Effective Go (На русском языке)

Это переведенный на русский язык Effective Go.

Содержание

Введение

Go — это новый язык. Хотя он заимствует идеи из существующих языков, у него есть необычные свойства, которые делают эффективные программы на Go отличными по характеру от программ, написанных на его "родственных" языках. Прямой перевод программы на C++ или Java в Go вряд ли приведет к удовлетворительному результату — программы на Java пишутся для Java, а не для Go. С другой стороны, размышление о проблеме с точки зрения Go может привести к успешной, но совершенно иной программе. **Иными словами, чтобы писать на Go эффективно, важно понимать его особенности и идиомы**. **Также важно знать установленные соглашения по программированию на Go, такие как правила именования, форматирования, структуры программы и так далее, чтобы программы, которые вы пишете, были понятны другим программистам Go.**
Этот документ содержит советы по написанию ясного, идиоматичного кода на Go. Он дополняет спецификацию языка, "Tour of Go" и "How to Write Go Code", которые вы должны прочитать в первую очередь.
Примечание, добавленное в январе 2022 года: Этот документ был написан для релиза Go в 2009 году и с тех пор не обновлялся значительно. Хотя он остается хорошим руководством для понимания того, как использовать сам язык, благодаря его стабильности, в нем мало говорится о библиотеках и ничего о значительных изменениях в экосистеме Go, таких как система сборки, тестирование, модули и полиморфизм. Обновлять его не планируется, так как произошло много изменений, и обширный набор документов, блогов и книг хорошо описывает современные способы использования Go. **"Effective Go" по-прежнему полезен, но читателю следует понимать, что это далеко не полное руководство.** См. задачу 28782 для контекста.

Примеры

Исходные коды пакетов Go предназначены не только для работы в качестве базовой библиотеки, но и как примеры того, как использовать язык. Более того, многие пакеты содержат рабочие, автономные исполняемые примеры, которые можно запустить прямо с веб-сайта `go.dev`, как этот (при необходимости, нажмите на слово "Example", чтобы открыть его). Если у вас есть вопрос о том, как подойти к решению проблемы или как может быть реализовано что-то, документация, код и примеры в библиотеке могут предоставить ответы, идеи и бэкграунд.

Форматирование

Вопросы форматирования - самые спорные, но **наименее значимые**. Люди могут приспособиться к разным стилям форматирования, но лучше, если им не придется этого делать, и меньше времени будет уделено теме, если все будут

придерживаться одного стиля. Проблема в том, как подойти к этой утопии без длинного предписывающего руководства по стилю.
В Go мы используем необычный подход и позволяем машине позаботиться о большинстве вопросов форматирования. Программа `gofmt` (также доступная как `go fmt`, которая работает на уровне пакетов, а не исходных файлов) читает программу на Go и выдает исходный текст в стандартном стиле с отступами и вертикальным выравниванием, сохраняя и при необходимости переформатируя комментарии. Если вы хотите узнать, как поступить в новой ситуации с верстkой, запустите `gofmt`; если ответ покажется вам неправильным, перестройте свою программу (или напишите об ошибке в `gofmt`), а не обходите ее.
Например, не нужно тратить время на выстраивание комментариев к полям структуры. `Gofmt` сделает это за вас. Учитывая эту структуру

```
type T struct {
    name string // имя объекта
    value int // его значение
}
```

`gofmt` выстроит столбцы в ряд:

```
type T struct {
    name    string  // имя объекта
    value   int     // его значение
}
```

Весь код Go в стандартных пакетах был отформатирован с помощью `gofmt`.
Некоторые детали форматирования остались. **Очень коротко:**

Отступы

Мы используем табуляции для отступов, и `gofmt` выдает их по умолчанию. Используйте пробелы только в случае необходимости.

Длина строки

В Go нет ограничений на длину строки. Не бойтесь переполнить перфокарту. Если строка кажется слишком длинной, оберните ее и сделайте отступ с помощью дополнительной табуляции.
Пример:

```
package main

import "fmt"

func main() {
    // Длинная строка без переноса
    fmt.Println("This is an example of a very long line of code in Go that exceeds the usual length limit and should b
}
```

```
package main

import "fmt"

func main() {
    // Длинная строка с множественными переносами
    fmt.Println(
        "This is an example of a very long line of code in Go that exceeds " +
        "the usual length limit and should be wrapped "+
        "according to best practices for readability " +
        "and maintainability.",
    )
}
```

Круглые скобки

В Go требуется меньше круглых скобок, чем в C и Java: управляющие структуры (if, for, switch) не содержат круглых скобок в своем синтаксисе. Кроме того, иерархия старшинства операторов короче и понятнее, так что

```
x<<8 + y<<16
```

означает то, что подразумевает интервал, в отличие от других языков.

Комментирование

В Go предусмотрены блочные комментарии `/* */` в стиле `C` и строчные комментарии `//` в стиле `C++`. Строчные комментарии являются нормой; блочные комментарии появляются в основном как комментарии к пакетам, но они полезны внутри выражения или для отключения больших участков кода.
Комментарии, появляющиеся перед декларациями верхнего уровня, без промежуточных строк, считаются документированием самой декларации. Эти "doc-комментарии" являются основной документацией для данного пакета или команды Go. Подробнее о комментариях см. в разделе Комментарии к документам Go.

Имена

Имена важны в Go, как и в любом другом языке. Они даже имеют семантический эффект: **видимость имени вне пакета определяется тем, является ли его первый символ верхним регистром**. Поэтому стоит уделить немного времени обсуждению соглашений об именовании в программах на Go.

Имена пакетов

Когда пакет импортируется, его имя становится указателем доступа к его содержимому. После импорта пакета:

```
import "strings"
```

Мы можем использовать его содержимое:

```
package main

import "strings"

func main() {
    str := "This is the new string!"
    words := strings.SplitN(str, " ", 3)
    // words = ["This","is","the new string!"]
}
bytes.Clone()
```

Мы обращаемся к **экспортируемым** функциям/переменным/структурам через имя пакета. Это удобно, так как все, кто будут использовать этот пакет, будут использовать одинаковое обращение к его содержимому, что подразумевает, что имя пакета должно быть хорошим: коротким, лаконичным, вызывающим. По соглашению, пакетам присваиваются имена в нижнем регистре, состоящие из одного слова; подчеркивания или смешанные прописные буквы не нужны. Отдавайте предпочтение краткости, поскольку все, кто будет использовать ваш пакет, будут набирать это имя. И не беспокойтесь о коллизиях априори. Имя пакета - это только имя по умолчанию для импорта; оно не обязательно должно быть уникальным во всем исходном коде, и в редких случаях коллизии импортирующий пакет может выбрать другое имя для локального использования.
Например:

```
import (
    "strings"
    "github.com/exampleuser/examplelibrary/strings"
    // Будет конфликт, так как оба пакета имеют одинаковые имена указателей доступа.
    // strings и strings
)
```

Сделаем так:

```
import (
    "strings"
```

```
    strings2 "github.com/exampleuser/examplelibrary/strings"
    // Конфликта не будет, как ты локально дали другое название пакету.
)
```

В любом случае, путаница возникает редко, поскольку имя файла в импорте определяет, какой именно пакет используется. Другое соглашение заключается в том, что имя пакета является базовым именем его исходного каталога; пакет в `src/encoding/base64` импортируется как `"encoding/base64"`, но имеет имя `base64`, а не `encoding_base64` и не `encodingBase64`.
Импортер пакета будет использовать это имя для ссылки на его содержимое, поэтому экспортируемые имена в пакете могут использовать этот факт, чтобы избежать повторений. (Не используйте нотацию `import .`, которая может упростить тесты, которые должны выполняться вне проверяемого пакета, но в остальном ее следует избегать). Например, `buffered reader type` в пакете `bufio` называется `Reader`, а не `BufReader`, потому что пользователи видят его как `bufio.Reader`, **что является ясным и кратким именем**. Более того, поскольку к импортируемым сущностям всегда обращаются по имени их пакета, `bufio.Reader` не конфликтует с `io.Reader`. Аналогично, функция создания новых экземпляров `ring.Ring` - а это определение конструктора в Go - обычно называется `NewRing`, но поскольку `Ring` - единственный тип, экспортируемый пакетом, и поскольку пакет называется `ring`, она называется просто `New`, что клиенты пакета видят как `ring.New`.
Используйте структуру пакета, чтобы помочь вам выбрать хорошие имена.
Например, у нас есть пакет `internal/server`, где лежит структура сервера и есть функция, которая создает объект этого сервера. так как мы будем обращаться к содержимому пакета через `server`, во всех экспортируемых именах не стоит ипсользовать `server`.

```
package server

type server struct {
    // ...
}

func New(...) *server {
    return &server{
        ...
    }
}

// ...


package client

import ".../internal/server"

func main() {
    s := server.New(...)
    // ...
}
```

Что здесь у нас?
У нас есть пакет со структурой сервера и функция, которая инициализирует объект сервера и возвращает указатель на него. **Структура НЕэкспортируемая, а функция экспортируемая**. В другом месте мы импортируем пакет сервера и хотим создать сервер + получить на него указатель. Мы вызываем `server.New()`, **что уже явно говорит нам, что создается именно сервер**. Не надо делать никаких `NewServer()` и других.
Конечно, если будет еще другая структура в этом пакете и у нее будет функция `New()`, то тогда придется выбрать другое название. **А вообще надо просто создать отдельный пакет для этой структуры и всего, что с ней связано.**

Геттеры (Getters)

Go не предоставляет автоматической поддержки геттеров и сеттеров. Нет ничего плохого в том, чтобы предоставлять геттеры и сеттеры самостоятельно, и часто это бывает уместно, **но нет ни идиоматизма, ни необходимости добавлять Get в имя геттера**. Если у вас есть поле с именем `owner` (**нижний регистр, неэкспортированное**), **метод геттера должен называться** `Owner` (**верхний регистр, экспортированное**), **а не** `GetOwner`. Использование имен в верхнем регистре при экспорте дает крючок для различения поля и метода. Функция сеттера, если она необходима, скорее всего, будет называться `SetOwner`. Оба имени хорошо читаются на практике:

```
owner := obj.Owner()
if owner != user {
```

```
    obj.SetOwner(user)
}
```

Имена интерфейсов

По соглашению, интерфейсы с одним методом называются по имени метода с добавлением суффикса `-er` или аналогичного изменения для образования существительного-агента: `Reader`, `Writer`, `Formatter`, `CloseNotifier` и т.д. Существует множество таких имен, и важно уважать их и функции, которые они описывают. `Read`, `Write`, `Close`, `Flush`, `String` и так далее **имеют канонические сигнатуры и значения**. Чтобы избежать путаницы, не давайте вашему методу одно из этих имен, если у него нет такой же сигнатуры и значения. Напротив, если ваш тип реализует метод с тем же значением, что и метод в известном типе, дайте ему такое же имя и сигнатуру; назовите ваш метод преобразования строки `String`, а не `ToString`.

Что за канонические сигнатуры и значения?

В Go существуют **стандартные (канонические) имена и сигнатуры методов для интерфейсов**. Например, методы `Read`, `Write`, `Close`, и `String` имеют общепринятые значения и способы использования в языке.

Вот что это означает:

1. *Стандартные имена и сигнатуры*. Методы с определенными именами имеют ожидаемые сигнатуры (определение параметров и возвращаемых значений). Например, метод `Read` в стандартной библиотеке имеет сигнатуру `Read(p []byte) (n int, err error)`, **что означает, что он читает данные в срез байтов и возвращает количество прочитанных байтов и ошибку**.
2. *Согласованность и предсказуемость*. Когда вы используете стандартные имена и сигнатуры, это делает ваш код предсказуемым и совместимым с другими пакетами и библиотеками, которые следуют тем же стандартам. Это помогает другим разработчикам быстрее понять ваш код и использовать его без необходимости разбираться в нестандартных соглашениях.

Не используйте стандартные имена без соответствия сигнатурам: не называйте ваш метод, например, `Read`, **если он не реализует ту же функциональность или сигнатуру, что и стандартный метод** `Read`. Если вы используете стандартное имя, оно должно точно соответствовать общепринятым стандартам, чтобы избежать путаницы. Например, метод `Read` должен принимать **срез байтов** и **возвращать количество прочитанных байтов и ошибку**, как это предусмотрено стандартом.

Соблюдайте соглашения для методов с известными именами: если ваш тип реализует метод, аналогичный методу в известных типах (например, `String` для метода, возвращающего строку), используйте те же имена и сигнатуры. Это делает ваш код согласованным с другими библиотеками и типами, что упрощает его использование и понимание. Например, метод, **который возвращает строковое представление вашего типа, должен называться** `String`**, а не** `ToString`**.**

MixedCaps

Наконец, в Go принято использовать **MixedCaps** или **mixedCaps**, **а не подчеркивание для записи многословных имен**.

Точки с запятой

Как и в `C`, в формальной грамматике Go используется точка с запятой для завершения операторов, но в отличие от C, эти точки с запятой не появляются в исходном коде. Вместо этого лексический анализатор **(Лексер)** использует простое правило для автоматического добавления точек с запятой во время сканирования, поэтому в тексте программы их почти нет.

Правило такое: *если последний токен перед новой строкой является идентификатором (включая такие слова, как int и float64), простым литералом, таким как число или строковая константа, или одним из следующих токенов:*

```
break, continue, fallthrough, return, ++, --, ), },
```

лексический анализатор всегда вставляет точку с запятой после этого токена. Это можно резюмировать так: **"если новая строка идет после токена, который может завершать оператор, вставьте точку с запятой"**.

Точку с запятой также можно опустить непосредственно перед закрывающей скобкой, поэтому оператор, такой как

```
go func() { for { dst <- <-src } }()
```

не требует точек с запятой. Идиоматические программы на Go содержат точки с запятой только в таких местах, как условие в цикле for, чтобы разделить инициализацию, условие и продолжение. Они также необходимы для разделения нескольких операторов в одной строке, если вы пишете код таким образом.

Одним из последствий правил вставки точек с запятой является то, **что вы не можете ставить открывающую скобку управляющей структуры (if, for, switch или select) на следующей строке**. Если вы это сделаете, точка с запятой будет вставлена перед скобкой, что может вызвать нежелательные эффекты. Пишите их так:

```
if i < f() {
    g()
}
```

а не так:

```
if i < f()  // неправильно!
{           // неправильно!
    g()
}
```

Почему именно так?
Потому что `if i < f()` не заканчивается во втором случае открывающей скобкой, а **значит условие окончено**, а значит лексер поставит точку с запятой. Когда лексер видит конструкцию вида `if i < f() {`, он понимает, что дальше идет тело условия.
Лексический анализатор сам расставляет точки с запятой перед компиляцией.

