1 Начало работы с Go

В этой главе

- ✓ Знакомство с языком Go.
- ✓ Место Go среди других языков программирования.
- ✓ Первые шаги в работе с Go.

В этой главе дается обзор языка Go и его ключевых особенностей. В ней закладывается основа, необходимая профессиональному разработчику для применения Go в процессе решения реальных рабочих задач. Вы познакомитесь с самим языком и сопровождающим его инструментарием, узнаете, какое место Go занимает среди других языков программирования, а также научитесь устанавливать Go и сможете приступить к разработке собственного приложения или модуля.

Если вы уже давно используете Go, то часть материала может показаться знакомой и вам наверняка захочется сразу перейти к следующим главам. Тем не менее за последние годы язык претерпел значительные изменения. Он по-прежнему остается лаконичным и сфокусированным на практических задачах, но благодаря отзывам участников сообщества в него были добавлены мощные новые возможности, включая поддержку обобщений.

1.1. Язык Go

Go (иногда называемый *Golang*) — это статически типизированный компилируемый язык программирования с открытым исходным кодом, изначально разработанный компанией Google. В 2007 году Роберт Гризмер, Роб Пайк и Кен

Томпсон начали работу над языком системного программирования нового поколения, предназначенным для решения проблем, возникающих при масштабировании больших систем. Публичный анонс языка состоялся в ноябре 2009 года, и вскоре он начал стремительно набирать популярность.

Вместо стремления к теоретической чистоте создатели Go сосредоточились на реальных практических задачах. Они черпали вдохновение из лучших языков прошлого, таких как C, Pascal, Java и Python, стараясь перенять их сильные стороны и при этом свести сложность к минимуму. Результатом стал мощный, но по-прежнему легкий язык. Go обладает лаконичным синтаксисом, выдающимся набором инструментов и производительностью, которая делает его особенно привлекательным на фоне многих современных альтернатив.

Go — это не обычный компилируемый язык со статической типизацией. Благодаря ряду особенностей статическая типизация в Go создает впечатление динамической, а скомпилированные исполняемые файлы работают в среде выполнения со встроенным сборщиком мусора. Разработчики Go изначально ориентировались на проекты, характерные для Google, в частности связанные с огромными кодовыми базами, над которыми работают большие команды разработчиков.

В своей основе Go представляет собой язык программирования, определенный спецификацией, реализация которой может быть выполнена любым компилятором. Стандартная реализация поставляется через инструмент командной строки `go`. Однако Go — это не только язык. Как показано на рис. 1.1, поверх языка выстроено несколько уровней инфраструктуры.

Рис. 1.1. Уровни языка Go

Помимо самого языка программирования, для разработки приложения требуются средства компиляции, тестирования, документирования и форматирования. Во многих языках соответствующие возможности предоставляются

с привлечением сторонних инструментов. Однако утилита `go`, которая используется для компиляции приложений, обеспечивает поддержку всех этих функций. Одним из самых заметных элементов этого инструментария является система управления пакетами. Язык Go и связанная с ним утилита `go` изначально поддерживают работу с зависимостями как на локальном, так и на глобальном уровне. Встроенная система управления пакетами, наряду с единым инструментарием для решения основных задач разработки, способствовала формированию целой экосистемы вокруг этого языка.

Одной из характерных особенностей Go является его простота. При создании этого языка Гриземер, Пайк и Томпсон придерживались подхода, согласно которому ни одна функция не добавлялась в язык, пока все трое не соглашались с тем, что она действительно нужна. Такая модель принятия решений, подкрепленная многолетним опытом самих разработчиков, привела к созданию простого, но мощного языка — настолько лаконичного, что его можно удержать в голове, и при этом достаточно выразительного для написания широкого спектра программ.

