

Содержание (сводка)

	Введение	25
1	Знакомство с Go. <i>Основы синтаксиса</i>	35
2	Какой код будет выполняться? <i>Условные команды и циклы</i>	65
3	Вызовы функций. <i>Функции</i>	113
4	Запаковка кода. <i>Пакеты</i>	147
5	И далее по списку. <i>Массивы</i>	183
6	Проблема с присоединением. <i>Сегменты</i>	209
7	Значения и метки. <i>Карты</i>	239
8	Совместное хранение. <i>Структуры</i>	265
9	Ты – мой тип! <i>Определяемые типы</i>	299
10	Все при себе. <i>Инкапсуляция и встраивание</i>	323
11	Что можно сделать? <i>Интерфейсы</i>	355
12	Снова на ногах. <i>Восстановление после сбоев</i>	383
13	Совместное выполнение. <i>Горютины и каналы</i>	413
14	Контроль качества кода. <i>Автоматизация тестирования</i>	435
15	Запросы и ответы. <i>Веб-приложения</i>	459
16	Пример для подражания. <i>Шаблон HTML</i>	479
A	Функция os.OpenFile: <i>Открытие файлов</i>	515
B	Еще шесть тем. <i>Напоследок</i>	529

Содержание (настоящее)

Введение

Ваш мозг и Go. Вы сидите за книгой и пытаетесь что-нибудь выучить, но ваш мозг считает, что вся эта писанина не нужна. Ваш мозг говорит: «Выгляни в окно! На свете есть более важные вещи, например сноуборд». Как заставить мозг изучить программирование на Go?

Для кого написана эта книга?	26
Мы знаем, о чем вы думаете	27
И мы знаем, о чем думает ваш мозг	27
Метапознание: наука о мышлении	29
Вот что сделали МЫ	30
Примите к сведению	32
Благодарности	33

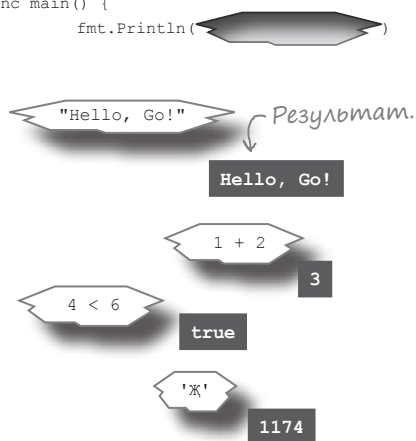
Знакомство с Go

1

ОСНОВЫ СИНТАКСИСА

Готовы поднять свой код на новый уровень? Нужен простой язык программирования, который **быстро компилируется** и **быстро выполняется**? Язык, с которым вы сможете **легко и удобно распространять** свое ПО среди пользователей? Тогда знакомьтесь: **Go** — язык программирования, ориентированный на **простоту** и **скорость**. Он проще других языков, и поэтому вы быстрее освоите его. Кроме того, Go эффективно использует мощь современных многоядерных процессоров, а значит, программы будут выполняться быстрее. В этой главе представлены возможности Go, которые упростят **вам работу** и наверняка придется по вкусу **пользователям**.

```
package main
import "fmt"
func main() {
    fmt.Println(
```



На старт... внимание... Go!	36
Интерактивная среда Go Playground	37
Что это все означает?	38
Структура типичного файла Go	38
А если что-то пойдет не так?	39
Вызов функций	41
Функция Println	41
Использование функций из других пакетов	42
Возвращаемые значения функций	43
Шаблон программы Go	45
Строки	45
Руны	46
Логические значения	46
Числа	47
Математические операции и сравнения	47
Типы	48
Объявление переменных	50
Нулевые значения	51
Короткие объявления переменных	53
Правила выбора имен	55
Преобразования	56
Установка Go на вашем компьютере	59
Компиляция кода Go	60
Инструменты Go	61
Быстрый запуск кода командой «go run»	61
Ваш инструментарий Go	62

Какой код будет выполняться?

2

Условные команды и циклы

В каждой программе есть части, которые должны выполняться только в определенных ситуациях. «Этот код должен выполняться, *если* произошла ошибка. А если нет — должен выполняться этот код». Почти в каждой программе присутствует код, который должен выполняться только в том случае, если некоторое *условие* истинно. Почти в каждом языке программирования существуют **условные команды**, которые позволяют определить, нужно ли выполнять те или иные сегменты кода. Язык Go не исключение. Возможно, какие-то части кода должны выполняться *множественно*. Как и многие языки, Go поддерживает **циклы** для многократного выполнения блоков кода. В этой главе вы научитесь применять как условные команды, так и циклы!

```

if 1 < 2 {
    fmt.Println("It's true!")
}

```

Ключевое слово «if».

Условие.

Начало условного блока.

Тело условного блока.

Конец условного блока.

