для выяснения типа переменных, но и для чтения значений переменных, присвоения им значений и создания с нуля новых значений.

С помощью функции `reflect.ValueOf` можно создать экземпляр типа `reflect.Value`, представляющий значение переменной:

```
vValue := reflect.ValueOf(v)
```

Так как каждая переменная в Go обладает типом, у типа `reflect.Value` есть метод `Type`, который возвращает экземпляр типа `reflect.Type` для экземпляра типа `reflect.Value`. Как и у типа `reflect.Type`, у типа `reflect.Value` также есть метод `Kind`.

Подобно тому как у типа `reflect.Type` есть методы для получения информации о типе переменной, у типа `reflect.Value` есть методы для получения информации о значении переменной. Мы не будем рассматривать здесь все эти методы, но посмотрим, как можно использовать тип `reflect.Value` для работы со значениями переменных.

Для начала разберемся с тем, как производится считывание значений из экземпляров типа `reflect.Value`. Метод `Interface` возвращает значение переменной как пустой интерфейс. При этом теряется информация о типе, и поэтому при занесении возвращаемого значения в переменную его нужно снова привести к правильному типу с помощью операции утверждения типа:

```
s := []string{"a", "b", "c"}
sv := reflect.ValueOf(s)           // переменная sv обладает типом reflect.Value
s2 := sv.Interface().([]string)    // переменная s2 обладает типом []string
```

Метод `Interface` можно вызывать для экземпляров типа `reflect.Value`, содержащих значения любого типа, однако имеются и специализированные методы для тех случаев, когда переменная относится к одному из встроенных простых типов: `Bool`, `Complex`, `Int`, `Uint`, `Float` и `String`. Есть также и метод `Bytes` для того случая, когда переменная представляет собой байтовый срез. При вызове метода, не соответствующего типу значения, содержащегося в экземпляре типа `reflect.Value`, ваш код выдаст панику.

Рефлексию также можно использовать и для присвоения значения переменной, однако эта операция выполняется в три этапа. Сначала нужно передать указатель на переменную в функцию `reflect.ValueOf`, которая возвратит экземпляр типа `reflect.Value`, представляющий этот указатель:

```
i := 10
iv := reflect.ValueOf(&i)
```

Затем необходимо добраться непосредственно до того значения, которое вам нужно поменять. Вызвав метод `Elem` в экземпляре типа `reflect.Value`, мы можем получить то значение, на которое указывает указатель, переданный в функцию `reflect.ValueOf`. Подобно тому как метод `Elem` типа `reflect.Type` возвращает тип, на который указывает вмещающий тип, метод `Elem` типа `reflect.Value` возвращает значение, на которое указывает указатель, или значение, содержащееся в экземпляре интерфейса:

```
ivv := iv.Elem()
```

Теперь осталось непосредственно применить метод, используемый для установки значения. Наряду со специализированными методами для чтения простых типов в пакете `reflect` имеются и специализированные методы для установки значений простых типов: `SetBool`, `SetInt`, `SetFloat`, `SetString` и `SetUint`. В рассматриваемом нами случае мы можем изменить значение переменной `i`, произведя следующий вызов метода: `ivv.SetInt(20)`. Если мы выведем значение переменной `i`, то увидим, что теперь оно равно 20:

```
ivv.SetInt(20)
fmt.Println(i) // выводит 20
```

В случае всех остальных типов следует использовать метод `Set`, который принимает переменную типа `reflect.Value`. При этом присваиваемое значение может не быть указателем, потому что мы просто считываем это значение, не изменяя его. И подобно тому, как метод `Interface()`, помимо прочего, может использоваться для чтения простых типов, метод `Set` также может использоваться для записи простых типов.

Тот факт, что мы должны передавать указатель в функцию `reflect.ValueOf` для изменения значения входного параметра, объясняется тем, что эта функция ведет себя так же, как любая другая функция в языке Go. Как упоминалось в разделе «Указатели служат для указания изменяемых параметров» на с. 146, использование параметров указательного типа означает, что функция должна модифицировать значение параметра. Эта модификация производится путем разыменования указателя и присвоения значения. Так, например, следующие две функции производят одни и те же действия:

```
func changeInt(i *int) {
    *i = 20
}

func changeIntReflect(i *int) {
    iv := reflect.ValueOf(i)
    iv.Elem().SetInt(20)
}
```



Если переданный в функцию `reflect.ValueOf` параметр не будет представлять собой указатель, вы по-прежнему сможете прочитать значение переменной с помощью рефлексии. Но если вы при этом попытаетесь использовать один из тех методов, которые изменяют значение переменной, это (что неудивительно) приведет к панике.

Создание новых значений

Прежде чем переходить к разговору о том, как лучше использовать рефлекссию, мы должны узнать еще одну вещь: как создавать новые значения. Для этой цели служит функция `reflect.New` — рефлексивный аналог функции `new`. Она принимает экземпляр типа `reflect.Type` и возвращает экземпляр типа `reflect.Value`, представляющий собой указатель на экземпляр типа `reflect.Value`, соответствующий указанному типу. Поскольку это указатель, вы можете модифицировать его и присвоить модифицированное значение переменной с помощью метода `Interface`.

Подобно тому как метод `reflect.New` можно использовать для создания указателя на скалярный тип, вы также можете применять рефлекссию для создания тех же вещей, что и ключевое слово `make`, используя следующие функции:

```
func MakeChan(typ Type, buffer int) Value

func MakeMap(typ Type) Value

func MakeMapWithSize(typ Type, n int) Value

func MakeSlice(typ Type, len, cap int) Value
```

Эти функции принимают экземпляр типа `reflect.Type`, который вместо вложенного типа представляет составной тип.

Конструирование экземпляра типа `reflect.Type` всегда следует начинать со значения. Однако если у вас нет значения, то для создания переменной, представляющей экземпляр типа `reflect.Type`, можно использовать следующий хитрый прием:

```
var stringType = reflect.TypeOf((*string)(nil)).Elem()

var stringSliceType = reflect.TypeOf([]string(nil))
```

Переменная `stringType` содержит экземпляр типа `reflect.Type`, представляющий тип `string`, а переменная `stringSliceType` — экземпляр типа `reflect.Type`, представляющий тип `[]string`. Разобраться в том, что делает первая строка, не так уж просто. Мы преобразуем здесь значение `nil` в указатель на тип `string` и используем функцию `reflect.TypeOf` для создания экземпляра типа `reflect.Type` для этого указательного типа, после чего вызываем в этом экземпляре метод `Elem`, чтобы получить базовый тип. В силу используемого в Go порядка выполнения операций мы заключили `*string` в скобки, чтобы компилятор не посчитал, что мы хотим преобразовать значение `nil` в значение типа `string`, что является недопустимой операцией.

В случае с переменной `stringSliceType` дело обстоит чуть проще, потому что срез может быть равным значению `nil`. Нам остается лишь привести значение `nil` к типу `[]string` и передать его в функцию `reflect.TypeOf`.

Теперь, уже располагая этими типами, мы можем воспользоваться методами `reflect.New` и `reflect.MakeSlice`, как показано ниже:

```
ssv := reflect.MakeSlice(stringSliceType, 0, 10)

sv := reflect.New(stringType).Elem()
sv.SetString("hello")

ssv = reflect.Append(ssv, sv)
ss := ssv.Interface().([]string)
fmt.Println(ss) // выводит [hello]
```

Вы можете запустить этот пример кода в онлайн-песочнице (<https://oreil.ly/ak2PG>).

Используйте рефлексии для проверки значения интерфейса на равенство значению nil

Как упоминалось в разделе «Интерфейсы и значение `nil`» на с. 186, если вы присвоите равную `nil` переменную конкретного типа переменной интерфейсного типа, эта переменная интерфейсного типа не будет равна `nil`. Это объясняется тем, что с интерфейсной переменной ассоциируется некоторый тип. Если вам нужно проверить, равно ли значению `nil` ассоциированное с интерфейсом значение, то это можно сделать с помощью рефлексии и методов `IsValid` и `IsNil`:

```
func hasNoValue(i interface{}) bool {
    iv := reflect.ValueOf(i)
    if !iv.IsValid() {
        return true
    }
    switch iv.Kind() {
    case reflect.Ptr, reflect.Slice, reflect.Map, reflect.Func,
        reflect.Interface:
        return iv.IsNil()
    default:
        return false
    }
}
```

Метод `IsValid` возвращает значение `true`, если экземпляр типа `reflect.Value` содержит любое другое значение, кроме интерфейса, равного `nil`. Это нужно проверять в первую очередь, потому что, когда метод `IsValid` выдает значение `false`, вызов любого другого метода типа `reflect.Value` (что неудивительно)

выдает панику. Метод `IsNil` возвращает значение `true`, если экземпляр типа `reflect.Value` содержит значение `nil`, но его можно использовать лишь в том случае, когда разновидность типа (`reflect.Kind`) допускает равенство значению `nil`. Если вы вызовете этот метод для типа, нулевое значение которого отличается от `nil`, то это (как вы уже догадались) приведет к панике.

Вы можете запустить эту функцию в онлайн-песочнице (<https://oreil.ly/D-HR9>).

Вы, конечно, можете учитывать те случаи, когда интерфейс равен значению `nil`, но старайтесь писать свой код так, чтобы он работал корректно даже в том случае, когда ассоциированное с интерфейсом значение будет равно `nil`. Предусмотрите такой код на тот случай, когда у вас не будет других вариантов.

Используйте рефлексия для создания маршализатора данных

Как уже упоминалось ранее, рефлексия используется в стандартной библиотеке для реализации маршализации и демаршализации. Чтобы наглядно увидеть, как это делается, попробуем создать собственный маршализатор. Язык Go предоставляет вам функции `csv.NewReader` и `csv.NewWriter` для чтения CSV-файла в срез строковых срезов и записи среза строковых срезов в CSV-файл, но не предусматривает никакого способа отображения этих данных на поля структуры. Мы сами определим эту недостающую функциональность.



Для краткости изложения приводимые здесь примеры были немного сокращены с помощью уменьшения количества поддерживаемых типов. С полной версией этого кода вы можете ознакомиться в онлайн-песочнице (<https://oreil.ly/VDytK>).