Такой поступательный и осторожный подход к проектированию позволяет сохранять компактность языка и внедрять новые возможности после тщательного согласования. Например, обобщения, давно существующие в C++ и Java, долго не включались в Go. Команда разработчиков на протяжении длительного времени вела открытое обсуждение достоинств и недостатков их внедрения. В конечном счете сообщество пользователей и создатели языка пришли к выводу, что поддержка обобщений действительно полезна и ее можно добавить, не нарушая удобочитаемость кода, не усложняя язык и не ломая обратную совместимость. О поддержке обобщений мы подробно поговорим в главе 3. Хорошей иллюстрацией простоты Go является синтаксис объявления переменных:

```
var i int = 2
```

Здесь создается переменная целочисленного типа (`int`) и ей сразу присваивается значение `2`. Однако, поскольку переменная инициализируется конкретным значением, этот синтаксис можно сократить:

```
var i = 2
```

Компилятор сам выведет тип переменной, исходя из присвоенного значения. В данном случае — `int`, так как переменной присвоено целое число `2`.

Но и это еще не все. Если вы не хотите указывать ключевое слово `var`, то можете использовать *краткую форму объявления переменной*:

```
i := 2
```

Этот компактный эквивалент первого варианта почти вдвое короче, легко читается и работает благодаря тому, что компилятор сам заполняет пропущенные фрагменты. Такая переменная выглядит как динамически типизированная, но на деле использует все механизмы контроля, присущие статически типизированным

компилируемым языкам. Подобная простота делает изучение основ Go особенно доступным.

Хотя ядро языка Go остается довольно простым, встроенная система управления зависимостями позволяет подключать недостающие элементы, выходящие за рамки стандартной библиотеки. Такие внешние компоненты можно импортировать как сторонние пакеты и использовать в приложениях через встроенный менеджер пакетов.

1.2. Важные особенности Go

Поскольку Go ориентирован на решение реальных практических задач, он обладает рядом примечательных особенностей, которые, если их использовать вместе, составляют строительные блоки Go-приложений.

1.2.1. Возврат нескольких значений

В Go функции и методы могут принимать и возвращать несколько значений. В большинстве языков программирования, даже разработанных после Go, функция возвращает только одно значение. При необходимости вернуть несколько значений их обычно упаковывают в кортеж, хеш-таблицу, массив или другую структуру данных. Go — это один из немногих языков, в котором поддержка множественных возвращаемых значений реализована на уровне синтаксиса. Эта возможность используется практически во всех библиотеках и приложениях, написанных на Go. Рассмотрим функцию из листинга 1.1, возвращающую две строки.

Листинг 1.1. Возврат нескольких значений

```
package main

import (
    "fmt"
)

func getStrings() (string, string) {
    return "Foo", "Bar"
}

func main() {
    n1, n2 := getStrings()
    fmt.Println(n1, n2)
    n3, _ := getStrings()
    fmt.Println(n3)
}
```

Функция без входных параметров, возвращающая два значения типа string

Возвращаются две строки: "Foo" и "Bar"

Вызывающая сторона получает оба значения и отображает их

Одно из значений можно проигнорировать, используя вместо переменной символ нижнего подчеркивания (_)

СОВЕТ Используемые в этой главе пакеты (например, `fmt`, `bufio` и `net`) входят в состав стандартной библиотеки Go. API и особенности их работы подробно описываются на сайте https://golang.org/pkg.

В этом примере оба возвращаемых значения определены в сигнатуре функции после списка аргументов. Поскольку функция `getStrings` не принимает параметров, но возвращает два значения типа `string`, при ее вызове необходимо подготовить две переменные для приема возвращаемых значений. Если одно из значений вам не нужно, то вместо имени переменной можно использовать символ нижнего подчеркивания (`_`).

Наиболее часто такой метод применяется при возврате ошибки. В Go принято значение ошибки указывать последним. Поэтому конструкции вроде `a, b, err := someFunc()` встречаются довольно часто. Рекомендуется придерживаться этого шаблона при возврате ошибки (или значения `nil`, если ошибка отсутствует). Подробнее об ошибках мы поговорим в главе 4.