Управляющие конструкции

Управляющие конструкции в Go похожи на те, что используются в `C`, но имеют важные отличия. В Go нет циклов `do` или `while`, есть только немного более универсальный `for`; конструкция `switch` более гибкая; `if` и `switch` могут принимать необязательный оператор инициализации, подобно `for`; операторы `break` и `continue` могут принимать необязательную метку, чтобы указать, какую конструкцию завершить или продолжить; также есть новые управляющие конструкции, включая `type switch` и мультиплексор многоканальной связи `select`. Синтаксис также немного отличается: **в Go нет круглых скобок**, а **тело конструкций всегда должно быть заключено в фигурные скобки**.

If

В Go простая конструкция `if` выглядит так:

```
if x > 0 {
    return y
}
```

Обязательное использование фигурных скобок поощряет написание простых операторов `if` на нескольких строках. **Это хороший стиль, особенно когда тело содержит управляющий оператор, такой как `return` или `break`.**
Поскольку `if` и `switch` могут принимать оператор инициализации, часто можно увидеть, как его используют для создания локальной переменной.

```
if err := file.Chmod(0664); err != nil {
    log.Print(err)
    return err
}
```

В библиотеках Go вы заметите, что если оператор `if` не переходит к следующему оператору — то есть, если тело заканчивается `break`, `continue`, `goto` или `return` **— то лишний `else` опускается.**

```
f, err := os.Open(name)
if err != nil {
    return err
}
codeUsing(f)
```

Это пример распространенной ситуации, когда код должен защититься от ряда ошибок. Такой код легко читается, если успешное выполнение продолжается вниз по странице, устраняя случаи ошибок по мере их возникновения. Поскольку случаи ошибок обычно заканчиваются операторами `return`, итоговый код не нуждается в операторах `else`.

```
f, err := os.Open(name)
if err != nil {
    return err
}
```

```
d, err := f.Stat()
if err != nil {
    f.Close()
    return err
}
codeUsing(f, d)
```

Пример `type switch`:

```
func process(value interface{}) {
    switch v := value.(type) {
    case int:
        fmt.Printf("Integer: %d\n", v)
    case string:
        fmt.Printf("String: %s\n", v)
    case bool:
        fmt.Printf("Boolean: %t\n", v)
    default:
        fmt.Printf("Unknown type\n")
    }
}
```

Повторное объявление и присваивание

Отступление: последний пример в предыдущем разделе демонстрирует одну деталь, касающуюся того, как работает краткая форма объявления `:=`. Объявление, которое вызывает `os.Open`, выглядит так:

```
f, err := os.Open(name)
```

Эта инструкция объявляет две переменные: `f` и `err`. Несколькими строками ниже вызов `f.Stat` выглядит так:

```
d, err := f.Stat()
```

что, на первый взгляд, кажется объявлением переменных `d` и `err`. Обратите внимание, однако, что `err` появляется в обоих выражениях. Это дублирование допустимо: `err` объявляется первым оператором, но во втором только повторно присваивается. Это означает, что вызов `f.Stat` использует существующую переменную `err`, объявленную выше, и просто присваивает ей новое значение.
В декларации `:=` переменная `v` может появляться, даже если она уже была объявлена, при условии, что:

1. Это объявление находится в той же области видимости, что и существующее объявление `v` (если `v` уже объявлена во внешней области видимости, то объявление создаст новую переменную).
2. Соответствующее значение в инициализации может быть присвоено `v`.
3. По крайней мере одна другая переменная создается этим объявлением.

Это необычное свойство является чистым прагматизмом, что упрощает использование единственного значения `err`, например, в длинной цепочке `if-else`. Вы часто увидите его использование.
Примечание: стоит отметить, что в Go область видимости параметров функции и возвращаемых значений совпадает с областью видимости тела функции, даже если они появляются лексически вне фигурных скобок, заключающих тело.

For

Цикл `for` в Go похож на цикл в языке `C`, но не идентичен ему. В Go объединены конструкции `for` и `while`, и отсутствует конструкция `do-while`. **Существует три формы цикла `for`, и только в одной из них используются точки с запятой**.

```
// Как for в C
for init; condition; post { }

// Как while в C
for condition { }

// Как for(;;) в C
for { }
```

Краткие объявления позволяют легко объявлять переменные-счетчики прямо в цикле.

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

```
    sum += i
}
```

Если вы итерируетесь по массиву, срезу, строке, карте или читаете из канала, оператор `range` может управлять циклом.

```
for key, value := range oldMap {
    newMap[key] = value
}
```

Если вам нужен только первый элемент в `range` (**ключ** или **индекс**), **опустите второй**:

```
for key := range m {
    if key.expired() {
        delete(m, key)
    }
}
```

Если вам нужен **только второй элемент в `range` (значение), используйте идентификатор-заглушку (подчеркивание) для отбрасывания первого**:

```
sum := 0
for _, value := range array {
    sum += value
}
```

Идентификатор-заглушка имеет много применений, которые будут описаны в следующих разделах.
Для строк `range` выполняет дополнительную работу, **разбивая отдельные символы Unicode путем парсинга UTF-8**. Ошибочные кодировки потребляют один байт и выводят символ замены `U+FFFD`. (Термин `rune` (и связанный с ним встроенный тип) в Go обозначает **один кодовый пункт Unicode**. Подробнее см. в спецификации языка.)
Цикл

```
for pos, char := range "日本\x80語" { // \x80 - недопустимая кодировка UTF-8
    fmt.Printf("character %#U starts at byte position %d\n", char, pos)
}
```

выводит:

```
character U+65E5 '日' starts at byte position 0
character U+672C '本' starts at byte position 3
character U+FFFD '' starts at byte position 6
character U+8A9E '語' starts at byte position 7
```

Наконец, **в Go нет оператора запятой, а ++ и -- — это операторы, а не выражения**. Поэтому, если вы хотите управлять несколькими переменными в `for`, **вы должны использовать параллельное присваивание (хотя это исключает использование ++ и --).**

```
// Перевернуть массив a
for i, j := 0, len(a)-1; i < j; i, j = i+1, j-1 {
    a[i], a[j] = a[j], a[i]
}
```

Switch

Конструкция `switch` в Go более универсальна, чем в `C`. Выражения в `switch` **не обязательно должны быть константами или даже целыми числами**, случаи (`case`) проверяются **сверху вниз до тех пор, пока не будет найдено совпадение**, **и если в `switch` не указано выражение, он переключается на `true`**. Поэтому можно (и это является идиоматикой Go) написать цепочку `if-else-if-else` в виде `switch`.

```
func unhex(c byte) byte {
    switch {
    case '0' <= c && c <= '9':
        return c - '0'
    case 'a' <= c && c <= 'f':
        return c - 'a' + 10
    case 'A' <= c && c <= 'F':
        return c - 'A' + 10
    }
```

```
    return 0
}
```

Автоматического перехода к следующему случаю (`fall through`) нет, но можно указать несколько значений для одного случая через запятую.

```
func shouldEscape(c byte) bool {
    switch c {
    case ' ', '?', '&', '=', '#', '+', '%':
        return true
    }
    return false
}
```

Хотя `break` в Go используется не так часто, как в некоторых других языках, похожих на `C` , его можно применять для досрочного завершения `switch` . Иногда, однако, требуется выйти из окружающего цикла, а не из `switch` , **и в Go это можно сделать, поместив метку на цикл и используя `break` с указанием этой метки.** Этот пример демонстрирует оба использования.

```
Loop:
    for n := 0; n < len(src); n += size {
        switch {
        case src[n] < sizeOne:
            if validateOnly {
                break
            }
            size = 1
            update(src[n])

        case src[n] < sizeTwo:
            if n+1 >= len(src) {
                err = errShortInput
                break Loop
            }
            if validateOnly {
                break
            }
            size = 2
            update(src[n] + src[n+1]<<shift)
        }
    }
```

Конечно, оператор `continue` также принимает необязательную метку, **но он применяется только к циклам**.
Для завершения этого раздела вот функция сравнения для срезов байтов, которая использует два оператора `switch` :

```
// Compare возвращает целое число, сравнивающее два среза байтов
// в лексикографическом порядке.
// Результат будет 0, если a == b, -1, если a < b, и +1, если a > b
func Compare(a, b []byte) int {
    for i := 0; i < len(a) && i < len(b); i++ {
        switch {
        case a[i] > b[i]:
            return 1
        case a[i] < b[i]:
            return -1
        }
    }
    switch {
    case len(a) > len(b):
        return 1
    case len(a) < len(b):
        return -1
    }
    return 0
}
```

Type switch

`Switch` **также может использоваться для определения динамического типа переменной интерфейса**. Такая проверка типа использует синтаксис проверки типа с ключевым словом `type` внутри скобок. Если `switch` объявляет переменную в выражении, эта переменная будет иметь соответствующий тип в каждом случае. Также **является идиоматическим переиспользование имени в таких случаях**, фактически объявляя новую переменную с тем же именем, но с другим типом в каждом случае.

```
var t interface{}
t = functionOfSomeType()
switch t := t.(type) {
default:
    fmt.Printf("неожиданный тип %T\n", t)     // %T выводит тип переменной t
case bool:
    fmt.Printf("логическое значение %t\n", t)  // t имеет тип bool
case int:
    fmt.Printf("целое число %d\n", t)          // t имеет тип int
case *bool:
    fmt.Printf("указатель на логическое значение %t\n", *t) // t имеет тип *bool
case *int:
    fmt.Printf("указатель на целое число %d\n", *t) // t имеет тип *int
}
```

Функции

Множественные возвращаемые значения

Одной из необычных особенностей Go является то, что функции и методы могут возвращать несколько значений. Эта возможность может быть использована для улучшения некоторых неудобных идиом в программах на языке `C`: например, возврат ошибок с использованием специальных значений, таких как `-1` для обозначения конца файла (`EOF`), и изменение аргумента, переданного по адресу.
В `C` ошибка записи сигнализируется отрицательным числом, при этом код ошибки хранится в каком-то изменяемом месте. В Go метод `Write` **может вернуть количество записанных байтов и ошибку**: "Да, вы записали некоторые байты, **но не все, потому что заполнили устройство**". Сигнатура метода `Write` для файлов из пакета `os` выглядит следующим образом:

```
func (file *File) Write(b []byte) (n int, err error)
```

И, как указано в документации, метод возвращает количество записанных байтов и ненулевую ошибку, когда `n != len(b)` . Это распространённый стиль; для получения дополнительных примеров обратитесь к разделу обработки ошибок.
Аналогичный подход устраняет необходимость передавать указатель на возвращаемое значение, чтобы симулировать параметр-ссылку. Вот простая функция, которая извлекает число из указанной позиции в срезе байтов, возвращая само число и следующую позицию.

```
func nextInt(b []byte, i int) (int, int) {
    for ; i < len(b) && !isDigit(b[i]); i++ {
    }
    x := 0
    for ; i < len(b) && isDigit(b[i]); i++ {
        x = x*10 + int(b[i]) - '0'
    }
    return x, i
}
```

Вы могли бы использовать её для сканирования чисел в входном срезе `b` следующим образом:

```
for i := 0; i < len(b); {
    x, i = nextInt(b, i)
    fmt.Println(x)
}
```

Именованные параметры результата

Возвращаемые или результативные "параметры" функции в Go могут иметь имена и использоваться как обычные переменные, так же, как и входящие параметры. **Когда им присваиваются имена, они инициализируются нулевыми значениями для их типов при начале выполнения функции; если функция выполняет оператор `return` без аргументов, то в качестве возвращаемых значений используются текущие значения параметров результата.** Имена не являются обязательными, но они могут сделать код короче и понятнее: **они служат документацией**. Если мы дадим имена результатам функции `nextInt`, станет очевидно, какой из возвращаемых `int` является каким.

```
func nextInt(b []byte, pos int) (value, nextPos int)
```

Поскольку именованные результаты инициализируются и привязаны к **неукрашенному `return`**, они могут не только упростить код, но и сделать его более ясным. Вот версия функции `io.ReadFull`, которая хорошо их использует:

```
func ReadFull(r Reader, buf []byte) (n int, err error) {
    for len(buf) > 0 && err == nil {
        var nr int
        nr, err = r.Read(buf)
        n += nr
        buf = buf[nr:]
    }
    return
}
```

Отложенные вызовы (defer)

Оператор `defer` в Go откладывает выполнение функции (отложенной функции) до момента, когда вызывающая её функция завершится и начнёт возвращать результат. Это необычный, но эффективный способ решения **таких задач, как освобождение ресурсов, которые должны быть освобождены независимо от того, каким образом функция завершает свою работу**. **Каноническими примерами являются разблокировка мьютекса или закрытие файла**.

```
// Contents возвращает содержимое файла в виде строки.
func Contents(filename string) (string, error) {
    f, err := os.Open(filename)
    if err != nil {
        return "", err
    }
    defer f.Close()  // f.Close будет вызвана, когда функция завершит работу.

    var result []byte
    buf := make([]byte, 100)
    for {
        n, err := f.Read(buf[0:])
        result = append(result, buf[0:n]...) // append будет написана позже.
        if err != nil {
            if err == io.EOF {
                break
            }
            return "", err  // f будет закрыт, если мы выйдем здесь.
        }
    }
    return string(result), nil // f будет закрыт, если мы выйдем здесь.
}
```

Откладывание вызова функции, такой как `Close`, имеет два преимущества. Во-первых, это гарантирует, что вы никогда не забудете закрыть файл, что легко сделать, если вы позже измените функцию, добавив новый путь возврата. Во-вторых, это означает, что закрытие файла находится рядом с его открытием, что делает код более понятным, чем если бы закрытие было размещено в конце функции.
Аргументы для отложенной функции (включая получателя, если функция является методом) вычисляются в момент выполнения `defer`, а не в момент вызова отложенной функции, **то есть в момент объявления блока `defer`**. Это позволяет избежать проблем с изменением значений переменных по мере выполнения функции, а также позволяет с помощью одного вызова `defer` отложить выполнение нескольких функций. Вот простой пример:

```
for i := 0; i < 5; i++ {
    defer fmt.Printf("%d ", i)
}
```

Отложенные функции выполняются в порядке **LIFO** (**Last In, First Out**), поэтому этот код выведет **4 3 2 1 0**, когда функция завершится. Более правдоподобный пример — простой способ отслеживания выполнения функций в программе. Мы могли бы написать несколько простых функций для трассировки:

```
func trace(s string)   { fmt.Println("entering:", s) }
func untrace(s string) { fmt.Println("leaving:", s) }
```

Использовать их можно так:

```
func a() {
    trace("a")
    defer untrace("a")
    // что-то делаем...
}
```

Мы можем улучшить код, используя тот факт, что аргументы для отложенных функций вычисляются в момент выполнения defer. Функция трассировки может подготовить аргумент для функции отслеживания завершения. Этот пример:

```
func trace(s string) string {
    fmt.Println("entering:", s)
    return s
}

func un(s string) {
    fmt.Println("leaving:", s)
}

func a() {
    defer un(trace("a"))
    fmt.Println("in a")
}

func b() {
    defer un(trace("b"))
    fmt.Println("in b")
    a()
}

func main() {
    b()
}
```

выводит:

```
entering: b
in b
entering: a
in a
leaving: a
leaving: b
```

Для программистов, привыкших к управлению ресурсами на уровне блоков кода в других языках, `defer` может показаться странным, **но его самые интересные и мощные применения связаны с тем, что он работает на уровне функции, а не блока кода**. В разделе о `panic` и `recover` будет показан ещё один пример его возможностей.