Вызов методов	66
Проверка результата	68
Комментарии	68
Получение значения от пользователя	69
Множественные возвращаемые значения функций или методов	70
Вариант 1. Игнорировать возвращаемое значение ошибки	71
Вариант 2. Обработка ошибки	72
Условные команды	73
Условная выдача фатальной ошибки	76
Избегайте замещения имен	78
Преобразование строк в числа	80
Блоки	83
Создание игры	89
Имена пакетов и пути импортирования	90
Генерирование случайных чисел	91
Получение целого числа с клавиатуры	93
Сравнение предположения с загаданным числом	94
Циклы	95
Операторы инициализации и приращения необязательны	97
Циклы и области видимости	97
Использование цикла в игре	100
Пропуск частей цикла командами continue и break	102
Выход из цикла	103
Вывод загаданного числа	104
Поздравляем, игра готова!	106
Ваш инструментарий Go	108

3

Вызовы функций

Функции

И все же чего-то не хватало. Вы вызывали функции как настоящий профи. Но могли вызывать только те функции, которые были определены для вас в Go. Настала ваша очередь. В этой главе мы покажем, как создавать собственные функции. Вы научитесь объявлять функции с параметрами и без. Сначала вы узнаете, как объявлять функции, которые возвращают одно значение, а потом мы перейдем к возвращению нескольких значений, чтобы функция могла сигнализировать об ошибке. А еще в этой главе рассматриваются **указатели**, которые повышают эффективность вызовов функций по затратам памяти.



Повторяющийся код	114
Форматирование вывода функциями Printf и Sprintf	115
Глаголы форматирования	116
Форматирование значений ширины	117
Форматирование с дробными значениями ширины	118
Использование Printf в программе	119
Объявление функций	120
Объявление параметров функции	121
Использование функций в программе	122
Функции и области видимости переменных	124
Возвращаемые значения функций	125
Использование возвращаемого значения в программе	127
Функции paintNeeded нужна обработка ошибок	129
Значения ошибок	130
Объявление нескольких возвращаемых значений	131
Использование множественных возвращаемых значений с функцией paintNeeded	132
Всегда обрабатывайте ошибки!	133
В параметрах функций хранятся копии аргументов	136
Указатели	137
Типы указателей	138
Чтение или изменение значения по указателю	139
Использование указателей с функциями	141
Исправление функции «double» с использованием указателей	142
Ваш инструментарий Go	144

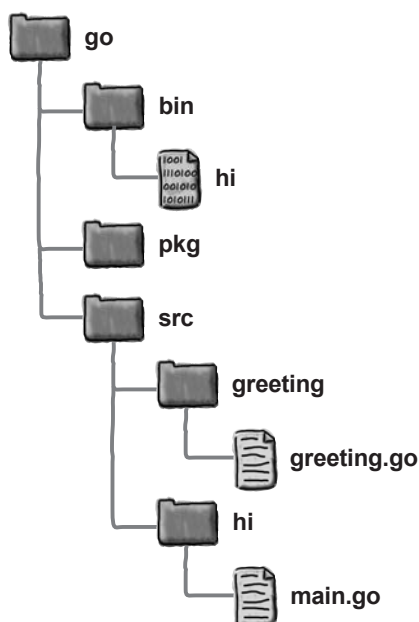
4

Запаковка кода

Пакеты

Время навести порядок! До сих пор мы сваливали весь свой код в один файл. Но чем больше и сложнее будут становиться наши программы, тем скорее такой подход приведет к хаосу. В этой главе мы научим вас создавать **пакеты** для хранения взаимосвязанного кода в одном месте. Но пакеты нужны не только для организации кода. Пакеты предоставляют простые средства для *повторного использования кода в разных программах*. А еще это простой способ *распространения кода среди разработчиков*.

Разные программы, одна функция	148
Пакеты и повторное использование кода в программах	150
Хранение кода пакетов	151
Создание нового пакета	152
Импорт пакета в программу	153
Файлы пакетов имеют одинаковую структуру	154
Соглашения по выбору имен пакетов	157
Уточнение имен	157
Перемещение общего кода в пакет	158
Константы	160
Вложенные каталоги и пути импорта пакетов	162
Установка исполняемых файлов командой «go install»	164
Переменная GOPATH и смена рабочих областей	165
Настройка GOPATH	166
Публикация пакетов	167
Загрузка и установка пакетов командой «go get»	171
Чтение документации пакетов командой «go doc»	173
Документирование пакетов	175
Просмотр документации в браузере	177
Запуск сервера документации HTML командой «godoc»	178
Сервер «godoc» включает ВАШИ пакеты!	179
Ваш инструментарий Go	180



5

И далее по списку

Массивы

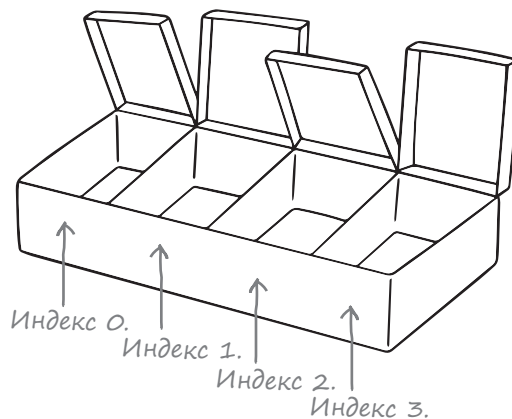
Многие программы работают со списками. Списки адресов. Списки телефонных номеров. Списки товаров. В Go существуют два встроенных способа хранения списков. В этой главе мы разберем первый способ: **массивы**. Вы научитесь создавать массивы, заполнять их данными и извлекать сохраненные данные. Далее рассмотрим способы обработки всех элементов в массиве: сначала *сложный* вариант с циклами `for`, а затем *простой* — с циклами `for...range`.