Начнем с определения API. Как и в случае любого другого маршализатора, мы должны определить теги структуры, устанавливающие соответствие между полями данных и полями структуры:

```
type MyData struct {
    Name  string `csv:"name"`
    Age   int    `csv:"age"`
    HasPet bool   `csv:"has_pet"`
}
```

Публичный API включает в себя две функции:

```
// Функция Unmarshal отображает все строки, содержащиеся в срезе
// строковых срезов, на срез структур.
// Предполагается, что первая строка представляет собой
// заголовок с именами столбцов.
```

```
func Unmarshal(data [][]string, v interface{}) error

// Функция Marshal отображает все структуры, содержащиеся в срезе структур,
// на срез строковых срезов.
// Первая записываемая строка представляет собой заголовок с именами столбцов.
func Marshal(v interface{}) ([][]string, error)
```

Сначала мы рассмотрим функцию `Marshal`, а затем — две вспомогательные функции, которые она будет использовать:

```
func Marshal(v interface{}) ([][]string, error) {
    sliceVal := reflect.ValueOf(v)
    if sliceVal.Kind() != reflect.Slice {
        return nil, errors.New("must be a slice of structs")
    }
    structType := sliceVal.Type().Elem()
    if structType.Kind() != reflect.Struct {
        return nil, errors.New("must be a slice of structs")
    }
    var out [][]string
    header := marshalHeader(structType)
    out = append(out, header)
    for i := 0; i < sliceVal.Len(); i++ {
        row, err := marshalOne(sliceVal.Index(i))
        if err != nil {
            return nil, err
        }
        out = append(out, row)
    }
    return out, nil
}
```

Поскольку эта функция должна производить маршализацию структур любого типа, мы используем параметр типа `interface{}`. Это не указатель на срез структур, потому что нам нужно лишь считывать данные из среза, не изменяя его.

Поскольку первая строка наших CSV-данных будет представлять собой заголовок с именами столбцов, мы извлекаем эти имена столбцов из тегов структур, содержащихся в полях структурного типа. Мы используем метод `Type`, чтобы получить соответствующий срез экземпляра типа `reflect.Type` на основе экземпляра типа `reflect.Value`, после чего вызываем метод `Elem`, чтобы получить тип (`reflect.Type`) элементов среза. Затем мы передаем этот тип в функцию `marshalHeader` и добавляем то, что она возвращает, в общий результат.

Затем мы обходим все элементы среза структур, используя рефлексия, передаем экземпляр типа `reflect.Value` каждого элемента в функцию `marshalOne` и добавляем в общий результат то, что она возвращает. После выполнения цикла мы возвращаем полученный срез строковых срезов.

Теперь взглянем на реализацию первой вспомогательной функции, `marshalHeader`:

```
func marshalHeader(vt reflect.Type) []string {
    var row []string
    for i := 0; i < vt.NumField(); i++ {
        field := vt.Field(i)
        if curTag, ok := field.Tag.Lookup("csv"); ok {
            row = append(row, curTag)
        }
    }
    return row
}
```

Эта функция просто обходит поля экземпляра типа `reflect.Type`, считывает в каждом поле тег `csv`, добавляет его в строковый срез и возвращает этот срез.

Второй вспомогательной функцией является функция `marshalOne`:

```
func marshalOne(vv reflect.Value) ([]string, error) {
    var row []string
    vt := vv.Type()
    for i := 0; i < vv.NumField(); i++ {
        fieldVal := vv.Field(i)
        if _, ok := vt.Field(i).Tag.Lookup("csv"); !ok {
            continue
        }
        switch fieldVal.Kind() {
        case reflect.Int:
            row = append(row, strconv.FormatInt(fieldVal.Int(), 10))
        case reflect.String:
            row = append(row, fieldVal.String())
        case reflect.Bool:
            row = append(row, strconv.FormatBool(fieldVal.Bool()))
        default:
            return nil, fmt.Errorf("cannot handle field of kind %v",
                fieldVal.Kind())
        }
    }
    return row, nil
}
```

Эта функция принимает экземпляр типа `reflect.Value` и возвращает строковый срез. Мы создаем строковый срез, а затем для каждого поля структуры на основе его разновидности (`reflect.Kind`) выбираем подходящий способ преобразования в строку и добавляем его в результат.

Таким образом, наш простой маршализатор готов. Теперь посмотрим, как можно осуществить демаршализацию:

```
func Unmarshal(data [][]string, v interface{}) error {
    sliceValPtr := reflect.ValueOf(v)
```

```

if sliceValPtr.Kind() != reflect.Ptr {
    return errors.New("must be a pointer to a slice of structs")
}
sliceVal := sliceValPtr.Elem()
if sliceVal.Kind() != reflect.Slice {
    return errors.New("must be a pointer to a slice of structs")
}
structType := sliceVal.Type().Elem()
if structType.Kind() != reflect.Struct {
    return errors.New("must be a pointer to a slice of structs")
}

// Предполагается, что первая строка представляет собой заголовок
header := data[0]
namePos := make(map[string]int, len(header))
for k, v := range header {
    namePos[v] = k
}

for _, row := range data[1:] {
    newVal := reflect.New(structType).Elem()
    err := unmarshalOne(row, namePos, newVal)
    if err != nil {
        return err
    }
    sliceVal.Set(reflect.Append(sliceVal, newVal))
}
return nil
}

```

Поскольку эта функция должна копировать данные в срез структур произвольного типа, мы используем параметр типа `interface{}`. Кроме того, поскольку значение этого параметра подвергается модификации, мы *должны* передавать указатель на срез структур. Функция `Unmarshal` преобразует этот указатель на срез структур в экземпляр типа `reflect.Value`, затем получает базовый срез, после чего получает тип содержащихся в нем структур.

Как уже говорилось выше, в данном примере предполагается, что первая строка данных представляет собой заголовок с именами столбцов. Мы используем эту информацию для создания карты, позволяющей ассоциировать значение тега структуры `csv` с правильным элементом данных.

Затем мы обходим все оставшиеся строковые срезы. При этом мы создаем новый экземпляр типа `reflect.Value`, использующий тип (`reflect.Type`) каждой структуры, вызываем функцию `unmarshalOne`, чтобы скопировать данные текущего строкового среза в структуру, а затем добавляем структуру в общий срез. После обхода всех строк данных мы выходим из функции.

Теперь нам осталось рассмотреть лишь реализацию функции `unmarshalOne`:

```
func unmarshalOne(row []string, namePos map[string]int, vv reflect.Value) error
{
    vt := vv.Type()
    for i := 0; i < vv.NumField(); i++ {
        typeField := vt.Field(i)
        pos, ok := namePos[typeField.Tag.Get("csv")]
        if !ok {
            continue
        }
        val := row[pos]
        field := vv.Field(i)
        switch field.Kind() {
        case reflect.Int:
            i, err := strconv.ParseInt(val, 10, 64)
            if err != nil {
                return err
            }
            field.SetInt(i)
        case reflect.String:
            field.SetString(val)
        case reflect.Bool:
            b, err := strconv.ParseBool(val)
            if err != nil {
                return err
            }
            field.SetBool(b)
        default:
            return fmt.Errorf("cannot handle field of kind %v",
                               field.Kind())
        }
    }
    return nil
}
```

Эта функция обходит все поля вновь созданного экземпляра типа `reflect.Value`, использует тег структуры `csv` текущего поля, чтобы получить его имя, находит элемент в срезе `data`, используя карту `namePos`, преобразует это значение из типа `string` в подходящий тип и присваивает это текущему полю. После заполнения всех полей производится выход из функции.

Теперь, располагая маршализатором и демаршализатором, мы можем интегрировать их с поддержкой формата CSV, предоставляемой стандартной библиотекой языка Go:

```
data := `name,age,has_pet
Jon,"100",true
"Fred ""The Hammer"" Smith",42,false
Martha,37,"true"
`
```

```
r := csv.NewReader(strings.NewReader(data))
allData, err := r.ReadAll()
if err != nil {
    panic(err)
}
var entries []MyData
Unmarshal(allData, &entries)
fmt.Println(entries)

// преобразование записей в конечный результат
out, err := Marshal(entries)
if err != nil {
    panic(err)
}
sb := &strings.Builder{}
w := csv.NewWriter(sb)
w.WriteAll(out)
fmt.Println(sb)
```

Создавайте с помощью рефлексии функции для автоматизации повторяющихся задач

Еще одной областью применения рефлексии в Go является создание функций. Мы можем использовать рефлексии для обертывания существующих функций с некоторой часто используемой функциональностью без написания повторяющегося кода. Например, вот как может выглядеть фабричная функция, снабжающая любую переданную ей функцию информацией о времени выполнения:

```
func MakeTimedFunction(f interface{}) interface{} {
    ft := reflect.TypeOf(f)
    fv := reflect.ValueOf(f)
    wrapperF := reflect.MakeFunc(ft, func(in []reflect.Value) []reflect.Value {
        start := time.Now()
        out := fv.Call(in)
        end := time.Now()
        fmt.Println(end.Sub(start))
        return out
    })
    return wrapperF.Interface()
}
```

Поскольку эта функция должна принимать на вход любую функцию, она принимает параметр типа `interface{}`. Затем она передает экземпляр типа `reflect.Type`, представляющий эту функцию, в функцию `reflect.MakeFunc`, вместе с замыканием, которое фиксирует начальный момент времени, вызы-

вает исходную функцию с помощью рефлексии, фиксирует конечный момент времени, выводит разницу между началом и концом и возвращает значение, вычисленное исходной функцией. Поскольку функция `reflect.MakeFunc` возвращает экземпляр типа `reflect.Value`, мы вызываем метод `Interface` этого типа, чтобы получить возвращаемое значение. Использовать эту функцию можно следующим образом:

```
func timeMe(a int) int {
    time.Sleep(time.Duration(a) * time.Second)
    result := a * 2
    return result
}

func main() {
    timed := MakeTimedFunction(timeMe).(func(int) int)
    fmt.Println(timed(2))
}
```

Вы можете запустить полную версию этой программы в онлайн-песочнице (<https://oreil.ly/NDfp1>).

Каким бы умным и продвинутым ни казалось это решение, будьте крайне внимательны при его использовании. Следите за тем, чтобы было вполне понятно, когда используется сгенерированная функция и какую функциональность она добавляет. В противном случае будет сложно понять, как перемещаются данные в пределах вашей программы. Кроме того, как мы еще увидим в подразделе «Используйте рефлексия только тогда, когда в этом есть смысл» на с. 378, рефлексия делает ваши программы медленнее, и поэтому использование ее для генерации и вызова функций сильно сказывается на производительности. Исключение составляет лишь тот случай, когда генерируемый вами код уже и так выполняет медленную операцию, например сетевой вызов. Помните, что рефлексия работает лучше всего, когда она используется для сопоставления данных на внешней границе вашей программы.

Примером проекта, соблюдающего эти принципы генерирования функций, в частности, является созданная мной библиотека `Proteus` для сопоставления данных SQL-запросов. Она создает типобезопасный API базы данных путем генерирования функции на основе SQL-запроса и функционального поля или переменной. Для получения более подробной информации о библиотеке `Proteus` ознакомьтесь с моим докладом на конференции `GopherCon 2017` «Генерирование на этапе выполнения, типобезопасность и декларативность: выбирайте любое из этих трех свойств» (<https://oreil.ly/ZUE47>); ее исходный код можно найти на сайте `GitHub` (<https://oreil.ly/KtFyj>).

Рефлексию можно использовать для создания структур, но лучше этого не делать

Существует и еще один, немного странный способ применения рефлексии. Функция `reflect.StructOf` принимает срез полей типа `reflect.StructField` и возвращает экземпляр типа `reflect.Type`, представляющий собой новую структуру. Такие структуры можно присваивать только переменным типа `interface{}`, а их поля можно считывать и записывать только с использованием рефлексии.