Данные

Выделение (аллокация) памяти с помощью `new`

В Go есть два примитива выделения памяти: **встроенные функции `new` и `make`.** Они выполняют разные задачи и применяются к различным типам, что может быть немного запутанным, но правила просты. Сначала поговорим о `new`. Это встроенная функция, **которая выделяет память, но, в отличие от аналогичных функций в некоторых других языках, она *не инициализирует* память, а лишь *обнуляет* её**. То есть `new(T)` выделяет память для нового объекта типа `T`, обнуляет её и возвращает его адрес, значение типа `*T`. В терминологии Go, она возвращает указатель на только что выделенное нулевое значение типа `T`.

Поскольку память, возвращаемая `new`, обнулена, полезно при проектировании ваших структур данных предусмотреть, чтобы нулевое значение каждого типа можно было использовать без дальнейшей инициализации. Это означает, что пользователь структуры данных может создать её с помощью `new` и сразу начать работу. Например, в документации для `bytes.Buffer` говорится, что "**нулевое значение для `Buffer` — это пустой буфер, готовый к использованию**". Аналогично, `sync.Mutex` не имеет явного конструктора или метода `Init`. **Вместо этого нулевое значение для `sync.Mutex` определяется как разблокированный мьютекс.**
Свойство "**нулевое значение полезно**" работает транзитивно. Рассмотрим это объявление типа.

```
type SyncedBuffer struct {
    lock    sync.Mutex
    buffer  bytes.Buffer
}
```

Значения типа `SyncedBuffer` также готовы к использованию сразу после выделения или просто объявления. В следующем фрагменте кода и `p`, и `v` будут работать корректно без дальнейшей настройки.

```
p := new(SyncedBuffer)  // тип *SyncedBuffer
var v SyncedBuffer      // тип  SyncedBuffer
```

Транзитивность означает, что при объявлении через `new` или как `var` ***поля структуры будут обнулены, то есть нулевыми, то есть у каждого поля в зависимости от типа будет нулевое значение.***

Конструкторы и составные литералы

Иногда нулевое значение недостаточно, и необходим конструктор для инициализации, как в следующем примере из пакета `os`.

```
func NewFile(fd int, name string) *File {
    if fd < 0 {
        return nil
    }
    f := new(File)
    f.fd = fd
    f.name = name
    f.dirinfo = nil
    f.nepipe = 0
    return f
}
```

В этом примере много шаблонного кода (так называемый `boilerplate code`). Мы можем упростить его, **используя составной литерал, который является выражением, создающим новый экземпляр при каждом его выполнении.**

```
func NewFile(fd int, name string) *File {
    if fd < 0 {
        return nil
    }
    f := File{fd, name, nil, 0}
    return &f
}
```

Обратите внимание, что в отличие от `C`, **возвращать адрес локальной переменной в Go вполне допустимо**; **память**, связанная с переменной, **сохраняется после завершения функции**. На самом деле, **взятие адреса составного литерала выделяет новый экземпляр при каждом выполнении, поэтому мы можем объединить последние две строки:**

```
return &File{fd, name, nil, 0}
```

Поля составного литерала располагаются в порядке и должны быть все указаны. Однако, указывая элементы явно в виде пар "**поле: значение**", инициализаторы могут быть в любом порядке, а пропущенные будут иметь нулевые значения. Так мы можем написать:

```
return &File{fd: fd, name: name}
```

В качестве крайнего случая, если составной литерал не содержит полей, **он создаст нулевое значение для типа**. Выражения `new(File)` и `&File{}` ***эквивалентны***.
Составные литералы также могут быть созданы для массивов, срезов и карт, где метки полей будут индексами или ключами карты. В этих примерах инициализация работает независимо от значений `Enone`, `Eio` и `Einval`, если они уникальны.

```
const Enone = 0
const Eio = 1
const Einval = 2
a := [...]string{Enone: "no error", Eio: "Eio", Einval: "invalid argument"}
s := []string{Enone: "no error", Eio: "Eio", Einval: "invalid argument"}
m := map[int]string{Enone: "no error", Eio: "Eio", Einval: "invalid argument"}
```

Выделение (аллокация) памяти с помощью `make`

Вернемся к выделению памяти. Встроенная функция `make(T, args)` имеет другую цель по сравнению с `new(T)`. Она создает **только срезы, карты и каналы и возвращает *инициализированное* (*а не обнуленное*) значение типа `T` (а не `*T`)**. Причина различия в том, что **эти три типа представляют собой ссылки на структуры данных, которые должны быть инициализированы перед использованием**. Например, *срез — это трехкомпонентный дескриптор, содержащий указатель на данные (внутри массива), длину и емкость, и пока эти элементы не инициализированы, срез равен `nil`*. Для срезов, карт и каналов функция `make` инициализирует внутреннюю структуру данных и подготавливает значение к использованию. Например,

```
make([]int, 10, 100)
```

выделяет массив из 100 целых чисел и затем создает структуру среза с длиной 10 и емкостью 100, указывающую на первые 10 элементов массива. (При создании среза емкость можно опустить; см. раздел о срезах для получения дополнительной информации.) В отличие от этого, `new([]int)` возвращает указатель на только что выделенную, обнуленную структуру среза, то есть указатель на значение `nil` среза.
Эти примеры иллюстрируют разницу между new и make.

```
var p *[]int = new([]int)       // выделяет структуру среза; *p == nil; редко полезно
var v  []int = make([]int, 100) // срез v теперь ссылается на новый массив из 100 целых чисел

// Неоправданно сложно (оно не надо, не делайте так):
var p *[]int = new([]int)
*p = make([]int, 100, 100)

// Идиоматично:
v := make([]int, 100)
```

Помните, что `make` **применяется только к картам, срезам и каналам и не возвращает указатель**. Чтобы получить явный указатель, используйте `new` или **явно возьмите адрес переменной**.

Массивы

Массивы полезны при планировании детальной структуры памяти и иногда помогают избежать выделения памяти, но в основном они служат строительным блоком для срезов, которые будут обсуждаться в следующем разделе. Чтобы заложить основу для этой темы, вот несколько слов о массивах.
Есть значительные различия в том, как массивы работают в Go и `C`. В Go:

- *Массивы — это значения. Присваивание одного массива другому копирует все элементы.*
- *В частности, если передать массив в функцию, она получит копию массива, а не указатель на него.*
- *Размер массива является частью его типа. Типы `[10]int` и `[20]int` различны.*

Свойство ***"значения"*** может быть полезным, но также дорогим по ресурсам; **если вам нужно поведение и эффективность, как в `C`, можно передать указатель на массив.**

```
func Sum(a *[3]float64) (sum float64) {
    for _, v := range *a {
        sum += v
    }
    return
}

array := [...]float64{7.0, 8.5, 9.1}
x := Sum(&array)  // Обратите внимание на явное использование оператора взятия адреса
```

Но даже этот стиль не является идиоматичным для Go.
Вместо этого используйте срезы.

Слайсы (Срезы, Slices)

Срезы оборачивают массивы, предоставляя более общий, мощный и удобный интерфейс к последовательностям данных. За исключением элементов с явными размерами, таких как матрицы преобразований, большинство программирования с массивами в Go осуществляется с использованием срезов, а не простых массивов.
Срезы содержат ссылки на лежащие в основе массивы, и если присвоить один срез другому, оба будут ссылаться на один и тот же массив. Если функция принимает аргумент в виде среза, изменения, которые она вносит в элементы среза, будут видны вызывающей стороне, аналогично передаче указателя на массив. Функция чтения (например, `Read`) может принять срез в качестве аргумента вместо указателя и количества данных; длина среза устанавливает верхний предел объема данных для чтения. Вот сигнатура метода `Read` типа `File` из пакета `os`:

```
func (f *File) Read(buf []byte) (n int, err error)
```

Метод возвращает количество прочитанных байт и значение ошибки, если таковая имеется. Чтобы прочитать первые 32 байта из большего буфера `buf`, нужно просто "нарезать" буфер.

```
n, err := f.Read(buf[0:32])
```

Такое нарезание является обычной практикой и очень эффективно. **Фактически, если оставить эффективность в стороне, следующий фрагмент кода также прочитает первые 32 байта буфера.**

```
var n int
var err error
for i := 0; i < 32; i++ {
    nbytes, e := f.Read(buf[i:i+1])  // Чтение по одному байту.
    n += nbytes
    if nbytes == 0 || e != nil {
        err = e
        break
    }
}
```

Длину среза можно изменять, пока она укладывается в пределы исходного массива; для этого достаточно присвоить срез самому себе. Вместимость среза (доступная через встроенную функцию `cap`) показывает максимальную длину, которую может иметь срез. **Вот функция для добавления данных в срез. Если данных больше, чем позволяет вместимость, срез перераспределяется**. **Возвращается изменённый срез**. В функции используется тот факт, что для нулевого среза длина и вместимость равны 0.

```
func Append(slice, data []byte) []byte {
    l := len(slice)
    if l + len(data) > cap(slice) {  // перераспределение
        // Выделяем память с запасом для будущего роста.
        newSlice := make([]byte, (l+len(data))*2)
        // Функция copy встроена и работает с любыми типами срезов.
        copy(newSlice, slice)
        slice = newSlice
    }

    slice = slice[0:l+len(data)]
    copy(slice[l:], data)
    return slice
}
```

Мы должны вернуть срез после изменения, потому что, хотя функция Append может изменять элементы среза, сам срез (структура, содержащая указатель, длину и вместимость) передается по значению.
Идея добавления в срез настолько полезна, что она реализована во встроенной функции `append`. Чтобы понять дизайн этой функции, нам нужно немного больше информации, к которой мы вернёмся позже.

Двумерные срезы

Массивы и срезы в Go **одномерны**. Чтобы создать эквивалент двумерного массива или среза, необходимо определить **массив массивов или срез срезов**, например:

```
type Transform [3][3]float64  // 3x3 массив, на самом деле массив массивов.
```

```
type LinesOfText [][]byte     // Срез срезов байтов.
```

Поскольку срезы имеют переменную длину, **каждый внутренний срез может иметь разную длину**. Это может быть распространенной ситуацией, как в нашем примере `LinesOfText`: каждая строка имеет независимую длину.

```
text := LinesOfText{
    []byte("Now is the time"),
    []byte("for all good gophers"),
    []byte("to bring some fun to the party."),
}
```

Иногда необходимо выделить двумерный срез, например, при обработке сканирующих строк пикселей. **Существует два способа достичь этого**. Один из способов — выделять каждый срез независимо; другой — выделить один массив и направить в него отдельные срезы. Какой способ использовать, зависит от вашего приложения. Если срезы могут увеличиваться или уменьшаться, их следует выделять независимо, чтобы избежать перезаписи следующей строки; если нет, то может быть эффективнее создать объект с помощью одного выделения памяти. Для справки, вот схемы двух методов. Сначала — строка за строкой:

```
// Для примера они содержат такие значения.
YSize := 20
XSize := 10
// Выделим срез верхнего уровня.
picture := make([][]uint8, YSize) // Одна строка на единицу y.
// Переберите строки, выделяя срез для каждой строки.
for i := range picture {
    picture[i] = make([]uint8, XSize)
}
```

А теперь — одно выделение, нарезанное на строки:

```
YSize := 20
XSize := 10
// Выделите срез верхнего уровня, так же как и прежде.
picture := make([][]uint8, YSize) // Одна строка на единицу y.
// Выделите один большой срез для всех пикселей.
pixels := make([]uint8, XSize*YSize) // Имеет тип []uint8, хотя picture — это [][]uint8.
// Переберите строки, нарезая каждую строку из начала оставшегося среза пикселей.
for i := range picture {
    picture[i], pixels = pixels[:XSize], pixels[XSize:]
}
```

Карты

Карты — это удобная и мощная встроенная структура данных, которая связывает значения одного типа (ключ) со значениями другого типа (элемент или значение). Ключ может быть любого типа, для которого определён оператор равенства, например, целые числа, числа с плавающей точкой и комплексные числа, строки, указатели, интерфейсы (если динамический тип поддерживает равенство), структуры и массивы. Срезы не могут использоваться в качестве ключей карт, потому что на них не определено равенство. Подобно срезам, карты хранят ссылки на основную структуру данных. **Если вы передадите карту функции, которая изменяет содержимое карты, изменения будут видны вызывающему коду**. Карты можно создавать с помощью обычного синтаксиса составного литерала с парами ключ-значение, разделёнными двоеточием, поэтому их легко строить при инициализации.

```
var timeZone = map[string]int{
    "UTC":  0*60*60,
    "EST": -5*60*60,
    "CST": -6*60*60,
    "MST": -7*60*60,
    "PST": -8*60*60,
}
```

Присваивание и извлечение значений из карты синтаксически выглядит так же, как и для массивов и срезов, за исключением того, что индекс не обязательно должен быть целым числом.

```
offset := timeZone["EST"]
```

Попытка получить значение карты по ключу, которого нет в карте, **вернёт нулевое значение для типа элементов карты**. Например, если карта содержит целые числа, поиск несуществующего ключа вернёт 0. Множество можно реализовать как

карту с типом значения bool. Установите значение карты в true, чтобы добавить значение в множество, а затем проверьте его с помощью простого индексирования.

```
attended := map[string]bool{
    "Ann": true,
    "Joe": true,
    ...
}

if attended[person] { // будет false, если person нет в карте
    fmt.Println(person, "was at the meeting")
}
```

Иногда нужно отличить отсутствующую запись от нулевого значения. Есть ли запись для `"UTC"`, или это 0, потому что её вообще нет в карте? Вы можете узнать это с помощью формы множественного присваивания.

```
var seconds int
var ok bool
seconds, ok = timeZone[tz]
```

По очевидным причинам это называется идиомой **“comma-ok”**. В этом примере, если `tz` присутствует, `seconds` будет установлен соответственно, а `ok` будет `true`; если нет, `seconds` будет установлен в `0`, а `ok` будет `false`. Вот функция, которая объединяет это с хорошим отчётом об ошибке:

```
func offset(tz string) int {
    if seconds, ok := timeZone[tz]; ok {
        return seconds
    }
    log.Println("unknown time zone:", tz)
    return 0
}
```

Чтобы проверить присутствие в карте, не беспокоясь о фактическом значении, вы можете использовать пустой идентификатор (_) вместо обычной переменной для значения.

```
_, present := timeZone[tz]
```

Чтобы удалить запись из карты, используйте встроенную функцию `delete`, аргументы которой — это карта и ключ, который нужно удалить. **Это безопасно, даже если ключ уже отсутствует в карте.**

```
delete(timeZone, "PDT") // Удаляем ключ "PDT" и его значение, соответственно
```