В массивах хранятся наборы значений	184
Нулевые значения в массивах	186
Литералы массивов	187
Функции пакета «fmt» умеют работать с массивами	188
Обращение к элементам массива в цикле	189
Проверка длины массива функцией «len»	190
Безопасный перебор массивов в цикле «for...range»	191
Пустой идентификатор в циклах «for...range»	192
Суммирование чисел в массиве	193
Вычисление среднего значения	195
Чтение текстового файла	197
Чтение текстового файла в массив	200
Чтение текстового файла в программе «average»	202
Наша программа может обрабатывать только три значения!	204
Ваш инструментарий Go	206

Количество элементов в массиве.

Тип элементов в массиве.

```
var myArray [4]string
```



6 Проблема с присоединением Сегменты

Вы уже знаете, что в массив нельзя добавить новые элементы.

В нашей программе это создает настоящие проблемы, потому что количество значений данных в файле неизвестно заранее. На помощь приходят **сегменты Go**. Сегменты — разновидность коллекций, которые могут расширяться для хранения дополнительных элементов; а это как раз то, что нужно! Вы также увидите, как использовать сегменты для простой передачи данных *любым* программам и как с их помощью пишутся функции, которые удобно вызывать.



Сегменты	210
Литералы сегментов	211
Оператор сегмента	214
Базовые массивы	216
При изменении базового массива изменяется сегмент	217
Расширение сегментов функцией «append»	218
Сегменты и нулевые значения	220
Чтение строк из файла с использованием сегментов и «append»	221
Тестирование измененной программы	223
Возвращение сегмента nil в случае ошибки	224
Аргументы командной строки	225
Получение аргументов командной строки из сегмента os.Args	226
Оператор сегмента может использоваться с другими сегментами	227
Использование аргументов командной строки в программе	228
Функции с переменным количеством аргументов	229
Использование функций с переменным количеством аргументов	231
Использование функции с переменным количеством параметров для вычисления среднего значения	232
Передача сегментов функциям с переменным количеством аргументов	233
Сегменты спасли положение!	235
Ваш инструментарий Go	236

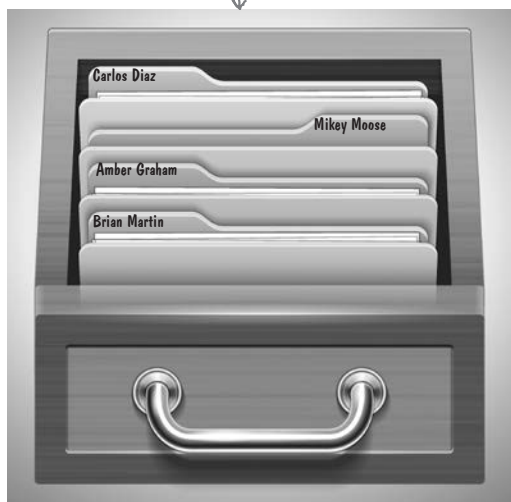
Значения и Метки

7

Карты

Сваливать все в одну кучу удобно до тех пор, пока не потребуется что-нибудь найти. Вы уже видели, как создавать списки значений в *массивах* и *сегментах*. Вы знаете, как применить одну операцию к *каждому значению* в массиве или сегменте. Но что, если требуется поработать с *конкретным* значением? Чтобы найти его, придется начать с начала массива или сегмента и *просмотреть*. *Каждое. Существующее. Значение.* А если бы существовала разновидность коллекций, в которой каждое значение снабжается специальной меткой? Тогда нужное значение можно было бы быстро найти по метке! Эта глава посвящена **картам**, которые предназначены именно для этого.

Ключи ускоряют
поиск данных!



Карта

Подсчет голосов	240
Чтение имен из файла	241
Подсчет имен: сложный способ с сегментами	243
Карты	246
Карты (продолжение)	247
Литералы карт	248
Нулевые значения с картами	249
Нулевое значение для карты равно nil	249
Как отличить нулевые значения от присвоенных	250
Удаление пар «ключ/значение» функцией «delete»	252
Версия программы с использованием карты	253
Циклы for...range с картами	255
Цикл for...range обрабатывает карты в случайном порядке!	257
Обновление программы подсчета голосов циклом for...range	258
Программа подсчета голосов готова!	259
Ваш инструментарий Go	261

8 Совместное хранение Структуры

Иногда требуется хранить вместе несколько типов данных.

Сначала вы познакомились с сегментами, предназначенными для хранения списков. Затем были рассмотрены карты, связывающие список ключей со списком значений. Но обе структуры данных позволяют хранить значения только *одного* типа, а в некоторых ситуациях требуется сгруппировать значения *нескольких* типов. Например, в почтовых адресах названия улиц (строки) группируются с почтовыми индексами (целые числа). Или в информации о студентах имена (строки) объединяются со средними оценками (вещественные числа). Сегменты и карты не позволяют смешивать разные типы. Тем не менее это *возможно* при использовании другого типа данных, называемого **структурой**. В этой главе мы подробно изучим структуры.



В сегментах и картах хранятся значения одного типа	266
Структуры формируются из значений многих типов	267
Обращение к полям структуры	268
Хранение данных подписчиков в структуре	269
Определения типов и структуры	270
Использование определяемого типа для информации о подписчиках	272
Использование определяемых типов с функциями	273
Изменение структуры в функции	276
Обращение к полям структур по указателю	278
Передача больших структур с помощью указателей	280
Перемещение типа структуры в другой пакет	282
Экспорт определяемых типов	283
Экспорт полей структур	284
Литералы структур	285
Создание типа структуры Employee	287
Создание типа структуры Address	288
Добавление структуры как поля в другой тип	289
Создание вложенной структуры	289
Анонимные поля структур	292
Встроенные структуры	293
Наши определяемые типы готовы!	294
Ваш инструментарий Go	295

9

Ты — Мой тип!