Эта функция фактически представляет интерес только с научной точки зрения. Если вы хотите увидеть, как функция `reflect.StructOf` используется на практике, вы можете ознакомиться в онлайн-песочнице с примером мемоизирующей функции (<https://oreil.ly/fXDk2>), в которой динамически генерируемые структуры используются в качестве ключей карты, кэширующей результаты мемоизируемой функции.

Рефлексия не позволяет создавать методы

Как мы видели, рефлексия позволяет вам делать почти все что угодно, но существует одна вещь, которую вы не сможете сделать с ее помощью. Ее можно использовать для создания новых функций и структур, но она не дает возможности снабдить тип дополнительными методами. Это означает, что вы не можете применять рефлексию для создания нового типа, реализующего некоторый интерфейс.

Используйте рефлексию только тогда, когда в этом есть смысл

Хотя рефлексия может играть важную роль при преобразовании данных на внешней границе Go-кода, будьте крайне осмотрительны с ее применением в других ситуациях, поскольку это несет с собой определенные издержки. Для наглядной демонстрации этого реализуем с помощью рефлексии функцию `Filter`. Это широко используемая функция во многих языках; она принимает на вход список значений, подвергает каждое из них проверке и возвращает список, содержащий только те элементы, которые успешно прошли проверку. Go не позволяет вам написать одну типобезопасную функцию, способную работать со срезами любого типа, но вы можете реализовать функцию `Filter` с использованием рефлексии:

```
func Filter(slice interface{}, filter interface{}) interface{} {  
    sv := reflect.ValueOf(slice)  
    fv := reflect.ValueOf(filter)
```

```

sliceLen := sv.Len()
out := reflect.MakeSlice(sv.Type(), 0, sliceLen)
for i := 0; i < sliceLen; i++ {
    curVal := sv.Index(i)
    values := fv.Call([]reflect.Value{curVal})
    if values[0].Bool() {
        out = reflect.Append(out, curVal)
    }
}
return out.Interface()
}

```

Эту функцию можно использовать следующим образом:

```

names := []string{"Andrew", "Bob", "Clara", "Hortense"}
longNames := Filter(names, func(s string) bool {
    return len(s) > 3
}).([]string)
fmt.Println(longNames)

ages := []int{20, 50, 13}
adults := Filter(ages, func(age int) bool {
    return age >= 18
}).([]int)
fmt.Println(adults)

```

Этот код выдаст следующий результат:

```

[Andrew Clara Hortense]
[20 50]

```

Хотя принцип действия нашей фильтрующей функции на базе рефлексии вполне понятен, она определенно будет работать медленнее специализированной функции, рассчитанной на тот или иной тип данных. Посмотрим, какую производительность она демонстрирует на моей машине с процессором i7-8700, 32 Гбайт оперативной памяти и версией Go 1.14 в случае фильтрации срезов, содержащих по 1000 строк или целых чисел, по сравнению со специализированными функциями для фильтрации строк и целых чисел:

BenchmarkFilterReflectString-12	4822	229099	ns/op	87361	B/op	2219	allocs/op
BenchmarkFilterString-12	158197	7795	ns/op	16384	B/op	1	allocs/op
BenchmarkFilterReflectInt-12	4962	232885	ns/op	72256	B/op	2503	allocs/op
BenchmarkFilterInt-12	348441	3440	ns/op	8192	B/op	1	allocs/op

Вы можете запустить этот пример кода самостоятельно, скачав его с сайта GitHub (<https://oreil.ly/Mj3SR>).

Функция на базе рефлексии работает примерно в 30 раз медленнее специализированной функции для фильтрации строк и почти в 70 раз медленнее

специализированной функции для фильтрации целых чисел. Она использует намного больше памяти и выполняет тысячи операций перераспределения памяти, что порождает дополнительный объем работы для сборщика мусора. В зависимости от того, какие требования предъявляются к вашей программе, это может не выходить за рамки допустимых издержек, но в то же время вы должны принять это во внимание.

Более серьезным недостатком является то, что компилятор не сможет остановить вас в том случае, если вы решите передать в качестве параметра `slice` или `filter` значение неподходящего типа. Если использование нескольких лишних тысяч наносекунд времени процессора может не представлять большой проблемы, то передача в функцию `Filter` среза или функции неподходящего типа приведет к аварийному прекращению работы программы в ходе ее эксплуатации. Это может привести к непозволительно высоким расходам на сопровождение программы. Хотя написание нескольких версий одной и той же функции для разных типов потребует многократного использования одинакового кода, сокращение объема кода в таком случае за счет использования одной версии не будет стоить связанных с этим издержек.

Использовать пакет `unsafe` небезопасно

Подобно тому как пакет `reflect` разрешает манипулировать типами и значениями, пакет `unsafe` позволяет манипулировать памятью. Это очень небольшой и очень странный пакет. Он определяет только три функции и один тип, которые ведут себя совершенно не так, как типы и функции, определенные в других пакетах.

Функция `Sizeof` принимает переменную любого типа и возвращает количество используемых ею байтов, функция `Offsetof` принимает поле структуры и возвращает количество байтов, содержащихся в структуре перед этим полем, а функция `Alignof` принимает поле или переменную и возвращает информацию о том, какой способ выравнивания байтов для нее нужно использовать. В отличие от других невстроенных функций, используемых в Go, эти функции могут принимать значения любого типа, а возвращаемые ими значения представляют собой константы, чтобы их можно было использовать в константных выражениях.

Тип `unsafe.Pointer` является особенным в том плане, что он существует лишь для того, чтобы указатель любого типа можно было преобразовать в этот тип или из него. Наряду с указателями, в тип `unsafe.Pointer` или из него также можно преобразовывать значения специального целочисленного типа `uintptr`. Над значениями этого типа можно производить математические действия,

как в случае любого другого целочисленного типа. Это позволяет «заходить» в экземпляр определенного типа и извлекать отдельные байты. Вы также можете использовать адресную арифметику, как это можно делать при работе с указателями в C и C++. Эти байтовые манипуляции ведут к изменению значения переменной.

Существует два основных паттерна использования небезопасного кода. Первый паттерн сводится к преобразованию друг в друга двух типов переменных, которые обычно не могут преобразовываться друг в друга. Это обеспечивается путем использования цепочки преобразований типа с небезопасным указателем (`unsafe.Pointer`) посередине. Второй паттерн сводится к чтению или модификации байтов переменной путем преобразования этой переменной в небезопасный указатель (`unsafe.Pointer`), преобразования этого указателя в значение типа `uintptr` и копирования или модификации низкоуровневых байтов. Посмотрим, когда следует и когда не следует использовать небезопасный код.

Используйте пакет `unsafe` для преобразования внешних двоичных данных

Возможно, вас удивляет, почему в Go вообще включили пакет `unsafe`, если этот язык ориентирован на безопасное использование памяти. Пакет `unsafe` применяется для преобразования двоичных данных, подобно тому как пакет `reflect` используется для преобразования текстовых данных на внешней границе Go-кода, и для его применения есть две основные причины. В 2020 году Коста (Costa), Муджахид (Mujahid), Абдалкарим (Abdalkareem) и Шихаб (Shihab) опубликовали статью «Нарушение типобезопасности в Go: эмпирическое исследование применения пакета `unsafe`»¹ (https://oreil.ly/N_6JX), в которой приводятся следующие результаты исследования 2438 популярных Go-проектов с открытым исходным кодом:

- 24 % изученных Go-проектов используют пакет `unsafe` в кодовой базе как минимум один раз;
- в большинстве случаев пакет `unsafe` применялся для интеграции с операционными системами и C-кодом (45,7 %);
- разработчики также часто используют пакет `unsafe` для написания более эффективного Go-кода (23,6 %).

Как видим, пакет `unsafe` чаще всего используется для интеграции с другими системами. В стандартной библиотеке языка Go пакет `unsafe` предназначен

¹ Costa D. E. D., Mujahid S., Abdalkareem R., Shihab E. Breaking Type-Safety in Go: An Empirical Study on the Usage of the `unsafe` Package. ArXiv abs/2006.09973 (2020).

для чтения и записи данных в операционную систему. Примеры такого его использования можно найти в пакете `syscall` и в более высокоуровневом пакете `sys` (<https://oreil.ly/ueHY3>). Более подробно о применении пакета `unsafe` для взаимодействия с операционной системой можно прочитать в замечательном блог-посте Мэтта Лэйхера (Matt Layher) (<https://oreil.ly/VtE1t>).

Вторым основанием для использования пакета `unsafe` является производительность, особенно при чтении данных из сети. Если вам нужно обеспечить сопоставление данных, передаваемых в структуру данных языка Go или из нее, то это можно очень быстро сделать с помощью типа `unsafe.Pointer`. Посмотрим, как это выглядит на практике, используя следующий, слегка надуманный пример. Допустим, что у нас есть протокол проводной связи с такой структурой.

- Значение: 4 байта; 32-разрядное беззнаковое целое число со «старшеконецным» порядком следования байтов.
- Метка: 10 байт; имя значения в формате ASCII.
- Активность: 1 байт; булев флаг, указывающий, является ли поле активным.
- Дополнение: 1 байт; используется для того, чтобы общий размер данных составлял 16 байт.



При пересылке данных по сети обычно используется «старшеконецный» порядок следования байтов (при котором старшие байты располагаются в начале), или, как его еще называют, «сетевой порядок байтов». Поскольку в большинстве современных процессоров используется «младшеконецный» порядок следования байтов (или двухконецный, работающий в «младшеконецном» режиме), следует быть крайне осторожными при чтении и записи данных в сеть.

Мы можем определить для этого протокола следующую структуру данных:

```
type Data struct {
    Value  uint32    // 4 байта
    Label  [10]byte   // 10 байт
    Active bool      // 1 байт
    // Go дополняет эти данные 1 байтом до "круглого" числа
}
```

Допустим, что мы считали из сети следующие байты:

```
[0 132 95 237 80 104 111 110 101 0 0 0 0 1 0]
```

Нам нужно считать эти байты в массив из 16 элементов и преобразовать этот массив в рассмотренную выше структуру языка Go.



Почему мы используем здесь массив, а не срез? Если вы помните, массивы, как и структуры, представляют собой значимые типы, для которых память выделяется напрямую. О том, как пакет `unsafe` можно использовать в сочетании со срезами, будет рассказано в следующем разделе.

С помощью безопасного Go-кода это сопоставление можно произвести следующим образом:

```
func DataFromBytes(b [16]byte) Data {
    d := Data{}
    d.Value = binary.BigEndian.Uint32(b[:4])
    copy(d.Label[:], b[4:14])
    d.Active = b[14] != 0
    return d
}
```

С другой стороны, мы можем использовать тип `unsafe.Pointer`:

```
func DataFromBytesUnsafe(b [16]byte) Data {
    data := *(*Data)(unsafe.Pointer(&b))
    if isLE {
        data.Value = bits.ReverseBytes32(data.Value)
    }
    return data
}
```

Здесь может сбивать с толку первая строка, но вам станет понятно, что делает этот код, как только мы разберем его на составные части. Сначала мы принимаем указатель на байтовый массив и преобразуем его в небезопасный указатель (`unsafe.Pointer`). Затем мы преобразуем небезопасный указатель в указатель `*Data` (который нужно заключить в скобки по причине порядка выполнения операций в Go). Поскольку нам нужно возвращать структуру, а не указатель на нее, мы производим разыменование указателя. Далее мы проверяем флаг, указывающий, работаем ли мы на «младшеконечной» платформе. Если это так, мы инвертируем порядок байтов в поле `Value`. Наконец, мы возвращаем значение.

