



Go语言编程技巧

<https://github.com/smallnest/gotips>

这是 Phuong Le 在X上发布的一系列的技巧。 Phuong Le也将推文整理到了一个github仓库中[go-practical-tips](#)

征得作者同意， 翻译成了中文。

本书是基于Phuong Le的推文进行翻译的， 同时也新建一个说明， 按照作者github项目上的划分整理各个tip。

感谢以下网友共同进行了翻译， 翻译进展迅速。

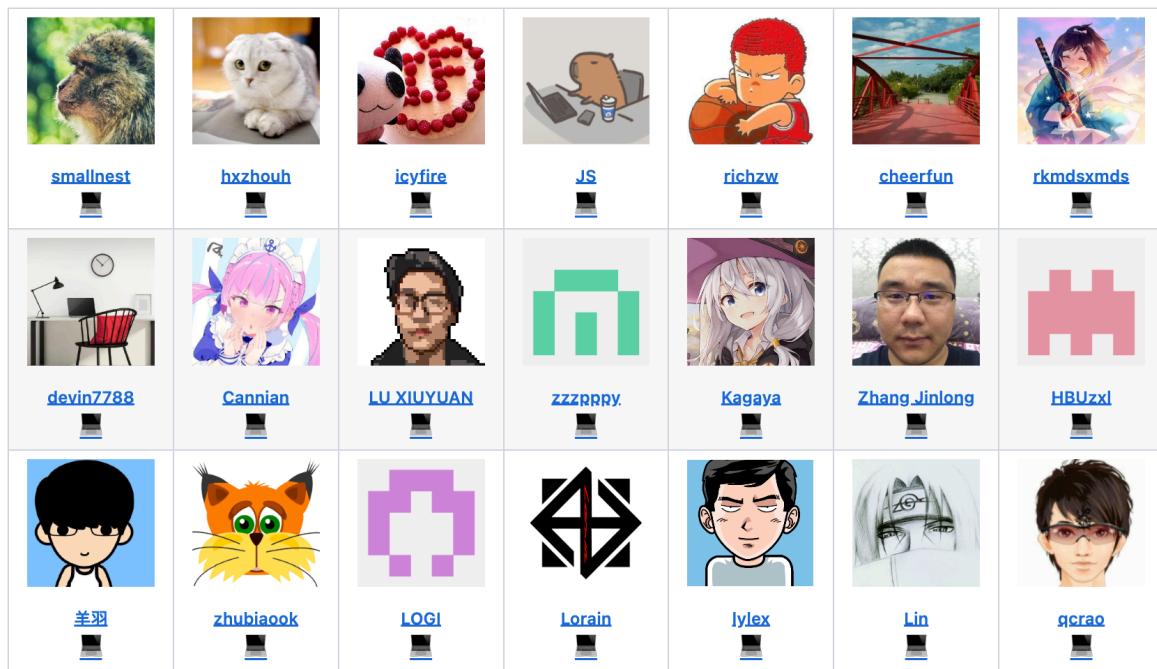
正如作者所说， 有些tip可能有一些错误：

Some of the tips were awkward, with typos and naive explanations.

我们翻译的时候也发现了作者的一些手误， 或者错误， 及时做了注解和纠正。难免还有一些问题， 欢迎大家到github提issue或者提Pull request.

翻译网站：[gotips](#)。

贡献者 ✨



Tip #1 一行代码测量函数的执行时间

原始链接: [Golang #1: Measure the execution time of a function in just one line of code.](#)

```
func main() {
    defer TrackTime(time.Now()) // <--- THIS

    time.Sleep(500 * time.Millisecond)
}

func TrackTime(pre time.Time) time.Duration {
    elapsed := time.Since(pre)
    fmt.Println("elapsed:", elapsed)

    return elapsed
}

// elapsed: 501.11125ms
```

Tip #2 多阶段 defer

原始链接: [Golang Tips #2: Multistage defer](#)

通过简单的'defer'关键字, 你可以借助一个小技巧实现在另一个函数的开头和结尾处执行一个函数。下面的图片展示了这一实现方式。



```
func main() {
    defer MultistageDefer()() // <-----
    fmt.Println("Main function called")
}

func MultistageDefer() func() {
    fmt.Println("Run initialization")

    return func() {
        fmt.Println("Run cleanup")
    }
}

// Output:
// Run initialization
// Main function called
// Run cleanup
```

Tip #3 预分配切片以提高性能

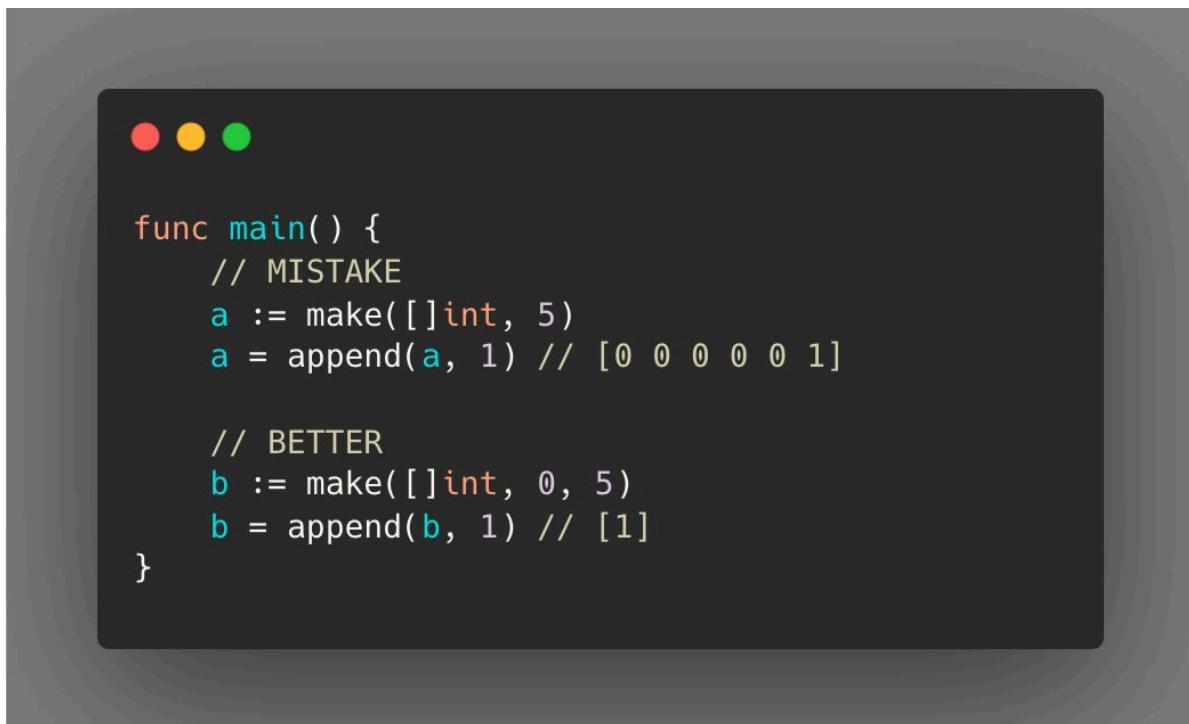
原始链接: [Golang Tip #3: Pre-allocate slices for performance](#)

为什么注重性能? 请参考以下回答。

过去, 我曾使用 `make(a, 10)` 预分配数组空间, 但是,

我经常习惯性地误用 `append()` 方法, 导致数组中出现了许多前导零 (参见下图)

为了避免这种情况, 我改用了一种更有效率的预分配方法: `make(a, 0, 10)`。



The image shows a dark-themed terminal window on a Mac OS X desktop. In the top-left corner, there are the red, yellow, and green window control buttons. The terminal window contains the following Go code:

```
func main() {
    // MISTAKE
    a := make([]int, 5)
    a = append(a, 1) // [0 0 0 0 1]

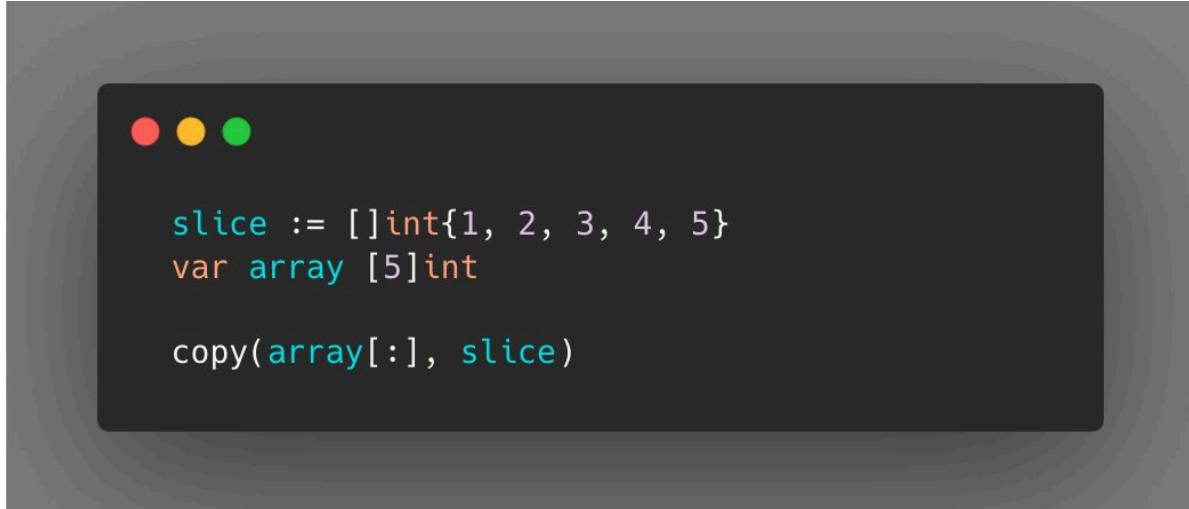
    // BETTER
    b := make([]int, 0, 5)
    b = append(b, 1) // [1]
}
```

Tip #3 将数组解析为切片

原始链接: [Golang Tip #4: Parse an Array into a Slice.](#)

译者注: 标题其实写反了, 应该是“把切片转换成数组”

你可能想到使用copy()方法, 对吧?



但是这是多余的。

如果你的项目已经更新到Go 1.20版本, 你可以更轻松地解析它, 类似于处理其他类型的解析 (例如int转int32)。

```
// Go 1.20

a := []int{0, 1, 2, 3, 4, 5}
b := [3]int(a[0:3])

fmt.Println(b) // [0 1 2]
```

但是，如果你还在使用旧版本，Go 1.17仍然为你提供了一行代码解决方案。

```
// Go 1.17

a := []int{0, 1, 2, 3, 4, 5}
b := *[3]int(a[0:3])

fmt.Println(b) // [0 1 2]
```

个人而言，我不常用这个方法，但它确实是一个值得了解的实用技巧。

Tip #5 方法链

原始链接: [Method Chaining](#)

在为一个类型定义接收器方法时, 返回其自身值。

```
func (p *Person) AddAge() *Person {
    p.Age++
    return p
}

func (p *Person) Rename(name string) *Person {
    p.Name = name
    return p
}
```

这种做法允许你在单个、流畅的处理序列中调用多个方法, 即所谓的“方法链”, 从而实现代码更简洁、更高效。

```
// NOT BAD
p := Person{Name: "Aiden", Age: 30}
p.AddAge()
p.Rename("Aiden 2")

// BETTER
p := Person{Name: "Aiden", Age: 30}.AddAge().Rename("Aiden 2")

// STILL BETTER
p := Person{Name: "Aiden", Age: 30}.
    AddAge().
    Rename("Aiden 2")
```

Tip #6 下划线导入

原始链接: [Golang Tip #6: Underscore Import](#)

有时候, 你会在很多库里看到 `import` 和下划线 (`_`) 一起使用的情况, 类似这样:

```
import (
    _ "google.golang.org/genproto/googleapis/api/annotations"
)
```

这样的作用是什么呢?

它会在不创建那个包的引用的情况下, 执行那个包里的初始化代码 (`init()` 函数)。

例如, 在 `underscore` 包里, 我写了个 `init` 函数:

```
package underscore

func init() {
    fmt.Println("init called from underscore package")
}
```

然后在 `main()` 里, 使用下划线导入, 我甚至什么都没做, 它依然会打印出来:

```
● ● ●

package main

import (
    _ "lab/underscore"
)

func main() {}

// Output: init called from underscore package
```

Tip #7 作者已删除了本tip

Tip #8 包裹错误

原始链接: [Golang Tip #8: Wrapping Errors](#)

通常, 我们会使用 `fmt.Errorf` 和 `%w` 把一个错误包裹到另外一个错误里, 像这样:

```
err := errors.New("error from Func1")

return fmt.Errorf("error from Func2: %w",
err)
```

但是在Go 1.20, 我们有一个更直接和友好的方法去包裹错误, 那就是使用 `errors.Join()`:

```
func Func2() error {
    err := Func1()
    if err != nil {
        return errors.Join(err, errors.New("error from Func2"))
    }

    return nil
}
```

Tip #9 编译时接口检查

原始链接: [Golang Tip #9: Compile-Time Interface Verification](#)

假设有一个 `Buffer` 接口, 它具有一个 `Write()` 方法, 然后 `StringBuffer` 这个结构体实现了这个接口。

如果你不小心打错字, 例如把 `Write()` 写成了 `Writeeee()` :

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

type StringBuffer struct{}`

func (s *StringBuffer) Writeeee(p []byte) (n int, err error) {
```

那么直到运行时才会抛出错误。使用了下面这个技巧后, 编译时就会报错:

```
var _ Buffer = (*StringBuffer)(nil)

// cannot use (*StringBuffer)(nil) (value of type
// *StringBuffer)
// as Buffer value in variable declaration: *StringBuffer
// does not implement Buffer (missing method Write)
```

Tip #10 避免裸露参数

原始链接: [Golang #1: Measure the execution time of a function in just one line of code.](#)

这是一个简单易行的技巧, 可以提高函数的可读性, 特别是在你的集成开发环境(IDE)不支持内联提示的情况下。我们可以通过使用结构体来实现这一目标, 需要注意的是, 结构体中的字段将是可选的而非必填项。你对此有何看法?

```
// NOT BAD
printInfo("foo", true, true) // What do "true" values
represent?