Определяемые типы

Мы еще не все рассказали об определяемых типах. В предыдущей главе вы узнали, как определяются типы, у которых базовым типом является тип структуры. Но при этом мы *не сказали*, что в качестве базового может использоваться *любой* тип.

Помните о методах — особой разновидности функций, связываемой со значениями определенного типа? В книге неоднократно встречались примеры вызова методов для разных значений, но *собственные* методы вы еще не умеете определять. В этой главе мы исправим это. А теперь за дело!

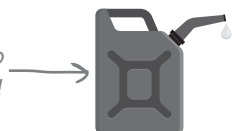
Ошибки типов в реальной жизни	300
Определяемые типы с базовыми основными типами	301
Определяемые типы и операторы	303
Преобразования типов с помощью функций	305
Разрешение конфликтов имен с использованием методов	308
Определение методов	309
Параметр получателя (почти) не отличается от других параметров	310
Метод (почти) не отличается от функции	311
Указатели и параметры получателей	313
Преобразование литров и миллилитров в галлоны с помощью методов	317
Преобразование Gallons в Liters и Milliliters с помощью методов	318
Ваш инструментарий Go	319

Сколько куплено топлива по представлению Стива



10 галлонов

Сколько куплено на самом деле!



10 литров

10

Все при себе

Инкапсуляция и встраивание

Ошибки случаются. Иногда программа получает недействительные данные от пользователя, из файла или другого источника. В этой главе рассматривается **инкапсуляция**: механизм защиты полей структурного типа от недействительных данных. И будьте уверены — теперь с данными полями можно безопасно работать! Мы также покажем, как **встраивать** другие типы в структуры. Если вашему типу структуры потребуются методы, которые уже существуют у другого типа, вам не придется копировать и вставлять код метода. Вы можете встроить другой тип в свой тип структуры, а затем воспользоваться методами встроеного типа так, если бы они были определены для вашего собственного типа!

Проверка данных в set-методах прекрасно работает... когда пользователи вызывают их. Но они обращаются к полям структур напрямую и продолжают вводить недопустимые данные!



Создание типа структуры Date	324
Пользователи заполняют поле структуры Date недопустимыми значениями!	325
Set-методы	326
Set-методам необходимы указатели на получателей	327
Добавление остальных set-методов	328
Включение проверки данных в set-методы	330
Полям все равно можно присвоить недопустимые значения!	332
Перемещение типа Date в другой пакет	333
Отмена экспортирования полей Date	335
Обращения к неэкспортируемым полям через экспортируемые методы	336
Get-методы	338
Инкапсуляция	339
Встраивание типа Date в тип Event	342
Неэкспортируемые поля не повышаются	343
Экспортируемые методы повышаются так же, как и поля	344
Инкапсуляция поля Title типа Event	346
Повышенные методы существуют наряду с методами внешнего типа	347
Пакет calendar готов!	348
Ваш инструментарий Go	350

11

Что можно сделать?

Интерфейсы

Иногда конкретный тип значения не важен. Вас не интересует, с чем вы работаете. Вы просто хотите быть уверены в том, что оно может делать то, что нужно вам. Тогда вы сможете вызывать для значения *определенные методы*. Неважно, с каким значением вы работаете — Pen или Pencil; вам просто нужно нечто, содержащее метод Draw. Именно эту задачу решают **интерфейсы** в языке Go. Они позволяют определять переменные и параметры функций, которые могут хранить *любой* тип при условии, что этот тип определяет некоторые методы.

Два разных типа с одинаковыми методами	356
В параметре метода может передаваться только один тип	357
Интерфейсы	359
Определение типа, поддерживающего интерфейс	360
Конкретные типы и типы интерфейсов	361
Присваивание любого типа, поддерживающего интерфейс	362
Вызывать можно только методы, определенные как часть интерфейса	363
Исправление функции playList с помощью интерфейса	365
Утверждения типа	368
Ошибки утверждений типа	370
Предотвращение паники при ошибках утверждений типов	371
Тестирование TapePlayer и TapeRecorder с утверждениями типов	372
Интерфейс еггор	374
Интерфейс Stringer	376
Пустой интерфейс	378
Ваш инструментарий Go	381



12

Снова на нoГax

Восстановление после сбоев

Любая программа сталкивается с ошибками. Учтите их при планировании. Иногда обработка ошибки сводится к простому выводу сообщения и завершению программы. Другие ошибки требуют дополнительных действий: например, закрытия открытых файлов или сетевых подключений или иного освобождения ресурсов, чтобы после вашей программы оставался порядок. В этой главе мы покажем, как **отложить** завершающие действия, чтобы они выполнились даже в случае ошибки. Также вы узнаете, как поднять **панику** в тех (редких) ситуациях, когда это уместно, и как **восстановиться** после нее.