Как же производится проверка работы на «младшеконечной» платформе? Это делается с помощью следующего кода:

```
var isLE bool

func init() {
    var x uint16 = 0xFF00
    xb := *(*[2]byte)(unsafe.Pointer(&x))
    isLE = (xb[0] == 0x00)
}
```

Как упоминалось в подразделе «По возможности не используйте функцию init» на с. 231, функцию `init` не рекомендуется применять для чего-либо иного, кроме инициализации фактически неизменных значений пакетного уровня. Поскольку используемый процессором порядок следования байтов не будет изменяться во время работы программы, данный код не противоречит этой рекомендации.

На «младшеконечной» платформе значение переменной `x` будет представлено памяти как `[00 FF]`, а на «старшеконечной» — как `[FF 00]`. Мы используем небезопасный указатель (`unsafe.Pointer`), чтобы преобразовать число в массив байтов, а затем проверяем, чему равен первый байт, чтобы определить значение переменной `isLE`.

Для записи содержимого структуры `Data` обратно в сеть мы точно так же могли бы использовать безопасный Go-код:

```
func BytesFromData(d Data) [16]byte {
    out := [16]byte{}
    binary.BigEndian.PutUint32(out[:4], d.Value)
    copy(out[4:14], d.Label[:])
    if d.Active {
        out[14] = 1
    }
    return out
}
```

Или делать это с помощью пакета `unsafe`:

```
func BytesFromDataUnsafe(d Data) [16]byte {
    if isLE {
        d.Value = bits.ReverseBytes32(d.Value)
    }
    b := *(*[16]byte)(unsafe.Pointer(&d))
    return b
}
```

Насколько оправданным будет такое решение? На моей машине с процессором `i7-8700` (который использует «младшеконечный» формат) применение типа `unsafe.Pointer` дает примерно двукратный прирост производительности:

BenchmarkBytesFromData-12	112741796	10.4 ns/op
BenchmarkBytesFromDataUnsafe-12	298846651	4.01 ns/op
BenchmarkDataFromBytes-12	100000000	10.3 ns/op
BenchmarkDataFromBytesUnsafe-12	235992582	5.95 ns/op



Все примеры из этого раздела можно найти на сайте GitHub (<https://oreil.ly/E4MEF>).

Если в вашей программе производится много подобных преобразований, то использование описанных низкоуровневых приемов будет вполне оправданным. Но в подавляющем большинстве программ будет лучше обойтись без небезопасного кода.

Пакет `unsafe` в сочетании со строками и срезами

Пакет `unsafe` также можно использовать для взаимодействия со срезами и строками. Как упоминалось в разделе «Строки в сочетании с рунами и байтами» на с. 72, для представления типа `string` в Go применяется указатель на последовательность байтов и длина этой последовательности. В пакете `reflect` есть тип с такой структурой, который называется `reflect.StringHeader`. Его можно использовать для чтения и модификации базового представления типа `string`:

```
s := "hello"
sHdr := (*reflect.StringHeader)(unsafe.Pointer(&s))
fmt.Println(sHdr.Len) // выводит 5
```

Мы можем произвести чтение байтов строки с применением адресной арифметики, используя поле `Data` структуры `sHdr`, которое относится к типу `uintptr`:

```
for i := 0; i < sHdr.Len; i++ {
    bp := (*byte)(unsafe.Pointer(sHdr.Data + uintptr(i)))
    fmt.Print(string(bp))
}
fmt.Println()
runtime.KeepAlive(s)
```

Поле `Data` типа `reflect.StringHeader` относится к типу `uintptr`, а, как упоминалось ранее, значение этого типа обеспечивает обращение к корректным данным в памяти лишь в пределах одного оператора. Каким же образом мы сможем не позволить сборщику мусора сделать эту ссылку некорректной? Мы можем добиться этого, разместив в конце нашей функции вызов функции `runtime.KeepAlive(s)`. Тем самым мы даем среде выполнения языка Go указание, что сборщик мусора не должен удалять переменную `s`, пока не будет вызвана функция `KeepAlive`.

Вы можете выполнить этот код в онлайн-песочнице (<https://oreil.ly/1DTR5>).

Подобно тому как с помощью пакета `unsafe` можно извлечь из строки экземпляр типа `reflect.StringHeader`, мы можем извлечь из среза экземпляр типа `reflect.SliceHeader` с тремя полями: `Len`, `Cap` и `Data`, содержащими соответственно длину, емкость и указатель на данные среза:

```
s := []int{10, 20, 30}
sHdr := (*reflect.SliceHeader)(unsafe.Pointer(&s))
fmt.Println(sHdr.Len) // выводит 3
fmt.Println(sHdr.Cap) // выводит 3
```

Точно так же, как это делалось в случае строк, мы используем преобразование типа для преобразования указателя на срез значений типа `int` в тип `unsafe.Pointer`. Затем мы преобразуем небезопасный указатель (экземпляр типа `unsafe.Pointer`) в указатель на экземпляр типа `reflect.SliceHeader`. После этого мы можем получить доступ к длине и емкости среза посредством полей `Len` и `Cap`. Теперь мы можем произвести обход среза:

```
intByteSize := unsafe.Sizeof(s[0])
fmt.Println(intByteSize)
for i := 0; i < sHdr.Len; i++ {
    intVal := *(*int)(unsafe.Pointer(sHdr.Data + intByteSize*uintptr(i)))
    fmt.Println(intVal)
}
runtime.KeepAlive(s)
```

Поскольку размер значений типа `int` может составлять 32 или 64 бита, мы должны использовать функцию `unsafe.Sizeof`, чтобы узнать, сколько байтов занимает каждое значение в том блоке памяти, на который указывает поле `Data`. Затем мы приводим значение переменной `i` к типу `uintptr` и умножаем его на размер типа `int`, добавляем результат к содержимому поля `Data`, преобразовываем значение типа `uintptr` в небезопасный указатель (экземпляр типа `unsafe.Pointer`), а затем — в указатель на тип `int` и, наконец, производим разыменование указателя на тип `int`, чтобы получить значение.

Вы можете запустить этот код в онлайн-песочнице (<https://oreil.ly/u9qB9>).

Вспомогательные инструменты для пакета `unsafe`

Будучи языком, поощряющим использование вспомогательных инструментов, Go предоставляет вам флаг компилятора, позволяющий выявлять случаи неправильного использования типов `uintptr` и `unsafe.Pointer`. Запустив свой код с флагом `-gcflags=-d=checkptr`, вы обеспечите проведение такой дополнительной проверки на этапе выполнения. Как и детектор состояний гонки, эта проверка не гарантирует выявления абсолютно всех проблем с небезопасным кодом и замедляет работу программы. В то же время использование этого флага будет вполне уместным на этапе тестирования кода.

Дополнительную информацию о пакете `unsafe` можно найти в соответствующей документации (<https://oreil.ly/xmihF>).



Пакет `unsafe` — мощное и низкоуровневое средство! Используйте его лишь в том случае, когда вы точно знаете, что делаете, и вам требуется обеспечить прирост производительности.

Пакет `cgo` предназначен для обеспечения интеграции, а не повышения производительности

Как и пакеты `reflect` и `unsafe`, пакет `cgo` чаще всего используется на внешней границе Go-программы. Если пакет `reflect` обеспечивает интеграцию с внешними текстовыми данными, а пакет `unsafe` — интеграцию с операционной системой и сетевыми данными, то пакет `cgo` предназначен главным образом для интеграции с С-библиотеками.

Несмотря на то что язык С существует уже почти 50 лет, это по-прежнему *лингва франка* в мире языков программирования. Поскольку все основные операционные системы написаны главным образом на С или С++, в их «комплект поставки» входят библиотеки, написанные на С. Это также означает, что практически все современные языки программирования предоставляют определенный способ интеграции с С-библиотеками. В Go роль интерфейса внешней функции (FFI, Foreign Function Interface) для сопряжения с С-кодом выполняет пакет `cgo`.

Как мы уже не раз видели, Go поощряет более явное определение тех или иных вещей. Go-разработчики иногда иронично называют «магией» существующее в других языках автоматическое поведение, однако использование пакета `cgo` вызывает во многом сходные ощущения. Посмотрим, как выглядит этот «магический» связующий код. Поскольку примеры использования пакета `cgo` нельзя запустить в онлайн-песочнице, они размещены на сайте GitHub (<https://oreil.ly/ct9xd>). Мы начнем с очень простой программы, которая вызывает С-код для выполнения некоторых математических действий:

```
package main

import "fmt"

/*
#cgo LDFLAGS: -lm
#include <stdio.h>
#include <math.h>
#include "mylib.h"

int add(int a, int b) {
```

```
        int sum = a + b;
        printf("a: %d, b: %d, sum %d\n", a, b, sum);
        return sum;
    }
    */
import "C"

func main() {
    sum := C.add(3, 2)
    fmt.Println(sum)
    fmt.Println(C.sqrt(100))
    fmt.Println(C.multiply(10, 20))
}
```

Заголовочный файл `mylib.h` находится в том же каталоге, что и наш файл `main.go`, вместе с файлом `mylib.c`:

```
int multiply(int a, int b);

#include "mylib.h"

int multiply(int a, int b) {
    return a * b;
}
```

Если на вашей машине уже установлен компилятор языка C, вам останется лишь скомпилировать свою программу, используя стандартные инструменты языка Go:

```
$ go build $ ./example1
a: 3, b: 2, sum 5
5
10
200
```

Что же здесь происходит? Стандартная библиотека языка Go не содержит реального пакета с именем `C`. Пакет `C` здесь представляет собой автоматически генерируемый пакет, идентификаторы которого в основном берутся из C-кода, вложенного в блок комментариев, расположенный непосредственно перед оператором для импорта этого пакета. В данном примере мы объявляем C-функцию с именем `add`, и пакет `cgo` делает ее доступной внутри Go-программы в виде функции `C.add`. Мы также можем использовать в Go-коде функции и глобальные переменные, импортируемые в блок комментариев из библиотек посредством заголовочных файлов. Таким образом, внутри функции `main` мы вызываем функцию `C.sqrt` (которая импортируется из файла `math.h`) и функцию `C.multiply` (которая импортируется из файла `mylib.h`).

Наряду с идентификаторами, которые определяются в блоке комментариев (или импортируются в него), псевдопакет `C` также определяет такие типы, как `C.int` и `C.char`, для представления встроенных типов языка `C` и функции наподобие функции `C.CString` для преобразования строк языка `Go` в строки языка `C`.