// BETTER
printInfo("foo", true /* isLocal */, true /* done */)

// STILL
printInfo("foo",
    true /* isLocal */,
    true /* done */
)
```

Tip #11 数字分隔符

原始链接: [Golang Tip #11: Numeric separators.](#)

这个技巧在处理长数字时很有用, 可以让你的代码更加的可读以及没有那么容易出错。

这样你就不用再眯着眼睛看一串这么长的数字了, 把它分隔开看起来会更加的清晰。

```
○ ○ ○

// NOT BAD
const oneBillion = 1000000000

// BETTER
const oneBillion = 1_000_000_000

// float
const pi = 3.141_592_653_589_793 //  
3.141592653589793
```

Tip #12 使用crypto/rand生成密钥，避免使用math/rand

原始链接：[Golang Tip #12: Avoid using math/rand, use crypto/rand for keys instead.](#)

当你所在的项目需要生成一些密钥用来加密或者创建唯一标识的时候，那密钥的质量和安全性就尤其重要了。

为什么不使用math/rand？

math/rand 这个包生成的是**伪随机数**。

这意味着如果你知道那些数字是怎么生成的（就是知道用于生成随机数序列的种子），那你就能预知到会生成哪些数字。

```
○ ○ ○

import "math/rand"

func Key() string {
    // rand.Seed(time.Now().UnixNano()) // DEPRERCATED
    r := rand.New(rand.NewSource(time.Now().UnixNano()))

    buf := make([]byte, 16)

    for i := range buf {
        buf[i] = byte(r.Intn(256))
    }

    return fmt.Sprintf("%x", buf)
}
```

就算你使用当前的时间（例如 `time.Nanoseconds()`）作为种子，不可预知性（熵）也很低，因为在两次执行之间当前时间并没有太多的变化。

为什么使用crypto/rand?

`crypto/rand` 提供了一个生成密码学安全随机数的方式。

它被设计成无法被预测，使用了你操作系统上提供的更加难以预测的随机数源。

```
○ ○ ○

import "crypto/rand"

func Key() string {
    buf := make([]byte, 16)

    _, err := rand.Read(buf)
    if err != nil {
        panic(err)
    }

    return fmt.Sprintf("%x", buf)
}
```

`crypto/rand` 适用于加密、认证和其他对于安全敏感的操作。

Tip #13 使用空切片还是更好的NIL切片

原始链接: [Golang Tip #13: Empty slice or, even better, NIL SLICE.](#)

在Go里面使用切片的时候, 有两个可以让你得到看起来像空切片的方法:

- 使用var关键字: `var t []int`

这个方法声明了一个 `int` 类型的切片 `t`, 但并没有对它进行初始化。这时候这个切片被认为是 `nil` 的。

这意味着它并没有指向任何的底层数组。它的长度 (`len`) 和容量 (`cap`) 都是0。

- 使用切片字面量: `t := []int{}`

跟使用 `var` 声明的切片不一样, 这个切片不是 `nil` 的。这个切片的指向了一个底层的数组, 但这个数组并没有包含任何的元素。

所以, 哪种方式更惯用呢?

1. nil切片没有分配任何的内存。

`nil`切片只是一个没有指向任何地方的指针, 而空切片 (`[]int{}`) 则分配了很小的内存去指向一个空数组。

大多数情况下, 这种差异是可以忽略的, 但是对于有性能要求的应用来说, 这个差异影响就比较明显了。

2. Go社区更倾向于使用nil切片的方式, 因为这更加符合Go语言简单的哲学以及切片本身的零值。

3. 当然, 也有例外的情况。

例如，在使用JSON的时候，nil切片和空切片的表现是不一样的。

nil切片（`var t []int`）编码成JSON后的值是`null`，而空切片（`t := []int{}`）编码成JSON后的值是一个空的JSON数组（`[]`）。

4. 在设计代码的时候，你应该同等对待非空切片、空切片和nil切片。

如果你比较熟悉Go，你可能已经知道，对nil切片进行`for range`、`len`和`append`等操作是不会引起panic的。

```
// NOT BAD
emptySlice := []int{}
...
for _, value := range emptySlice {}

// BETTER?
var nilSlice []int
...
for _, value := range nilSlice {}
```

Tip #14 错误信息不要大写或者以标点结尾

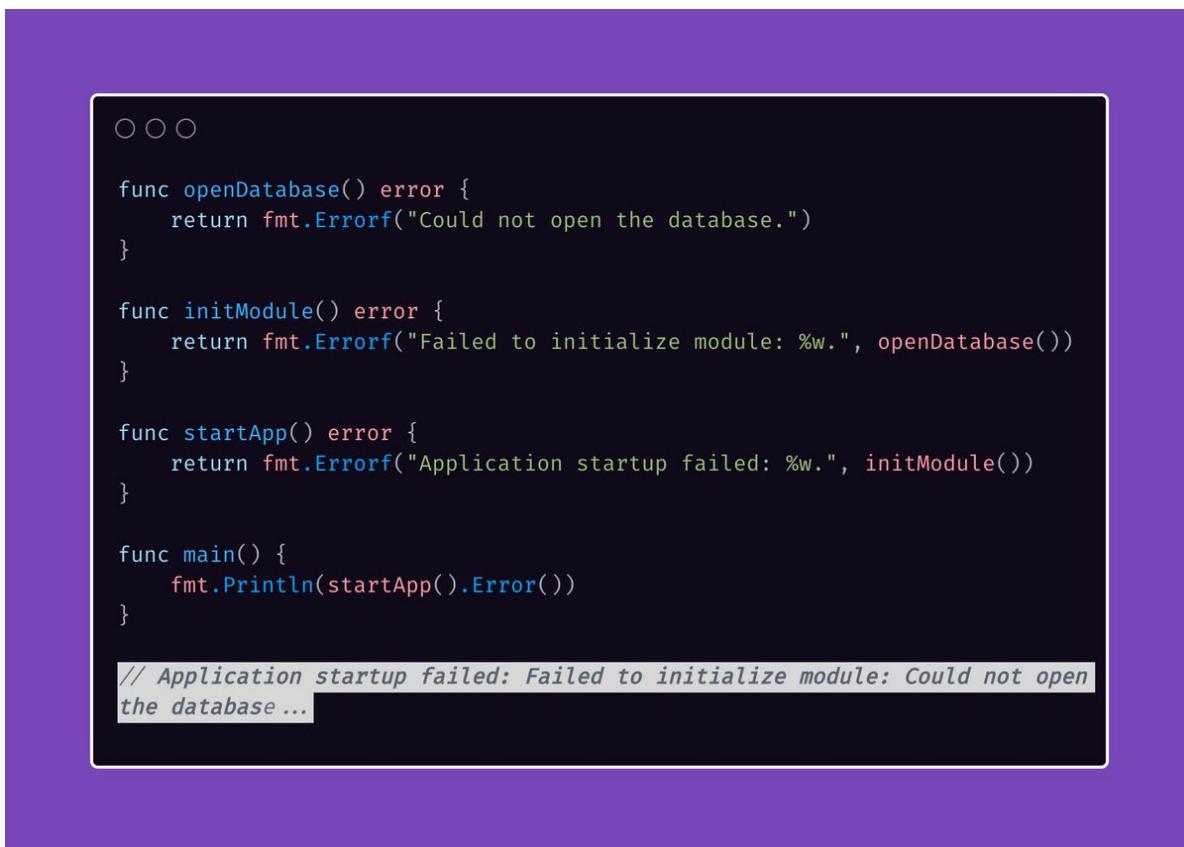
原始链接: [Golang Tip #14: Error messages should not be capitalized or end with punctuation.](#)

乍看起来有点不寻常, 但背后是有实际理由的。

为什么要小写?

错误信息经常会被包裹或者合并到其他错误信息里。

如果一条错误信息以大写字母开头, 那么当它出现在句子中间的时候, 看起来就会很怪或者显得格格不入。



```
○○○

func openDatabase() error {
    return fmt.Errorf("Could not open the database.")
}

func initModule() error {
    return fmt.Errorf("Failed to initialize module: %w.", openDatabase())
}

func startApp() error {
    return fmt.Errorf("Application startup failed: %w.", initModule())
}

func main() {
    fmt.Println(startApp().Error())
}

// Application startup failed: Failed to initialize module: Could not open
// the database ...
```

而以小写字母开头会有让它们融合得更加自然。

// application startup failed: failed to initialize module: could not open the database

还有个需要关注的点就是关于在消息结尾出现的“...”。

这意味着，任何跟在格式化错误字符串里 %w 后面的文本，都会被添加到整个消息的结尾。

为什么不要标点？

这是为了确保当一个消息被追加到另外一个消息后，看起来更像连贯的句子，而不是一堆杂乱的短句。

Tip #15 什么时候使用空白导入和点导入？

原始链接: [Golang Tip #15: When to use Dot\(.\) Import and Blank\(_\) Import?](#)

空白导入 (`import _ "package"`)

当你使用“空白导入”时，你引入了一个包，但并不是为了直接访问它的内容（如函数或变量），而是为了它的副作用。

那么，什么是副作用呢？

副作用是指一个包在被导入时可能执行的任何操作，如初始化、注册、设置环境等。

这通常发生在包的 `init()` 函数中，该函数在包被导入时自动运行。

```
// ----- logger
package logger

import "fmt"

func init() {
    fmt.Println("Logger package initialized")
}

// ----- main
package main

import _ "path/to/logger"

func main() {}

// RESULT: "Logger package initialized"
```

即使 main 函数是空的，当你运行上面的代码时，logger 包的 init() 函数会在导入时运行，向控制台打印 “Logger package initialized”。

使用时机是什么？

主要规则是：

- 通常在 main 包中使用。
- 在那些需要引入副作用才能正确运行的测试中使用。

一个常见的例子是在使用 database/sql 包的程序中导入数据库驱动包。

数据库驱动包被导入是因为其副作用（例如，将自己注册为 database/sql 的驱动）。

点导入 (`import . "package"`)

使用“点导入”有点特别。

这意味着你可以直接使用那些导出的项，就像它们是在当前包中定义的一样。

```
import (
    "fmt"
    . "math"
)

func main() {
    fmt.Println(Abs(-5))
    fmt.Println(Pi)
}

// 5
// 3.14159265358979
```

看，我们不需要指定 `math.Abs` 或 `math.Pi`。

那么，何时使用它？

这种形式在测试中特别有用。

尤其是在处理难以轻易解决的循环依赖时。

给你举个例子。

想象你有两个包：

- `mypackage`: 这是你的 `main` 包, 包含你正在测试的功能。
- `testhelpers`: 一个提供测试辅助函数的独立包。

其中一个辅助函数需要使用 `mypackage`, 从而创建了从 `testhelpers` 到 `mypackage` 的依赖。

```
package mypackagetest

import (
    . "path/to/mypackage"
    "path/to/testhelpers"
    "testing"
)

func TestMyFunction(t *testing.T) {
    testhelpers.DoSomething(MyPackageFunction())
}
```

现在, 你正在为 `mypackage` 编写测试, 并希望使用 `testhelpers` 中的辅助函数。

但由于 `testhelpers` 已经导入了 `mypackage`, 你不能简单地将 `testhelpers` 导入到你的 `mypackage_test.go` 中, 否则会创建循环依赖。

译者注: 上面隐含了 `mypackage_test.go` 所在的包是 `main` 包。

为了解决这个问题, 测试文件声明自己为 `mypackagetest` 包, 并使用点导入直接访问 `mypackage` 的标识符, 就像它在 `mypackage` 内部定义的一样。

建议谨慎使用这两种包导入方式, 因为它们可能会使代码更难阅读。

Tip #16 不要通过返回 -1 或者 nil 来表示错误

原始链接: [Golang Tip #16: Don't Return -1 or nil to Indicate Error.](#)

在其他语言中, 函数通常通过返回特殊值如-1、null、""等来表示错误或缺失的结果。

这被称为“带内错误” (**in-band errors**)。

```
function OpenFile(filename) {
    if (cannotOpenFile) {
        return null;
    }

    return fileContent;
}
```

使用 in-band 错误的主要问题是, 需要调用者记住每次都要检查返回的特殊值。

但这是...非常容易出错的。

此外, 在 Go 中这种方法其实并不是最好的方法 (甚至不是好的), 因为 Go 可以支持多返回值。

Go的解决方案是: 多返回值

函数可以返回其通常的结果以及额外的值（错误或布尔值），明确表示操作是否成功。

这可以使得代码更加清晰。

```
func OpenFile(filename string) (string, error) {
    if cannotOpenFile {
        return "", fmt.Errorf("cannot open file %q", filename)
    }

    return fileContent, nil // Success
}
```

在不检查表明是否成功的返回值（ok bool）的情况下使用结果会导致编译时错误。

这迫使我们必须明确处理可能的错误：

```
content, err := OpenFile("example.txt")
if err != nil {
    log.Printf("Failed to open file: %v", err)
    return
}

// Safely use the content
processFileContent(content)
```

现在，您的代码便拥有了 3 个优势（您甚至不需要额外关心）：

明确的关注点分离

返回值明确的表示了的哪部分是实际结果，哪部分表示操作的成功或失败。

强制错误处理

Go 编译器要求开发人员处理错误的可能性，从而降低忽略错误的风险（因此，请勿使用“_”来忽略错误）。

提高可读性和可维护性

代码可以明确地解释自身的行为（documents itself）。

Tip #17 理解“尽快返回、尽早返回”，避免代码嵌套

原始链接：[Golang Tip #17: Understanding "Return fast, return early" to avoid nested code.](#)

当你写代码的时候，你会想让它尽可能的清晰易懂。

要做到这点，其中一个方法就是组织你的代码，让它的“快乐路径”（预期的或者正常的执行流程）更加的突出和简单明了。

举一个（潜在的）反例：

```
○ ○ ○

if fileExists(fileName) {
    content, readErr := readFile(fileName)
    if readErr != nil {
        // handle error
    } else {
        // process content
    }
} else {
    // handle file not existing
}
```

所以，指导原则是什么？

很简单：提前处理错误，别让他们碍事。

这意味着当出现一个错误时：

- 立刻处理它。
- 使用 `return`、`break`、`continue` 等等语句停止当前操作的执行。
- 或者如果可以的话，处理错误时让正常的执行流程可以安全的得到处理。

回到最开始的那个例子，更好的方法是：

```
○ ○ ○

if !fileExists(fileName) {
    // handle file not existing
    return
}

content, readErr := readFile(fileName)
if readErr != nil {
    // handle error
    return
}

// process content
```

“如果我的方法返回两个值，例如获取 `user` 返回 `(user, error)`，然后值需要在短期内使用呢？”

就算 `user` 仅仅用在 `else` 的作用域里，我也建议把初始化和错误检测分开。

这样可以避免深层的嵌套以及可以简化错误的处理。

```
○○○

-- BEFORE
if user, err := database.FetchUser(userID); err != nil {
    return err
} else {
    // Process the user...
}

// Continuing without using 'user' ...

return nil

-----
-- AFTER
user, err := database.FetchUser(userID);
if err != nil {
    return err
}

// Process the user...