Снова о чтении чисел из файла	384
Любая ошибка мешает закрытию файла!	386
Отложенные вызовы функций	387
Восстановление после ошибок	388
Использование отложенного вызова для гарантированного закрытия файлов	389
Получение списка файлов в каталоге	392
Рекурсивные вызовы функций	394
Рекурсивный вывод содержимого каталога	396
Обработка ошибок в рекурсивной функции	398
Запуск паники	399
Трассировка стека	400
Отложенные вызовы завершаются перед аварийным завершением	400
Использование panic с scanDirectory	401
Когда стоит паниковать	402
Функция recover	404
Возвращаемое значение recover	405
Восстановление после паники в scanDirectory	407
Возобновление состояния паники	408
Ваш инструментарий Go	410

Не преоб-
разуется
в float64!



bad-data.txt

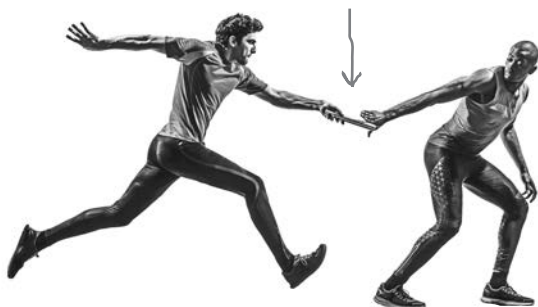
13

Совместное Выполнение Горутин и каналы

Одновременная работа над одной задачей не всегда быстрее всего приводит к цели. Некоторые большие задачи можно разбить на мелкие. **Горутин** (**goroutines**) позволяют программам работать над несколькими задачами одновременно. Они координируют свою работу при помощи **каналов**, по которым могут отправлять данные друг другу и синхронизировать выполнение, чтобы одна горутин не опережала другую. Горутин позволяют использовать всю мощь многопроцессорных компьютеров, чтобы программы выполнялись как можно быстрее!

Загрузка веб-страниц	414
Многозадачность	416
Конкурентность на базе горутин	417
Использование горутин	418
Использование горутин с функцией <code>responseSize</code>	420
Порядок выполнения горутин	422
Горутин не могут использоваться с возвращаемыми значениями	423
Отправка и получение значений в каналах	425
Каналы и синхронизация горутин	426
Соблюдение синхронизации в горутине	427
Использование каналов в программе для вывода размера веб-страниц	430
Изменение канала для передачи структуры	432
Ваш инструментарий Go	433

*Получающая горутин ожидает,
когда другая горутин отправит
значение.*



14

Контроль качества кода

Автоматизация тестирования

А вы уверены, что ваша программа работает правильно?

Точно уверены? Прежде чем рассылать новую версию пользователям, вы, скорее всего, опробовали ее новые возможности и убедились, что все работает. Но опробовали ли вы *старые* возможности? А вдруг что-нибудь сломалось в процессе доработки? Все старые возможности? Если от этого вопроса вам стало слегка не по себе, значит, вашим программам необходимо **автоматизированное тестирование**. Автоматизированные тесты гарантируют, что компоненты программы работают правильно даже после того, как вы внесете изменения в код. Средства тестирования Go и команда `go test` упрощают написание автотестов.

Автотесты обнаруживают ошибки до того, как их обнаружат пользователи	436
Функция, для которой нужны автотесты	437
Мы внесли ошибку!	439
Написание тестов	440
Выполнение тестов командой «go test»	441
Тестирование возвращаемых значений	442
Метод «ErrGotf» и подробные сообщения о непрохождении тестов	444
«Вспомогательные» тестовые функции	445
Прохождение тестов	446
Разработка через тестирование	447
Еще одна ошибка	448
Выполнение определенных наборов тестов	451
Табличные тесты	452
Табличные тесты (продолжение)	453
Решение проблемы с паникой при помощи тестов	454
Ваш инструментарий Go	456



Проходит.



```
For []slice{"apple", "orange", "pear"}, JoinWithCommas
should return "apple, orange, and pear".
```

Ошибка!



```
For []slice{"apple", "orange"}, JoinWithCommas should
return "apple and orange".
```

15

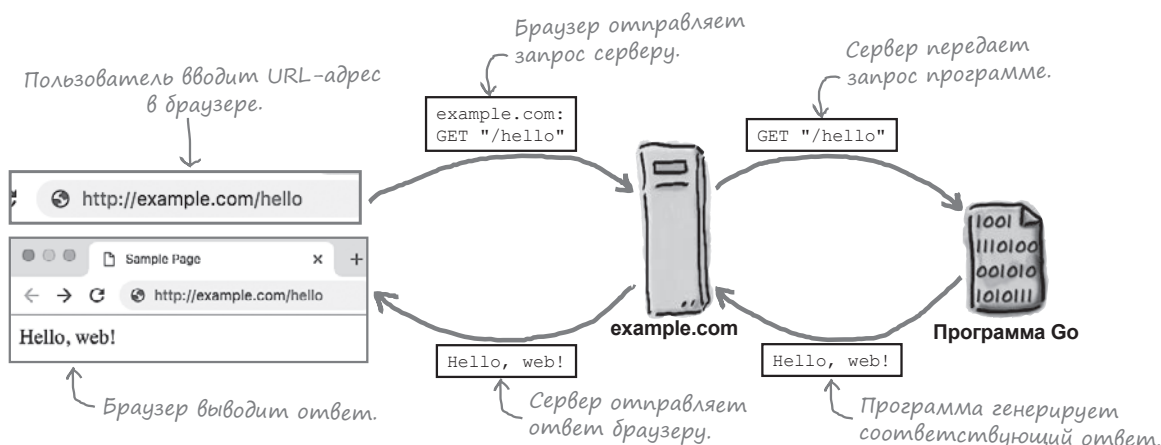
Запросы и ответы

Веб-приложения

Мы живем в XXI веке. Пользователям нужны веб-приложения.