Вы можете использовать еще больше «магии», сделав возможным вызов `Go`-функций из `C`-функций. `Go`-функцию можно сделать доступной для `C`-кода путем размещения перед определением функции комментария `//export`:

```
//export doubler
func doubler(i int) int {
    return i * 2
}
```

При этом вы уже не сможете определить `C`-код непосредственно в блоке комментариев, расположенном перед оператором `import "C"`. В этом блоке можно будет только объявить функции, но не определять их:

```
/*
    extern int add(int a, int b);
*/
import "C"
```

После этого вы должны сохранить свой `C`-код в файле с расширением `.c` в том же каталоге, где находится ваш `Go`-код, и подключить «магический» заголовочный файл `"_cgo_export.h"`:

```
#include "_cgo_export.h"

int add(int a, int b) {
    int doubleA = doubler(a);
    int sum = doubleA + b;
    return sum;
}
```

Хотя ничего из сказанного выше не должно вызывать каких-либо затруднений, в использовании пакета `cgo` все же имеется одна «загвоздка»: в языке `Go` есть сборка мусора, а в языке `C` ее нет. Это затрудняет интеграцию с `C`-кодом нетривиального `Go`-кода. В `C`-код можно передать указатель, но не что-то содержащее указатель. Это очень существенное ограничение, поскольку такие вещи, как строки, срезы и функции, реализуются с помощью указателей, и поэтому их не может содержать структура, передаваемая в `C`-функцию. И это еще не все: `C`-функция не может сохранить копию `Go`-указателя, используемую после выхода из этой функции. Если вы нарушите эти правила, то ваша программа скомпилируется и запустится, но даст сбой или начнет вести себя некорректно на этапе выполнения, когда сборщик мусора высвободит память, на которую указывает такой указатель.

Есть и другие ограничения. Например, пакет `cgo` нельзя использовать для вызова вариативной C-функции (такой как `printf`). Объединенные типы языка C преобразуются в байтовые массивы. И вы не можете вызвать указатель на C-функцию (но его можно присвоить Go-переменной и передать в C-функцию).

Эти правила существенно усложняют работу с пакетом `cgo`. Если вам приходилось писать код на языке Python или Ruby, то вы можете подумать, что использование пакета `cgo` оправдывается соображениями производительности. Разработчики, использующие эти языки, переписывают на C те части своих программ, в которых нужно обеспечить хорошую производительность. Так, высокая скорость работы библиотеки NumPy обеспечивается за счет использования C-библиотек, обернутых Python-кодом.

Go-код обычно работает намного быстрее, чем код, написанный на Python или Ruby, поэтому переписывать алгоритмы на низкоуровневом языке требуется гораздо реже. По идее, пакет `cgo` можно было бы использовать в тех случаях, когда нужно обеспечить дополнительный прирост производительности, но, к сожалению, с его помощью очень сложно сделать код быстрее. По причине того, что в Go и C используются разные модели обработки данных и управления памятью, C-функция, вызываемая из Go-кода, работает примерно в 29 раз медленнее, чем C-функция, вызываемая из другой C-функции. На конференции CapitalGo 2018 Филиппо Вальсорта (Filippo Valsorda) выступил с докладом, который назывался «Почему пакет `cgo` работает медленно». К сожалению, во время этого выступления не производилась видеозапись, но вы можете ознакомиться с соответствующими слайдами (<https://oreil.ly/MLRFY>). Ознакомившись с ними, вы поймете, почему пакет `cgo` работает медленно и почему не стоит надеяться на то, что в будущем он будет работать сколь-нибудь быстрее.

Учитывая то, что пакет `cgo` не позволяет повысить производительность, и то, что его сложно использовать в нетривиальных программах, его стоит подключать лишь в том случае, когда требуется C-библиотека, которую нельзя заменить кодом, написанным на Go. Вместо того чтобы применять пакет `cgo` самостоятельно, стоит поискать сторонний модуль, предоставляющий нужную вам обертку. Например, если вам нужно встроить в Go-приложение поддержку баз данных SQLite, скачайте с сайта GitHub пакет `go-sqlite3` (<https://oreil.ly/IEskN>). Если нужно обеспечить поддержку редактора изображений ImageMagick, попробуйте использовать пакет `imagemick` (<https://oreil.ly/I58-1>).

Если же вам нужно использовать внутреннюю библиотеку языка C или стороннюю библиотеку, для которой пока не создано обертки, то подробную информацию об обеспечении такой интеграции можно найти в документации языка Go (<https://oreil.ly/9JvNI>). О том, с какими видами проблем производительности

и дизайна вам придется столкнуться при использовании пакета `sgo`, можно прочитать в блог-посте Тобиаса Григера (Tobias Grieger) под названием «Издержки и сложность применения пакета `sgo`» (<https://oreil.ly/Oj9Tw>).

Резюме

В этой главе мы рассмотрели пакеты `reflect`, `unsafe` и `sgo`. Это, пожалуй, наиболее интересные составляющие языка Go, позволяющие вам нарушать скучные правила безопасного использования типов и памяти. Что еще важнее, вы узнали, *почему* иногда требуется нарушать эти правила и почему этого не стоит делать в большинстве случаев.

В следующей главе мы рассмотрим нововведение, которое должно появиться в ближайших версиях языка Go: обобщенные типы.

Взгляд в будущее: обобщенные типы в Go

Хотя Go и уделяет меньше внимания функциональным возможностям, чем другие языки, нельзя сказать, что он совершенно статичен и не подвержен изменениям. В Go вводятся новые функциональные возможности, только более медленно и после продолжительных обсуждений и экспериментов. Так, после выхода первой версии Go 1.0 значительным изменениям подверглись паттерны идиоматического Go-кода. Первым таким изменением было введение контекста в версии Go 1.7, за которым последовало введение модулей в версии Go 1.11 и обертывания ошибок в версии Go 1.13.

В ближайшем будущем ожидается следующее крупное изменение. Команда разработчиков языка Go ознакомила публику с предварительным дизайном параметров типа, или, как их еще называют, обобщенных типов, которые планируется включить в версию Go 1.18. В эту предварительную реализацию еще могут быть внесены изменения, но в этой главе мы обсудим то, что известно на момент написания данной книги. Мы разберемся в этой главе, зачем разработчикам нужны обобщенные типы, что может и что не может делать та реализация обобщенных типов, которая будет введена в Go, и как это скажется на паттернах идиоматического кода.

Обобщенные типы уменьшают количество повторяющегося кода и повышают типобезопасность

Go — язык со статической типизацией, а это значит, что типы переменных и параметров проверяются на этапе компиляции кода. При этом встроенные типы (карты, срезы, каналы) и функции (такие как `len`, `cap` и `make`) еще могут

принимать и возвращать значения разного типа, однако пользовательские типы и функции не могут этого делать. Если вы знакомы с Java, C++ или каким-либо другим языком, в котором есть определенная разновидность обобщенных типов, то такое простое устройство системы типов языка Go может вызвать у вас разочарование.

Если раньше вы использовали языки с динамической типизацией, в которых типы определяются лишь в момент выполнения кода, то вы, возможно, не понимаете, к чему весь этот шум по поводу обобщенных типов, и имеете смутное представление о том, что это такое. Возможно, вам будет проще рассматривать их как «параметры типа». Вы привыкли использовать функции, входные параметры которых определяются в момент их вызова. Таким же образом можно создавать структуры, в которых значения полей определяются в момент создания экземпляра структуры. Концепция обобщенных типов используется в тех случаях, когда нужно написать функции или структуры, в которых конкретный *тип* параметров и полей определяется в момент их использования.

Когда речь идет о типах данных, применение этой концепции не представляет больших сложностей. В подразделе «Пишите код методов с расчетом на экземпляр, равный nil» на с. 168 в качестве примера рассматривалось двоичное дерево для значений типа `int`. Если бы нам потребовалось типобезопасное двоичное дерево для строк или значений типа `float64`, то пришлось бы создать отдельные деревья для каждого типа. При этом мы написали бы большое количество кода, что чревато ошибками. Было бы лучше, если бы мы могли создать одну структуру данных, способную работать с любым типом, позволяющим выполнять сравнение с помощью оператора `<`, но на данный момент Go не позволяет этого сделать.

Мы можем создать дерево, которое будет работать с интерфейсом, определяющим способ упорядочения значений:

```
type Orderable interface {  
    // Метод Order возвращает:  
    // value < 0, если Orderable меньше предоставленного значения,  
    // value > 0, если Orderable больше предоставленного значения,  
    // и 0 в случае равенства значений.  
    Order(interface{}) int  
}
```

Теперь, располагая интерфейсом `Orderable`, мы можем вставлять в свое дерево значения любого типа, обладающего методом `Order`:

```
type Tree struct {  
    val      Orderable  
    left, right *Tree  
}
```

```
func (t *Tree) Insert(val Orderable) *Tree {
    if t == nil {
        return &Tree{val: val}
    }

    switch comp := val.Order(t.val); {
    case comp < 0:
        t.left = t.left.Insert(val)
    case comp > 0:
        t.right = t.right.Insert(val)
    }
    return t
}
```

Так, для вставки целочисленных значений можно использовать тип `OrderableInt`:

```
type OrderableInt int

func (oi OrderableInt) Order(val interface{}) int {
    return int(oi - val.(OrderableInt))
}

func main() {
    var it *Tree
    it = it.Insert(OrderableInt(5))
    it = it.Insert(OrderableInt(3))
    // и так далее...
}
```

Хотя интерфейс `Orderable` позволяет упорядочивать значения, компилятор при этом не может проследить за тем, чтобы все вставленные в структуру значения относились к одному и тому же типу. Если мы также определим тип `OrderableString`:

```
type OrderableString string

func (os OrderableString) Order(val interface{}) int {
    return strings.Compare(string(os), val.(string))
}
```

то следующий код успешно скомпилируется:

```
var it *Tree
it = it.Insert(OrderableInt(5))
it = it.Insert(OrderableString("nope"))
```

Для представления передаваемого ей значения функция `Order` использует тип `interface{}`. Это фактически сводит на нет одно из главных преимуществ языка Go — контроль в отношении безопасности типов на этапе компиляции. В данном случае код, который пытается вставить значение типа `OrderableString` в струк-

туру `Tree`, уже содержащую значение типа `OrderableInt`, успешно проходит компиляцию. Однако если мы запустим этот код, он выдаст панику:

```
panic: interface conversion: interface {} is main.OrderableInt, not string
```

Как видим, в случае структур данных отсутствие обобщенных типов вызывает большие неудобства, но действительно серьезным ограничением это является в случае функций. Вместо того чтобы использовать отдельные версии функций для разных числовых типов, Go реализует такие функции, как `math.Max`, `math.Min` и `math.Mod`, используя параметры типа `float64`, который обладает достаточно большим диапазоном значений для точного представления значений других числовых типов (за исключением значений типа `int`, `int64` или `uint`, превышающих $2^{53} - 1$ или составляющих меньше $-2^{53} - 1$). Кроме того, вы не можете создать новый экземпляр переменной, которая определяется интерфейсом, или указать, что два параметра одного и того же интерфейсного типа также относятся к одному и тому же конкретному типу. Go также не позволяет обрабатывать срезы произвольного типа: вы не можете присвоить срез типа `[]string` или типа `[]int` переменной типа `[]interface{}`. Это означает, что в случае функций, выполняющих операции над срезами, нужно либо создавать отдельные версии функций для каждого типа среза, либо использовать рефлексию, смирившись с некоторой потерей производительности и отказавшись от контроля безопасности типов на этапе компиляции (именно так работает функция `sort.Slice`).