// Continuing without using 'user' ...

return nil
```

“但如果我只是想在 `else` 的作用域里使用 `user` 呢？”

如果 `user` 的使用严格限制在 `else` 里面，并且不会影响外面的逻辑，那么可能是时候把这部分逻辑封装到一个新的方法里了。

○ ○ ○

```
func DoSomethingWithUser(userID string) error {
    user, err := database.FetchUser(userID);
    if err != nil {
        return err
    }

    // Process the user...
}



---


if err := DoSomethingWithUser(userID); err != nil {
    return err
}

// Continuing without using 'user' ...

return nil
```

现在我们在 `DoSomethingWithUser` 方法上使用了这个原则。

当然，并不存在一个“放之四海而皆准”的解决方案。

Tip #18 在使用者的包中定义接口，而不是提供者的包中定义

原始链接: [Define interfaces in the consumer package, not the producer](https://twitter.com/func25/status/1738890734349201903)

我之前在推特上提到过这个话题
(<https://twitter.com/func25/status/1738890734349201903>)，但它的重要性使
我把它列入了这个tips的列表里。

现在，有3个原则需要记住：

1. 在使用者的包中定义接口，而不是提供者的包中定义

接口应该由使用者（使用这个接口的代码）而不是提供者（实现这些接口的代
码）来定义。

这种方法使得添加一个新的函数实现更容易，不会影响到使用者。

2. 在提供者的包中使用具体类型作为返回值

这很简单，因为我们没有再提供者的包里定义这个接口。

它允许我们在这些类型上添加新方法，而不破坏这个API。

3. 避免过早定义接口

只在有明确使用场景下定义接口，确保它们是必要的且设计得当的。

好了，说够了理论和假设。

你有没有做过类似的事情？

```
package logger

// Logger interface defined in the producer package - not recommended.
type Logger interface {
    Log(message string)
}

type consoleLogger struct{}

func (l consoleLogger) Log(message string) {
    fmt.Println(message)
}

func NewLogger() Logger {
    return consoleLogger{}
}
```

在consoleLogger同一个的包中定义Logger接口。

然后，每当你想使用它时，你在使用者的包中创建接口 (?)。

```
package service

import "logger"

// Service using the Logger interface from the logger
// package.
func PerformLogging(l logger.Logger) {
    l.Log("Performing service operation")
}
```

我这么做好多年了，不使用Logger，而是使用库的接口。

“这么做不好吗？”

可能吧

但是遵循我们原则考虑一下这个方法，让我们修改一下它。

首先，这是我们新的提供者的logger包：

```
package logger

// Return a concrete type rather than an interface.
type ConsoleLogger struct{}

func (l ConsoleLogger) Log(message string) {
    fmt.Println(message)
}

func NewConsoleLogger() ConsoleLogger {
    return ConsoleLogger{}
}
```

1. 我们不再在提供者的包中保留Logger这个接口了。
2. 在创建一个提供者包中的Logger时，我们返回一个具体类型，对吧？
3. 我们避免过早的定义接口，不需要猜测使用者需要什么功能。

现在让我们看看我们的使用者是如何用它的呢

```
package service

// Define the Logger interface in the consumer package
// where it's used.
type Logger interface {
    Log(message string)
}

func PerformLogging(l Logger) {
    l.Log("Performing service operation")
}

// Example of a fake logger for testing, defined in
// the consumer package.
type fakeLogger struct{}

func (f fakeLogger) Log(message string) {}
```

通过在使用接口的地方(在高级模块中)定义接口，可以确保这些模块依赖于抽象接口而不是具体的实现。

啊，这也可以增强模块化，mock测试和设计思维。

Tip #19 除非出于文档说明需要，否则避免使用命名结果

原始链接: [Golang Tip #19: Avoid named results unless necessary for documentation.](#)

注意：个人而言，我总是避免使用命名结果，因为它们会鼓励使用裸露返回语句。

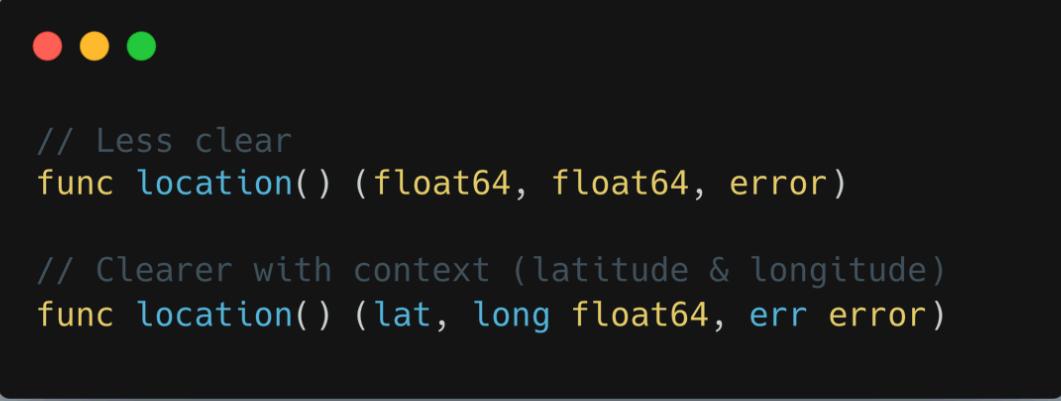
命名结果可以在源代码及生成的文档（如godoc、pkgsite (<http://pkg.go.dev>)）中增强代码可读性。

但了解何时使用它们至关重要，以下是一些关键要点：

必要时进行澄清

建议：

- 如果函数返回多个同类型值，使用命名结果。
- 若其用途不明显，为清晰起见应为其命名。



```
// Less clear
func location() (float64, float64, error)

// Clearer with context (latitude & longitude)
func location() (lat, long float64, err error)
```

不要

- 仅仅为了避免在函数内部声明变量而使用命名结果。
- 为了避免反复书写“return nil, err”，而倾向于简单地使用“return”。

长函数中避免使用裸露(naked)返回

人们常常对裸露返回持谨慎态度，因为它们可能导致代码可读性降低并影响清晰度。

但在短小的函数中，它们完全是可以接受的：

```
func calculateStats(a, b int) (sum int, product int) {  
    sum = a + b  
    product = a * b  
    return  
}
```

一眼即可明了其意图，在较长函数中则应避免使用。

为保持代码可读性，与命名结果结合使用时，您有权选择是否使用裸露返回。

对于延迟闭包是必要的

若需在延迟函数调用中修改返回值，为结果参数命名至关重要。

```
func operation() (result int, err error) {
    defer func() {
        if r := recover(); r != nil {
            err = fmt.Errorf("caught panic: %v", r)
        }
    }()

    // Some operations that might panic
    result = someComputation()
    return
}
```

此处为结果参数赋予名称 `result` 和 `err` 具有特定目的。

这使得这些变量在延迟闭包内可访问，从而根据函数执行结果或针对 panic 进行相应修改。

即使在返回多个结果的情况下，有些情况也不必命名

当函数返回相同类型的对象时，特别是在某一类型的成员方法中，为每个返回的对象命名可能会造成冗余，并使我们的文档显得杂乱。

或者，该类型本身可能已经具有自解释性。

```
// BAD
func (n *Node) Parent1() (node *Node) {}
func (n *Node) Parent2() (node *Node, err error) {}

// BETTER
func (n *Node) Parent1() *Node {}
func (n *Node) Parent2() (*Node, error) {}

-----
// BAD
func fetchDetails(id int) (user *User, details *Details, err error) {
    // Fetching logic
    return
}

// BETTER
func fetchDetails(id int) (*User, *Details, error) {
    // Fetching logic
    return user, details, nil
}
```

Tip #20 传递值，而不是指针

原始链接: [Golang Tip #20: Pass values, not pointers](#)

这是我们在刚开始接触 Go 时绊倒了许多人（包括我自己）的一个话题。

很多时候，出于以下几个原因，我们会倾向于在函数中传递指针：

- 我们试图避免复制结构体时带来的开销。
- 或许是因为我们已经有了一个指针，感觉为了传值而去解引用它显得多余 ($*T$)。

**** 0. 关于指针的常见观念****

人们普遍认为这是一种节省内存的巧妙方法。

既然可以通过传递一个小巧的地址（指向数据存储的位置）代替复制所有数据传递给函数，何乐而不为呢？

但是建议是**优先直接传递值给函数，而不是传递指针**。

为什么呢？以下是关于何时应该传递值的5个关键点。

1. 固定大小的类型

这里我们讨论的是整数、浮点数、小型结构体以及（小型）数组。

这类类型的内存占用是固定的，通常在很多系统上与指针的大小相当，甚至更小。

```
○ ○ ○

type Rectangle struct {
    Width, Height int
}

func CalculateArea(rect Rectangle) int {
    return rect.Width * rect.Height
}
```

2. 不变性和清晰度

传递值意味着函数接收到一份数据的副本，不受外部影响。

这样，你无需担忧意外的副作用，因为任何改变都将局限于函数内部。

同时，传递值意味着你在向团队传达这样一个信息：“我不会改动你的原始数据，我只是需要操作这份数据”。

此处作者写成传递指针，应该是手误了

这种方法清晰且安全。

```
○ ○ ○

func AdjustWidthByValue(rect Rectangle, newWidth int) Rectangle {
    rect.Width = newWidth
    return rect
}

func AdjustWidthByPointer(rect *Rectangle, newWidth int) {
    rect.Width = newWidth
}
```

两个例子都被认为是好的。

如果你想在调用的函数内部改变值，当然可以通过指针来实现。

3. 小型或不太可能增长的类型

对于本质上较小或不太可能显著扩展的数据类型，直接传递可以避免解引用指针的额外步骤。

**** 4. 传递值的速度很快，而且很少比传递指针慢****

这可能会因为复制而显得有悖常理，但原因如下：

- 复制少量数据非常高效，通常比使用指针时所需的间接操作更快。
- 当值直接传递时，垃圾收集器的工作量会减少，因为它需要跟踪的指针引用更少。
- 通过值传递的数据在内存中往往存储得更紧密，这使得CPU能够更快地访问数据。

你很少会遇到一个足够大的结构体，以至于通过指针传递对其有利。

**** 5. 将传递值设为默认值****

只有在基准测试显示指针传递有明显优势时，才考虑使用指针。

一点点性能提升通常不值得牺牲清晰度。

当然，指针可以加速大型或不断增长的（无界的）结构体的处理速度，但你必须证明这是值得的。

Golang Tip #21: 定义方法时，优先使用指针作为接收器(receiver)

原文链接: [Golang Tip #21: Prefer using a pointer receiver when defining methods.](https://twitter.com/func25/status/1757759982354026636)

在继续之前，这里有一个关于指针接收器与值接收器的简短介绍：

<https://twitter.com/func25/status/1757759982354026636>

在Go中，规则并非是非黑即白的：“使用指针接收器进行修改，否则使用值接收器”。

一些指导原则：

何时选择指针接收器？

- 修改接收器的状态时
- 对于被认为是“大型”的结构体。这可能有点主观，就像我之前的推文中提到的那样
- 当结构体包含同步字段时，例如sync.Mutex，选择指针可以避免复制锁



```
type SafeCounter struct {
    mu    sync.Mutex
    count int
}

func (s *SafeCounter) Increment() {
    s.mu.Lock()
    defer s.mu.Unlock() // Requires a pointer to
    correctly lock/unlock the same mutex
    s.count++
}
```

- 如果不确定，选择指针接收器是比较明智的选择

何时适合使用值接收器？

- 小型且不会被改变的类型
- 如果你的类型是 map、func、channel，或者涉及到切片，而且切片的大小和容量不会改变（尽管元素可能会改变）

“为什么切片的大小和容量不会被改变？”

尽管你可以通过值接收器修改切片的元素或底层数组的内容（影响原始切片），但调整切片的大小（例如，使用append来增加容量）不会影响方法外部的原始切片。

这里有一个例子：<https://twitter.com/func25/status/1731181436282208375>

最后，一致性至关重要。

避免在给定结构体中混合使用不同的接收器类型，以保持一致性。

如果任何方法因为需要进行修改而使用指针接收器，通常最好为该结构体的所有方法都使用指针接收器，即使其中一些方法并不会引起修改。

我想到的主要理由有：

- 混合使用两种接收器可能会导致对该结构体进行操作时出现混乱和不一致，尤其是在更改接收器类型时
- 为了保持对象与接口的交互一致和简单（更多详情，参考我之前在 4/4 部分的推文）

Tip #22 使用结构体或变长参数简化函数签名

原始链接: [Golang Tip #22: Simplify function signatures with structs or variadic options](#)

在 Go 语言中设计函数时, 会遇到需要传递大量参数的情况。

```
func ConnectToService(  
    host string,  
    port int,  
    username string,  
    password string,  
    ssl bool  
) {  
    // Connection logic ...  
}
```

这会使得函数不够简明, 代码难以维护, 尤其是多个参数类型相同时。

为了保持代码整洁, 可以考虑以下两种策略:

- 结构体作为参数
- 变长参数

1. 结构体作为参数

将你的参数放入结构体中，不仅增加代码可读性，还使传参更方便。

什么时候使用结构体作为参数？

- 你的函数具有冗长的参数列表。
- 你的目的是编写自文档化的代码，因为结构体字段名本身就对其功能具有描述性。
- 你希望方便的为函数参数设置默认值或灵活的修改函数参数。