Печать

Форматированная печать в Go использует стиль, похожий на семейство функций `printf` в `C`, но более богатый и универсальный. Функции находятся в пакете `fmt` и имеют имена с заглавных букв: `fmt.Printf`, `fmt.Fprintf`, `fmt.Sprintf` и т.д. **Функции строк (например, `Sprintf` и др.) возвращают строку, а не заполняют предоставленный буфер.**
Не обязательно указывать строку формата. Для каждой из функций `Printf`, `Fprintf` и `Sprintf` существуют парные функции, например `Print` и `Println`. Эти функции не принимают строку формата, а вместо этого генерируют формат по умолчанию для каждого аргумента. **Версии `Println` также вставляют пробел между аргументами и добавляют новую строку в вывод, тогда как версии `Print` добавляют пробелы только в случае, если операнд с обеих сторон не является строкой**. В этом примере каждая строка производит одинаковый вывод.

```
fmt.Printf("Hello %d\n", 23)
fmt.Fprint(os.Stdout, "Hello ", 23, "\n")
fmt.Println("Hello", 23)
fmt.Println(fmt.Sprint("Hello ", 23))
```

Форматированные функции печати `fmt.Fprint` и другие принимают в качестве первого аргумента любой объект, реализующий интерфейс `io.Writer`; переменные `os.Stdout` и `os.Stderr` являются знакомыми примерами.
Здесь начинаются отличия от `C`. Во-первых, числовые форматы, такие как `%d`, не принимают флаги для знаковости или размера; вместо этого функции печати используют тип аргумента для определения этих свойств.

```
var x uint64 = 1<<64 - 1
fmt.Printf("%d %x; %d %x\n", x, x, int64(x), int64(x))
```

печатает

```
18446744073709551615 ffffffffffffffff; -1 -1
```

Если вы хотите получить формат по умолчанию, такой как десятичный для целых чисел, вы можете использовать универсальный формат `%v` (для "значения"); результат будет таким же, как `Print` и `Println`. **Более того, этот формат может печатать любое значение, включая массивы, срезы, структуры и карты**. Вот пример для карты временных зон, определённой в предыдущем разделе.

```
fmt.Printf("%v\n", timeZone)  // или просто fmt.Println(timeZone)
```

что выводит:

```
map[CST:-21600 EST:-18000 MST:-25200 PST:-28800 UTC:0]
```

Для карт функции `Printf` и другие сортируют вывод лексикографически по ключу.
При печати структуры модифицированный формат `%+v` аннотирует поля структуры их именами, а для любого значения альтернативный формат `%#v` печатает значение в полном синтаксисе Go.

```
type T struct {
    a int
    b float64
    c string
}
t := &T{ 7, -2.35, "abc\tdef" }
fmt.Printf("%v\n", t)
fmt.Printf("%+v\n", t)
fmt.Printf("%#v\n", t)
fmt.Printf("%#v\n", timeZone)
```

печатает

```
&{7 -2.35 abc   def}
&{a:7 b:-2.35 c:abc     def}
&main.T{a:7, b:-2.35, c:"abc\tdef"}
map[string]int{"CST":-21600, "EST":-18000, "MST":-25200, "PST":-28800, "UTC":0}
```

(Обратите внимание на амперсанды.) Этот формат строки также доступен через `%q`, если применить его к значению типа `string` или `[]byte`. Альтернативный формат `%#q` будет использовать обратные кавычки, если это возможно. (Формат `%q` также применяется к целым числам и рунам, создавая строковую константу с одинарными кавычками.) Также `%x` работает со строками, массивами байтов и срезами байтов, а также с целыми числами, создавая длинную строку в шестнадцатеричном формате, **и при использовании пробела в формате (`% x`) добавляет пробелы между байтами**.
Другой полезный формат — `%T`, который печатает тип значения.

```
fmt.Printf("%T\n", timeZone)
```

печатает

```
map[string]int
```

Если вы хотите контролировать формат по умолчанию для пользовательского типа, всё, что нужно сделать, это определить метод с сигнатурой `String() string` для типа. Для нашего простого типа `T` это может выглядеть так.

```
func (t *T) String() string {
    return fmt.Sprintf("%d/%g/%q", t.a, t.b, t.c)
}
fmt.Printf("%v\n", t)
```

чтобы напечатать в формате

```
7/-2.35/"abc\tdef"
```

(Если вам нужно печатать значения типа `T`, а также указатели на `T`, метод `String` должен быть методом типа значения; в этом примере использован указатель, потому что это более эффективно и идиоматично для структур. См. раздел ниже о получателях значений и указателей для получения дополнительной информации.)
Наш метод `String` может вызывать `Sprintf`, **поскольку функции печати полностью реентерабельны и могут быть обёрнуты таким образом**. Однако есть важная деталь, которую нужно понять об этом подходе: **не конструируйте метод `String`, вызывая `Sprintf` таким образом, чтобы он рекурсивно вызывал ваш метод `String` бесконечно. Это может**

произойти, если вызов `Sprintf` попытается напечатать получателя напрямую как строку, что, в свою очередь, снова вызовет метод. Это обычная и легкая ошибка, как показано в этом примере.

```
type MyString string

func (m MyString) String() string {
    return fmt.Sprintf("MyString=%s", m) // Ошибка.
    // Будет рекурсивно вызываться бесконечно.
}
```

Это также легко исправить: **преобразуйте аргумент к базовому строковому типу, у которого нет метода.**

```
type MyString string
func (m MyString) String() string {
    return fmt.Sprintf("MyString=%s", string(m)) // OK
    // Обратите внимание на преобразование.
}
```

В разделе инициализации мы увидим ещё одну технику, которая избегает этой рекурсии.
Ещё одна техника печати — это передача аргументов функции печати напрямую другой такой функции. Сигнатура функции `Printf` использует тип `...interface{}` для своего последнего аргумента, чтобы указать, что может быть произвольное количество параметров (произвольного типа) после формата.

```
func Printf(format string, v ...interface{}) (n int, err error) {
```

Внутри функции `Printf` `v` действует как переменная типа `[]interface{}`, но если она передана другой вариадической функции, она действует как обычный список аргументов. Вот реализация функции `log.Println`, которую мы использовали выше. Она передаёт свои аргументы напрямую в `fmt.Sprintln` для фактического форматирования.

```
// Println печатает в стандартный логгер, подобно fmt.Println.
func Println(v ...interface{}) {
    std.Output(2, fmt.Sprintln(v...))  // Output принимает параметры (int, string)
}
```

Мы пишем `...` после `v` в вложенном вызове `Sprintln`, чтобы указать компилятору рассматривать `v` как список аргументов; в противном случае он просто передаст `v` как один аргумент в виде среза.
Есть ещё много аспектов печати, которые мы не рассмотрели здесь. См. документацию `godoc` для пакета `fmt` для получения подробной информации.
Кстати, параметр `...` может быть конкретного типа, например, `...int` для функции `min`, которая выбирает минимальное значение из списка целых чисел:

```
func Min(a ...int) int {
    min := int(^uint(0) >> 1)  // наибольшее целое число
    for _, i := range a {
        if i < min {
            min = i
        }
    }
    return min
}
```

Добавление (Append)

Теперь у нас есть недостающее звено, которое необходимо для объяснения конструкции встроенной функции `append`. Сигнатура `append` отличается от нашей пользовательской функции `Append`, показанной выше. Схематически она выглядит так:

```
func append(slice []T, elements ...T) []T
```

где `T` — это заполнитель для любого типа (*`T` — это просто кастомный тип, какой мы захотим. Можем `int`, можем `string`, `rune` или свой тип любой*). В Go нельзя написать функцию, в которой тип `T` определяется вызывающей стороной. Именно поэтому `append` встроена в язык: ей требуется поддержка компилятора.
Что делает `append`, так это добавляет элементы в конец среза и возвращает результат. Результат нужно вернуть, потому что, как и в нашей вручную написанной функции `Append`, исходный массив может измениться. Этот простой пример:

```
x := []int{1,2,3}
x = append(x, 4, 5, 6)
fmt.Println(x)
```

выводит [1 2 3 4 5 6]. Таким образом, `append` работает немного как `Printf`, собирая произвольное количество аргументов. Но что, если бы мы захотели сделать то, что делает наша функция `Append`, и добавить один срез к другому срезу? Легко: используйте `...` в месте вызова, как мы сделали выше в вызове `Output`. Этот фрагмент кода выводит тот же результат, что и предыдущий.

```
x := []int{1,2,3}
y := []int{4,5,6}
x = append(x, y...)
fmt.Println(x)
```

Без `...` код не скомпилируется, потому что типы будут неправильными; `y` не является типом `int`.

Инициализация (Initialization)

Хотя инициализация в Go внешне не сильно отличается от инициализации в `C` или `C++`, в Go она более мощная. Сложные структуры могут быть построены в процессе инициализации, и проблемы с порядком инициализации объектов, даже из разных пакетов, обрабатываются корректно.

Константы

Константы в Go являются именно константами. Они создаются во время компиляции, даже если они определены как локальные переменные внутри функций, и **могут быть только числами, символами (рунами), строками или булевыми значениями**. Из-за ограничения компиляции, выражения, определяющие их, должны быть константными выражениями, которые могут быть вычислены компилятором. Например, `1<<3` — это константное выражение, в то время как `math.Sin(math.Pi/4)` — нет, потому что вызов функции `math.Sin` должен произойти во время выполнения программы. В Go перечисленные константы создаются с помощью перечислителя `iota`. Поскольку `iota` может быть частью выражения, а выражения могут повторяться неявно, легко создать сложные наборы значений.

```
type ByteSize float64

const (
    _           = iota // пропустить первое значение, присвоив его пустому идентификатору
    KB ByteSize = 1 << (10 * iota)
    MB
    GB
    TB
    PB
    EB
    ZB
    YB
)
```

Возможность прикрепить метод, такой как `String`, к любому пользовательскому типу делает возможным автоматическое форматирование произвольных значений для их вывода. Хотя это чаще применяется к структурам, эта техника также полезна для скалярных типов, таких как типы с плавающей запятой, например, `ByteSize`.

```
func (b ByteSize) String() string {
    switch {
    case b >= YB:
        return fmt.Sprintf("%.2fYB", b/YB)
    case b >= ZB:
        return fmt.Sprintf("%.2fZB", b/ZB)
    case b >= EB:
        return fmt.Sprintf("%.2fEB", b/EB)
    case b >= PB:
        return fmt.Sprintf("%.2fPB", b/PB)
    case b >= TB:
        return fmt.Sprintf("%.2fTB", b/TB)
    case b >= GB:
        return fmt.Sprintf("%.2fGB", b/GB)
    case b >= MB:
```

```
        return fmt.Sprintf("%.2fMB", b/MB)
    case b >= KB:
        return fmt.Sprintf("%.2fKB", b/KB)
    }
    return fmt.Sprintf("%.2fB", b)
}
```

Выражение `YB` выводится как `1.00YB`, а `ByteSize(1e13)` выводится как `9.09TB`.
Использование здесь `Sprintf` для реализации метода `String` типа `ByteSize` безопасно (избегает бесконечной рекурсии) не из-за преобразования, а потому, что оно вызывает `Sprintf` с `%f`, что не является строковым форматом: `Sprintf` вызовет метод `String` только тогда, когда ему нужна строка, а `%f` ожидает значение с плавающей запятой.

Переменные

Переменные могут инициализироваться аналогично константам, но инициализатор может быть общим выражением, вычисляемым во время выполнения программы.

```
var (
    home   = os.Getenv("HOME")
    user   = os.Getenv("USER")
    gopath = os.Getenv("GOPATH")
)
```

Функция init

Наконец, каждый исходный файл может определить свою собственную функцию `init` без параметров, чтобы настроить всё необходимое состояние. (На самом деле каждый файл может иметь несколько функций `init`.) И "наконец" здесь означает следующее: **`init` вызывается после того, как все объявления переменных в пакете инициализировали свои значения, а это происходит только после того, как все импортированные пакеты были инициализированы.**
Помимо инициализаций, которые не могут быть выражены как декларации, распространённое использование функций `init` заключается в проверке или исправлении корректности состояния программы перед началом её реального выполнения.

```
func init() {
    if user == "" {
        log.Fatal("$USER not set")
    }
    if home == "" {
        home = "/home/" + user
    }
    if gopath == "" {
        gopath = home + "/go"
    }
    // gopath может быть переопределён флагом --gopath в командной строке.
    flag.StringVar(&gopath, "gopath", gopath, "переопределить стандартный GOPATH")
}
```

Методы

Указатели против значений

Как мы видели с `ByteSize`, методы могут быть определены для любого именованного типа (кроме указателя или интерфейса); получатель метода не обязательно должен быть структурой.
В обсуждении выше мы написали функцию `Append` для срезов. Мы можем определить её как метод для срезов. Для этого сначала объявим именованный тип, к которому мы можем привязать метод, а затем сделаем получателем метода значение этого типа.

```
type ByteSlice []byte

func (slice ByteSlice) Append(data []byte) []byte {
```

```
    // Тело метода такое же, как у функции Append, описанной выше.
}
```

Однако этот метод всё ещё требует возврата обновлённого среза. Мы можем избавиться от этого неудобства, переопределив метод так, чтобы получателем был указатель на `ByteSlice`, и тогда метод сможет изменять срез вызывающей стороны.

```
func (p *ByteSlice) Append(data []byte) {
    slice := *p
    // Тело как и выше, без возврата значения.
    *p = slice
}
```

На самом деле, мы можем сделать ещё лучше. Если мы изменим нашу функцию так, чтобы она выглядела как стандартный метод `Write`, это будет выглядеть так:

```
func (p *ByteSlice) Write(data []byte) (n int, err error) {
    slice := *p
    // Снова, как и выше.
    *p = slice
    return len(data), nil
}
```

Тогда тип `*ByteSlice` будет удовлетворять стандартному интерфейсу `io.Writer`, что очень удобно (***он имплементирует этот метод, а значит в местах, где нужен в будет io.Writer, мы сможем использовать наш тип*** `ByteSlice`). Например, мы можем записать данные в этот тип с помощью форматированного вывода.

```
var b ByteSlice
fmt.Fprintf(&b, "This hour has %d days\n", 7)
```