В 2017 году я написал блог-пост под заголовком «Замыкания — это обобщенные типы языка Go» (<https://oreil.ly/2pKYt>), в котором была рассмотрена возможность решения этих проблем путем использования замыканий. Однако такой подход обладает целым рядом недостатков. Он делает код менее читабельным, вызывает убегание значений в кучу и просто не работает во многих распространенных ситуациях.

Как следствие, многие часто используемые алгоритмы, такие как `map`, `reduce` и `filter`, в итоге все равно приходится реализовывать для каждого типа в отдельности. И хотя в случае простых алгоритмов такое копирование кода не представляет больших проблем, многих (если не всех) разработчиков раздражает тот факт, что они должны дублировать код только по той причине, что компилятор недостаточно сообразителен для того, чтобы делать это автоматически.

Добавление обобщенных типов в Go

Призывы добавить в Go обобщенные типы раздаются еще с того момента, как этот язык был впервые представлен публике. В 2009 году Расс Кокс (Russ Cox), руководитель команды разработчиков языка Go, написал пост, в котором объ-

яснялось, почему в Go не были изначально включены обобщенные типы (<https://oreil.ly/U4huA>). Разработчики хотели получить быстрый компилятор, читабельный код и хорошую производительность, а ни одна из известных им реализаций обобщенных типов не позволяла обеспечить и первое, и второе, и третье. После изучения этой проблемы в течение десяти лет разработчики Go нашли, как им кажется, работоспособное решение, суть которого была изложена в документе «Предварительный дизайн параметров типа» (<https://oreil.ly/POhSg>).

Посмотрим, как будут работать обобщенные типы в Go, на примере простого стека. Если бы мы решили реализовать стек для значений любого типа, не используя обобщенные типы, это можно было бы сделать вот так:

```
type Stack struct {
    vals []interface{}
}

func (s *Stack) Push(val interface{}) {
    s.vals = append(s.vals, val)
}

func (s *Stack) Pop() (interface{}, bool) {
    if len(s.vals) == 0 {
        return nil, false
    }
    top := s.vals[len(s.vals)-1]
    s.vals = s.vals[:len(s.vals)-1]
    return top, true
}
```

Использовать такой стек можно следующим образом:

```
func main() {
    var s Stack
    s.Push(10)
    s.Push(20)
    s.Push(30)
    v, ok := s.Pop()
    fmt.Println(v, ok)
}
```

Этот код успешно работает и выводит на экран `30 true`, но в этот стек можно вставить значение любого типа, и если вы захотите сделать что-то большее, а не просто вывести значение, возвращаемое методом `Pop`, вам потребуется привести это значение к конкретному типу с помощью операции утверждения типа. Посмотрим, как этот стек можно сделать типобезопасным, используя обобщенные типы языка Go:

```
type Stack[T any] struct {
    vals []T
}
```

```
}

func (s *Stack[T]) Push(val T) {
    s.vals = append(s.vals, val)
}

func (s *Stack[T]) Pop() (T, bool) {
    if len(s.vals) == 0 {
        var zero T
        return zero, false
    }
    top := s.vals[len(s.vals)-1]
    s.vals = s.vals[:len(s.vals)-1]
    return top, true
}
```

Здесь следует обратить внимание на три детали. Прежде всего, после объявления типа указан параметр типа `[T any]`. Параметры типа применяются в квадратных скобках. Подобно обычным параметрам, сначала указывается имя, а следом — ограничение типа. В принципе, можно использовать любое имя, но обычно в качестве имени параметра типа используются заглавные буквы. Для определения допустимых типов в Go применяются интерфейсы. Если допускается использование любого типа, это указывается с помощью нового идентификатора `any`, который определен во всеобщем блоке и абсолютно эквивалентен пустому интерфейсу `interface{}`, но может применяться только в качестве ограничения типа. Внутри определения типа `Stack` мы объявляем переменную `vals`, используя тип `[]T` вместо типа `[]interface{}`.

Второй интересной деталью здесь являются объявления методов. Как и в случае переменной `vals`, мы заменили здесь `interface{}` на `T`. В секции для определения приемника мы также указали тип `Stack[T]` вместо типа `Stack`.

Третьим любопытным моментом здесь является то, как при использовании обобщенных типов приходится обращаться с нулевым значением. В методе `Pop` мы не можем просто возвращать значение `nil`, потому что это недопустимое значение для значимого типа, такого как `int`. Самый простой способ получить нулевое значение для обобщенного типа сводится к тому, чтобы просто объявить переменную с помощью ключевого слова `var` и вернуть ее, поскольку по определению ключевое слово `var` всегда инициализирует переменную соответствующим нулевым значением, если ей не присваивается другое значение.

Используются обобщенные типы практически так же, как обычные типы:

```
func main() {
    var s Stack[int]
    s.Push(10)
```

```

    s.Push(20)
    s.Push(30)
    v, ok := s.Pop()
    fmt.Println(v, ok)
}

```

Единственное отличие состоит в том, что при объявлении переменной нужно указать, с каким типом будет работать наш стек: в данном случае это тип `int`. Теперь переменная `v` относится к типу `int`, а не к типу `interface{}`, что позволяет использовать ее без утверждения типа. Кроме того, если мы попытаемся добавить в стек строку, компилятор не позволит нам этого сделать. Если мы добавим следующий код:

```
s.Push("nope")
```

компилятор выдаст сообщение об ошибке:

```
cannot convert "nope" (untyped string constant) to int
```

Поскольку обобщенные типы еще не были официально выпущены, они не поддерживаются в используемой нами онлайн-песочнице The Go Playground. Однако их поддерживает временно существующая онлайн-песочница Gotip (<https://oreil.ly/MikKu>), где вы можете опробовать реализацию обобщенного стека.

Дополним наш стек еще одним методом, который будет сообщать о том, содержит ли стек значение:

```

func (s Stack[T]) Contains(val T) bool {
    for _, v := range s.vals {
        if v == val {
            return true
        }
    }
    return false
}

```

К сожалению, этот код не скомпилируется, выдав сообщение об ошибке:

```
cannot compare v == val (operator == not defined for T)
```

Как и пустой интерфейс `interface{}`, ограничение `any` ничего не сообщает о своем значении. Мы можем лишь добавлять и извлекать значения любого типа. Чтобы можно было использовать оператор `==`, мы должны использовать другой тип. Поскольку почти все типы языка Go позволяют выполнять сравнение с помощью операторов `==` и `!=`, новый встроенный интерфейс `comparable`

был определен во всеобщем блоке. Если мы заменим `any` на `comparable` в определении нашего типа `Stack`:

```
type Stack[T comparable] struct {
    vals []T
}
```

это позволит нам использовать новый метод `Contains`:

```
func main() {
    var s Stack[int]
    s.Push(10)
    s.Push(20)
    s.Push(30)
    fmt.Println(s.Contains(10))
    fmt.Println(s.Contains(5))
}
```

Этот код выдаст следующий результат:

```
true
false
```

Опробуйте эту обновленную версию в онлайн-песочнице (<https://oreil.ly/S6lwP>).

Вернемся к нашему двоичному дереву и посмотрим, как его можно сделать обобщенным. Нам нужно, чтобы в нем использовались только типы, реализующие интерфейс `Orderable`, и чтобы при этом обеспечивалась безопасность типов на этапе компиляции. Этого можно добиться с помощью параметров типа. Сначала внесем изменения в интерфейс `Orderable`:

```
type Orderable[T any] interface {
    Order(T) int
}
```

Это объявление говорит о том, что интерфейс `Orderable` определяет один параметр типа с именем `T`, который может обладать любым типом. Чтобы тип реализовывал этот новый обобщенный интерфейс, его метод `Order` должен принимать параметр, обладающий типом `T`. Поэтому внесем соответствующие изменения в метод `Order` для типа `OrderableInt`:

```
func (oi OrderableInt) Order(val OrderableInt) int {
    return int(oi - val)
}
```

Этот код проще, чем предыдущее определение этого метода, где входной параметр обладал типом `interface{}`, что вынуждало нас использовать опе-

рацию утверждения типа. Аналогичные изменения нужно внести и в метод `OrderableString`:

```
func (os OrderableString) Order(val OrderableString) int {
    return strings.Compare(string(os), string(val))
}
```

Теперь изменим определение типа `Tree` таким образом, чтобы в нем использовался обобщенный интерфейс `Orderable`:

```
type Tree[T Orderable[T]] struct {
    val      T
    left, right *Tree[T]
}
```

Хотя это определение на первый взгляд кажется довольно сложным, оно говорит лишь о том, что структура `Tree` может содержать только типы, реализующие интерфейс `Orderable`. Изменим также определения методов типа `Tree`. Метод `Insert` теперь будет выглядеть так:

```
func (t *Tree[T]) Insert(val T) *Tree[T]
```

а метод `Contains` так:

```
func (t *Tree[T]) Contains(val T) bool
```

Небольшое изменение также нужно внести и в тело метода `Insert`. Новый экземпляр структуры `Tree` теперь будет создаваться следующим образом:

```
return &Tree[T]{val: val}
```

Теперь объявим обобщенное дерево, используя параметр типа:

```
var it *Tree[OrderableInt]
```

Хотя нетипизированные константы можно присваивать без явного преобразования типа, чтобы присвоить переменную типа `int`, необходимо выполнить преобразование типа (потому что Go не выполняет преобразование типа автоматически):

```
a := 10
it = it.Insert(OrderableInt(a))
```

Обратите также внимание, что если мы передадим в метод `Insert` значение типа `OrderableString`, это приведет к ошибке на этапе компиляции:

```
cannot use OrderableString("nope") (constant "nope" of type OrderableString)
as OrderableInt value in argument
```

С полной версией этого кода вы можете ознакомиться в онлайн-песочнице (https://oreil.ly/_Z5vP).