```
type ServiceOptions struct {
    Host      string
    Port      int
    Username string
    Password string
    SSL       bool
}

func ConnectToService(options ServiceOptions) {
    // Connection logic ...
}
```

当使用这种模式时，将 `context.Context` 作为单独的参数，而不是将其放到结构体中。

```
func ConnectToService(ctx context.Context, options ServiceOptions) {  
    // Connection logic ...  
}
```

这是因为 `context.Context` 在控制请求作用域的值、截止日期和取消信号方面扮演着独特的角色。

不过在使用结构体作为参数时，这里有些小的技巧：

- 保持结构体向后兼容性，当我们添加新字段时，不会破坏之前的任何功能。
- 在结构体被使用之前，我们总是能对其进行校验。
- 考虑隐藏结构体（设置成不可导出），通过暴露 `NewXXX()` 函数来给结构体赋默认值。

2. 变长参数

这种方法利用 Go 函数的能力，允许你以更简洁的方式传递不定数量的参数。

非常适合下面的情况：

- 函数需要高度可配制。
- 大多数参数是可选的或很少用到。
- 你喜欢简洁的函数调用。

```
type ServiceConfig struct {
    ssl bool
}

type ServiceOption func(*ServiceConfig)

func WithSSL(ssl bool) ServiceOption {
    return func(cfg *ServiceConfig) {
        cfg.ssl = ssl
    }
}

func ConnectToDatabase(options ...ServiceOption) {
    cfg := ServiceConfig{}
    for _, option := range options {
        option(&cfg)
    }

    // Connect using cfg...
}

// Usage:
ConnectToDatabase(WithSSL(true))
```

变长参数设置参数默认值比使用结构体作为参数更方便，你不需要隐藏它，直接在 ConnectToDatabase 函数中设置默认值即可。

Tip #23 省略 getter 方法的'Get'前缀

原始链接: [Golang Tip #23: Skip the 'Get' prefix for getters](#)

在编写代码时, 我们通常以动词开头给函数命名, 比如 get、set、fetch、update、calculate 等等...

但是在Go语言中 getter 方法是一个例外。

“为什么需要 getter 和 setter 方法? ”

在 Go 语言 中, 封装是通过方法的可见性和命名约定来实现的, 这巧妙地支持了封装, 而不需要严格使用 getter/setter 方法。

然而, 如果需要额外的逻辑, 或者我们想要访问一个计算字段, 手动定义 getter 和 setter 方法也是没有什么问题的。

“定义 getter 名称的惯用方法是什么? ”

惯用的方法是简单地使用字段的首字母大写作为 getter 的名称 (以便将其导出):

```
● ● ●

type Book struct {
    title string // unexported field
}

// GETTER
func (b *Book) Title() string {
    if len(b.title) == 0 {
        return "Unknown"
    }

    return b.title
}

// SETTER
func (b *Book) SetTitle(newTitle string) {
    b.title = newTitle
}
```

另一个示例涉及提供计算属性或配置的方法，这些属性或配置并不直接作为字段存储在结构体中：

```
package "http" // net/http

func (r *Request) UserAgent() string {
    return r.Header.Get("User-Agent")
}

func (r *Request) Cookies() []*Cookie {
    return readCookies(r.Header, "")
}
```

Tip #24 避免命名中的重复

原始链接: [Golang Tip #24: Avoid repetition in naming](#)

在编写代码时, 我们通常以动词开头给函数命名, 比如 get、set、fetch、update、calculate 等等...

1、包名与导出符号名称

在为对外可见 (即在包外可见) 的元素命名时, 应避免重复使用包名。

否则, 由于在包外使用这些符号时包名已经可见, 会导致名称过长且更为重复:

```
○ ○ ○  
// BAD  
chocolate.NewChocolateBar()  
userrepository.NewUserRepository()  
  
// BETTER  
chocolate.NewBar()  
userrepository.New()
```

这个“改进版”消除了重复。

当我们使用它时, 语义自然: `chocolate.NewBar()`, 清晰地创建了一个新的巧克力棒, 没有冗余。

2、变量名与类型

我们通常不需要在变量名中重复其类型。

通常从上下文或使用方式即可清楚得知。

```
○ ○ ○  
// BAD  
var secondaryHero *Hero  
var employeeList []*Employee  
  
// BETTER  
var secondary *Hero  
var cars []Car
```

然而，存在一些例外情况，应当予以考虑。

如果你同时拥有 `[]Car` 和 `map[string]Car` (可能是出于快速查找的目的)，那么为了清晰起见，可以这样做。

“但如何命名呢？`carList` 和 `carMap`？”

`CarList` 和 `carMap` 是不错的解决方案。

但我们可以指出数据的形式或状态使其更清晰，如：`[]Car cars` 和 `map[string]Car carLookup`

以下为另一个示例：

```
○ ○ ○  
// GOOD  
dateString := "2024-02-18"  
date, err := time.Parse("2006-01-02", dateString)  
  
// ALSO GOOD  
dateInput := "2024-02-18"  
date, err := time.Parse("2006-01-02", dateInput)
```

在第二种方案中，显而易见其为字符串和输入值。

3、避免重复归结于“上下文”

迄今为止我们讨论的所有内容都归结于“上下文”

- 包名
- 方法名
- 类型名
- 文件名

这些应指导你选择既简单又具有信息性、避免不必要的重复的名称。

接下来讨论一些与“上下文”相关的其他情况：

- 带有类型名的方法:

```
○○○

// BAD
func (u *User) GetUserName() string {}

// BETTER
func (u *User) Name() string {}

_____

// BAD
func (l *Logger) LogMessageToConsole(message string) {}

// BETTER
func (l *Logger) ToConsole(message string) {}
```

- 函数及其参数:

```
○○○

// BAD
func SendEmail(
    recipientEmailAddress string,
    emailSubject string,
    emailBody string
) {}

// BETTER
func SendEmail(recipient, subject, body string) {}
```

- 在函数内部, 特别是在处理与函数目的密切相关参数或数据时, 以一个不好的示例为例:

```
○○○

func (o *Order) CalculateOrderTotalCost() float64 {
    var orderItemsTotalCost float64
    for _, item := range o.Items {
        orderItemsTotalCost += item.Price * item.Quantity
    }

    return orderItemsTotalCost
}
```

我们将函数名和局部变量名都进行重命名：

```
○○○

func (o *Order) CalculateTotal() float64 {
    var total float64
    for _, item := range o.Items {
        total += item.Price * float64(item.Quantity)
    }

    return total
}
```

Tip #25 在 goroutines 之间进行信号传递时，使用 'chan struct{}' 而不是 'chan bool'

原始链接: [Golang Tip #25: Prefer 'chan struct{}' over 'chan bool' for signaling between goroutines.](#)

- **chan bool:** 也可以用作信号传递，但是传递的信号为布尔值 (true 或 false)，表达的意义可能不太清晰。
 - **chan struct{}:** 存粹用作信号传递，因为 struct{} 类型不占用内存。
-

"为什么倾向于选择 'chan struct{}'"

考虑一个使用 'chan bool' 的例子：

```
type JobDispatcher struct {
    startCh chan bool
}

func NewJobDispatcher() *JobDispatcher {
    return &JobDispatcher{
        startCh: make(chan bool),
    }
}

// Unclear: What does sending true or false mean?
```

这样的用法可能会令人困惑：我们应该发送 true 还是 false 来停止？

选择 `chan struct{}` 意味着，“我只对发生的事件感兴趣，而不关心传递的数据是什么。”

```
type JobDispatcher struct {
    startCh chan struct{}
}

func NewJobDispatcher() *JobDispatcher {
    return &JobDispatcher{
        startCh: make(chan struct{}, 1),
    }
}

func (d *JobDispatcher) SignalStart() {
    d.startCh <- struct{}{}
}

// Clear: Sending anything means "start job."
```

所以，使用 `chan struct{}` 有 2 个（主要的）优点：

- 由于 `struct{}` 不占用内存，通过 '`chan struct{}`' 不会在 channel 之间传递任何数据，只传递一个信号通知（一种微妙的内存优化手段）。
- 当开发者看到代码中的 '`chan struct{}`'，可以立刻清楚的知道这个 channel 是用于信号传递的，从而减少了歧义。

使用 `chan struct{}` 的缺点可能是有些笨拙的 "`struct{}{}`" 语法。

这种解决方案防止出现本应用于信号传递的 channel 被用于数据传输的情况。

Tip #25 使用空标识符（_）明确忽略值，而不是无声地忽略它们

原始链接：[Golang Tip #26: Explicitly ignore values with blank identifier（_）instead of silently ignoring them](#)

在编写Go语言代码时，函数可能会返回一些你可能想使用也可能不想使用的值。

在这种情况下，有两种处理方式：

- 隐式：调用函数但不将其返回值分配给任何变量，这种方式简短且简洁。

```
○ ○ ○

func PerformOperation() string {
    // Operation logic here...
    return "Operation completed successfully."
}

func main() {
    PerformOperation()
}
```

- 显式：稍显冗长一些，通过将返回值分配给空标识符 _ 来显式地忽略它。

“为什么即使显式方式更冗长且不如隐式方式简洁，我们仍然更倾向于使用它呢？”

在编程中，清晰性总是优于简洁性。

这种显式方式清楚地表明我们有意忽略了 `PerformOperation()` 的返回值。

使用 `_ =` 向其他开发者（或我们自己在不久的将来）发出信号，表明这种省略是故意的，而不是疏忽。

“那错误怎么办呢？”

无论如何，如果函数返回一个错误，一定要处理它，或者至少记录它。

同时，为了更好地提高清晰性，可以考虑添加注释来解释原因。

Tip #27 原地过滤

原始链接: [Golang Tip #27: Filter without any allocation.](#)

在Go语言中, 通常的做法是为过滤后的元素创建一个新的切片。但是, 这种方法会导致额外的内存分配。

```
var filtered []int

for _, num := range numbers {
    if isOdd(num) {
        filtered = append(filtered, num)
    }
}
```

更聪明的方法是利用原始切片的底层数组, 在原地切片过滤, 操作方式如下:

```
filtered := numbers[:0]

for _, num := range numbers {
    if isOdd(num) {
        filtered = append(filtered, num)
    }
}
```

- '`filtered := numbers[:0]`' 创建了一个新的切片 `filtered`, 它与 `numbers` 共享底层数组, 但长度为零, 同时保留了 `numbers` 的容量。
- 当我们将 `num` 添加到 `filtered` 中时, 我们避免了额外的内存分配, 因为我们实际上是修改了 `numbers` (或者是 `numbers` 的底层数组)。

因此，我们没有分配新的内存，而是在现有数组上进行修改。

```
numbers := []int{1, 2, 3, 4, 5, 6, 7, 8, 9}  
// ...  
filtered: [1 3 5 7 9]  
numbers: [1 3 5 7 9 6 7 8 9]
```

记住，这种技术最适用于以下情况：

- 在过滤后不再需要 `numbers` 切片。
- 性能至关重要，特别是在处理大型数据集的时候。

Tip #28 将多个if-else语句转换为switch

原始链接: [Golang Tip #28: Converting multiple if-else statements into switch cases.](#)

通过多个 `if-else` 语句处理复杂的条件逻辑是很常见的:

```
○○○

if canProcess && userActive {
    // ...
} else if !canProcess && !quotaReached {
    // ...
} else if quotaReached {
    // ...
} else {
    // ...
}
```

这种方法并没有错。

但是有一个更简洁、更易读的替代方案: 将 `if-else` 转换成 `switch` 语句

首先，我们应该了解 `switch-case` 结构是如何工作的：

○ ○ ○

```
switch initialization; expression {  
  case value1: ..  
  case value2: ..  
  ...  
  default: ..  
}
```

我们可以忽略“初始化”，也可以忽略“表达式”。当我们这样做时，我们本质上是在写：`switch true {}`，但 `true` 是隐式的

随后，回到我们的例子，让我们用我们刚刚讨论的内容来增强它：

```
○ ○ ○

switch {
  case canProcess && userActive:
    // ...
  case !canProcess && !quotaReached:
    // ...
  case quotaReached:
    // ...
  default:
    // ...
}
```

更多信息：[twitter.com/func25/status/...](https://twitter.com/func25/status/1100000000000000000)

Tip #29 避免使用 `context.Background()`, 使你的协程具备承诺性

原始链接: [Golang Tip #29: Avoid `context.Background\(\)`, make your goroutines promisable.](#)

译者注: 这里的承诺性 (promisable) 指的是协程运行的最终状态应该是确定的, 而不是无期限地一直运行下去, 这就像协程给了使用方一个承诺: 我要么执行成功, 要么因为超时等原因取消执行, 但最终在有限时间内一定会有一个明确的状态。

我们经常使用 Golang 协程处理并发任务, 并且这些协程经常需要执行阻塞任务, 比如:

- 执行 HTTP 请求;
 - 执行数据库查询、命令;
 - 从通道读取和写入数据
 - ...
-

“为什么要避免直接使用 `context.Background()` 呢? ”

我们必须确保这些操作不会无限期地挂起或阻塞协程 (而没有逃逸途径), 以避免资源泄漏、应用程序无响应、死锁等问题。

一般来说, 有两种方法可以使你的协程具有承诺性: 取消和超时。

```
context.WithTimeout(ctx, duration)
context.WithTimeoutCause(ctx, duration, errors.New("custom message"))

context.WithCancel(ctx)
context.WithCancelCause(ctx)

context.WithDeadline(ctx)
context.WithDeadlineCause(ctx, deadline, errors.New("custom message"))
```

因此，你启动的每一个协程都在做出一个承诺：“我要么完成我的任务，要么及时告诉你为什么我不能完成，并且你可以在任何时候取消我的任务。”

以下是一些关键点：

- 在底层实现中，`WithTimeout` 实际上是使用 `WithDeadline` 封装的；
- 一些 `XXXCause` 函数是在 Go 1.20 和 Go 1.21 版本中刚刚新增的；
- 如果在使用 `XXXCause` 类函数时发生超时，它会提供更详细的错误信息：“`context deadline exceeded: custom message`”。

“那么关于通道呢？我可不想在一个通道上永远等待。”

有很多种办法可以避免在通道上永远等待，但都会用到 `select{}`。

```
select {
    case result := <-ch:
        fmt.Println("Received:", result)
    case <-time.After(2 * time.Second):
        fmt.Println("Timed out")
}
```

上面代码中有一个微妙的注意事项：`time.After` 可能会导致内存泄露，请考虑使用 `time.NewTimer(duration)`。

Tip #30 使用context.WithoutCancel() 继续上下文操作

原始链接: [Golang Tip #30: Keep contexts going with context.WithoutCancel\(\)](#)

我们已经知道,当父上下文被取消时,它的所有子上下文也会被取消,对吗?