Мы передаем адрес `ByteSlice`, потому что только `*ByteSlice` удовлетворяет интерфейсу `io.Writer`. Правило для указателей и значений в методах состоит в том, что методы значений могут вызываться как на указателях, так и на значениях, а методы указателей могут вызываться только на указателях.
Такое правило: ***если метод имеет приемник-указатель (например,*** `*ByteSlice`***), то его можно вызывать только на указателе****, а не на значении.* ***Если метод имеет приемник-значение (например,*** `ByteSlice`***), его можно вызывать и на значениях, и на указателях****. Это сделано для того, чтобы при передаче значений методы могли изменять внутреннее состояние, если они работают с указателями.*
Это правило возникает из-за того, что методы указателей могут изменять получателя; вызов их на значении приведет к тому, что метод получит копию значения, и любые изменения будут отброшены. Поэтому язык запрещает такую ошибку.
Однако есть удобное исключение: **когда значение доступно по адресу, язык автоматически вставляет оператор адресации**. В нашем примере переменная `b` доступна по адресу, поэтому мы можем вызвать метод `Write` просто как `b.Write`. **Компилятор перепишет это на** `(&b).Write` **за нас.**
Кстати, идея использования `Write` для среза байтов является ключевой в реализации `bytes.Buffer`.
Полный пример из раздела:

```
package main

import (
    "fmt"
)

// Определяем новый тип ByteSlice на основе среза байтов
type ByteSlice []byte

// Метод Append для добавления данных в срез
// Принимает УКАЗАТЕЛЬ на ByteSlice, чтобы изменять исходные данные
func (p *ByteSlice) Append(data []byte) {
    slice := *p
    // Добавляем новые данные в срез
    slice = append(slice, data...)
    // Обновляем исходное значение через указатель
    *p = slice
}

// Метод Write для удовлетворения интерфейса io.Writer
// Он добавляет данные в срез и возвращает количество записанных байтов и ошибку (nil в данном случае)
func (p ByteSlice) Write(data []byte) (n int, err error) {
```

```
    // Добавляем данные в срез
    p.Append(data)

    // Выведем, чтобы узнать, какие данные лежат в data
    fmt.Println(string(data))
    // Вывод: Это заняло 7 дней
    //

    // Возвращаем количество добавленных байтов и nil, так как ошибок нет
    return len(data), nil
}

func main() {
    // Создаем переменную типа ByteSlice
    var b ByteSlice

    // Используем fmt.Fprintf для записи в наш ByteSlice
    fmt.Fprintf(b, "Это заняло %d дней\n", 7)

    // Печатаем результат
    fmt.Println(string(b)) // Преобразуем срез байтов в строку для вывода
    // Вывод: Это заняло 7 дней
    //
}
```

Интерфейсы и другие типы

Интерфейсы

Интерфейсы в Go предоставляют способ задать поведение объекта: **если что-то может выполнять определенное действие, его можно использовать здесь**. Мы уже видели пару простых примеров; пользовательские принтеры можно реализовать с помощью метода `String`, в то время как `Fprintf` может генерировать вывод для всего, что имеет метод `Write`. **Интерфейсы, содержащие только один или два метода, распространены в Go-коде и обычно получают имя, производное от метода**, например, `io.Writer` для чего-то, что реализует `Write`.
*То есть, если метод `Write`, то интерфейс `Writer`. Если метод `Read`, то интерфейс `Reader`. **Интерфейсы с одним методом принято в Go называть как "Имя метода + `er`"**. Другой пример: `String` -> `Stringer`.*
Тип может реализовать несколько интерфейсов. Например, коллекцию можно отсортировать с помощью функций из пакета `sort`, если она реализует интерфейс `sort.Interface`, который содержит методы `Len()`, `Less(i, j int) bool` и `Swap(i, j int)`, и у нее также может быть свой метод форматирования. В этом условном примере `Sequence` удовлетворяет обоим.

```
type Sequence []int

// Методы, необходимые для sort.Interface.
func (s Sequence) Len() int {
    return len(s)
}
func (s Sequence) Less(i, j int) bool {
    return s[i] < s[j]
}
func (s Sequence) Swap(i, j int) {
    s[i], s[j] = s[j], s[i]
}

// Copy возвращает копию Sequence.
func (s Sequence) Copy() Sequence {
    copy := make(Sequence, 0, len(s))
    return append(copy, s...)
}

// Метод для вывода - сортирует элементы перед выводом.
func (s Sequence) String() string {
    s = s.Copy() // Сделать копию; не перезаписывать аргумент.
    sort.Sort(s)
    str := "["
    for i, elem := range s { // Цикл O(N²); исправим это в следующем примере.
```

```
        if i > 0 {
            str += " "
        }
        str += fmt.Sprint(elem)
    }
    return str + "]"
}
```

Полный пример для "потыкать":

```
package main

import (
    "fmt"
    "sort"
)

type Sequence []int

// Методы, необходимые для sort.Interface.
func (s Sequence) Len() int {
    return len(s)
}
func (s Sequence) Less(i, j int) bool {
    return s[i] < s[j]
}
func (s Sequence) Swap(i, j int) {
    s[i], s[j] = s[j], s[i]
}

// Copy возвращает копию Sequence.
func (s Sequence) Copy() Sequence {
    copy := make(Sequence, 0, len(s))
    return append(copy, s...)
}

// Метод для вывода - сортирует элементы перед выводом.
func (s Sequence) String() string {
    s = s.Copy() // Сделать копию; не перезаписывать аргумент.
    sort.Sort(s)
    str := "["
    for i, elem := range s { // Цикл O(N²); исправим это в следующем примере.
        if i > 0 {
            str += " "
        }
        str += fmt.Sprint(elem)
    }
    return str + "]"
}

func main() {
    // Создаем объект seq типа Sequence
    var seq Sequence

    // Аллоцируем память под 10 элементов
    seq = make(Sequence, 10)

    // Закинем в seq случайные элементы
    seq = append(seq, 123, 632, 73, -123, 5, 6)

    // Выведем в консоль seq
    fmt.Println(seq)
    // Вывод: [-123 0 0 0 0 0 0 0 0 0 5 6 73 123 632]
    // Потому что мы для нашего типа Sequence имплементировали интерфейс Stringer,
    // написав реализацию метода String.
    // Поэтому при вызове fmt.Println у нас вызывается метод String() и идет сортировка массива
}
```

Преобразования

Метод `String` для типа `Sequence` повторяет работу, которую уже делает `Sprint` для срезов. (Кроме того, его сложность $O(N^2)$, что **неэффективно**.) **Мы можем разделить эту работу (и также ускорить её), если преобразуем `Sequence` в обычный `[]int` перед вызовом `Sprint`.**

```
func (s Sequence) String() string {
    s = s.Copy()
    sort.Sort(s)
    return fmt.Sprint([]int(s))
}
```

*Таким образом, мы убрали собственное преобразование к строке, потому что стандартная библиотека умеет работать со стандартными типами данных (в данном случае с `[]int`). При помощи `[]int(s)` мы преобразуем Sequence в `[]int` на время вызова функции, что **ПОЗВОЛЯЕТ** нам вызвать собственно функцию `fmt.Sprint` и не писать собственную реализацию.*
Этот метод — еще один пример техники преобразования для безопасного вызова `Sprintf` из метода `String`. **Поскольку эти два типа (`Sequence` и `[]int`) одинаковы, если игнорировать имя типа, их можно преобразовать друг в друга. Преобразование не создает новое значение, оно временно делает так, будто существующее значение имеет новый тип**. (Есть и другие допустимые преобразования, такие как из целого числа в число с плавающей запятой, которые действительно создают новое значение.)
Это идиома в программах на Go — преобразовывать тип выражения, чтобы получить доступ к другому набору методов. Например, мы можем использовать существующий тип `sort.IntSlice`, чтобы свести весь пример к следующему:

```
type Sequence []int

// Метод для вывода - сортирует элементы перед выводом.
func (s Sequence) String() string {
    s = s.Copy()
    sort.IntSlice(s).Sort()
    return fmt.Sprint([]int(s))
}
```

Теперь, вместо того чтобы заставлять `Sequence` реализовывать несколько интерфейсов (сортировка и вывод), мы используем возможность преобразования данных к разным типам (`Sequence`, `sort.IntSlice` и `[]int`), каждый из которых выполняет свою часть задачи. **Это менее распространено на практике, но может быть эффективным**.

Преобразования интерфейсов и утверждения типов

Переключатели типов (***Type Switches***) — это форма преобразования: **они берут интерфейс и, для каждого случая в переключателе, своего рода преобразуют его в тип этого случая**. Вот упрощенная версия того, как код в `fmt.Printf` преобразует значение в строку с помощью переключателя типов. Если это уже строка, мы хотим получить фактическое значение строки, хранимое в интерфейсе, в то время как если у него есть метод `String`, мы хотим результат вызова этого метода.

```
type Stringer interface {
    String() string
}

var value interface{} // Значение, предоставленное вызывающим.
switch str := value.(type) {
case string:
    return str
case Stringer:
    return str.String()
}
```

Первый случай находит конкретное значение; второй преобразует интерфейс в другой интерфейс. Смешивать типы таким образом абсолютно нормально.
Что если нас интересует только один тип? Если мы знаем, что значение содержит строку, и просто хотим извлечь её? Одного случая в переключателе типов достаточно, но также можно использовать утверждение типа. Утверждение типа берет значение интерфейса и извлекает из него значение указанного явного типа. Синтаксис заимствован из блока, открывающего переключатель типов, но с явным типом вместо ключевого слова `type`:

```
value.(typeName)
```

Результат — это новое значение со статическим типом `typeName`. Этот тип должен быть либо конкретным типом, хранимым в интерфейсе, либо вторым типом интерфейса, к которому можно преобразовать значение. Чтобы извлечь строку, которую мы знаем, что значение содержит, можно написать:

```
str := value.(string)
```

Но если окажется, что значение не содержит строку, программа аварийно завершится с ошибкой во время выполнения. *Чтобы защититься от этого, используйте идиому "запятая, ok" для безопасной проверки, является ли значение строкой:*

```
str, ok := value.(string)
if ok {
    fmt.Printf("строковое значение: %q\n", str)
} else {
    fmt.Printf("значение не является строкой\n")
}
```

Если утверждение типа не удастся, `str` все равно будет существовать и будет типа `string`, но оно будет иметь нулевое значение, пустую строку.
В качестве иллюстрации возможности вот условное выражение, эквивалентное переключателю типов, открывающему этот раздел.

```
if str, ok := value.(string); ok {
    return str
} else if str, ok := value.(Stringer); ok {
    return str.String()
}
```

Общие принципы

Если тип существует только для того, чтобы реализовать интерфейс и никогда не будет иметь экспортируемых методов, кроме тех, что описаны в интерфейсе, нет необходимости экспортировать сам тип. Экспортирование только интерфейса ясно показывает, что у значения нет интересного поведения, кроме того, что описано в интерфейсе. Это также избавляет от необходимости повторять документацию для каждого экземпляра общих методов.
В таких случаях конструктор должен возвращать значение интерфейса, а не реализующего типа. Например, в библиотеках хеширования как `crc32.NewIEEE`, так и `adler32.New` возвращают интерфейсный тип `hash.Hash32`. Замена алгоритма `CRC-32` на `Adler-32` в программе Go требует только изменения вызова конструктора; **остальной код не затронут изменением алгоритма.**
Подобный подход позволяет алгоритмам потокового шифрования в различных криптографических пакетах быть отделенными от блочных шифров, с которыми они работают. Интерфейс `Block` в пакете `crypto/cipher` описывает поведение блочного шифра, который обеспечивает шифрование одного блока данных. Затем, по аналогии с пакетом `bufio`, криптографические пакеты, реализующие этот интерфейс, могут быть использованы для создания потоковых шифров, представленных интерфейсом `Stream`, без знания деталей блочного шифрования.
Интерфейсы в пакете `crypto/cipher` выглядят так:

```
type Block interface {
    BlockSize() int
    Encrypt(dst, src []byte)
    Decrypt(dst, src []byte)
}

type Stream interface {
    XORKeyStream(dst, src []byte)
}
```

Вот определение потока в режиме счётчика (`CTR`), который превращает блочный шифр в потоковый шифр; обратите внимание, что детали блочного шифра абстрагированы:

```
// NewCTR возвращает Stream, который шифрует/дешифрует с использованием
// переданного блочного шифра в режиме счётчика. Длина iv должна быть
// такой же, как размер блока блочного шифра.
func NewCTR(block Block, iv []byte) Stream
```

`NewCTR` применяется не только к одному конкретному алгоритму шифрования и источнику данных, но и к любой реализации интерфейса `Block` и любому `Stream`. Поскольку они возвращают интерфейсные значения, замена

шифрования в режиме `CTR` на другие режимы — это локализованное изменение. Нужно только изменить вызовы конструкторов, но поскольку остальной код должен обрабатывать результат только как `Stream`, он не заметит разницы.

Интерфейсы и методы

Так как метод можно прикрепить практически к чему угодно, почти всё может удовлетворять интерфейсу. Один из наглядных примеров — это пакет `http`, в котором определяется интерфейс `Handler`. Любой объект, который реализует `Handler`, может обслуживать HTTP-запросы.

```
type Handler interface {
    ServeHTTP(ResponseWriter, *Request)
}
```

`ResponseWriter` сам по себе является интерфейсом, который предоставляет доступ к методам, необходимым для возврата ответа клиенту. Эти методы включают стандартный метод `Write`, так что `http.ResponseWriter` может использоваться везде, где используется `io.Writer`. `Request` — это структура, содержащая разобранное представление запроса от клиента.
Для краткости будем игнорировать **POST-запросы** и предполагать, что **HTTP-запросы** всегда являются **GET-запросами**; *это упрощение не влияет на способ настройки обработчиков.* **Вот простейшая реализация обработчика, который считает, сколько раз страница была посещена.**

```
// Простой сервер-счётчик.
type Counter struct {
    n int
}

func (ctr *Counter) ServeHTTP(w http.ResponseWriter, req *http.Request) {
    ctr.n++
    fmt.Fprintf(w, "counter = %d\n", ctr.n)
}
```

(В соответствии с нашей темой, обратите внимание, как `Fprintf` может выводить данные в `http.ResponseWriter`.) В реальном сервере доступ к `ctr.` n нужно было бы защитить от конкурентного доступа. *Ознакомьтесь с пакетами* `sync` *и* `atomic` *для предложений по этому поводу.*
Для справки, вот как прикрепить такой сервер к узлу в дереве `URL`.

```
import "net/http"

ctr := new(Counter)
http.Handle("/counter", ctr)
```

Но зачем делать `Counter` структурой? Достаточно одного целого числа. (**Получатель должен быть указателем, чтобы приращение было видно вызывающему коду**.)

```
// Более простой сервер-счётчик.
type Counter int

func (ctr *Counter) ServeHTTP(w http.ResponseWriter, req *http.Request) {
    *ctr++
    fmt.Fprintf(w, "counter = %d\n", *ctr)
}
```

Что если в вашей программе есть какое-то внутреннее состояние, которое должно быть уведомлено о том, что страница была посещена? **Свяжите канал с веб-страницей.**

```
// Канал, который отправляет уведомление при каждом посещении.
// (Вероятно, канал стоит сделать буферизированным.)
type Chan chan *http.Request

func (ch Chan) ServeHTTP(w http.ResponseWriter, req *http.Request) {
    ch <- req
    fmt.Fprint(w, "уведомление отправлено")
}
```

Наконец, допустим, мы хотим отобразить на `/args` аргументы, использованные при запуске серверного бинарного файла. Написать функцию для вывода аргументов очень просто.

```
func ArgServer() {
    fmt.Println(os.Args)
}
```