Используйте списки типов для определения операторов

Таким образом, мы можем использовать структуру `Tree` для встроенных типов путем использования привязанного к ним типа, соответствующего интерфейсу `Orderable`, однако было бы еще лучше, если бы ее можно было использовать для встроенных типов без каких-либо оберток. Для этого нужно определенным образом указать, что значения типа можно сравнивать с помощью оператора `<`. Средства обобщенного программирования языка Go предусматривают для этой цели *списки типов*, которые представляют собой просто список типов, указываемый внутри интерфейса:

```
type BuiltInOrdered interface {
    type string, int, int8, int16, int32, int64, float32, float64,
        uint, uint8, uint16, uint32, uint64, uintptr
}
```

Когда ограничение типа задается с помощью интерфейса, содержащего список типов, допускается использование любого из перечисленных типов. Однако при этом можно применять только операторы, которые являются допустимыми для *всех* перечисленных типов. В данном случае это операторы `==`, `!=`, `>`, `<`, `>=`, `<=` и `+`. Чтобы посмотреть, как это выглядит на практике, создадим еще одну версию структуры `Tree`. Сначала мы должны заменить интерфейс `Orderable` на интерфейс `BuiltInOrdered`:

```
type Tree[T BuiltInOrdered] struct
```

Затем внутри методов `Insert` и `Contains` нужно снова записать операторы `switch` так же, как они выглядели в специализированной версии типа `Tree` для типа `int`:

```
switch {
case val < t.val:
case val > t.val:
}
```

Теперь мы можем использовать тип `Tree` для встроенного типа `int` (или любого другого типа, указанного в интерфейсе `BuiltInOrdered`):

```
func main() {
    var it *Tree[int]
    it = it.Insert(5)
    a := 10
    it = it.Insert(a)
    // и так далее...
}
```

С полной версией этого кода вы можете ознакомиться в онлайн-песочнице (<https://oreil.ly/N5cd2>).

Обобщенные функции абстрагируют алгоритмы

Мы также можем создавать обобщенные функции. Ранее уже упоминалось, что отсутствие в Go обобщенных типов затрудняло создание реализации функций `Map`, `Reduce` и `Filter`, способных работать с любыми типами. С появлением обобщенных типов эта задача перестает вызывать какие-либо затруднения. Вот примеры реализации этих функций, взятые из описания предварительного дизайна параметров типа:

```
// Функция Map преобразует срез типа []T1 в срез типа []T2,
// используя отображающую функцию.
// Эта функция принимает два параметра типа, T1 и T2.
// Ее можно использовать для срезов любого типа.
func Map[T1, T2 any](s []T1, f func(T1) T2) []T2 {
    r := make([]T2, len(s))
    for i, v := range s {
        r[i] = f(v)
    }
    return r
}

// Функция Reduce редуцирует срез типа []T1 до одного значения,
// используя редуцирующую функцию.
func Reduce[T1, T2 any](s []T1, initializer T2, f func(T2, T1) T2) T2 {
    r := initializer
    for _, v := range s {
        r = f(r, v)
    }
    return r
}

// Функция Filter фильтрует значения среза, используя функцию фильтрации.
// Она возвращает новый срез, содержащий только те элементы среза s,
// для которых функция f возвращает значение true.
func Filter[T any](s []T, f func(T) bool) []T {
    var r []T
    for _, v := range s {
        if f(v) {
            r = append(r, v)
        }
    }
    return r
}
```

Параметры типа указываются в определении функции после ее имени, перед переменными параметрами. Функции `Map` и `Reduce` принимают по два параметра типа `any`, а функция `Filter` — один такой параметр. Запустив следующий код:

```
words := []string{"One", "Potato", "Two", "Potato"}
filtered := Filter(words, func(s string) bool {
```

```
    return s != "Potato"
})
fmt.Println(filtered)
lengths := Map(filtered, func(s string) int {
    return len(s)
})
fmt.Println(lengths)
sum := Reduce(lengths, 0, func(acc int, val int) int {
    return acc + val
})
fmt.Println(sum)
```

мы получим следующий результат:

```
[One Two]
[3 3]
6
```

Вы можете запустить этот код сами, воспользовавшись онлайн-песочницей (<https://oreil.ly/xauU3>).

Как и в случае присвоения значений с помощью оператора `:=`, Go позволяет вам использовать упрощенный способ вызова обобщенных функций за счет автоматического вывода типов. В тех случаях, когда произвести автоматический вывод типов невозможно (например, когда параметр типа используется только в качестве возвращаемого значения), необходимо указывать все параметры типа:

```
type Integer interface {
    type int, int8, int16, int32, int64, uint, uint8, uint16, uint32, uint64
}

func Convert[T1, T2 Integer](in T1) T2 {
    return T2(in)
}

func main() {
    var a int = 10
    b := Convert[int, int64](a)
    fmt.Println(b)
}
```

Списки типов накладывают ограничения на константы и реализации

Списки типов также определяют, какие константы могут присваиваться переменным обобщенного типа. Поскольку типы, перечисленные в интерфейсе

`BuiltInOrdered`, не допускают присваивание констант, вы не можете присвоить константу переменной этого обобщенного типа. В случае интерфейса `Integer` следующий код не скомпилируется, потому что восьмизначному целому числу нельзя присвоить значение `1000`:

```
// НЕ СКОМПИЛИРУЕТСЯ!
func PlusOneThousand[T Integer](in T) T {
    return in + 1_000
}
```

Однако следующий код успешно скомпилируется:

```
// СКОМПИЛИРУЕТСЯ
func PlusOneHundred[T Integer](in T) T {
    return in + 100
}
```

Помимо встроенных типов, списки типов также могут содержать типовые литералы срезов, карт, массивов, каналов или структур. В следующем, слегка надуманном примере эта возможность используется для создания функции, способной копировать значения из любой карты или среза:

```
type Rangeable[E comparable, T any] interface {
    type []T, map[E]T
}

func CopyVals[E comparable, T any, R Rangeable[E, T]](vals R) []T {
    var out []T
    for _, v := range vals {
        out = append(out, v)
    }
    return out
}
```

Этот пример довольно сложен для понимания, поскольку мы используем здесь параметры типа в интерфейсе, который, в свою очередь, тоже используется как параметр типа. Вот как функцию `CopyVals` можно использовать для среза типа `[]string` и карты типа `map[string]float64`:

```
func main() {
    x := []string{"a", "b", "c"}
    var out []string = CopyVals[interface{}, string, []string](x)
    fmt.Println(out)

    y := map[string]float64{"a": 1.2, "b": 3.1, "c": 10}
    var out2 []float64 = CopyVals[string, float64, map[string]float64](y)
    fmt.Println(out2)
}
```

К сожалению, текущая реализация автоматического вывода типов не позволяет производить автоматический вывод параметров типа, и поэтому их нужно указывать при вызове функции. Этот код выдает следующий результат:

```
[a b c]
[3.1 10 1.2]
```

Вы можете запустить этот код сами, воспользовавшись онлайн-песочницей (<https://oreil.ly/K9SWe>).

Элемент списка типов, представляющий собой встроенный тип или литерал типа, позволяет использовать любой тип, базовый тип которого соответствует этому встроенному типу или литералу типа. Если вы укажете в списке типов пользовательские типы, то в качестве параметров типа можно будет использовать только указанные типы.



Если вы укажете в списке типов и встроенные, и пользовательские типы, то этому интерфейсу будет соответствовать любой из указанных пользовательских типов и любой тип, базовый тип которого соответствует указанным встроенным типам.

При указании пользовательского типа вы не получаете автоматически доступ к методам этого типа. Нужные вам методы пользовательских типов должны быть определены в интерфейсе, и все типы, включаемые в список типов, должны реализовывать эти методы; в противном случае они не будут соответствовать интерфейсу. Лучше разобраться в этом нам поможет следующий небольшой пример. Определим несколько типов:

```
type MyInt int
func (mi MyInt) String() string {
    return strconv.Itoa(int(mi))
}

type AlsoMyInt int
func (ami AlsoMyInt) String() string {
    return "42"
}

type MyFloat float64
func (mf MyFloat) String() string {
    return strconv.FormatFloat(float64(mf), 'E', -1, 64)
}
```

Теперь определим функцию, которая будет принимать значение типа `T`, удваивать его, вызывать для него метод `String` и возвращать полученный результат. В качестве ограничения типа мы попробуем использовать несколько разных

интерфейсов, поэтому пока поставим в этом месте знаки вопроса: ??? (в Go не допускается использование такого синтаксиса):

```
func DoubleString[T ???](val T) string {  
    x := val * 2  
    return x.String()  
}
```

Наш первый вариант ограничения типа выглядит следующим образом:

```
type IntOrFloat interface {  
    type int, float64  
}
```

Если мы попытаемся использовать в качестве ограничения типа интерфейс `IntOrFloat`, код не скомпилируется, выдав следующее сообщение об ошибке:

```
x.String undefined (interface IntOrFloat has no method String)
```

Что ж, попробуем применить следующий вариант ограничения типа, включающий в себя типы `MyInt` и `MyFloat`:

```
type MyTypes interface {  
    type MyInt, MyFloat  
}
```

Поскольку мы упустили из виду то, какими методами должны обладать типы, включаемые в список типов, мы получим ту же ошибку на этапе компиляции:

```
x.String undefined (interface MyTypes has no method String)
```

Теперь попробуем включить метод в ограничение типа:

```
type StringableIntOrFloat interface {  
    type int, float64  
    String() string  
}
```

При использовании интерфейса `StringableIntOrFloat` наш код успешно скомпилируется. Поскольку типы `int` и `float64` не реализуют ни метод `String`, ни какие-либо другие методы, в функцию `DoubleString` при этом нельзя передавать переменные этих типов. Если мы так сделаем, это приведет к ошибке на этапе компиляции:

```
int does not satisfy StringableIntOrFloat (missing method String)  
float64 does not satisfy StringableIntOrFloat (missing method String)
```

В то же время мы можем передавать в функцию значения типа `MyInt`, `MyFloat` или `AlsoMyInt`. Наконец, попробуем использовать еще один, последний вариант ограничения типа:

```
type StringableMyType interface {
    type MyInt, MyFloat
    String() string
}
```

Наша функция успешно скомпилируется и при использовании интерфейса `StringableMyType`. При этом в функцию можно передавать только значения типа `MyInt` или `MyFloat`. Если мы попытаемся передать значение типа `AlsoMyInt`, это приведет к ошибке на этапе компиляции:

```
AlsoMyInt does not satisfy StringableMyType (AlsoMyInt or int not found in
MyInt, MyFloat)
```

И еще одно, последнее замечание в отношении списков типов: интерфейс со списком типов можно использовать только в качестве параметра типа.

Что остается «за бортом»

Go по-прежнему остается небольшим и «прозрачным» языком, и в добавляемой в него реализации обобщенных типов отсутствует многое из того, что включают в себя средства обобщенного программирования в других языках. Посмотрим, какие возможности останутся «за бортом» первоначальной реализации обобщенных типов в Go.

Если мы еще раз взглянем на наш пример с двоичным деревом, то увидим, что обобщенные типы позволили нам создать два варианта типобезопасной реализации, один из которых работает со встроенными типами, для которых обеспечена возможность использования операторов `>` и `<` посредством интерфейса `BuiltInOrdered`, а второй — с пользовательскими типами, которые соответствуют определенному нами интерфейсу `Orderable`. Будучи более удачным решением, чем создание отдельных реализаций дерева для каждого типа, эти подходы все же неидеальны. Во многих языках, в частности, в Python, Ruby и C++, можно выполнять *перегрузку операторов*, то есть определять отдельные реализации операторов для каждого пользовательского типа. Эта возможность не будет добавлена в Go, а это значит, что вы не сможете использовать ключевое слово `range` для обхода элементов пользовательского контейнерного типа или квадратные скобки (`[]`) для доступа к ним по индексу.