```
○ ○ ○

parentCtx, cancelFunc := context.WithCancel(context.Background())
childCtx, _ := context.WithCancel(parentCtx)

go func(ctx context.Context) {
    <-ctx.Done()
    fmt.Println("Child context cancelled")
}(childCtx)

cancelFunc()

// Give some time for the cancellation to propagate.
time.Sleep(time.Second)
```

但有时候,这不是我们想要的。在某些场景下,我们需要某些操作在父上下文被取消时继续进行,不被中断。想象一下,你正在处理一个HTTP请求,在请求被取消(客户端超时、断开连接等)的情况下,你仍然希望记录请求详细信息并收集指标。

"啊,我只需要为这些操作创建一个新的上下文就可以了"

这是一个解决方案,但新的上下文缺少原始事件上下文中的值,而这些值对于诸如记录、收集指标等任务很重要。只有子上下文才能传播这些值:

```
○○○

parentCtx := context.WithValue(context.Background(), "parent", "parent value")
childCtx, _ := context.WithCancel(parentCtx)

fmt.Println(
    "Value propagated to child context:",
    childCtx.Value("parent")
)

// Value propagated to child context: parent value
```

现在,回到我们的HTTP示例,这里是解决方案:

```
○○○

func handleRequest(req *http.Request) {
    ctx := req.Context()
    uncancelableCtx := context.WithoutCancel(ctx)

    go func() {
        // This logging operation won't be interrupted if
        // the parent context (ctx) is canceled.
        logRequestDetails(uncancelableCtx, req)
    }()
}
```

`WithoutCancel` 确保这些操作可以在请求被取消时仍然完成,而不会被中断。顺便说一下,这个函数是在Go 1.21中添加的。

Tip #31 使用跳转标签让break和continue语句更简洁

原始链接: [Golang Tip #31: Loop labels for cleaner breaks and continues](#)

通常避免使用标签和 `goto` 语句, 因为会降低代码可读性, 使其难以理解。



```
// Simulate checking user authentication
if userAuthenticated {
    fmt.Println("User is already authenticated.")
    goto SkipAuth
}

// "Starting authentication process..."
fmt.Println("User authenticated successfully.")

SkipAuth:
fmt.Println("Proceeding to the next step.")
```

上面简短的示例看起来很清晰易懂。但是随着代码复杂度的增加, 可读性会大大降低:

- 你可能要在距离 `goto` 语句“数百英里”之外才能找到目的标签。
- 你需要在代码上下文中去寻找 `goto` 语句跳转的目的标签在哪里。

跳转标签

例如，在处理嵌套循环时，某些情况下使用跳转标签是很不错的实践。

想象一下，我们在二维数组中搜索一个数字：

```
for i, row := range matrix {
    for j, value := range row {
        if value == target {
            fmt.Printf("Found %d at (%d,%d)\n", target, i, j)
            found = true
            break
        }
    }
    if found {
        break
    }
}
```

此时，你有一个更优雅的解决方案：在循环语句处声明跳转标签。

一旦声明了标签以后，你就可以使用 `break` 或 `continue` 后跟一个标签实现不仅本层循环的跳转，任何外层循环的跳转都可以。

这样做的结果是？

代码不仅简短，而且更加清晰、明了。

```
OuterLoop:  
for i, row := range matrix {  
    for j, value := range row {  
        if value == target {  
            fmt.Printf("Found %d at (%d,%d)\n", target, i, j)  
            break OuterLoop  
        }  
    }  
}
```

我们可以在 `break` 和 `continue` 语句中都使用标签。

另外一个有用的实例是当循环代码块中包含 `select{}`。

如果你在 `select` 代码块中使用了不带标签的 `break`，只会跳出 `select` 代码块，而不会跳出包含它的外层循环。



```
for {
    select {
        case <-someChannel:
            // ...
            break // This only exits the
        select, not the for loop
    }
}
```

本节的技巧主要针对循环，但也可以使用在其他地方，比如 `switch` 实例：

```
SwitchChoice:
    switch userChoice {
    case 1:
        fmt.Println("Option 1 selected. Exiting...")
    case 2:
        fmt.Println("Option 2 selected, run the loop")

        for i := 0; i < 5; i++ {
            if i == 2 {
                break SwitchChoice
            }
        }
    default:
        fmt.Println("Default option. No specific action taken.")
    }
```

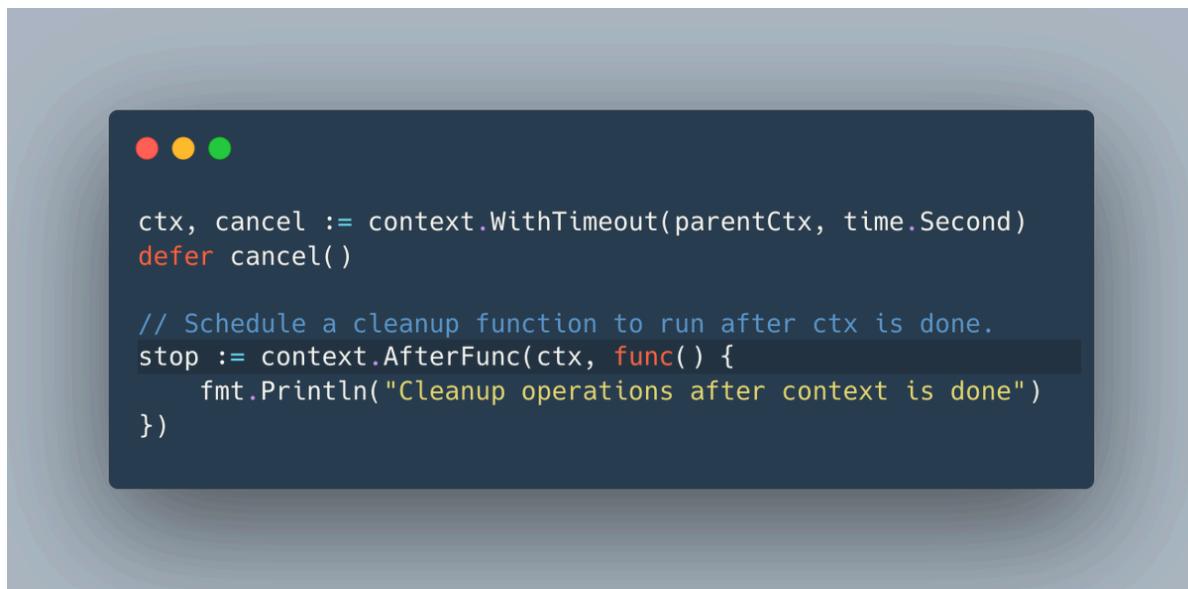
Tip #31 使用跳转标签让break和continue语句更简洁

原始链接: [Golang Tip #32: Scheduling functions after context cancellation with context.AfterFunc](#)

在 [tip #30](#)，我们学习了如何使一个Context在其Context停止时仍能继续运行：

现在，让我们来看一下Go 1.21引入的一项新特性。

`context.AfterFunc` 允许您设置一个回调函数 `f`，在ctx结束后（无论因取消还是超时）在新的goroutine中运行。



A screenshot of a macOS application window showing a dark-themed code editor. The code is written in Go and demonstrates the use of `context.AfterFunc`. The code is as follows:

```
ctx, cancel := context.WithTimeout(parentCtx, time.Second)
defer cancel()

// Schedule a cleanup function to run after ctx is done.
stop := context.AfterFunc(ctx, func() {
    fmt.Println("Cleanup operations after context is done")
})
```

该特性对于清理、日志记录或其他取消后的任务非常有用。

“回调函数何时运行？”

回调函数在一个新的goroutine中运行，该goroutine在接收到父级上下文的 `ctx.done` 通道发送的信号后被触发。

“如果上下文已经取消了怎么办？”

回调函数会立即运行，当然也是在一个新的 goroutine 中。

以下是几个要点：

- 自行运行：您可以多次使用同一上下文调用 `AfterFunc` 而没有任何问题，您设置的每个任务都会各自独立运行。
- 如果上下文已完成则立即运行：如果在调用 `AfterFunc` 时 `ctx` 已结束，则它会立即在一个新的goroutine中启动 `f`。
- 可以取消计划中的函数：它为您提供了一个 `stop` 函数，可以阻止`f`运行。
- 非阻塞：使用 `stop` 不会等待`f`完成，而是快速停止。如果您需要`f`和主线程工作保持同步，需要您自行安排。接下来我们谈谈`AfterFunc`返回的`stop()`函数：



```
stop := context.AfterFunc(ctx, func() {
    // ...
})

// Cancel the scheduled function before it runs
if stopped := stop(); stopped {
    fmt.Println("Callback was stopped before execution")
}
```

如果我们尚未完成上下文且回调尚未运行（实际上，goroutine尚未被触发）时就调用`stop()`，那么`stopped`将为`true`。

这意味着我们成功阻止了回调的运行。

如果`stop()`返回`false`，则可能意味着：

- 函数`f`已在新的goroutine中开始运行。

- 函数 f 已被停止。

Tip #33 尽量...不要使用panic()

原始链接: [Golang Tip #33: Just... Don't Panic\(\)](#)

“不要使用panic()”这句话听起来很激进，但实际上是在生产环境中应遵循的良好实践。

“为何这么说？难道我不能利用recover()来捕获panic()吗？”

即使你使用了recover(),也可能无法从panic()中恢复。下面我来解释一下：

```
func panicFunc() {
    panic("I'm panicking!")
}

func main() {
    defer func() {
        if r := recover(); r != nil {
            fmt.Println("Recovered from panic")
        }
    }()
}

go panicFunc()

time.Sleep(time.Second)
}

// panic: I'm panicking
```

(我之前在这篇[推文](#)也已稍微解释过)

在上面的代码片段中，panic是在一个新的goroutine中发生的（通过go panicFunc启动）。

关键是，只有panic触发和调用recover()是在同一个goroutine中时，使用recover()才有效。

因此，在主函数中的defer函数无法捕获或恢复panic，尽管尝试了恢复，程序仍然会崩溃。

但这并不是唯一的原因，还有另外两点考量：

1. **在生产环境中，代码必须具备极高的稳健性** 程序意外崩溃是绝对要避免的，因为它会导致系统宕机，从而影响用户体验，并可能对您的企业声誉造成不利影响。
2. **系统中某一部分的panic可能会引发连锁反应** 这可能导致系统（尤其是在微服务或者分布式系统）中其他部分接连出现故障（可能是级联失败）。

我们来看一个典型例子：

```
func initFile(filePath string) {
    f, err := os.Open(filePath)
    if err != nil {
        panic("failed to open the file")
    }
    defer f.Close()

    // Assume we do something with f next...
}

func main() {
    initFile("somefile.txt")
}
```

上面这种做法并非不可取，但它们鼓励了对panic的使用。一个更佳的做法应该如下：

```
func openFile(filePath string) error {
    f, err := os.Open(filePath)
    if err != nil {
        return fmt.Errorf("failed to open the file: %w", err)
    }
    defer f.Close()

    // Assume we do something with f next...

    return nil
}

func main() {
    if err := openFile("path/to/your/file.txt"); err != nil {
        fmt.Fprintf(os.Stderr, "Error: %s\n", err)
        os.Exit(1)
    }
}
```

当程序返回错误而非panic时，你的程序可以根据错误做进行相应的处理，例如：

- 重试操作
- 使用默认值
- 记录详细的调试信息
- 程序终止
- 等等...

这种灵活性对于构建健壮的系统至关重要。

应当把panic作为最后的手段

- 仅在遇到真正无法恢复的错误时才使用panic，即如果继续运行程序可能会引发更严重的问题，比如数据损坏或未知行为。
- 在程序初始化阶段，如果一个关键组件启动失败，panic或许是“可接受的”，因为它表明程序无法按预期运行。

Tip #34 以context开头，以options结尾，并且总是用error来关闭

原始链接：[Golang Tips #34: Lead with context, end with options, and always close with an error](#)

编写符合习惯的Go代码通常涉及遵循一定的模式和最佳实践，这些模式和实践能够提升代码的可预测性。

这里是3个设计函数签名时的关键准则：

1. `context.Context`放在前面

在函数签名中，`context.Context` 应当始终放在首位。

“为什么？”

`context` 通常与一个请求或者操作的生命周期息息相关。

当浏览代码时，一看到 `context.Context` 作为第一个参数，就立刻让人明白这个函数有如下的特性：

- 取消
- 截止期限
- 其他上下文相关的机制

这种一致性有助于提高代码的可读性，并且让代码库变得易于导览。

```
func FetchData(ctx context.Context, resource string) (*Data, error)
```

此外，别把context.Context放到 struct 中。

context 本质上意味着它注定是短暂的，旨在贯穿与一段程序，而非成为一个对象状态的一部分（这里有一些例外情况，比如，HTTP 的 handler，大家习以为常地从请求中提取 context，这是因为这里 context 早已跟请求的生命周期关联起来了）。

2. Options结构体置于最后

“options结构体”模式是一个灵活而强大的方法，它能让函数在不破坏兼容性的前提下随着时间去演进。

我曾在推特上探讨过 options 结构体和可变参 options：

<https://twitter.com/func25/status/1758435261183353308>

```
type FetchOptions struct {
    Cache    bool
    Priority int
}

func FetchData(context.Context, string, FetchOptions) (Data, error) {}
```

参数的顺序可能表征了这个参数的重要性。

把这个结构体作为一个函数的最后一个参数有两个目的：

- 保持一致性（与可变参 options 模式一致）
- 表明这些是可选配置项，而非函数操作逻辑的核心部分

3. 以error (或者bool) 结尾

Go在语义上表征一个操作是成功还是失败是通过它的最后一个返回值来达成的，通常这个最后的返回值是一个error。

某些情况下，如果布尔值更加妥当，比如说存在性检查，那么它也是放到最后的。

如果兼而有之，那么优先级应当是 (x, bool, error)。