Кроме того, вы можете задать имена возвращаемых значений прямо в сигнатуре функции и использовать их как обычные переменные. Благодаря этому можно явно не указывать эти значения в инструкции `return` в теле функции, сократив тем самым объем шаблонного кода. Чтобы это продемонстрировать, давайте перепишем предыдущий пример с использованием именованных возвращаемых значений (листинг 1.2).

Листинг 1.2. Именованные возвращаемые значения

```
package main

import (
    "fmt"
)

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

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

У возвращаемых значений есть имена

Именованным возвращаемым значениям присваиваются конкретные значения

Инструкция return вызывается без указания значений

Переменным присваиваются значения

Когда вызывается функция `getStrings`, именованные переменные уже доступны для присваивания значений. При вызове оператора `return` без аргументов возвращаются текущие значения этих именованных переменных. Такой способ называется *пустым возвратом* (naked return). Со стороны вызывающего кода поведение остается прежним: получение и применение результата будет происходить так же, как при обычном использовании оператора `return`. Хотя такой гибкий подход делает код более лаконичным и удобочитаемым, он становится менее распространенным по мере увеличения длины функций и методов, которое затрудняет отслеживание возвращаемых значений.

1.2.2. Современная стандартная библиотека

Многие современные приложения имеют общие черты, например работают в сети и используют шифрование. Go-разработчику не нужно тратить время на поиск сторонних пакетов, поскольку стандартная библиотека Go предоставляет средства для реализации сетевого взаимодействия, криптографии, сериализации данных, выполнения математических вычислений и многого другого. Давайте кратко рассмотрим некоторые компоненты стандартной библиотеки, чтобы вы имели представление о ее возможностях.

ПРИМЕЧАНИЕ Документация по стандартной библиотеке Go с примерами доступна по адресу https://pkg.go.dev/std.

Сетевое взаимодействие и протокол HTTP

Сетевое приложение, как правило, должно уметь подключаться к другим устройствам в сети (выступать в роли клиента) и принимать входящие соединения (выступать в роли сервера). Как показано в листинге 1.3, стандартный пакет Go `net` упрощает реализацию такого взаимодействия, независимо от того, используете ли вы HTTP или такие протоколы, как TCP (Transmission Control Protocol, протокол управления передачей), UDP (User Datagram Protocol, протокол пользовательских датаграмм) и другие распространенные схемы.

Листинг 1.3. Проверка состояния TCP-соединения

```
package main

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

func main() {
    conn, err := net.Dial("tcp", "golang.org:80")
    if err != nil {
        log.Fatal(err)
    }

    fmt.Fprintf(conn, "GET / HTTP/1.0\r\n\r\n")
    status, err := bufio.NewReader(conn).ReadString('\n')
    if err != nil {
        log.Fatal(err)
    }

    log.Println(status)
}
```

Устанавливает TCP-соединение с golang.org:80

Обрабатывает возникшую при подключении ошибку и завершает выполнение функции

Посылает через соединение отформатированную строку — в данном случае строку HTTP-запроса методом GET

Считывает ответ до первого символа новой строки (\n)

Обрабатывает ошибки, возникающие при чтении данных из соединения

Выводит на экран первую строку ответа

Прямое подключение к определенному порту по протоколу TCP реализуется с помощью пакета `net`, который предоставляет унифицированный интерфейс для установки различных типов соединений. Функция `Dial` устанавливает соединение, используя заданный тип и конечную точку. В данном случае мы устанавливаем TCP-соединение с `golang.org` на порте 80, через которое отправляется сформированный `GET`-запрос, после чего в консоль выводится первая строка ответа.

> **ПРИМЕЧАНИЕ** В листинге 1.3 мы обрабатываем ошибки по мере их возникновения и вызываем функцию `log.Fatal`, чтобы вывести информативное сообщение и завершить выполнение процесса. Язык Go предусматривает множество способов завершения работы программы, каждый из которых имеет свое назначение и преимущество. Помимо функции `log.Fatal` (имеющей аналог с поддержкой форматирования), можно использовать `os.Exit(STATUS_CODE)` или `panic()`. Эти подходы подробно рассматриваются в главе 4.

Возможность прослушивания порта реализуется столь же просто. Вместо обращения к конечной точке с помощью функции `Dial` вы можете использовать функцию `Listen` из пакета `net`, чтобы настроить приложение на прослушивание порта и обработку входящих соединений.

Для создания клиента и сервера в Go предусмотрен пакет `http`, который есть как у клиента, так и у сервера (листинг 1.4). Клиентская часть достаточно проста в использовании. Она охватывает большинство типовых сценариев, но при этом обладает гибкостью, необходимой для ее адаптации под более сложные задачи.

Листинг 1.4. HTTP-запрос методом GET