Как превратить это в HTTP-сервер? Мы могли бы сделать `ArgServer` методом какого-то типа, значение которого мы игнорируем, **но есть более элегантное решение**. Поскольку мы можем определить метод для любого типа, кроме указателей и интерфейсов, мы можем написать метод для функции. Пакет `http` содержит следующий код:

```
// Тип HandlerFunc — это адаптер, который позволяет
// использовать обычные функции в качестве HTTP-обработчиков.
// Если f — это функция с подходящей сигнатурой, HandlerFunc(f)
// будет объектом Handler, который вызывает f.
type HandlerFunc func(ResponseWriter, *Request)

// ServeHTTP вызывает f(w, req).
func (f HandlerFunc) ServeHTTP(w ResponseWriter, req *Request) {
    f(w, req)
}
```

`HandlerFunc` — это тип с методом `ServeHTTP`, так что значения этого типа могут обрабатывать **HTTP-запросы**. Посмотрите на реализацию метода: получателем является функция `f`, и метод вызывает `f`. **Это может показаться странным, но это не так уж отличается от того, когда получателем является канал, а метод отправляет данные в канал.**
Чтобы сделать из `ArgServer` **HTTP-сервер**, сначала изменим его сигнатуру.

```
// Сервер аргументов.
func ArgServer(w http.ResponseWriter, req *http.Request) {
    fmt.Fprintln(w, os.Args)
}
```

Теперь `ArgServer` имеет ту же сигнатуру, что и `HandlerFunc`, так что его можно преобразовать в этот тип, чтобы получить доступ к его методам, точно так же, как мы преобразовали `Sequence` в `IntSlice`, чтобы получить доступ к `IntSlice.Sort`. Код для его настройки лаконичен:

```
http.Handle("/args", http.HandlerFunc(ArgServer))
```

Когда кто-то заходит на страницу `/args`, обработчиком на этой странице является `ArgServer` с типом `HandlerFunc`. **HTTP-сервер** вызовет метод `ServeHTTP` этого типа, с `ArgServer` в качестве получателя, который в свою очередь вызовет `ArgServer` (через вызов `f(w, req)` внутри `HandlerFunc.ServeHTTP`). Тогда аргументы будут отображены.
В этом разделе мы сделали **HTTP-сервер** из структуры, целого числа, канала и функции, **потому что интерфейсы — это просто наборы методов, которые могут быть определены для (почти) любого типа.**
Полный пример кода из раздела:

```
package main

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

// Простой сервер-счётчик
type Counter struct {
    n int
}

func (ctr *Counter) ServeHTTP(w http.ResponseWriter, req *http.Request) {
    ctr.n++
    fmt.Fprintf(w, "counter = %d\n", ctr.n)
}

// Более простой сервер-счётчик
type SimpleCounter int

// Обратите внимание, интерфейс может быть реализован примитивом
func (ctr *SimpleCounter) ServeHTTP(w http.ResponseWriter, req *http.Request) {
    *ctr++
    fmt.Fprintf(w, "counter = %d\n", *ctr)
}
```

```go
// Канал для уведомления при посещении страницы
type Chan chan *http.Request

func (ch Chan) ServeHTTP(w http.ResponseWriter, req *http.Request) {
    ch <- req
    fmt.Fprint(w, "уведомление отправлено")
}

// Сервер аргументов
func ArgServer(w http.ResponseWriter, req *http.Request) {
    fmt.Fprintln(w, os.Args)
}

func main() {
    // Пример с сервером-счётчиком (структура)
    ctr := new(Counter)
    http.Handle("/counter", ctr)

    // Пример с простым счётчиком (целое число)
    var simpleCtr SimpleCounter
    http.Handle("/simple-counter", &simpleCtr)

    // Пример с каналом
    // Он должен быть буферизированным
    // Здесь, например, мы можем посетить 10 раз страницу
    // После чего страница не будет загружаться
    // Потому что канал не читается, а только заполняется
    ch := make(Chan, 10)
    http.Handle("/notify", ch)

    // Пример с выводом аргументов сервера
    http.Handle("/args", http.HandlerFunc(ArgServer))

    // Запуск сервера на порту 8080
    http.ListenAndServe(":8080", nil)
}
```

Интерфейсы в Go определяются ТОЛЬКО набором методов, и любой тип, который реализует эти методы, может удовлетворять интерфейсу.

Пустой идентификатор

Мы уже несколько раз упоминали пустой идентификатор в контексте циклов `for range` и работы с картами. Пустой идентификатор может быть присвоен или объявлен с любым значением любого типа, и это значение будет безопасно проигнорировано. Это немного похоже на запись в файл `/dev/null` в `Unix`: он представляет собой переменную, которую можно только записывать, и используется в тех случаях, когда переменная требуется, но само значение не имеет значения. Однако у пустого идентификатора есть и другие полезные применения, помимо тех, что мы уже видели.

Пустой идентификатор в множественном присваивании

Использование пустого идентификатора в цикле `for range` **является частным случаем общей ситуации — множественного присваивания.**
Если в присваивании требуется несколько значений с левой стороны, но одно из этих значений не будет использоваться программой, пустой идентификатор на левой стороне присваивания позволяет избежать создания фиктивной переменной и даёт понять, что это значение должно быть отброшено. Например, при вызове функции, которая возвращает значение и ошибку, но важно только наличие ошибки, пустой идентификатор используется для игнорирования неважного значения.

```go
if _, err := os.Stat(path); os.IsNotExist(err) {
    fmt.Printf("%s не существует\n", path)
}
```

Иногда можно встретить код, где ошибка игнорируется путём отбрасывания значения ошибки. **Это плохая практика**. **Всегда проверяйте возвращаемые ошибки; они предоставлены не просто так.**

```
// Плохо! Этот код упадёт, если path не существует.
fi, _ := os.Stat(path)
if fi.IsDir() {
    fmt.Printf("%s является директорией\n", path)
}
```

Неиспользуемые импорты и переменные

Импортирование пакета или объявление переменной без их фактического использования — это ошибка. Неиспользуемые импорты увеличивают размер программы и замедляют компиляцию, а переменная, которая инициализирована, но не используется, по крайней мере, является пустой вычислительной операцией, а возможно, указывает на более крупную ошибку. Однако, когда программа активно разрабатывается, часто возникают неиспользуемые импорты и переменные, и может быть неудобно их удалять только для того, чтобы компиляция прошла успешно, а затем снова добавлять их, когда они понадобятся. **Пустой идентификатор предоставляет решение этой проблемы.** Этот незаконченный код содержит два неиспользуемых импорта (`fmt` и `io`) и неиспользуемую переменную (`fd`), поэтому он не скомпилируется, но было бы полезно проверить, корректен ли код, написанный на данный момент.

```
package main

import (
    "fmt"
    "io"
    "log"
    "os"
)

func main() {
    fd, err := os.Open("test.go")
    if err != nil {
        log.Fatal(err)
    }
    // TODO: использовать fd.
}
```

Чтобы предотвратить жалобы на неиспользуемые импорты, можно использовать пустой идентификатор для ссылки на символ из импортированного пакета. Аналогично, присваивание неиспользуемой переменной `fd` пустому идентификатору избавит от ошибки о неиспользуемой переменной. **Эта версия программы компилируется.**

```
package main

import (
    "fmt"
    "io"
    "log"
    "os"
)

var _ = fmt.Printf // Для отладки; удалить, когда будет не нужно.
var _ io.Reader    // Для отладки; удалить, когда будет не нужно.

func main() {
    fd, err := os.Open("test.go")
    if err != nil {
        log.Fatal(err)
    }
    // TODO: использовать fd.
    _ = fd
}
```

По соглашению, глобальные объявления для подавления ошибок импорта должны находиться сразу после импорта и быть закомментированы, чтобы их было легко найти и напомнить о необходимости очистить код позже.

Импорт ради побочного эффекта

Неиспользуемый импорт, такой как `fmt` или `io`, в предыдущем примере должен быть либо использован, либо удален: **присваивания с использованием пустого идентификатора указывают на то, что код находится в процессе разработки. Но иногда полезно импортировать пакет только ради его побочных эффектов, без явного использования.** Например, во время выполнения функции `init` пакет `net/http/pprof` регистрирует **HTTP-обработчики**, которые предоставляют отладочную информацию. У этого пакета есть экспортируемый `API`, но большинству клиентов нужен только механизм регистрации обработчиков, и данные доступны через веб-страницу. **Чтобы импортировать пакет только ради его побочных эффектов, переименуйте пакет в пустой идентификатор:**

```
import _ "net/http/pprof"
```

Эта форма импорта явно указывает на то, что пакет импортируется ради его побочных эффектов, потому что других возможных целей для импорта нет: в этом файле у него нет имени. (Если бы оно было, и мы бы его не использовали, компилятор отклонил бы программу.)

Проверка интерфейсов

Как мы уже видели в обсуждении интерфейсов выше, тип не обязан явно заявлять, что он реализует интерфейс. Вместо этого тип реализует интерфейс просто за счет реализации методов интерфейса. На практике большинство преобразований интерфейсов статичны и поэтому проверяются во время компиляции. Например, **передача `*os.File` в функцию, которая ожидает `io.Reader`, не скомпилируется, если `*os.File` не реализует интерфейс `io.Reader`.**
Однако некоторые проверки интерфейсов происходят во время выполнения. Один пример — это пакет `encoding/json`, который определяет интерфейс `Marshaler`. Когда **JSON-кодировщик** получает значение, которое реализует этот интерфейс, он вызывает метод маршализации этого значения для преобразования его в `JSON` вместо стандартного преобразования. Кодировщик проверяет это свойство во время выполнения с помощью утверждения типа, такого как:

```
m, ok := val.(json.Marshaler)
```

Если необходимо только проверить, реализует ли тип интерфейс, без фактического использования самого интерфейса, например, в ходе проверки на наличие ошибки, **можно использовать пустой идентификатор, чтобы игнорировать значение, полученное через утверждение типа**:

```
if _, ok := val.(json.Marshaler); ok {
    fmt.Printf("значение %v типа %T реализует json.Marshaler\n", val, val)
}
```

Одна из ситуаций, где это возникает, — когда необходимо гарантировать, что внутри пакета, который реализует тип, этот тип действительно удовлетворяет интерфейсу. Если тип, например, `json.RawMessage`, нуждается в пользовательском представлении `JSON`, он должен реализовать интерфейс `json.Marshaler`, но статические преобразования не заставят компилятор автоматически это проверять. Если тип случайно не реализует интерфейс, **JSON-кодировщик** по-прежнему будет работать, но не будет использовать пользовательскую реализацию. Чтобы гарантировать правильность реализации, в пакете можно использовать глобальное объявление с использованием пустого идентификатора:

```
var _ json.Marshaler = (*RawMessage)(nil)
```

В этом объявлении присваивание, связанное с преобразованием `*RawMessage` в `Marshaler`, требует, чтобы `*RawMessage` реализовал интерфейс `Marshaler`, и это свойство будет проверено во время компиляции. Если интерфейс `json.Marshaler` изменится, этот пакет больше не будет компилироваться, и это даст нам сигнал о том, что его нужно обновить.
Появление пустого идентификатора в этой конструкции указывает на то, что объявление существует только для проверки типов, а не для создания переменной. Однако не стоит делать так для каждого типа, который удовлетворяет интерфейсу. По соглашению, такие объявления используются только тогда, когда в коде нет статических преобразований, что является редким случаем.
Полный пример для демонстрации:

```
package main

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

// 1. Проверка соответствия интерфейсу на этапе компиляции
// Гарантируем, что наш тип CustomData реализует интерфейс json.Marshaler
```

```go
var _ json.Marshaler = (*CustomData)(nil)

// Тип, который будет реализовывать интерфейс json.Marshaler для кастомной маршализации в JSON
type CustomData struct {
    Name string
    Age  int
}

// Реализуем метод MarshalJSON для кастомной маршализации в JSON
func (cd *CustomData) MarshalJSON() ([]byte, error) {
    return json.Marshal(map[string]interface{}{
        "Имя":     cd.Name,
        "Возраст": cd.Age,
    })
}

func main() {
    // Создадим экземпляр CustomData
    data := &CustomData{Name: "Иван", Age: 30}

    // 2. Проверка на соответствие интерфейсу json.Marshaler во время выполнения
    if _, ok := interface{}(data).(json.Marshaler); ok {
        fmt.Println("Тип CustomData реализует json.Marshaler")
    }

    // 3. Пример пустого идентификатора
    // Проверим наличие файла и игнорируем сам результат, оставив только проверку ошибки
    if _, err := os.Stat("test.txt"); os.IsNotExist(err) {
        fmt.Println("Файл test.txt не существует")
    }

    // Пример маршализации нашего кастомного типа
    jsonData, err := json.Marshal(data)
    if err != nil {
        log.Fatal("Ошибка при маршализации:", err)
    }
    fmt.Println("Кастомный JSON:", string(jsonData))
}
```

Вывод будет такой:

```
$ go run main.go
Тип CustomData реализует json.Marshaler
Файл test.txt не существует
Кастомный JSON: {"Возраст":30,"Имя":"Иван"}
```

Если мы уберем реализацию интерфейса, то увидим такую ошибку:

`cannot use (*CustomData)(nil) (value of type *CustomData) as json.Marshaler value in variable declaration: *CustomData does not implement json.Marshaler (missing method MarshalJSON)`

Мы ее заметим, так как у нас есть строка:

```go
var _ json.Marshaler = (*CustomData)(nil)
```

Именно в ней и будет ошибка.