```
func FetchData(ctx context.Context, resource string) (*Data, error)

func TryFetchData(ctx context.Context, resource string) (Data, bool, error) {
    //...
}
```

在譬如 TryFetchData 的场景中，bool 被用来表征存在性，特别是即便不存在也不被当做是一个错误的情况下。

Tip #35 转换字符串时优先使用 `strconv` 而非 `fmt`

原始链接: [Golang Tip #35: Prefer strconv over fmt for converting to/from string.](#)

当需要将数字转换为字符串时, 选择合适的工具可以加快处理速度。

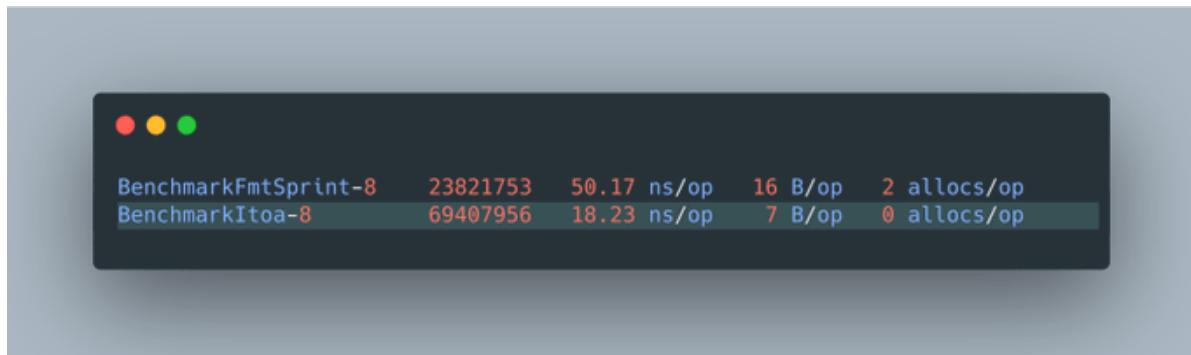
`strconv` 包专门为这个场景而设计, 每一点性能提升和内存节省都很重要。

我们来看一个简单的基准测试:

```
func BenchmarkFmtSprint(b *testing.B) {
    for i := 0; i < b.N; i++ {
        _ = fmt.Sprint(i)
    }
}

func BenchmarkItoa(b *testing.B) {
    for i := 0; i < b.N; i++ {
        _ = strconv.Itoa(i)
    }
}
```

基准测试显示出显着的性能差异。



Benchmark	Operations per second	Time per operation	Memory usage per operation	Allocations per operation
BenchmarkFmtSprint-8	23821753	50.17 ns/op	16 B/op	2 allocs/op
BenchmarkItoa-8	69407956	18.23 ns/op	7 B/op	0 allocs/op

(虽然我不确定编译器是否做了优化，但两者的上下文是相同的)

- **strconv** 的函数是为特定的转换任务设计的，这使得它们能比更通用的 **fmt** 函数执行得更快。
- **fmt.Sprint** 函数及其变体需要通过反射来识别其正在处理的类型，并确定如何将其格式化为字符串。



```
func (p *pp) doPrint(a []any) {
    prevString := false
    for argNum, arg := range a {
        isString := arg != nil && reflect.TypeOf(arg).Kind() == reflect.String
        // Add a space between two non-string arguments.
        if argNum > 0 && !isString && !prevString {
            p.buf.WriteByte(' ')
        }
        p.printArg(arg, 'v')
        prevString = isString
    }
}
```

这个反射过程并非无成本，它既增加了时间也增加了内存开销。

Tip #36 以下划线（_）作为前缀，命名非导出的全局变量

原始链接：[Golang Tip #36: Naming Unexported Global Variables with an Underscore \(_\) Prefix.](#)

（注：此命名规则在Go社区并没有被普遍认可为惯用做法，而是受到Uber编码风格指南的启发）

在Go语言中，声明在顶层的变量和常量可以在它们所属的整个包中被访问。

常规命名方式有何不妥？

如果没有明确的命名约定，我们很容易在更小的作用域内无意中覆盖这些包级别的变量。

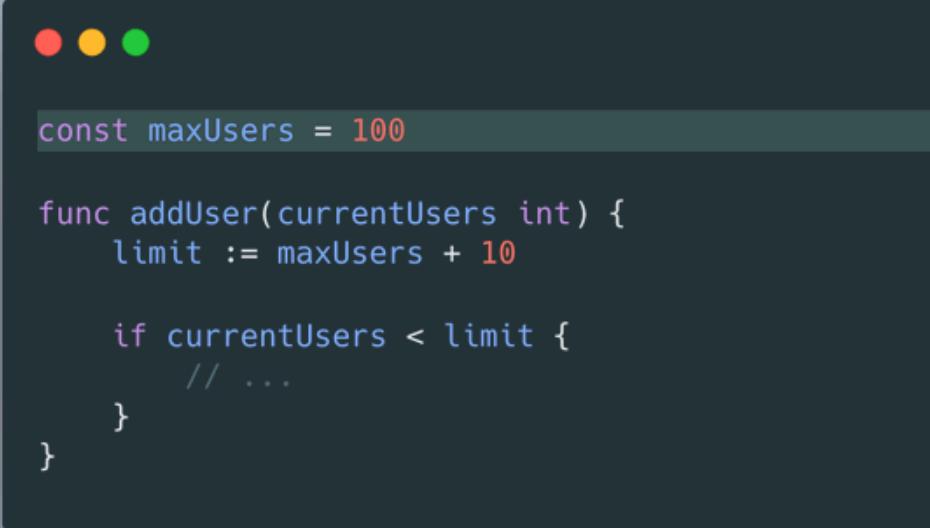


```
var dataSize = 100

func processData() {
    dataSize := 50
    fmt.Println("Processed data size:", dataSize)
}
```

设想一下，一个命名为 `dataSize` 的局部变量可能会覆盖同名的全局变量。

但如果它们的命名不同，还会有什么问题呢？”



```
const maxUsers = 100

func addUser(currentUsers int) {
    limit := maxUsers + 10

    if currentUsers < limit {
        // ...
    }
}
```

尽管这个例子看似简单，但它却引发了一个问题：我们如何知道 `maxUsers` 变量的来源？

- 它是一个像 `'limit'` 这样的局部变量？
- 还是函数的一个参数？
- 或者是来自全局作用域？

在更复杂的场景中，我们可能不得不四处搜索或使用IDE的快捷键（如cmd + click）来查找并跳转到变量的定义。这个过程可能会分散我们的注意力并打断我们的工作流程。

使用下划线前缀

通过在全局变量前添加下划线(_), 可以明确表示这些标识符是全局的:

```
const _maxUsers = 100

func addUser(currentUsers int) {
    limit := _maxUsers + 10

    if currentUsers < limit {
        // ...
    }
}
```

这种明确的标识使得“_maxUsers”被一眼识别为全局变量，大大降低了我们无意中覆盖或修改它的风险。

Tip #37：使用未导出的空结构体作为上下文键

原始链接：[Golang Tip #37: Using Unexported Empty Struct as Context Key](#)

context包不仅可用于传递取消信号和设置截止日期，也常用于传递请求范围的值。

我们可以在上下文中添加一个值，将其向下传递，然后再获取它：

```
● ● ●

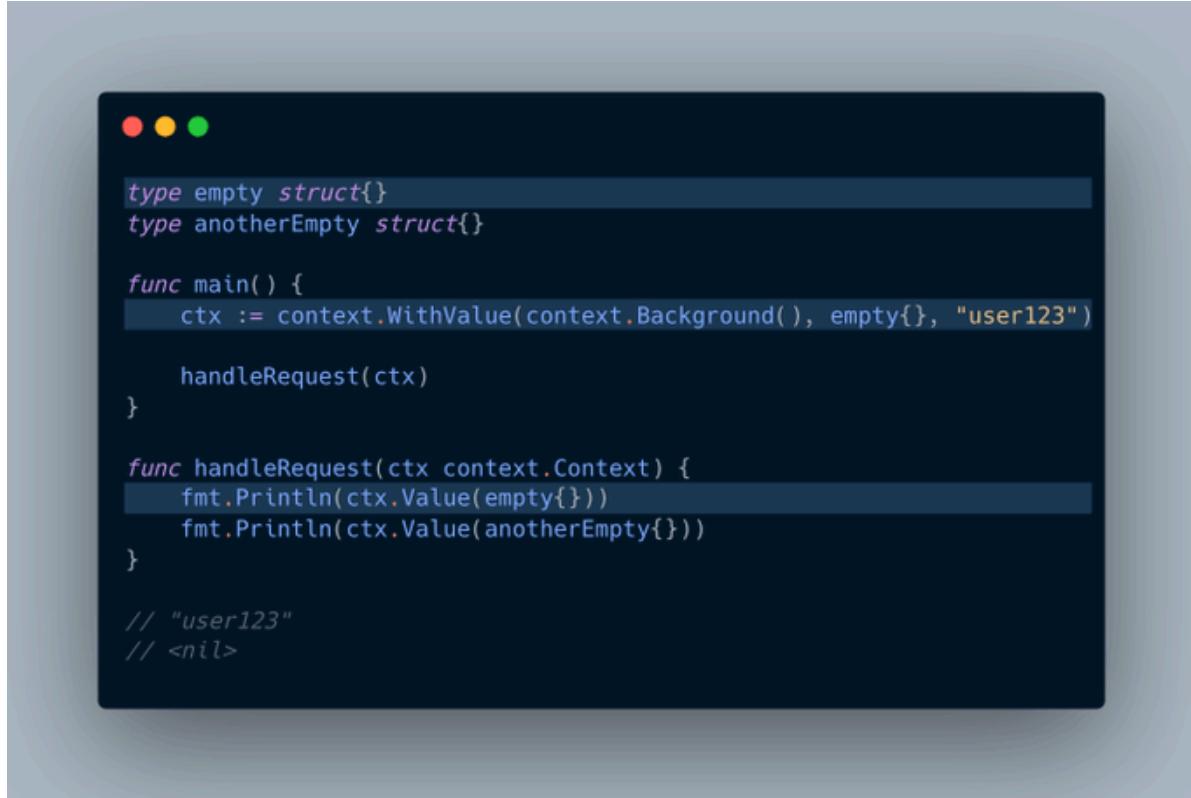
func main() {
    ctx := context.WithValue(context.Background(), "data", "request-scoped data")
    handleRequest(ctx)
}

func handleRequest(ctx context.Context) {
    fmt.Println(ctx.Value("data"))
}

// request-scoped data
```

挑战在于，我们如何确保我们的键（本例中是“data”）是唯一的？

完全有可能其他人已经使用“data”作为键，因此可能存在潜在的冲突。



```
type empty struct{}
type anotherEmpty struct{}

func main() {
    ctx := context.WithValue(context.Background(), empty{}, "user123")

    handleRequest(ctx)
}

func handleRequest(ctx context.Context) {
    fmt.Println(ctx.Value(empty{}))
    fmt.Println(ctx.Value(anotherEmpty{}))
}

// "user123"
// <nil>
```

这正是空结构体发挥作用的地方，每个结构体相比于其他结构体都是唯一的：

一般来说，使用未导出（私有）的空结构体，我们可以避免由其他包引起的任何潜在冲突。

“我可以使用其他类型吗，尽管其底层类型依然是字符串或整数？”

是的，我们当然可以使用其他类型，并且应该可以避免冲突。例如，一个底层类型为int，数值为0的number和一个int(0)是不同的：



```
type number int

func main() {
    ctx := context.WithValue(context.Background(), number(0), "value from number type")
    ctx = context.WithValue(ctx, 0, "value from int type")

    handleRequest(ctx)
}

func handleRequest(ctx context.Context) {
    fmt.Println(ctx.Value(number(0)))
    fmt.Println(ctx.Value(0))
}

// "value from number type"
// "value from int type"
```

这背后的原理归结为 Go 如何比较 interface{}，只有当两个 interface{} 的类型和值都匹配时，它们才相等。

- 第一个值：{ 类型：number， 值：0 }
- 第二个值：{ 类型：int， 值：0 }

它们是不同类型的，因此它们不相等。

“但为什么会选择使用一个空的struct{}呢？”

一个空结构体不会分配内存，它没有字段因而不包含数据，但它的类型仍然可以唯一地标识上下文值。

当然，我们仍然会在某些情况下使用具有底层基本类型的类型定义。

（使用上下文值是我一直避免的事情，尤其是在编写业务逻辑时。它不是编译时安全的，并且难以追踪和调试。）

Tip #38 使用 `fmt.Errorf` 使你的错误信息清晰明了，不要让它们过于赤裸

原始链接：[Golang Tip #38: Make your errors clear with `fmt.Errorf`, don't just leave them bare.](#)

在 Go 语言中，错误被当成值来处理。我们采用返回错误而非抛出错误的方式：

```
func doOperation() error {
    err := doSomething()

    if err != nil {
        return err
    }

    return nil
}
```

仅返回错误而不提供任何额外详情会导致难以确定错误来源及为何发生。

这会使调试错误和处理错误变得更加困难。

**使用 `fmt.Errorf` 和 `%w` **

Go 1.13 版本引入了一种在保留原始错误的同时为其添加更多信息的方法，这就是通过 `fmt.Errorf` 函数配合 `%w` 符号来实现。

它会将错误包装起来，以便您在后续需要时能够深入探究：

```
func doSomething() error {
    err := someOperation()
    if err != nil {
        // Add context to the error
        return fmt.Errorf("failed to do someOperation: %w", err)
    }
    return nil
}
```

“我还是没看出这样做的好处，反正最后都只是一个错误。”

让我们通过一个例子来看看添加详细信息的重要性：

```
func getResourceHandler(http.ResponseWriter, *http.Request)
| -> func authorizeUser(userID, resourceID string) error
|   | -> func getUserFromDB(userID int) (*User, error)
| -> func getResourceByID(resourceID string) (*Resource, error)
```

下面哪一项提供了更多信息？

- "Failed to retrieve resource: Authorization check failed: User 123 does not exist: mongo: no documents in result"
- "Failed to retrieve resource: mongo: no documents in result"

第一种表述清楚地显示出问题起始于一个不存在的用户，导致了操作失败。而第二种错误信息则未能帮助我们判断问题根源是在用户还是资源上。

如果没有这些细节信息，我们可能会错过关于到底哪里出错的重要线索。

另外，通过使用 `errors.Is()` 方法，我们可以精准地定位错误的确切类型：