Go поможет и в этом! Стандартная библиотека Go включает пакеты, при помощи которых можно размещать собственные веб-приложения и делать их доступными для любого веб-браузера. Последние две главы этой книги будут посвящены построению веб-приложений. Первое, что необходимо веб-приложению — способность отвечать на запросы, отправляемые браузером. В этой главе вы узнаете, как реализовать эту функцию с использованием пакета `net/http`.

Написание веб-приложений на языке Go	460
Браузеры, запросы, серверы и ответы	461
Простое веб-приложение	462
Ваш компьютер общается сам с собой	463
Простое веб-приложение: шаг за шагом	464
Пути к ресурсам	466
Разные ответы для разных путей к ресурсам	467
Функции первого класса	469
Передача функций другим функциям	470
Функции как типы	470
Что дальше	474
Ваш инструментарий Go	475



16

Пример для подражания

Шаблон HTML

Веб-приложение должно выдавать ответ с разметкой HTML, а не с простым текстом. Для сообщений электронной почты и постов в соцсетях достаточно простого текста. Тем не менее ваши страницы должны быть отформатированы с выделением заголовков и разбиением на абзацы. На них должны быть формы, в которых пользователь сможет ввести данные для приложения. Для решения таких задач вам понадобится разметка HTML.

Рано или поздно в разметку HTML потребуется вставлять данные. Для этого в Go существует пакет `html/template` — мощный инструмент для вставки данных в HTML-ответы приложения. Шаблоны играют ключевую роль при построении более масштабных и качественных веб-приложений, и в последней главе книги мы научим вас ими пользоваться!

Гостевая книга	480
Функции обработки запроса и проверки ошибок	481
Создание каталога проекта и пробный запуск	482
Создание списка записей в HTML	483
Как заставить приложение отвечать разметкой HTML	484
Пакет « <code>text/template</code> »	485
Использование интерфейса <code>io.Writer</code> с методом шаблона <code>Execute</code>	486
<code>ResponseWriter</code> и <code>os.Stdout</code> поддерживают <code>io.Writer</code>	487
Вставка данных в шаблоны при помощи действий	488
Необязательные действия « <code>if</code> » в шаблонах	489
Повторение частей шаблонов в действиях « <code>range</code> »	490
Вставка полей структуры в шаблон	491
Чтение сегмента записей из файла	492
Структура для хранения записей и количества записей	494
Обновление шаблона для включения записей	495
Ввод данных в формах HTML	498
Включение формы HTML в ответ	499
Запросы на отправку данных формы	500
Путь и метод HTTP для отправки данных формы	501
Получение значений полей формы из запроса	502
Сохранение данных формы	504
Перенаправления HTTP	506
Полный код приложения	508
Ваш инструментарий Go	511

Функция `os.OpenFile`

Приложение А. Открытие файлов

Некоторые программы не только читают данные, но и записывают их в файлы. Когда в этой книге мы собирались поработать с файлами, приходилось создавать их в текстовом редакторе, чтобы программа могла прочитать данные. Но некоторые программы *генерируют* данные, и когда это происходит, программа должна иметь возможность *записать* данные в файл. Функция `os.OpenFile` уже использовалась для открытия файла для записи. Но у нас не было возможности объяснить, как она работает. В этом приложении вы узнаете все, что необходимо знать для эффективного использования `os.OpenFile`!

Как работает <code>os.OpenFile</code>	516
Передача констант флагов функции <code>os.OpenFile</code>	517
Двоичная запись	519
Побитовые операторы	519
Побитовый оператор И	520
Побитовый оператор ИЛИ	521
Побитовый оператор ИЛИ и константы пакета «os»	522
Решение проблемы с параметрами <code>os.OpenFile</code>	523
Разрешения доступа к файлам в стиле Unix	524
Представление разрешений с типом <code>os.FileMode</code>	525
Восьмеричная система счисления	526
Преобразование восьмеричных значений в <code>FileMode</code>	527
Подробный анализ вызова <code>os.OpenFile</code>	528

На этот раз
новый текст
присоединя-
ется в конец
файла.



Еще шесть тем

Приложение Б. Напоследок

Позади большой путь, и книга почти подошла к концу. Мы будем скучать, но, пожалуй, было бы неправильно отпускать вас в свободное плавание без *небольшой* дополнительной подготовки. Мы приберегли шесть важных тем для этого приложения.

№ 1. Команды инициализации для «if»	530
№ 2. Команда switch	532
№ 3. Другие базовые типы	533
№ 4. О рунах	533
№ 5. Буферизованные каналы	537
№ 6. Дополнительные ресурсы	540

Все символы имеют печатное представление.

```

if count := 5; count > 4 {
    fmt.Println("count is", count)
}
  
```

Команда инициализации.

Условие.

0: A
 1: B
 2: C
 3: D
 4: E
 0: E
 2: Г
 4: Д
 6: Ж
 8: И

Отправка значения при заполненном буфере приводит к блокировке отправляющей горютины.

"d"

"c"

"b"

"a"

Другие отправляемые значения добавляются в буфер, пока он не заполнится.

Конкурентность на базе горутин

Когда `responseSize` вызывает `http.Get`, вашей программе приходится ожидать ответа сайта. Во время ожидания она не делает ничего полезного.

Возможно, другой программе придется дожидаться пользовательского ввода. А еще одной программе придется ждать чтения данных из файла. Существует множество ситуаций, в которых программам приходится просто простаивать в ожидании.