```
package main

import (
    "io"
    "log"
    "net/http"
)

func main() {
    resp, err := http.Get("http://example.com/")
    if err != nil {
        log.Fatal("could not retrieve example.com", err)
    }
    defer resp.Body.Close()

    body, err := io.ReadAll(resp.Body)
    if err != nil {
        log.Fatal("could not read body", err)
    }

    log.Println(string(body))
}
```

Выполняет HTTP-запрос методом GET с использованием пакета net/http

Обработчики ошибок, дополненные поясняющим контекстом

Откладывает закрытие объекта чтения (в данном случае ответа сервера)

Выводит полученный ответ в виде строки в стандартный поток вывода (или в другой логгер)

Обработчики ошибок, дополненные поясняющим контекстом

Данный пример демонстрирует формальный способ извлечения и отображения тела простого HTTP-запроса методом `GET`. HTTP-клиент в Go обладает гораздо более широкими возможностями: он может работать с прокси-серверами, поддерживать протокол TLS, устанавливать заголовки и cookie, создавать клиентские объекты и даже настраивать транспортный уровень передачи данных. Расширенные TCP-соединения, такие как долговременные сеансы WebSocket и события, отправляемые сервером, поддерживаются как стандартной библиотекой, так и сторонними решениями. Эти типы соединений будут рассмотрены в главе 8.

Обратите внимание на ключевое слово `defer` в листинге 1.4. Любой оператор с этим словом выполняется в конце соответствующего блока кода — в данном случае в конце функции `main()`. Обычно `defer` используется для освобождения ресурсов, например, при закрытии объектов типа `Reader`, хотя в этом примере его отсутствие не привело бы к негативным последствиям. Мы подробно обсудим этот механизм в главе 4. А пока просто имейте в виду, что оператор `defer` помещается в стек вызовов, и чем раньше вы его определите, тем меньше вероятность, что инициированная впоследствии паника (`panic`) помешает его выполнению.

Создание HTTP-сервера на языке Go — это весьма распространенная задача. Средства, предоставляемые стандартной библиотекой Go, являются достаточно мощными для создания масштабируемых приложений, при этом они просты в освоении и обладают гибкостью, позволяющей справляться со сложными случаями. Глава 8 посвящена запуску HTTP-сервера: в ней рассматриваются темы маршрутизации, работы с cookie-файлами и аутентификации.

Код HTML

При работе с веб-серверами часто возникает необходимость генерации HTML-кода. Для этого можно использовать пакеты `html` и `html/template`. Пакет `html` предназначен для экранирования и обратного преобразования HTML, а пакет `html/template` позволяет создавать многократно используемые HTML-шаблоны с возможностью подстановки переменных и управления логикой с помощью Go-кода. Модель безопасности при взаимодействии с данными описана в документации, а вспомогательные функции позволяют работать с HTML, JavaScript и не только. Система шаблонов является расширяемой, что делает ее удобной как для новичков, так и для специалистов, занимающихся построением более сложных решений.

Криптография

Криптография широко используется в современных приложениях как для работы с простым хешем, так и для шифрования и расшифровки конфиденциальной информации. В Go предусмотрены стандартные криптографические средства, включая хеш-функции MD5 и SHA (Secure Hash Algorithm — безопасный

алгоритм хеширования), протокол TLS, алгоритмы шифрования DES (Data Encryption Standard — стандарт шифрования данных), TDEA (или Triple DES — тройной DES), AES (Advanced Encryption Standard — расширенный стандарт шифрования), а также HMAC (Keyed-Hash Message Authentication Code — код аутентификации сообщений на основе хеш-функции) и многие другие. Кроме того, в стандартной библиотеке доступен криптографически безопасный генератор случайных чисел (`crypto/rand`).