```
func readConfig(path string) error {
    return fmt.Errorf("failed to read config: %w", ErrConfigNotFound)
}

func main() {
    err := readConfig("config.yaml")
    if err != nil {
        if errors.Is(err, ErrConfigNotFound) {
            // ...
        }
    }
}
```

@thedenisnikulin 提供了额外的建议，我们可以进一步改进错误处理方式。

"failed to do X: %w" is bad when you have deeply wrapped errors better
write "doing X: %w"

Tip #39 避免在循环中使用defer，否则可能会影响内存溢出

原始链接：[Golang Tip #39: Avoid defer in loops, or your memory might blow up.](#)

在 Go 中使用 defer 时，我们一般希望 defer 后面的函数能在当前函数返回之前被执行。

```
○ ○ ○

func main() {
    defer fmt.Println("World")
    fmt.Println("Hello")
}

// Output:
// Hello
// World
```

然而，像这样在循环中放置 defer 是不建议的：

```
○ ○ ○

for _, file := range files {
    f, err := os.Open(file)
    if err != nil {
        //...
    }

    defer f.Close()
}
```

(为简单起见，让我们不考虑 f.Close 的错误处理)。

以下是需要考虑的两个关键点：

1. 执行时间

所有这些延迟调用都在函数即将返回时执行，而不是在循环的每次迭代之后执行。

如果您的循环是长时间运行的函数的一部分，这意味着在很久之后才会执行任何延迟的任务。

当这些延迟的任务用于释放资源或清理时，这尤其成问题。这样我们不能在完成后立即释放资源，而是等到最后才释放。

2. 内存溢出的可能性

每个 defer 都会在内存中添加一个调用点。

在迭代了数百或上千次的循环中，因为每个调用都会消耗内存，导致堆栈被这些延迟调用填满。

每个延迟调用的细节需要被存储（例如要调用的函数及其参数），这些细节被分配在函数的堆栈帧中（或者根据编译器的策略分配在堆上）。

有几种策略可以减轻影响。其中一个偷懒的修复它的方法，可以考虑使用匿名函数：

```
○ ○ ○

for _, file := range files {
    func(filename string) error {
        f, err := os.Open(filename)
        if err != nil {
            // ...
        }

        defer f.Close()
    }(file)
}
```

我们也可以将这个功能修改成一个具名函数，尽量不要使用 defer（除非这个函数有 panic 的风险）。

Tip #40 在使用defer时处理错误以防止静默失败

原始链接: [Golang Tip #40: Handle errors while using defer to prevent silent failures](#)

有一个隐蔽的/易被忽视的陷阱,许多人都会落入其中:忘记检查延迟调用中的错误:

```
○ ○ ○

func doSomething() error {
    file, err := os.Open("example.txt")
    if err != nil {
        return err
    }
    defer file.Close()

    // ...
}
```

让我们以上面的代码片段为例。如果文件关闭操作失败(可能是由于写操作未刷新或文件系统出现问题),且这个错误没有被检查,我们就失去了优雅处理故障的机会。现在,仍然使用defer,我们有3种选择:

- 将其作为函数错误处理
- Panic
- 记录日志

Panic或记录日志都很直接,但如何将其作为函数错误处理呢?在这种情况下,使用命名返回值可能是一个简单的解决方案:

```
○○○

func OpenFile(path string) (err error) {
    file, err := os.Open(path)
    if err != nil {
        return err
    }

    defer func() {
        if cerr := file.Close(); cerr != nil {
            err = errors.Join(err, cerr)
        }
    }()
}

// ...
}
```

或者更简短的方式:

```
○○○

defer func( ) {
    err = errors.Join(err, file.Close())
}()
```

然而,由于需要创建一个匿名函数,这种方法仍然有些冗长,增加了嵌套层次。考虑到大多数延迟调用都涉及关闭资源,比如连接或I/O操作,我们可以使用一个更简洁的一行解决方案来简化,使用io.Closer:

```
○○○

defer closeWithError(&err, file)

// somewhere
func closeWithError(err *error, closer io.Closer) {
    *err = errors.Join(*err, closer.Close())
}
```

但是这段代码会导致panic,因为当err可能为nil时对其解引用,对吗?

实际上并非如此,这段代码可以正常工作。

(警告,以下是大段的分析)

幸运的是,由于error是一个接口, `nil error` 并不意味着它就是其他指针类型(如 `*int`)的nil指针。

一个nil(接口)error的结构是`{type=nil; value=nil}`,但它仍然...是一个值,即接口的零值。

当我们在 `defer closeWithError(&err, file)` 调用中使用 `&err` 取err的地址时,我们得到的不是一个nil指针。我们得到的是一个指向接口变量的指针,该变量的值为 `{type=nil, value=nil}`。

所以在 `closeWithError` 函数中,当我们使用 `*err` 解引用错误指针来赋予新值时,我们并没有解引用一个nil指针(那会导致panic)。

相反,我们是通过指针修改了一个接口变量的值。

Tip #41 将你结构体中的字段按从大到小的顺序排列

原始链接: [Golang Tip #41: Sort your fields in your struct from largest to smallest.](#)

(我之前发过一些关于字段填充和对齐的推文, 但这次我想把它作为一条 tip 分享)

结构体中字段的顺序确实会影响到结构体自身的大小, 这意味着我们可以利用这一点来优化内存的使用, 不是吗?

让我们来看一个示例 (暂时忽略每个字段的注释):

```
// 32 bytes
type StructA struct {
    A byte // 1-byte alignment
    B int32 // 4-byte alignment
    C byte // 1-byte alignment
    D int64 // 8-byte alignment
    E byte // 1-byte alignment
}

// 16 bytes
type OptimizedStructA struct {
    D int64 // 8-byte alignment
    B int32 // 4-byte alignment
    A byte // 1-byte alignment
    C byte // 1-byte alignment
    E byte // 1-byte alignment
}
```

结构体 StructA 使用了 32 字节，而 OptimizedStructA 仅需要 16 字节。

为了理解为什么具有相同字段的两个结构体大小不同，让我们探讨一下字段的对齐和填充：

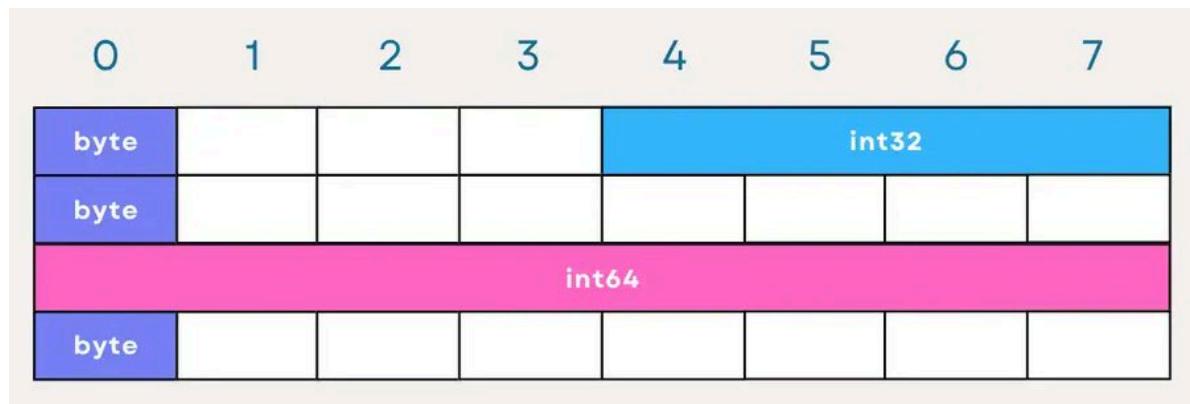
- **对齐**：数据类型根据其大小具有特定的对齐要求。

例如，一个 int32 类型可能需要在 4 字节边界上对齐，这意味着它的启始内存地址应该是 4 的倍数。

- **填充**：为了满足对齐要求，编译器可能会在结构体字段之间插入未使用的空间（填充）。

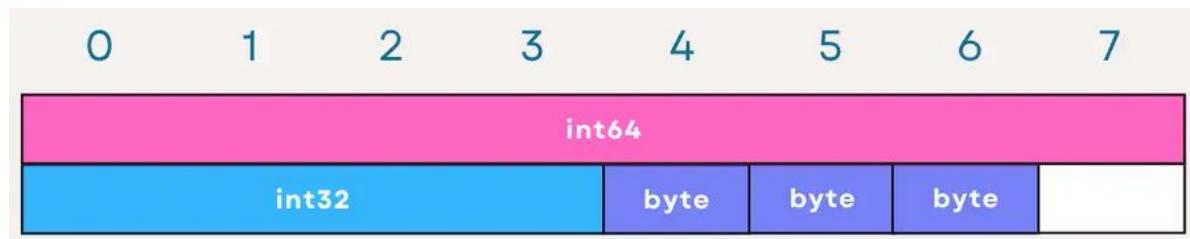
让我们看看 StructA 的内部表示，它的大小为 8x4 字节，让我们尝试使用上述的思路来解释：

以下是对 StructA 每个字段的解释：



- A (byte)：占用 1 字节，但由于下一个字段 B 需要 4 字节对齐，因此在 A 后面有 3 字节的填充以正确对齐 B。
- B (int32)：4 字节，后面不需要填充，因为下一个字段 C 是一个字节。
- C (byte)：同样占用 1 字节，但为了对齐 D (需要 8 字节对齐)，在 C 后面添加了 7 字节的填充。
- D (int64)：8 字节，完全利用了它的空间。
- E (byte)：最后一个字节，在内存中直接跟在 “D” 的后面，根据上下文可能会导致在结构体的末尾添加额外的填充以将整个结构体的大小对齐到边界。

现在来看看 OptimizedStructA：



- D (int64)：最先被放置以利用其 8 字节对齐的要求，无需前置填充。

- B (int32)：紧随其后，在 D 后自然对齐到 4 字节边界。
- A, C, E (byte)：随后组合在一起，由于它们是单字节类型，它们之间不需要额外的填充。

通过将字段按从大到小的顺序排列，我们可以让所需的填充最小化，从而减少结构体（和内存）的总大小。

像 **betteralign** 这样的工具可以检测到效率低下的对齐方式，并可能帮助自动重新排序以提高效率：

<https://github.com/dkorunic/betteralign>

需要注意的是，为了效率而重新排序并不总是适用或必要的。

保持结构体字段按照其使用方式或重要性进行有意义的顺序排列，即使这种方式并不使用最少的内存，也可以使代码更易于阅读和使用。

Tip #42 单点错误处理，降低噪音

原始链接：[Golang Tip #42: Single Touch Error Handling, Less Noise.](#)

这是我以前处理错误的方式，假设我们有一个函数 A 调用函数 B，两个函数都处理错误，如下所示：

```
func B() error {
    if err := doSomething(); err != nil {
        log.Printf("failed to doSomething: %v", err)
        return err
    }

    return nil
}

func A() error {
    if err := B(); err != nil {
        log.Printf("unable to do B: %v", err)
        return err
    }

    return nil
}
```

- 当 B 产生一条错误时，将问题记录日志，并将错误传递给 A。
- A 收到此错误后，重复同样的操作：记录日志，也可能将错误传递给上层的调用链。

这有什么问题？

这似乎是彻底的错误处理方式，因为我们可以从一条条日志中追溯错误来源，但实际上只会制造噪音。

这些问题：

- **重复记录日志：**这会在日志文件中制造噪音，使得诊断问题变困难，因为相同的错误被记录了多次。
- **错误处理变复杂：**它增加了错误处理逻辑的复杂度。
- **潜在的其他错误：**多次错误处理意味着更多的代码，更多的代码意味着更多的潜在 bug。

一条错误，只考虑处理一次，但是如何有效的做到这点呢？

更好的解决方案

一个更好的处理方法是决定在本层处理错误，还是将错误返回给上层处理（但不要同时都处理）。

如果你选择返回错误不记录日志，考虑给错误添加更多上下文（参考[Tip #38](#)，原链接<https://twitter.com/func25/status/...>）

```
func B() error {
    if err := doSomething(); err != nil {
        return fmt.Errorf("failed to doSomething: %w", err)
    }
    return nil
}

func A() error {
    if err := B(); err != nil {
        return fmt.Errorf("failed when calling B: %w", err)
    }
    return nil
}

// Centralized error logging for when A is called
if err := A(); err != nil {
    log.Printf("Operation A failed: %v", err)
}
```

让调用者来决定如何处理错误，是记录日志，产生恐慌，包装额外的上下文，还是采取一些纠正措施。

Tip #43 优雅关闭你的应用程序

原始连接: [Golang Tip #43: Gracefully Shut Down Your Application](#)

当我们讨论优雅地关闭应用程序时, 有几个关键保证是我们力求实现的:

- 不接收新请求: 服务器停止接受新的请求。
- 完成正在进行的任务: 等待当前处理的任务达到逻辑上的停止点。
- 资源清理: 释放诸如数据库连接、打开文件、网络连接等资源。

虽然存在一些不同的实现方式, 但为了简化起见, 我尝试给出最简短的方法:

```
○○○

ctx, cancel := signal.NotifyContext(
    context.Background(), os.Interrupt, syscall.SIGTERM)
defer cancel()

server := http.Server{Addr: ":8080"}

g, gCtx := errgroup.WithContext(ctx)
g.Go(func() error { return server.ListenAndServe() })
g.Go(func() error {
    ←gCtx.Done()
    // consider using a timeout here
    return server.Shutdown(context.Background())
})

if err := g.Wait(); err != nil {
    fmt.Printf("exit with: %v\n", err)
}
```

首先, 我们创建一个 (主) 上下文, 当接收到中断信号 (`Ctrl+C`) 或 `SIGTERM` 时将其取消。

接着我们创建两个goroutine，均由 `errgroup` 协调（如果您还不了解它，请考虑阅读 <https://blog.devtovert.com/p/go-errgroup-you-havent-used-goroutines>）：

第一个直接启动服务器，但要记住，`ListenAndServe` 始终返回非空错误。第二个更有趣，这是我们放置优雅清理代码的地方。这个goroutine等待 `gCtx.Done()` 关闭，该关闭状态由我们的主ctx传播而来。

如果您的服务运行在Kubernetes上，应考虑在接收到SIGTERM后不立即终止新请求。

您的应用程序不应立即终止，而应完成所有活跃请求，并继续监听Pod关闭开始后到达的传入连接。

Kubernetes可能需要一段时间来更新所有kube-proxy和负载均衡器。

这是简化版，您可能需要考虑为服务器配置添加超时、检查错误是否关闭、为关闭过程添加超时等。

Tip #44 有意地使用Must函数来停止程序

原始链接: [Intentionally Stop with Must Functions](#)

这是一个乍一看有点儿反直觉的技巧, 就是使用“Must”函数有意地允许程序停止。

通常情况下, 我们会尽可能的避免程序发生panic的情况, 但是在某些情况下, 这种解决方案可以避免一些冗余的逻辑。