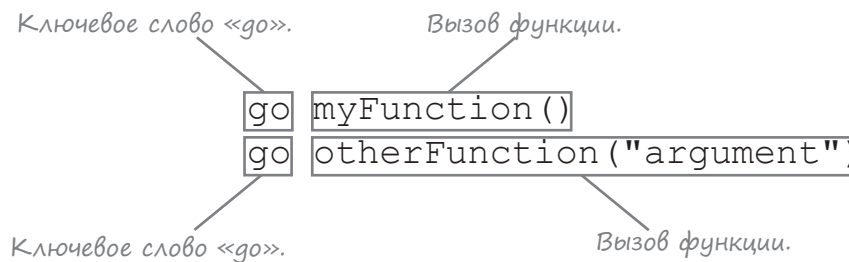
Конкурентность позволяет программе приостановить одну задачу и работать над другими задачами. Программа, ожидающая ввода от пользователя, может выполнять другие действия в фоновом режиме. Во время чтения из файла программа может обновлять индикатор прогресса. Наша программа `responseSize` может выдавать другие сетевые запросы, ожидая завершения первого запроса.

Если программа написана с поддержкой конкурентности, она также может поддерживать **параллельное выполнение**: *одновременное* выполнение задач. Компьютер с одним процессором может выполнять только одну задачу за раз. Тем не менее большинство современных компьютеров содержит несколько процессоров (или один процессор с несколькими ядрами). Ваш компьютер может распределить несколько конкурентных задач между несколькими процессорами, чтобы выполнять их одновременно. (Вам редко приходится управлять ими вручную: операционная система обычно делает все за вас.)

Разбиение больших задач на меньшие подзадачи, которые могут выполняться конкурентно, иногда приводит к существенному приросту скорости ваших программ.

В Go конкурентно выполняемые задачи называются **горутинами**. В других языках программирования существует аналогичная концепция *потоков*, но горутины расходуют меньше компьютерной памяти, чем потоки, а также быстрее запускаются и останавливаются, а это означает, что вы можете запускать больше горутин одновременно.

Кроме того, горутины проще в использовании. Для запуска новой горутины используется `go`-команда — обычный вызов функции или метода, перед которым находится ключевое слово `go`:



Обратите внимание: мы говорим «*другую* горутину». Функция `main` каждой программы Go запускается с помощью горутин, так что в каждой программе Go выполняется по крайней мере одна горутина. Выходит, мы все это время пользовались горутинами, не подозревая об этом!

Горутины обеспечивают возможность конкурентности: приостановки одной задачи для работы над другими задачами. А в других ситуациях они позволяют реализовать параллелизм: одновременную работу над несколькими задачами!

Использование горутин

Следующая программа вызывает функции по одной. В ней используется цикл, который выводит строку "a" 50 раз, а функция b выводит строку "b" 50 раз. Функция main вызывает a, затем b, а потом выводит сообщение при завершении.

```
package main

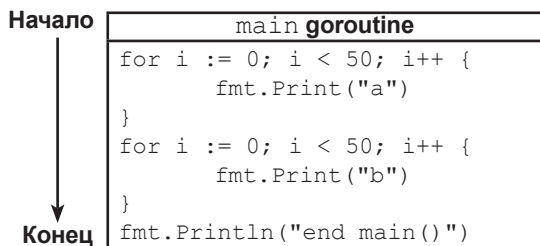
import "fmt"

func a() {
    for i := 0; i < 50; i++ {
        fmt.Print("a")
    }
}

func b() {
    for i := 0; i < 50; i++ {
        fmt.Print("b")
    }
}

func main() {
    a()
    b()
    fmt.Println("end main()")
}
```

Все происходит так, как если бы функция main со-держала весь код функции a, за которым следовал бы весь код функции b и собственный код main:

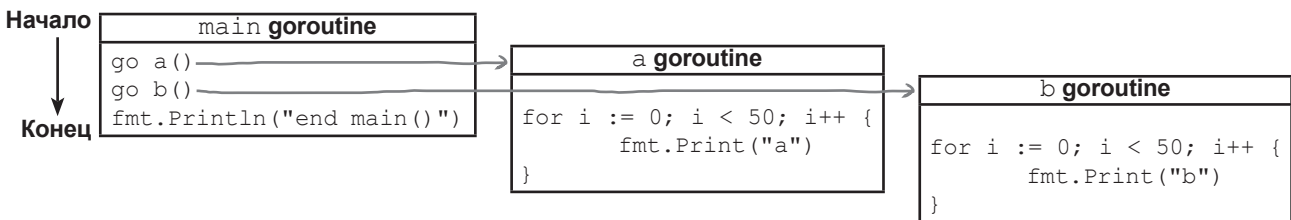


```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
bbbbend main()
```

Чтобы запустить функции a и b в новых горутинах, достаточно добавить ключевое слово go перед вызовами функций:

```
func main() {
    go a()
    go b()
    fmt.Println("end main()")
}
```

Тогда новые горутинны будут конкурентно выполняться в функции main:



Использование горутин (продолжение)

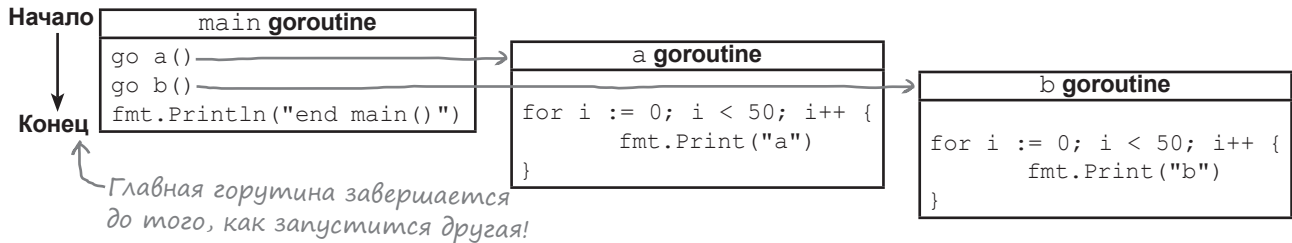
Но если запустить программу сейчас, то единственный результат, который мы увидим, будет выведен функцией `Println` в конце `main`. А вот функции `a` и `b` ничего не выведут!

```
func main() {
    go a()
    go b()
    fmt.Println("end main()")
}
```