Кодирование данных

При обмене данными между системами часто возникают вопросы о формате и кодировке. Например, получены ли данные в Base64? Нужно ли преобразовать данные JSON (JavaScript Object Notation) или XML (Extensible Markup Language) в локальный объект? Подобные задачи типичны для современной сетевой среды.

Язык Go изначально создавался с учетом задач кодирования данных. Внутри самого Go все данные обрабатываются в формате UTF-8, что неудивительно, ведь его авторы стояли и за разработкой самого стандарта UTF-8. Однако не все данные, которые передаются между системами, закодированы в этом формате. Иногда приходится иметь дело с другими представлениями, которые несут дополнительную смысловую нагрузку. Для преобразования и обработки таких данных в Go предусмотрены специальные пакеты и интерфейсы. Они позволяют, например, превратить строку JSON в полноценный объект или структуру, а также переключаться между кодировками или подключать внешние пакеты для поддержки других форматов.

1.2.3. Реализация конкурентности с помощью горутин и каналов

Сегодня многоядерные процессоры используются в самых разных компьютерах — от серверов и смартфонов до встраиваемых систем и микроконтроллеров. До недавнего времени большинство языков программирования создавалось с расчетом на однопроцессорные системы. В отличие от них Go изначально предназначался для многопроцессорных архитектур.

Одной из ключевых особенностей Go является *горутина* — функция, которая может выполняться в конкурентном режиме параллельно с основной программой или другими горутинами. За управление горутинами отвечает среда выполнения Go: она распределяет их по соответствующим потокам операционной системы (ОС) и освобождает ресурсы, когда они больше не нужны. Если системе доступно несколько ядер, горутины могут выполняться одновременно, поскольку разные потоки задействуют разные процессорные ядра. Однако даже в рамках одного потока Go самостоятельно управляет переключением контекста и организацией конкурентного выполнения операций.

С точки зрения разработчика, создать горутину так же просто, как написать функцию. На рис. 1.2 показан принцип работы горутин. Если вы привыкли к языкам, использующим событийно-ориентированный или асинхронный подход (async/await), данный способ может показаться непривычным, но он открывает широкие возможности для отправки и получения данных при выполнении операций в конкурентном режиме.

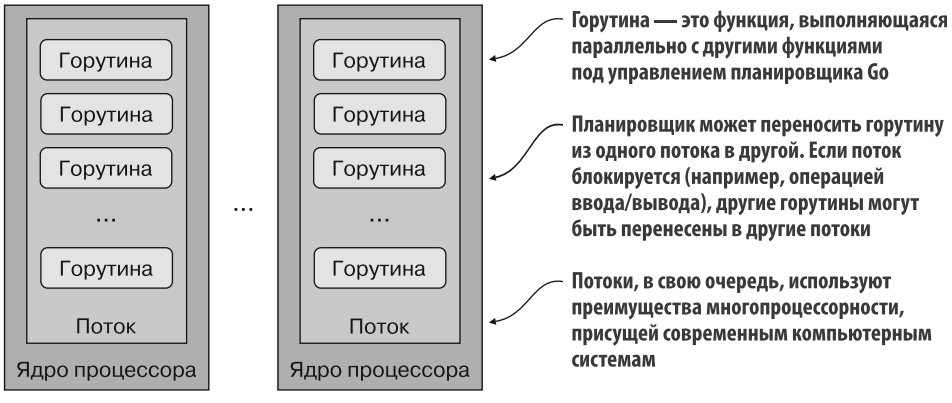

Рис. 1.2. Горутины, распределенные по потокам, запущенным на доступных процессорных ядрах

Чтобы проиллюстрировать работу горутин, рассмотрим пример из листинга 1.5, в котором горутина считает от 0 до 4, в то время как основная программа выводит на экран строку `Hello World`.

Листинг 1.5. Вывод значений в конкурентном режиме

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

Данный вывод представляет собой смесь двух функций, которые выводят результат в конкурентном режиме. Код, реализующий такое поведение, во многом похож на обычное процедурное программирование, но с небольшим нюансом, показанным в листинге 1.6.

Функция `count` — это обычная функция, которая считает от 0 до 4. Чтобы запустить ее параллельно, а не последовательно, используется ключевое слово `go`, из-за чего функция `main` продолжает выполняться незамедлительно. В результате `count` и `main` работают одновременно. Благодаря вызовам `time.Sleep()` порядок их выполнения становится предсказуемым и воспроизводимым.

Листинг 1.6. Отображение значений в конкурентном режиме