Встраивание (Embedding)

Go не предоставляет типичное, основанное на типах, понятие наследования, но в нем есть возможность "заимствовать" части реализации, встраивая типы в структуры или интерфейсы.
Встраивание интерфейсов очень простое. Мы уже упоминали интерфейсы `io.Reader` и `io.Writer`; вот их определения:

```go
type Reader interface {
    Read(p []byte) (n int, err error)
}

type Writer interface {
    Write(p []byte) (n int, err error)
}
```

Пакет `io` также экспортирует несколько других интерфейсов, которые определяют объекты, способные реализовать несколько таких методов. Например, существует интерфейс `io.ReadWriter`, который включает методы Read и Write. Мы могли бы явно указать методы, составляющие `io.ReadWriter`, но проще и понятнее встраивать два интерфейса для создания нового, вот так:

```
// ReadWriter — это интерфейс, объединяющий интерфейсы Reader и Writer.
type ReadWriter interface {
    Reader
    Writer
}
```

Это означает именно то, что и кажется: `ReadWriter` может выполнять функции как `Reader`, так и `Writer`; это объединение встроенных интерфейсов. Только интерфейсы могут быть встроены в другие интерфейсы.
Та же основная идея применима к структурам, но с более глубокими последствиями. Пакет `bufio` имеет две структуры: `bufio.Reader` и `bufio.Writer`, каждая из которых, разумеется, реализует соответствующие интерфейсы из пакета `io`. Пакет `bufio` также реализует буферизованный `Reader` / `Writer`, который объединяет `Reader` и `Writer` в одну структуру с помощью встраивания: типы перечислены внутри структуры, но им не присвоены имена полей.

```
// ReadWriter хранит указатели на Reader и Writer.
// Он реализует io.ReadWriter.
type ReadWriter struct {
    *Reader  // *bufio.Reader
    *Writer  // *bufio.Writer
}
```

Встроенные элементы являются указателями на структуры и, конечно, должны быть инициализированы для указания на действительные структуры, прежде чем их можно будет использовать. Структуру `ReadWriter` можно было бы написать как:

```
type ReadWriter struct {
    reader *Reader
    writer *Writer
}
```

Но тогда, чтобы "продвинуть" методы полей и удовлетворить интерфейсам `io`, нам пришлось бы предоставить методы пересылки, например:

```
func (rw *ReadWriter) Read(p []byte) (n int, err error) {
    return rw.reader.Read(p)
}
```

Встраивая структуры напрямую, мы избегаем этой работы. Методы встроенных типов автоматически становятся доступными, что означает, что `bufio.ReadWriter` не только имеет методы `bufio.Reader` и `bufio.Writer`, но и удовлетворяет всем трем интерфейсам: `io.Reader`, `io.Writer` и `io.ReadWriter`.
Есть важное отличие встраивания от наследования. Когда мы встраиваем тип, его методы становятся методами внешнего типа, но при их вызове получателем метода является внутренний тип, а не внешний. В нашем примере, когда вызывается метод `Read` у `bufio.ReadWriter`, это имеет такой же эффект, как если бы мы написали пересылающий метод: получателем будет поле `reader` структуры `ReadWriter`, а не сам `ReadWriter`.
Встраивание также может быть удобным инструментом. Этот пример показывает встроенное поле рядом с обычным именованным полем:

```
type Job struct {
    Command string
    *log.Logger
}
```

Теперь тип `Job` имеет методы `Print`, `Printf`, `Println` и другие методы `*log.Logger`. **Конечно, мы могли бы дать `Logger` имя поля, но это не обязательно**. И теперь, после инициализации, мы можем вести логирование с помощью `Job`:

```
job.Println("начинаю выполнение...")
```

`Logger` — это обычное поле структуры `Job`, поэтому мы можем инициализировать его обычным образом в конструкторе для `Job`, например так:

```
func NewJob(command string, logger *log.Logger) *Job {
    return &Job{command, logger}
```

```
}
```

или с помощью составного литерала:

```
job := &Job{command, log.New(os.Stderr, "Job: ", log.Ldate)}
```

Если нам нужно напрямую обратиться к встроенному полю, имя типа этого поля, без квалификатора пакета, служит именем поля, как это было в методе `Read` нашей структуры `ReadWriter`. Здесь, если нам нужно получить доступ к * `log.Logger` переменной `job`, мы бы написали `job.Logger`, что было бы полезно, если бы мы хотели изменить методы `Logger`.

```
func (job *Job) Printf(format string, args ...interface{}) {
    job.Logger.Printf("%q: %s", job.Command, fmt.Sprintf(format, args...))
}
```

Встраивание типов может привести к конфликтам имен, но правила их разрешения просты. Во-первых, поле или метод `X` скрывает любой другой элемент `X`, находящийся глубже в структуре. Если бы в `log.Logger` было поле или метод с именем `Command`, поле `Command` в `Job` имело бы приоритет.
Во-вторых, если одно и то же имя появляется на одном уровне вложенности, это обычно ошибка; было бы ошибкой встраивать `log.Logger`, если структура `Job` содержала другое поле или метод с именем `Logger`. Однако, если дублируемое имя никогда не упоминается в программе за пределами определения типа, это допустимо. Это условие обеспечивает некоторую защиту от изменений типов, встроенных из внешних источников.

Конкурентность

Обмен через коммуникации

Конкурентное программирование — это большая тема, и здесь есть место только для некоторых особенностей, специфичных для Go.
Конкурентное программирование во многих средах усложняется из-за тонкостей, необходимых для правильного доступа к общим переменным. **Go поощряет другой подход, при котором общие значения передаются через каналы и, фактически, никогда не разделяются между отдельными потоками выполнения**. **В любой момент времени только одна горутина имеет доступ к значению. По своему замыслу исключается возможность гонок данных. **Чтобы поощрить такое мышление, мы свели его к лозунгу:
Не обменивайтесь данными через общую память; вместо этого обменивайтесь памятью через коммуникации.
Этот подход можно довести до крайности. Например, для подсчета ссылок может быть лучше использовать мьютекс вокруг целочисленной переменной. Но как высокоуровневый подход, использование каналов для управления доступом упрощает написание понятных и правильных программ.
Один из способов подумать об этой модели — представить типичную однопоточную программу, работающую на одном процессоре. Ей не нужны примитивы синхронизации. Теперь запустите еще один такой экземпляр; ему тоже не нужна синхронизация. Теперь позвольте этим двум программам общаться; если коммуникация является синхронизатором, других примитивов синхронизации не требуется. Например, каналы Unix идеально вписываются в эту модель. Хотя подход Go к конкурентности берет начало в модели коммуницирующих последовательных процессов (Communicating Sequential Processes, CSP), он также может рассматриваться как типобезопасное обобщение каналов Unix.
Основная идея конкурентного программирования в Go заключается в том, что вместо использования общей памяти для обмена данными между потоками (горутинами), Go поощряет передачу данных через каналы. Это предотвращает типичные проблемы с гонками данных и синхронизацией, которые возникают, когда несколько потоков пытаются одновременно изменить одну и ту же переменную.
Главные принципы:

1. **"Не делитесь памятью напрямую".** Это значит, что потоки (или горутины) не должны одновременно иметь доступ к одним и тем же данным напрямую, потому что это требует сложных механизмов синхронизации, таких как мьютексы или блокировки.
2. **"Делитесь памятью через коммуникацийй".** Вместо этого, данные передаются через каналы, где одна горутина передает данные другой. Это гарантирует, что в любой момент времени доступ к данным есть только у одного потока.

Горутины

Их называют горутинами, потому что существующие термины — потоки, корутины, процессы и т. д. — создают неверные ассоциации. Модель горутин проста: **это функция, выполняющаяся одновременно с другими горутинами в одном и том же адресном пространстве.** Они легковесны и требуют лишь небольших затрат, в основном на выделение

места под стек. Размеры стеков начинаются с малых значений, что делает их дешевыми (в ресурсном плане), и они увеличиваются по мере необходимости, выделяя (и освобождая) память из кучи.
Горутины мультиплексируются на несколько потоков операционной системы, так что если одна горутина заблокируется, например, ожидая ввода-вывода, другие продолжают работать. Их дизайн скрывает многие сложности, связанные с созданием и управлением потоками.
Добавьте ключевое слово `go` перед вызовом функции или метода, чтобы выполнить вызов в новой горутине. Когда выполнение завершится, горутина завершит работу, не подавая сигналов. (*Эффект аналогичен использованию символа `&` в оболочке Unix для выполнения команды в фоновом режиме.*)

```
go list.Sort()  // выполняем list.Sort параллельно; не ожидаем завершения.
```

Функциональные литералы могут быть полезны при вызове горутин.

```
func Announce(message string, delay time.Duration) {
    go func() {
        time.Sleep(delay)
        fmt.Println(message)
    }()  // Обратите внимание на круглые скобки - нужно вызвать функцию.
}
```

В Go функциональные литералы являются замыканиями: реализация гарантирует, что переменные, на которые ссылается функция, будут существовать до тех пор, пока они активны.
Эти примеры не слишком практичны, потому что функции не могут сигнализировать о завершении. Для этого нужны каналы.

Каналы

Как и карты (**maps, отображения, мапы**), каналы создаются с помощью `make`, **и полученное значение действует как ссылка на базовую структуру данных.** Если передан необязательный целочисленный параметр, он задает размер буфера для канала. По умолчанию размер равен нулю, что делает канал неблокирующим или синхронным.

```
ci := make(chan int)            // неблокирующий канал целых чисел
cj := make(chan int, 0)         // неблокирующий канал целых чисел
cs := make(chan *os.File, 100)  // буферизованный канал указателей на файлы
```

Небуферизованные каналы совмещают коммуникацию — обмен значениями — с синхронизацией, гарантируя, что две вычислительные задачи (горутины) находятся в известном состоянии.
Существует множество полезных идиом для работы с каналами. Вот одна из них. В предыдущем разделе мы запустили сортировку в фоновом режиме. Канал может позволить горутине, которая запустила сортировку, дождаться её завершения.

```
c := make(chan int)  // Создаем канал.
// Запускаем сортировку в горутине; когда она завершится, подаем сигнал через канал.
go func() {
    list.Sort()
    c <- 1  // Отправляем сигнал; значение не имеет значения.
}()
doSomethingForAWhile()
<-c   // Ждем завершения сортировки; отбросить полученное значение.
```

Получатели всегда блокируются до тех пор, пока не появятся данные для получения. Если канал небуферизованный, отправитель блокируется, пока получатель не получит значение. Если канал буферизованный, отправитель блокируется только до тех пор, пока значение не будет скопировано в буфер; если буфер заполнен, это означает ожидание, пока какой-то получатель не извлечет значение.
Буферизованный канал можно использовать как семафор, например, чтобы ограничить пропускную способность. В этом примере входящие запросы передаются в функцию `handle`, которая отправляет значение в канал, обрабатывает запрос, а затем получает значение из канала, чтобы подготовить "семафор" для следующего потребителя. Емкость буфера канала ограничивает количество одновременных вызовов функции `process`.

```
var sem = make(chan int, MaxOutstanding)

func handle(r *Request) {
    sem <- 1    // Ожидание освобождения активной очереди.
    process(r)  // Может занять много времени.
    <-sem       // Завершено; разрешить выполнение следующего запроса.
}
```

```
func Serve(queue chan *Request) {
    for {
        req := <-queue
        go handle(req)  // Не ожидаем завершения функции handle.
    }
}
```

Как только количество активных обработчиков достигает значения `MaxOutstanding`, любые дополнительные вызовы блокируются, пытаясь отправить значение в заполненный буфер канала, пока один из существующих обработчиков не завершит работу и не извлечет значение из буфера.
Однако этот дизайн имеет проблему: `Serve` создает новую горутину для каждого входящего запроса, хотя в любой момент времени может выполняться только `MaxOutstanding` запросов. В результате программа может потреблять неограниченные ресурсы, если запросы поступают слишком быстро. Мы можем исправить этот недостаток, изменив `Serve`, чтобы ограничить создание горутин:

```
func Serve(queue chan *Request) {
    for req := range queue {
        sem <- 1
        go func() {
            process(req)
            <-sem
        }()
    }
}
```

(Обратите внимание, что в версиях Go до 1.22 этот код содержит ошибку: переменная цикла используется всеми горутинами. Подробнее об этом можно узнать на Go wiki.)
Другой подход, который хорошо управляет ресурсами, заключается в том, чтобы запустить фиксированное количество горутин `handle`, каждая из которых читает из канала запросов. Количество горутин ограничивает количество одновременных вызовов функции `process`. Эта функция `Serve` также принимает канал, через который ей сообщат о завершении работы; после запуска горутин она блокируется, ожидая сигнала на этом канале.

```
func handle(queue chan *Request) {
    for r := range queue {
        process(r)
    }
}

func Serve(clientRequests chan *Request, quit chan bool) {
    // Запуск обработчиков
    for i := 0; i < MaxOutstanding; i++ {
        go handle(clientRequests)
    }
    <-quit  // Ожидание сигнала для завершения работы.
}
```

Каналы каналов

Одним из важнейших свойств Go является то, что **канал — это полноценное значение, которое можно выделить и передавать, как и любое другое.** Часто это свойство используется для реализации безопасной параллельной демультиплексации.
В примере из предыдущего раздела `handle` был идеализированным обработчиком запроса, но мы не определили тип, который он обрабатывал. Если этот тип включает в себя канал для отправки ответа, каждый клиент может предоставить свой собственный путь для получения ответа. Вот схематическое определение типа `Request`.

```
type Request struct {
    args        []int
    f           func([]int) int
    resultChan  chan int
}
```

Клиент предоставляет функцию и её аргументы, а также канал внутри объекта запроса, по которому будет получен ответ.

```
func sum(a []int) (s int) {
    for _, v := range a {
        s += v
```

```
    }
    return
}

request := &Request{[]int{3, 4, 5}, sum, make(chan int)}
// Отправляем запрос
clientRequests <- request
// Ожидаем ответа.
fmt.Printf("ответ: %d\n", <-request.resultChan)
```

На стороне сервера единственное, что изменяется, — это функция обработчика.

```
func handle(queue chan *Request) {
    for req := range queue {
        req.resultChan <- req.f(req.args)
    }
}
```

Здесь очевидно, что нужно многое доработать для создания реальной системы, но этот код — это основа для системы параллельного неблокирующего `RPC` с ограничением скорости, причём без единого мьютекса.

Параллелизация

Ещё одно применение этих идей — параллелизация вычислений на нескольких ядрах процессора. Если вычисление можно разбить на отдельные части, которые могут выполняться независимо, его можно параллелизировать, используя канал для сигнализации о завершении каждой части.
Допустим, у нас есть ресурсоёмкая операция, которую нужно выполнить для вектора элементов, и значение этой операции для каждого элемента не зависит от других, как в этом идеализированном примере.

```
type Vector []float64

// Применяем операцию к v[i], v[i+1] ... вплоть до v[n-1].
func (v Vector) DoSome(i, n int, u Vector, c chan int) {
    for ; i < n; i++ {
        v[i] += u.Op(v[i])
    }
    c <- 1    // сигнализируем, что эта часть завершена
}
```

Мы запускаем части вычислений независимо в цикле, по одному на каждое ядро процессора. Они могут завершаться в любом порядке, и это не имеет значения; мы просто считаем сигналы завершения, считывая данные из канала после запуска всех горутин.

```
const numCPU = 4 // количество ядер процессора

func (v Vector) DoAll(u Vector) {
    c := make(chan int, numCPU)  // Буферизация не обязательна, но логична.
    for i := 0; i < numCPU; i++ {
        go v.DoSome(i*len(v)/numCPU, (i+1)*len(v)/numCPU, u, c)
    }
    // Считываем данные из канала.
    for i := 0; i < numCPU; i++ {
        <-c    // ждём завершения одной задачи
    }
    // Всё завершено.
}
```

Вместо того чтобы создавать константу для `numCPU`, мы можем запросить у среды выполнения подходящее значение. Функция `runtime.NumCPU` возвращает количество физических ядер процессора на машине, так что мы могли бы написать:

```
var numCPU = runtime.NumCPU()
```

Также существует функция `runtime.GOMAXPROCS`, которая сообщает (или задаёт) количество ядер, которое может одновременно использовать программа на Go. По умолчанию она принимает значение `runtime.NumCPU`, но его можно изменить, установив одноимённую переменную среды или вызвав функцию с положительным числом. Если вызвать её с нулём, она просто вернёт текущее значение. Поэтому, если мы хотим учитывать предпочтения пользователя по использованию ресурсов, мы можем написать:

```
var numCPU = runtime.GOMAXPROCS(0)
```

Не путайте понятия конкурентности — структурирования программы как набора независимо выполняемых компонентов — и параллелизма — выполнения вычислений параллельно для повышения эффективности на нескольких процессорах. Хотя особенности конкурентности Go могут упростить структуру некоторых задач как параллельных вычислений, **Go является языком для конкурентного программирования, а не параллельного**, и не все задачи параллелизации соответствуют модели Go. Для обсуждения этого различия посмотрите доклад, упомянутый в блоге.