Для того чтобы оставить «за бортом» перегрузку операторов, есть веские основания. Прежде всего, в Go удивительно много операторов. Кроме того, поскольку

в Go нет перегрузки функций и методов, возникает вопрос о том, как в таком случае пришлось бы определять разную функциональность оператора для разных типов. Более того, перегрузка операторов часто затрудняет понимание кода, поскольку разработчики могут придавать операторам особый, только им понятный смысл (так, например, в C++, оператор << означает для некоторых типов побитовый сдвиг влево, а для других типов — запись значения правого операнда в левый операнд). Разработчики языка Go стремятся по возможности исключать такие проблемы с читабельностью кода.

«За бортом» исходной реализации обобщенных типов в Go останется и такая полезная возможность, как использование параметров типа в методах. Рассмотренные нами функции Map/Reduce/Filter, например, было бы удобнее определить в виде методов:

```
type functionalSlice[T any] []T

// ЭТО НЕ РАБОТАЕТ
func (fs functionalSlice[T]) Map[E any](f func(T) E) functionalSlice[E] {
    out := make(functionalSlice[E], len(fs))
    for i, v := range fs {
        out[i] = f(v)
    }
    return out
}

// ЭТО НЕ РАБОТАЕТ
func (fs functionalSlice[T]) Reduce[E any](start E, f func(E, T) E) E {
    out := start
    for _, v := range fs {
        out = f(out, v)
    }
    return out
}
```

которые можно было бы использовать следующим образом:

```
var numStrings = functionalSlice[string>{"1", "2", "3"}
sum := numStrings.Map(func(s string) int {
    v, _ := strconv.Atoi(s)
    return v
}).Reduce(0, func(acc int, cur int) int {
    return acc + cur
})
```

К сожалению любителей функционального программирования, этот подход не работает. Вместо того чтобы использовать цепной способ вызова методов, вам придется либо вкладывать вызовы функций друг в друга, либо использовать более читабельный подход с вызовом функций по отдельности и присвоением промежуточных значений переменным. В описании предварительного дизайна

параметров типа подробно объясняется, почему разработчики решили отказаться от параметризованных методов, однако при этом остается неясным, как следует взаимодействовать с типом, соответствующим интерфейсу.

Вы также не сможете использовать вариативные параметры типа. В подразделе «Создавайте с помощью рефлексии функции для автоматизации повторяющихся задач» на с. 376 мы написали оберточную функцию, которая использовала рефлексии для вывода информации о времени выполнения любой другой функции. В таком случае вам по-прежнему придется использовать рефлексии, потому что это нельзя реализовать с помощью обобщенных типов. При каждом использовании параметров типа нужно явно указывать имена всех необходимых типов, что делает невозможным представление функции, принимающей произвольное количество параметров разного типа.

В Go не будут включены и следующие, гораздо менее распространенные средства обобщенного программирования.

- *Специализация*. В дополнение к обобщенной версии, функцию или метод можно было бы перегружать одной или несколькими типоспецифичными версиями. Поскольку в Go нет перегрузки функций и методов, добавление этой возможности не планируется.
- *Каррирование (currying)*. Объявление типа на основе обобщенного типа с указанием только определенной части параметров.
- *Метапрограммирование*. Определение кода, выполняемого на этапе компиляции и генерирующего код, выполняемый на этапе выполнения.

Идиоматический Go-код и обобщенные типы

Добавление обобщенных типов, безусловно, внесет определенные коррективы в рекомендации по идиоматическому использованию языка Go. Это положит конец использованию типа `float64` для представления любого числового типа. Мы больше не будем использовать пустой интерфейс (`interface{}`) для представления значений любого типа, применяемых в структуре данных или в качестве параметров функции. Кроме того, теперь можно будет обрабатывать разные типы срезов с помощью одной функции. Однако в то же время не стоит думать, что нам придется сразу же переключиться на использование параметров типа во всем своем коде. По мере того как будут вырабатываться и совершенствоваться новые паттерны проектирования, вы по-прежнему можете использовать свой старый код.

Поскольку на данный момент еще не выпущена окончательная реализация обобщенных типов, сложно сказать, как их использование скажется на

производительности. По всей вероятности, это в какой-то мере скажется и на времени компиляции, и на времени выполнения кода. При этом, как и всегда, нужно стремиться к тому, чтобы создаваемые вами программы были удобными в сопровождении и достаточно быстрыми для того, чтобы удовлетворять предъявляемые к ним требования. Для оценки и повышения производительности кода используйте средства сравнительного тестирования и профилирования, о которых мы говорили в разделе «Сравнительные тесты» на с. 343.

Какие нововведения нас ожидают

Первая версия обобщенных типов должна появиться в версии Go 1.18. Наряду с новым синтаксисом и добавлением идентификаторов `any` и `comparable` во всеобщий блок, также, вероятно, будут внесены изменения, обеспечивающие поддержку обобщенных типов в стандартной библиотеке. Это включает в себя новые интерфейсы для представления распространенных случаев (такие как `Orderable`), новые типы (такие как множество, дерево или упорядоченная карта) и новые функции.

За этим могут последовать и другие изменения. Если в первой версии идентификатор `any` можно использовать в качестве параметра типа для обозначения возможности использования любого типа, то в дальнейшем он, возможно, заменит пустой интерфейс (`interface{}`) во всех случаях его применения. Это никак не скажется на смысле кода и не даст вам никаких новых выразительных возможностей, но позволит вам изменить *внешний вид* кода. Поскольку в Go читабельность играет очень важную роль, изменения такого рода должны вводиться лишь после тщательного анализа.

Обобщенные типы могут послужить основой и для ряда других нововведений. Одним из таких нововведений могут, например, стать *вариантные типы* (*sum types*). Подобно тому как списки типов используются для указания того, какие типы можно подставлять вместо параметра типа, они также могут использоваться и в качестве интерфейсов в переменных параметрах. Это обеспечит нам некоторые интересные возможности. Так, например, при работе с форматом JSON поле часто может содержать или одно значение, или список значений. Даже при наличии обобщенных типов единственным возможным решением в таком случае является использование поля типа `interface{}`. Добавление в Go вариантных типов сделает возможным создание интерфейса, указывающего, что поле может содержать только строку или срез строк. При этом можно будет указать все возможные варианты в переключателе типов, тем самым обеспечив безопасность типов. Эта возможность определения ограниченного набора типов позволяет во многих современных языках (например, в Rust и Swift) использовать вариантные типы для представления перечислений. Учитывая то, насколько

слабой является текущая реализация перечислений в Go, этот подход является достаточно привлекательным решением, однако для оценки и изучения этих идей, вероятно, потребуется некоторое время.

Резюме

В этой главе мы рассмотрели добавляемую в Go реализацию обобщенных типов и выяснили, как их появление скажется на нашем подходе к решению различных задач. Поскольку эта реализация пока не была официально выпущена, в нее еще могут быть внесены изменения, однако, по всей вероятности, они будут минимальными.

Таким образом, мы завершили наш «круз» по возможностям языка Go и идиоматическим паттернам его использования, и, как водится в таком случае, я должен сказать пару заключительных слов своим «выпускникам». Если вы помните, в начале этой книги было сказано следующее: «...хорошо написанный код на Go выглядит скучно... Хорошо написанная программа на Go, как правило, отличается простотой, а часто и некоторым однообразием». Я надеюсь, что теперь вы понимаете, почему это улучшает процесс программной разработки. Идиоматический подход в Go включает в себя множество инструментов, практик и паттернов, которые позволяют упростить сопровождение программного обеспечения с течением времени и изменением состава команды разработчиков. Это не значит, что в других языках не ценится легкость сопровождения, просто этому обычно не придается столь важное значение. Вместо этого первостепенное внимание уделяется таким вещам, как производительность, наличие новейших функциональных возможностей или краткость синтаксиса. Хотя здесь, конечно, и есть место компромиссу, но будущее, как мне кажется, все же за созданием программ, способных работать долгие годы.

Желаю вам успехов в создании программного обеспечения, способного работать еще как минимум 50 лет!

Об авторе

Джон Боднер (Jon Bodner) более 20 лет работает ведущим разработчиком и архитектором. За это время ему приходилось создавать ПО самого разного назначения: для сферы образования, финансов, торговли, здравоохранения, для правоохранительных и государственных органов, инфраструктурное ПО для интернета. Джон исполняет обязанности главного инженера в компании Capital One, где совершенствует процесс разработки и тестирования. Джон часто выступает на конференциях, посвященных языку Go, а его посты по вопросам, связанным с языком Go и программной разработкой в целом, набирают сотни тысяч просмотров.

Об обложке

На обложке книги изображен равнинный карманный гофер (*Geomys bursarius*) — представитель землеройных млекопитающих, обитающий на Великих равнинах Северной Америки. Эти грызуны хорошо приспособлены к подземному образу жизни.

Тело равнинного карманного гофера покрывает коричневый мех, исключение составляет лишь практически голый хвост. Признаки физиологической адаптации к подземной жизни включают небольшой размер глаз, короткие уши и большие передние лапы с когтями. Эти зверьки также хорошо переносят низкое содержание в воздухе кислорода и высокое содержание углекислого газа. Их называют карманными из-за наличия у них защечных мешков (карманов), используемых для переноски пищи.

Гоферы агрессивно защищают свою территорию и редко пытаются зайти в чужую нору. Почти три четверти жизни они проводят в своих норах, где находятся их гнезда и хранятся запасы пищи в виде корней и травы. На поверхность земли эти зверьки выходят только для поиска пищи и партнеров для спаривания.

Равнинному карманному гоферу присвоен охранный статус «Вызывающие наименьшие опасения». Многие из тех видов животных, которые изображены на обложках книг от издательства O'Reilly, находятся под угрозой исчезновения, хотя каждый из них является важной частью нашего мира.

Представленную на обложке иллюстрацию создала Сьюзен Томпсон (Susan Thompson) на основе старинной черно-белой гравюры, взятой из неизвестного источника.

Джон Боднер

Go: идиомы и паттерны проектирования

Перевел с английского С. В. Черников

Заведующая редакцией	<i>Ю. Сергиенко</i>
Руководитель проекта	<i>А. Питиримов</i>
Ведущий редактор	<i>Н. Гринчик</i>
Литературный редактор	<i>Ю. Зорина</i>
Художественный редактор	<i>В. Мостипан</i>
Корректоры	<i>С. Беляева, Е. Павлович</i>
Верстка	<i>Л. Соловьева</i>

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 07.2022.

Наименование: книжная продукция.

Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014,
58.11.12 — Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск,
ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 13.05.22. Формат 70×100/16. Бумага офсетная.
Усл. п. л. 33,540. Тираж 700. Заказ 0000.