```
package main

import (
    "fmt"
    "time"
)

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

func main() {
    go count()
    time.Sleep(time.Millisecond * 20)
    fmt.Println("Hello World")
    time.Sleep(time.Millisecond * 10)
}
```

Функция, выполняемая в качестве горутины

Запуск горутины

Каналы позволяют горутинам взаимодействовать друг с другом или с другими процессами. По умолчанию они блокируют выполнение, способствуя синхронизации работы горутин. На рис. 1.3 показано, как это работает.

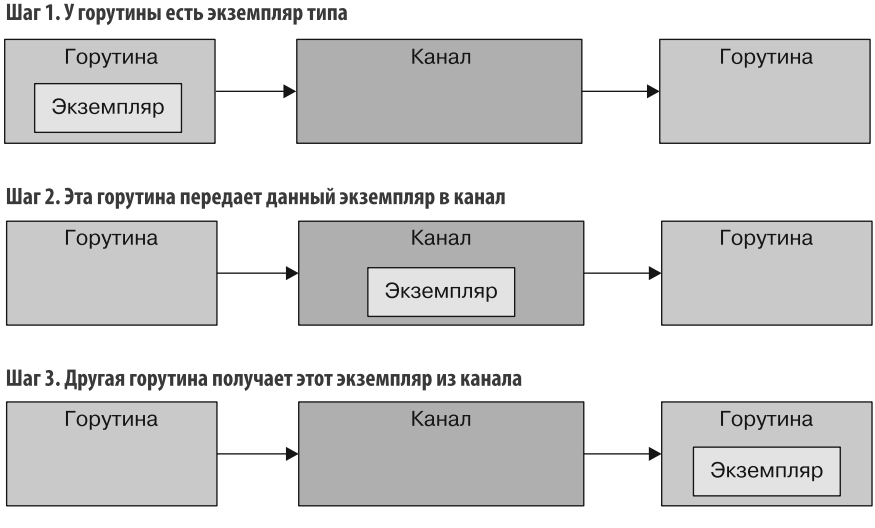

Рис. 1.3. Передача переменных от одной горутины к другой через канал

В этом примере переменная передается от одной горутины к другой через канал — и это работает даже тогда, когда горутины выполняются параллельно на разных

ядрах. Хотя на рис. 1.3 показана передача данных в одном направлении, каналы в Go могут быть как однонаправленными, так и двунаправленными. Направление передачи задается с помощью синтаксиса `<-` или `->`. В листинге 1.7 приведена наглядная схема этого процесса.

Листинг 1.7. Использование канала

