Мэтт Батчер
Мэтт Фарина

Go на практике

MANNING

ДМК ПРЕСС
ИЗДАТЕЛЬСТВО

УДК 004.438Go
ББК 32.973.26-018.1
Б28

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

ISBN 978-5-97060-477-9

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

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

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

УДК 004.438Go
ББК 32.973.26-018.1

ISBN 978-1-63343-007-5 (анг.)

ISBN 978-5-97060-477-9 (рус.)

Содержание

Глава 1

Введение в язык Go

В этой главе рассматриваются следующие темы:

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

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

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

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

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

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

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

1.1. Что представляет собой язык Go?

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

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

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

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

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

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

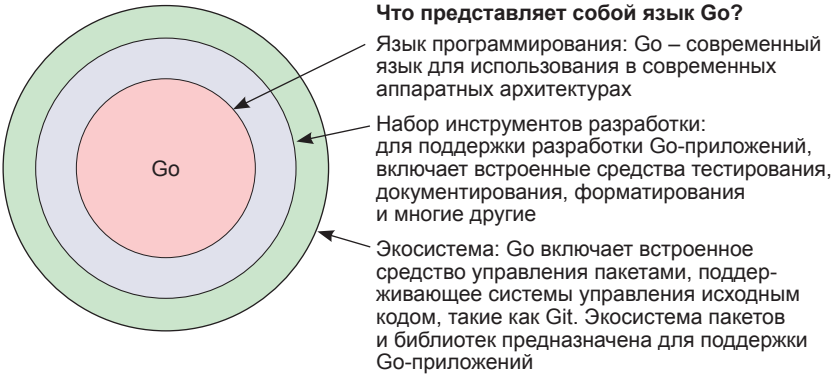

Рис. 1.1 ❖ *Слои языка Go*

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

Философию языка можно проиллюстрировать примером синтаксиса объявления переменной:

```
var i int = 2
```

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

```
var i = 2
```

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

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

```
i := 2
```

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

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

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

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

1.2. Примечательные особенности языка Go

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

1.2.1. Возврат нескольких значений

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

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

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

```
package main

import (
    "fmt"
)

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

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

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

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

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

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

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

```
package main

import (
    "fmt"
)

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

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

❶ Именованные возвращаемые значения

❷ Присваивание значений именованным возвращаемым переменным

❸ Оператор return вызывается без значений

❹ Значения присваиваются двум переменным

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

1.2.2. Современная стандартная библиотека

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

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

Сетевые операции и HTTP

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

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

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

```
package main

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

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

❶ Соединение по TCP

❷ Отправка строки через соединение

❸ Вывод первой строки ответа

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

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

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

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

```
package main

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

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

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

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

HTML

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

Криптография

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

Кодирование данных

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

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

1.2.3. Параллельная обработка с помощью сопрограмм и каналов

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

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

Язык Go изначально ориентировался на параллельную и конкурентную обработку.

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

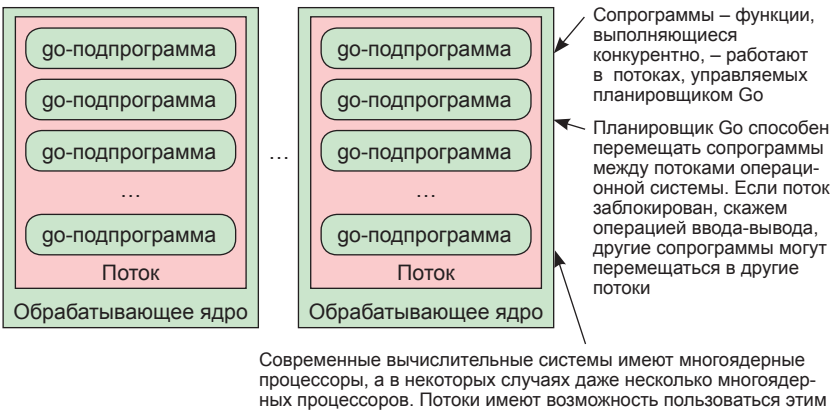

Рис. 1.2 ❖ *Go-подпрограммы выполняются в потоках, распространяемых на все доступные ядра*

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

Листинг 1.5 ❖ Результат конкурентного вывода

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

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

Листинг 1.6 ❖ Конкурентный вывод

```
package main

import (
    "fmt"
    "time"
)

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

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

❶ Функция, выполняемая как go-подпрограмма

← ❷ Вызов go-подпрограммы

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

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

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

В следующем листинге приводится пример использования канала.

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

Листинг 1.7 ❖ Использование каналов: `channel.go`

```
package main import (

    "fmt"
    "time"
)

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

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

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

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

Листинг 1.8 ❖ Вывод с использованием канала

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

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

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

1.2.4. Go – больше, чем язык

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

Управление пакетами

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

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

Листинг 1.9 ❖ Простой импорт пакета

```
package main

import "fmt" ← Импорт пакета fmt

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

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

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

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

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

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