Нет вывода функций
«a» и «b»?

end main()

Проблема вот в чем: программы Go перестают выполняться сразу же после завершения горутин `main` (той, которая вызывает функцию `main`), даже если другие горутин продолжают выполняться. Наша функция `main` завершается до того, как код функций `a` и `b` получит возможность выполниться.



Горутин `main` должна продолжать выполнение до того, как горутин функций `a` и `b` смогут завершиться. Чтобы правильно реализовать эту последовательность выполнения, понадобится еще одно средство Go — так называемые *каналы*, но этот механизм будет рассматриваться чуть позже в этой главе. Итак, пока мы просто приостановим горутин `main` на заданный период времени, чтобы смогли выполниться другие горутин.

Для этого мы воспользуемся функцией `Sleep` из пакета `time`. Эта функция приостанавливает текущую горутин на заданный промежуток времени. Вызов `time.Sleep(time.Second)` из функции `main` заставит горутин `main` приостановить выполнение на 1 секунду.

```
func main() {
    go a()
    go b()
    time.Sleep(time.Second)
    fmt.Println("end main()")
}
```

Горутин `main` приостанавливается на 1 секунду.

Дает другим горутинам достаточно времени для выполнения.

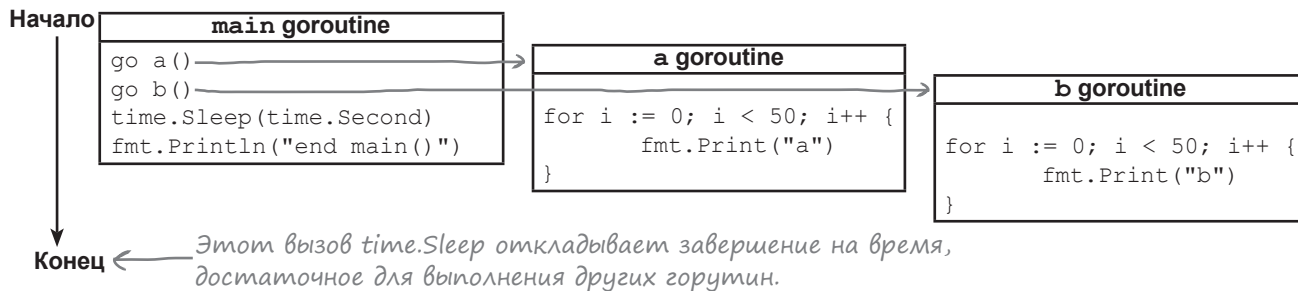
aaaaaaaaaaaaaaaaaaaaaabbbbbaaa
 aaaaaaabbbbbbbbbbaaaaaaaaaa
 abbaaaabbbbbbbbbbbbbbbbbbb
 bbbbbbbbbbend main()

Когда `time.Sleep` вернет управление, горутин `main` завершит выполнение.

Если повторно запустить программу, вы снова увидите выходы функций `a` и `b`, так как их горутин получают возможность выполниться. Вывод этих функций будет чередоваться, так как программа переключается между двумя горутинами. (Закономерность чередования может отличаться от той, которая показана здесь.) Когда Go-функция `main` снова активизируется, она вызовет `fmt.Println` и завершится.

Использование горутин (продолжение)

Вызов `time.Sleep` в главной горутине дает более чем достаточно времени для выполнения функций `a` и `b`.



Использование горутин с функцией `responseSize`

Программа для вывода размеров веб-страниц легко преобразуется для использования горутин. Для этого понадобится совсем немного — добавить ключевое слово `go` перед каждым вызовом `responseSize`.

Чтобы горутина `main` не завершилась перед тем, как горютины `responseSize` смогут завершиться, также нужно будет добавить вызов `time.Sleep` в функцию `main`.

```

package main

import (
    "fmt"
    "io/ioutil"
    "log"
    "net/http"
    "time" ← Добавляется пакет «time».
)

```

```

func main() {
    Вызовы responseSize преобразуются в go-команды.
    Пятисекундная пауза.
    go responseSize("https://example.com/")
    go responseSize("https://golang.org/")
    go responseSize("https://golang.org/doc")
    time.Sleep(5 * time.Second)
}

```

Впрочем, приостановка всего на 1 секунду может оказаться недостаточной для завершения сетевых запросов. Вызов `time.Sleep(5 * time.Second)` обеспечит приостановку горютины на 5 секунд. (Если вы попытаете запустить эту программу с медленной или перегруженной сетью, возможно, это время придется увеличить.)

```

func responseSize(url string) {
    fmt.Println("Getting", url)
    response, err := http.Get(url)
    if err != nil {
        log.Fatal(err)
    }
    defer response.Body.Close()
    body, err := ioutil.ReadAll(response.Body)
    if err != nil {
        log.Fatal(err)
    }
    fmt.Println(len(body))
}

```