```
package main

import (
    "fmt"
    "time"
)

func printCount(c chan int) {
    num := 0
    for num >= 0 {
        num = <-c
        fmt.Print(num, " ")
    }
}

func main() {
    c := make(chan int)
    a := []int{8, 6, 7, 5, 3, 0, 9, -1}
    go printCount(c)
    for _, v := range a {
        c <- v
    }

    time.Sleep(time.Millisecond * 1)
    fmt.Println("End of main")
}
```

Канал типа int передается в функцию

Ожидает поступления значения в канал

Создает канал

Запускает горутину

Передает целые числа в канал

Функция main приостанавливается перед завершением

В начале функции `main` создается небуферизованный канал типа `int` с именем `c`, предназначенный для передачи данных и сигналов между горутинами. При запуске функции `printCount` в качестве горутины этот канал передается ей в виде аргумента. Поскольку канал используется как параметр, его тип должен быть явно указан в сигнатуре `printCount`. В цикле `for` внутри функции `printCount` переменная `num` ожидает поступления целочисленного значения из канала `c`. Тем временем в функции `main` перебирается срез целых чисел, каждое из которых по одному передается в канал `c`. Как только значение поступает в канал в функции `main`, оно сразу же принимается переменной `num` внутри `printCount`. Функция `printCount` продолжает выполнение до тех пор, пока цикл `for` снова не дойдет до инструкции приема данных из канала, — в этот момент выполнение приостанавливается в ожидании нового значения. По завершении перебора целых чисел функция `main` продолжает выполняться. После завершения `main` программа полностью прекращает выполнение, поэтому перед выходом делается небольшая пауза в одну миллисекунду, чтобы функция `printCount` успела

завершиться до окончания работы `main`. Результат выполнения этой программы показан в листинге 1.8.

Листинг 1.8. Вывод значений с использованием канала

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

Совместное использование каналов и горутин обеспечивает функциональность, аналогичную той, которая свойственна легковесным потокам или внутренним сервисам, взаимодействующим через API-интерфейс с заданными типами данных. Для объединения этих компонентов в цепочки можно использовать различные техники. Если воспринимать каналы как внешние слушатели, способные выполняться параллельно с остальной частью программы, становится гораздо проще представить, как можно поддерживать работу конкурентных и/или асинхронных процессов наряду с линейным кодом.

В книге мы будем многократно возвращаться к горутинам и каналам — двум мощным средствам реализации конкурентности в Go. Вы увидите, как с их помощью создавать серверы, передавать сообщения и откладывать выполнение задач. Кроме того, мы рассмотрим паттерны проектирования, задействующие горутины и каналы, вы также научитесь применять `WaitGroup`, чтобы управлять выполнением кода в конкурентном режиме.

1.2.4. Инструментарий Go

Разработка масштабируемых и сопровождаемых приложений сегодня требует целого набора вспомогательных средств, помимо компилятора. С самого начала в Go это было предусмотрено, поэтому Go — это не только язык и компилятор. Исполняемый файл `go` представляет собой целый инструментарий, который не только позволяет компилировать код на Go в исполняемый файл, но и включает в себя средства управления пакетами, тестирования, генерации документации и не только. Давайте рассмотрим некоторые из этих компонентов.

Управление пакетами

Многие современные языки программирования предусматривают средства для управления пакетами/зависимостями, однако до недавнего времени они практически никогда не включались непосредственно в языковой инструментарий. В Go это реализовано с помощью `go mod` (Go-модулей), и у такого решения есть две причины. Первая заключается в повышении продуктивности разработчика, а вторая — в ускорении компиляции. Механизм работы с пакетами был спроектирован с учетом особенностей компилятора, и именно это во многом объясняет высокую скорость компиляции. Управление зависимостями реализовано чрезвычайно эффективно и позволяет избежать сложной обработки заголовков, свойственной языкам вроде C и C++. Для достижения такой скорости пришлось пойти на определенные компромиссы, но результат вполне себя оправдывает.