Протекающий буфер

Инструменты конкурентного программирования могут упростить даже идеи, не связанные с конкурентностью. Вот пример, взятый из пакета `RPC`. **Горутина клиента зацикливается, получая данные из какого-то источника, возможно, сети.** Чтобы избежать постоянного выделения и освобождения буферов, используется список свободных буферов, представленный буферизованным каналом. **Если канал пуст, создается новый буфер. Когда буфер для сообщения готов, он отправляется на сервер через `serverChan`.**

```
var freeList = make(chan *Buffer, 100)
var serverChan = make(chan *Buffer)

func client() {
    for {
        var b *Buffer
        // Получить буфер, если он доступен; создать новый, если нет.
        select {
        case b = <-freeList:
            // Получили буфер, ничего больше делать не нужно.
        default:
            // Нет свободных, создаём новый буфер.
            b = new(Buffer)
        }
        load(b)              // Считываем следующее сообщение из сети.
        serverChan <- b      // Отправляем серверу.
    }
}
```

Цикл сервера получает каждое сообщение от клиента, обрабатывает его и возвращает буфер в список свободных буферов.

```
func server() {
    for {
        b := <-serverChan    // Ждём работы.
        process(b)
        // Повторно используем буфер, если есть место.
        select {
        case freeList <- b:
            // Буфер возвращен в список, ничего больше делать не нужно.
        default:
            // Список заполнен, просто продолжаем работу.
        }
    }
}
```

Клиент пытается получить буфер из `freeList`. Если буфер недоступен, создается новый. Отправка на `freeList` возвращает буфер обратно в список, если там есть место; если список полон, буфер просто удаляется, и его забирает сборщик мусора. (Операторы `default` в выражениях `select` выполняются, когда ни один другой случай не готов, что означает, что `select` никогда не блокируется.) **Эта реализация создает протекающий буферный список всего в несколько строк, полагаясь на буферизованный канал и сборщик мусора для управления памятью.**

Ошибки

Библиотеки часто должны возвращать какое-то указание на ошибку вызывающему коду. Как упоминалось ранее, возможность Go возвращать несколько значений упрощает передачу подробного описания ошибки вместе с обычным возвращаемым значением. Хорошим стилем является использование этой функции для предоставления детальной

информации об ошибке. Например, как мы увидим, `os.Open` возвращает не только `nil-указатель` в случае ошибки, но также и значение ошибки, которое описывает, что пошло не так.
По соглашению, ошибки имеют тип `error`, который является простым встроенным интерфейсом.

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

Разработчик библиотеки может свободно реализовать этот интерфейс с использованием более сложной модели, что позволяет не только увидеть саму ошибку, но и предоставить некоторый контекст. Как уже упоминалось, помимо обычного значения `*os.File`, `os.Open` также возвращает значение ошибки. Если файл был успешно открыт, то ошибка будет `nil`, но если возникла проблема, будет возвращена ошибка типа `os.PathError`:

```
// PathError записывает ошибку, операцию и
// путь к файлу, который вызвал ошибку.
type PathError struct {
    Op string    // "open", "unlink" и т.д.
    Path string  // Соответствующий файл.
    Err error    // Ошибка, возвращенная системным вызовом.
}

func (e *PathError) Error() string {
    return e.Op + " " + e.Path + ": " + e.Err.Error()
}
```

Метод `Error` структуры `PathError` генерирует строку такого вида:

```
open /etc/passwx: no such file or directory
```

Такая ошибка, которая включает имя проблемного файла, операцию и системную ошибку, которая была вызвана, полезна даже если она выводится далеко от того места, где была вызвана. Это намного информативнее, чем просто "нет такого файла или директории".
Когда это возможно, строки ошибок должны идентифицировать их источник, например, добавляя префикс, обозначающий операцию или пакет, который сгенерировал ошибку. Например, в пакете `image` строка ошибки для ошибки декодирования из-за неизвестного формата выглядит как "`image: unknown format`".
Вызывающие коды, которым важны точные детали ошибки, могут использовать оператор `type switch` или утверждение типа (`type assertion`), чтобы найти конкретные ошибки и извлечь детали. Для `PathErrors` это может включать проверку внутреннего поля `Err` для выявления исправимых ошибок.

```
for try := 0; try < 2; try++ {
    file, err = os.Create(filename)
    if err == nil {
        return
    }
    if e, ok := err.(*os.PathError); ok && e.Err == syscall.ENOSPC {
        deleteTempFiles()  // Освобождаем место.
        continue
    }
    return
}
```

Второе условие `if` здесь является еще одним утверждением типа. Если оно не сработает, переменная `ok` будет ложной, а `e` будет равна `nil`. Если утверждение типа выполнится, `ok` будет истинным, что означает, что ошибка имеет тип `*os.PathError`, и тогда переменная `e` также будет иметь этот тип, и мы можем изучить ее для получения дополнительной информации об ошибке.

Panic (паника)

Обычный способ сообщить об ошибке вызывающему коду — это вернуть ошибку в виде дополнительного возвращаемого значения. Классическим примером является метод `Read`, который возвращает количество байт и ошибку. **Но что делать, если ошибка критическая и её невозможно исправить?** Иногда программа просто не может продолжить выполнение. Для таких случаев существует встроенная функция `panic`, которая по сути создает ошибку во время выполнения, приводящую к остановке программы (но об этом подробнее в следующем разделе). Эта функция принимает один аргумент произвольного типа (чаще всего строку), который будет выведен, когда программа завершится. Это также способ указать на то, что произошло нечто невозможное, например, выход из бесконечного цикла.

```go
// Игрушечная реализация вычисления кубического корня методом Ньютона.
func CubeRoot(x float64) float64 {
    z := x / 3   // Произвольное начальное значение
    for i := 0; i < 1e6; i++ {
        prevz := z
        z -= (z*z*z - x) / (3 * z * z)
        if veryClose(z, prevz) {
            return z
        }
    }
    // Миллион итераций не привели к сходимости — что-то пошло не так.
    panic(fmt.Sprintf("CubeRoot(%g) did not converge", x))
}
```

Это всего лишь пример, но реальным библиотечным функциям не следует использовать `panic` без крайней необходимости. Если проблему можно как-то скрыть или обойти, лучше продолжить выполнение программы, чем полностью её останавливать. Например, если программа не может корректно инициализировать важную часть системы, использование `panic` может быть оправдано.

```go
var user = os.Getenv("USER")

func init() {
    if user == "" {
        panic("no value for $USER")
    }
}
```

Здесь, если переменная окружения `$USER` не задана, программа вызывает `panic`, так как без этого значения она не может продолжить работу.

Восстановление (Recover)

Когда вызывается функция `panic` (включая неявный вызов для ошибок времени выполнения, таких как выход за пределы индекса массива или неудачная попытка приведения типа), она немедленно останавливает выполнение текущей функции и начинает "развёртывание" стека горутины, выполняя все отложенные функции по пути. Если развёртывание достигает верхушки стека горутины, программа завершится. Однако встроенная функция recover позволяет восстановить управление горутиной и возобновить нормальное выполнение.
Вызов `recover` останавливает развёртывание стека и возвращает аргумент, переданный в `panic`. Поскольку во время развёртывания выполняются только отложенные функции, `recover` полезен только внутри таких функций.
Одним из применений `recover` является остановка падающей горутины внутри сервера без нарушения работы других горутин.

```go
func server(workChan <-chan *Work) {
    for work := range workChan {
        go safelyDo(work)
    }
}

func safelyDo(work *Work) {
    defer func() {
        if err := recover(); err != nil {
            log.Println("work failed:", err)
        }
    }()
    do(work)
}
```

В этом примере, если `do(work)` вызывает `panic`, результат будет залогирован, а горутина завершится корректно, не нарушая работу других горутин. В отложенной функции больше ничего не требуется, так как вызов `recover` полностью обрабатывает ситуацию.
Так как `recover` всегда возвращает `nil`, если он вызван не из отложенной функции, отложенный код может вызывать библиотечные функции, которые сами используют `panic` и `recover`, без боязни сбоя. Например, в отложенной функции `safelyDo` можно сначала вызвать функцию логирования, прежде чем вызвать `recover`, и эта функция выполнится без проблем, несмотря на состояние паники.

Используя этот паттерн, функция `do` (и всё, что она вызывает) может завершить любую неудачную ситуацию вызовом `panic`. Мы можем применить этот принцип для упрощения обработки ошибок в сложном программном обеспечении. Рассмотрим идеализированную версию пакета для работы с регулярными выражениями, который сообщает об ошибках разбора, вызывая `panic` с локальным типом ошибки.

```
// Error — это тип ошибки разбора; он реализует интерфейс error.
type Error string

func (e Error) Error() string {
    return string(e)
}

// error — это метод *Regexp, который сообщает об ошибках разбора,
// вызывая панику с ошибкой типа Error.
func (regexp *Regexp) error(err string) {
    panic(Error(err))
}

// Compile возвращает разобранное представление регулярного выражения.
func Compile(str string) (regexp *Regexp, err error) {
    regexp = new(Regexp)
    // doParse вызовет панику, если возникнет ошибка разбора.
    defer func() {
        if e := recover(); e != nil {
            regexp = nil    // Очищаем возвращаемое значение.
            err = e.(Error) // Перепаника, если это не ошибка разбора.
        }
    }()
    return regexp.doParse(str), nil
}
```

Если `doParse` вызывает панику, блок восстановления изменит возвращаемое значение на `nil` (отложенные функции могут изменять именованные возвращаемые значения). Затем он проверит, была ли проблема ошибкой разбора, утверждая, что она имеет локальный тип `Error`. Если это не так, утверждение типа завершится ошибкой, что вызовет дальнейшее развёртывание стека, как будто ничего не было прервано. Это означает, что если произойдёт что-то неожиданное, например, выход за пределы индекса, программа завершится, несмотря на использование `panic` и `recover` для обработки ошибок разбора.
Таким образом, с реализацией обработки ошибок метод `error` позволяет легко сообщать об ошибках разбора без необходимости вручную обрабатывать развёртывание стека:

```
if pos == 0 {
    re.error("'*' illegal at start of expression")
}
```

Этот шаблон полезен, но его следует использовать только внутри одного пакета. Функция разбора превращает внутренние вызовы panic в значения ошибок, не выставляя панику внешним клиентам — это хорошее правило.
Кстати, использование шаблона перепаники изменяет значение паники в случае реальной ошибки. Однако в отчёте о сбое будут показаны как исходная, так и новая ошибка, так что основная причина проблемы останется видимой. Этот подход, как правило, достаточен, но если нужно показать только исходное значение ошибки, можно написать дополнительный код для фильтрации неожиданных проблем и перепаники с исходной ошибкой.

Веб-сервер

Закончим полным примером программы на Go — веб-сервером. Этот сервер на самом деле является своего рода «веб-пересервером». Google предоставляет сервис на chart.apis.google.com, который автоматически форматирует данные в графики и диаграммы. Однако его трудно использовать интерактивно, потому что необходимо вставлять данные в URL в виде запроса. Программа, представленная здесь, предлагает более удобный интерфейс для одного типа данных: получив короткий текст, она использует сервер диаграмм для создания QR-кода — матрицы клеток, которые кодируют текст. Этот образ можно захватить камерой мобильного телефона, и он будет интерпретирован как, например, URL, что избавляет вас от необходимости вручную вводить URL на крошечной клавиатуре телефона.
Вот полный код программы. Объяснение следует далее.

```
package main
```

```
import (
    "flag"
    "html/template"
    "log"
    "net/http"
)

var addr = flag.String("addr", ":1718", "http service address") // Q=17, R=18

var templ = template.Must(template.New("qr").Parse(templateStr))

func main() {
    flag.Parse()
    http.Handle("/", http.HandlerFunc(QR))
    err := http.ListenAndServe(*addr, nil)
    if err != nil {
        log.Fatal("ListenAndServe:", err)
    }
}

func QR(w http.ResponseWriter, req *http.Request) {
    templ.Execute(w, req.FormValue("s"))
}

const templateStr = `
<html>
<head>
<title>QR Link Generator</title>
</head>
<body>
{{if .}}
<img src="http://chart.apis.google.com/chart?chs=300x300&cht=qr&choe=UTF-8&chl={{.}}" />
<br>
{{.}}
<br>
<br>
{{end}}
<form action="/" name=f method="GET">
    <input maxLength=1024 size=70 name=s value="" title="Text to QR Encode">
    <input type=submit value="Show QR" name=qr>
</form>
</body>
</html>
`
```

Части программы до функции `main` должны быть легко понятны. Один флаг задает порт по умолчанию для **HTTP-сервера**. Переменная шаблона `templ` — это место, где начинается «магия». Она строит **HTML-шаблон**, который будет выполняться сервером для отображения страницы; подробнее об этом чуть позже.

Функция `main` разбирает флаги и, используя механизм, о котором мы говорили выше, связывает функцию `QR` с корневым путем сервера. Затем вызывается `http.ListenAndServe` для запуска сервера; он блокируется, пока сервер работает.

Функция `QR` просто получает запрос, который содержит данные формы, и выполняет шаблон на основе значения формы с именем `s`.

Пакет `html/template` мощный; эта программа лишь слегка демонстрирует его возможности. По сути, он переписывает часть текста `HTML` на лету, подставляя элементы, полученные из данных, переданных в `templ.Execute`, в данном случае это значение формы. Внутри текста шаблона (переменная `templateStr`), части, заключенные в двойные фигурные скобки, обозначают действия шаблона. Участок от `{{if .}}` до `{{end}}` выполняется только в том случае, если текущее значение данных, называемое `.` **(точка)**, непусто. То есть, если строка пуста, эта часть шаблона будет скрыта.

Два фрагмента `{{.}}` означают вывод данных, переданных в шаблон — строки запроса — на веб-странице. Пакет шаблонов автоматически обеспечивает правильное экранирование, чтобы текст был безопасен для отображения.

Остальная часть строки шаблона — это просто `HTML` для отображения страницы при её загрузке. Если это объяснение показалось слишком быстрым, обратитесь к документации по пакету шаблонов для более детального обсуждения.

Вот и всё: полезный веб-сервер в нескольких строках кода плюс немного `HTML`, основанного на данных. Go достаточно мощен, чтобы реализовать многое в нескольких строках кода.

API не доступен более, поэтому QR не будет генерироваться.