```
○ ○ ○  
var validID = regexp.MustCompile(`^[a-z]+[0-9]+\]$`)  
var tmpl = template.Must(template.New("name").Parse("{{.}}"))
```

如果你使用过Go语言, 你很可能已经在标准库里见过这类函数了。

这类函数有一个特定的命名模式, 它们以“Must” (或“must”) 开头, 这就是提醒你需要警惕一下, 如果程序没有按照预期执行的话就会导致panic。

```
○○○

func MustCompile(str string) *Regexp {
    regexp, err := Compile(str)
    if err != nil {
        panic(`regexp: Compile(` + quote(str) + `): ` + err.Error())
    }
    return regexp
}
```

Must函数主要用于：

- 通常情况下不应失败的**初始化**任务，例如：在应用程序开始时设置包级变量、设置正则表达式、连接数据库等。

```
○○○

func MustLoadConfig(path string) *Config {
    config, err := LoadConfig(path)
    if err != nil {
        panic(fmt.Sprintf("failed to load config: %v", err))
    }
    return config
}
```

- 它们在**单测**场景下也非常有用，允许使用t.Fatal立即失败这个测试用例。

```
○○○

func mustDecodeJSON(t *testing.T, jsonStr string, v any) {
    t.Helper()
    if err := json.Unmarshal([]byte(jsonStr), v); err != nil {
        t.Fatalf("mustDecodeJSON failed: %v", err)
    }
}
```

Must函数是在初始化和编写单元测试时候的工具，用于处理难以预料的情况。

它简化了在特定场景下的处理错误的方式，但是应该谨慎的使用避免panic。

(谈论一个鼓励我们的应用程序停止的小技巧可能会感觉有点奇怪 😊)

Tip #45 始终管理您协程的生命周期

原始链接: [Golang Tip #45: Always Manage Your Goroutine Lifetime.](#)

Golang 中的协程是有栈协程, 这意味着相比较于其他语言中的类似结构, Golang 中的协程会占用更多的内存, 每个 Golang 协程至少会占用 2KB 的内存。不要小看这 2KB 的内存占用量, 因为在 Golang 中, 协程的创建是非常便捷的, 很容易就快速增长到一个庞大的数量, 当协程数量达到 10K 时, 其内存占用将达到 20MB。

(!: 我会将相关信息放置在本 Tip 的底部)

我必须承认, 我运行过很多使用 `for` 循环和 `time.Sleep` 的任务, 其代码类似如下:

```
○ ○ ○

func Job() {
    for ;; time.Sleep(10 * time.Second) {
        // ....
    }
}

func Job() {
    for {
        // ....
        time.Sleep(10 * time.Second)
    }
}
```

以这种（懒惰）方式编写代码非常方便，但这样做也有缺点。

如果你阅读了关于优雅关闭的第 43 条 Tip，你就会明白，这个函数无法优雅地结束（除非在 `time.Sleep` 期间偶然发生），对吧？

睡眠（Sleep） -> 终止信号（SIGTERM） -> 运行中（Running） -> 被中断（Interrupted）。

因此，对于那些本质上没有明确终点的任务（例如：网络连接服务、配置文件监视等），应该使用取消信号或条件来明确定义这些任务何时应该结束。

```
○ ○ ○

func Job(ctx context.Context) {
    for {
        select {
        case <-ctx.Done():
            return
        default:
            // ...
            time.Sleep(10 * time.Second)
        }
    }
}
```

上面代码中使用了上下文 `context`，此上下文应该是由其他代码中的上下文（基础上下文）传播而来，当收到终止信号 `SIGTERM` 时，基础上下文会被取消，进而导致此处的代码在上下文处直接返回。

因此，至少我们知道上面这段代码何时停止，即使是在程序终止的时候。

以下是另一个可能导致 Golang 协程永远卡住的场景：

○ ○ ○

```
func worker(jobs <-chan int) {
    for job := range jobs {
        //...
    }
}

jobs := make(chan int)
go worker(jobs)
```

我们可能认为：一旦作业通道关闭，就很容易确定协程何时结束。

但是工作通道何时会关闭呢？

也许并非如此，我们可能会犯一个错误，即：没有关闭通道而是直接从函数中返回，这会导致协程无限期地挂起，进而引发内存泄漏。

因此，务必确保协程的启动和停止时机是显而易见的，并务必把上下文传递给长时间运行的任务。

阅读更多：

- Golang 协程是有栈协程：

<https://twitter.com/func25/status/1762632488219004931>；

- 第 43 条 Tip: <https://twitter.com/func25/status/1766104130303705226>。

Tip #46 避免在 switch 语句的 case 中使用 break, 除非与标签一起使用

原始链接: [Golang Tip #46: Avoid using break in switch cases, except when paired with labels.](#)

在 C、C#、Javascript 等语言中, 常常在 switch 语句的每个 case 末尾使用 break 来避免代码错误地继续执行下一个 case:

```
○ ○ ○

switch (1) {
case 1:
    printf("Play selected\n");
    break;
case 2:
    printf("Settings selected\n");
    break;
default:
    printf("Invalid choice\n");
}

// Output: Play selected
```

但是 Go 的处理方式有所不同，Go 的 switch 语句中的每个 case 自带一个隐式的 break。

这就意味着 Go 会在执行完匹配到的 case 后自动停止，无需显式添加 break 语句：

```
○ ○ ○

switch 1 {
case 1:
    fmt.Println("Play selected")
    fallthrough
case 2:
    fmt.Println("Settings selected")
default:
    fmt.Println("Invalid choice")
}

// Output:
// Play selected
// Settings selected
```

在 case 匹配后自动退出 switch 是 Go 语言的有意为之。

大多数情况下，这正是我们所希望的：一旦找到匹配项就停止执行，而继续执行到下一个case的情形很少见。Go语言的设计哲学是专注于最常见的需求，而非极端特例。

“如果我需要继续执行到下一个 case 怎么办？”

对于这种不常见的需求，Go 提供了**fallthrough**关键字，允许执行流程从当前 case 继续执行到下一个 case。

```
○ ○ ○

switch 1 {
case 1:
    fmt.Println("Play selected")
    fallthrough
case 2:
    fmt.Println("Settings selected")
default:
    fmt.Println("Invalid choice")
}

// Output:
// Play selected
// Settings selected
```

如果我们在循环内使用 switch 并且想要完全跳出循环，该怎么操作呢？

switch内部的break并不会影响外部的循环，如果我们希望基于switch中的条件退出循环，我们需要使用标签：

○ ○ ○

```
loop: // This is a label
  for {
    switch x {
      case "A":
        // This exits the loop because of the label
        break loop
    }
  }
```

使用标签，可以精确指定从哪里跳出，以达到从循环中退出的效果。

[探索Go中switch语句的6种使用方式](#) [技巧 #31：使用循环标签实现更清晰的break和continue操作](#)

Tip #47 表驱动测试，测试集和并行运行测试

原始链接：[Golang Tip #47: Table-driven tests, subtests, and parallel tests.](#)

如果我们的这个提示被忽略了，那可就不妙了，因为测试是确保我们在部署后，能够睡个安稳觉不可或缺的环节。

1. 表驱动测试

表驱动测试是一种通过表格的方式描述单元测试的方法，详细列出输入和预期结果。

让我们看一个简单的例子：我们有一个名为 `add()` 的函数，用于计算两个操作数的和。

以下是我们的测试用例：

○ ○ ○

```
testCases := [ ]struct {  
    a, b, expected int  
}{  
    {1, 2, 3},  
    {5, 0, 5},  
    {-1, -2, -3},  
    {-5, 10, 5},  
}
```

设置好这些之后，我们只需运行一个测试函数，就可以将所有的测试用例跑一遍：

```
○ ○ ○

func TestAdd(t *testing.T) {
    // ...testCases

    for _, tc := range testCases {
        got := add(tc.a, tc.b)

        if got != tc.expected {
            t.Errorf("add(%d, %d) = %d; want %d",
                    tc.a, tc.b, got, tc.expected)
        }
    }
}
```

就像上面那样，我们可以添加任意数量的测试用例，如果有任何失败，它们将很清晰地打印在控制台（或任何终端输出）上。

例如，将add()函数误认为是将a, b两个数相乘后，以下会生成一些失败的测试结果：

```
--- FAIL: TestAdd (0.00s)
    add(1, 2) = 2; want 3
    add(5, 0) = 0; want 5
    add(-1, -2) = 2; want -3
    add(-5, 10) = -50; want 5
```

“但如果有一个测试失败，我不想运行其余的测试，因为会很慢。”

我们可以使用 `t.Fatalf` 而不是 `t.Errorf`，它相当于 `t.Logf + t.FailNow`。

现在，我们还有一件事情忘记做了：给测试用例定义名称。

当测试失败时，命名变得非常重要。它能帮助我们快速定位哪个测试未通过，而无需仔细查看输入和预期结果。

2. 测试集和并行运行测试

测试集让你以逻辑方式组织测试，并将它们作为较大测试函数的一部分运行。

首先，让我们给每个测试用例一个名称：

```
○ ○ ○

func TestAdd(t *testing.T) {
    testCases := []struct {
        name      string
        a, b, expected int
    }{
        {"two positives", 1, 2, 3},
        {"positive and zero", 5, 0, 5},
        {"two negatives", -1, -2, -3},
        {"negative and positive", -5, 10, 5},
    }
}
```

然后，我们通过稍微修改，就能生成一个测试集，并进行并行运行，注意这两处更新：

```
○○○

for _, tc := range testCases {
    tc := tc // before Go 1.22

    t.Run(tc.name, func(t *testing.T) {
        t.Parallel() // run this subtest in parallel

        got := add(tc.a, tc.b)
        if got != tc.expected {
            t.Errorf("add(%d, %d) = %d; want %d",
                     tc.a, tc.b, got, tc.expected)
        }
    })
}
```

现在控制台上打印的结果非常清晰，它们以层次结构显示了哪个测试和哪个子测试失败：

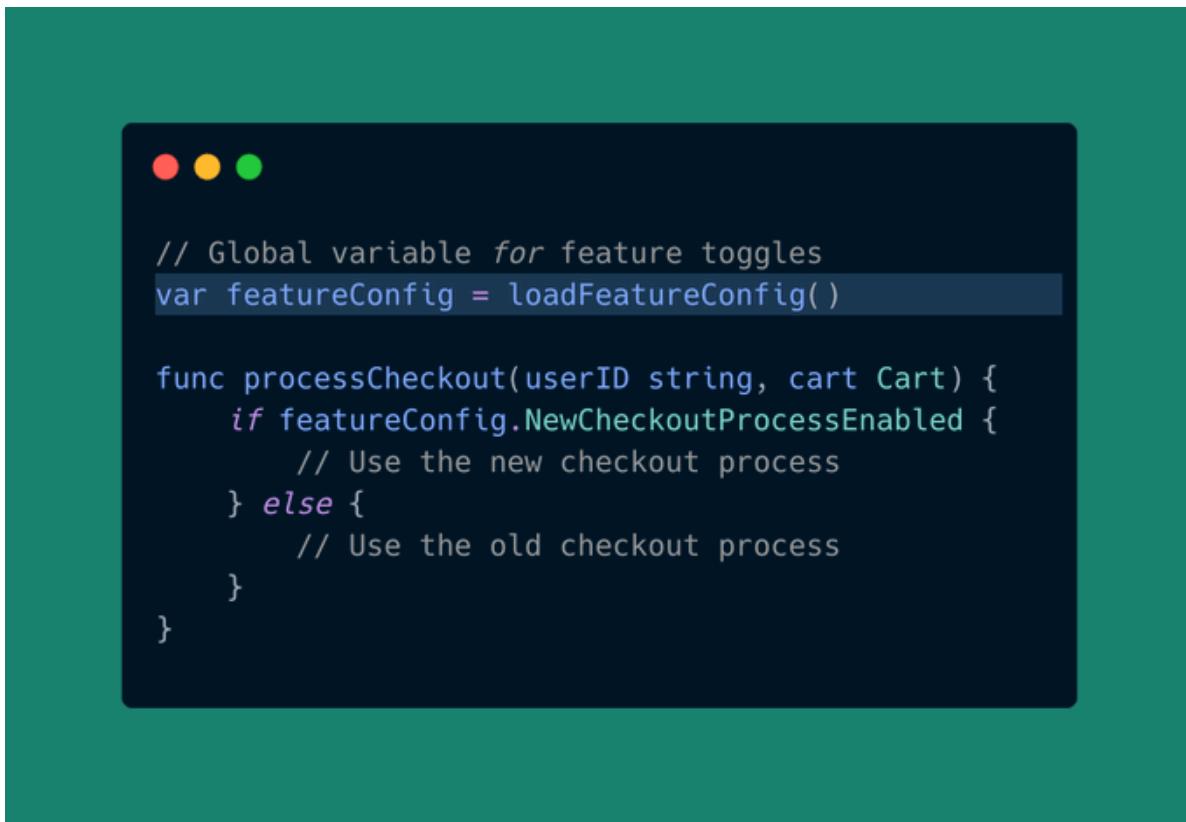
```
--- FAIL: TestAdd (0.00s)
    --- FAIL: TestAdd/two_positives (0.00s)
        add(1, 2) = 2; want 3
```

这种设置使你的测试输出清晰易懂，帮助你更快地识别和修复问题。

Tip #48 避免使用全局变量，尤其是可变变量

原始链接：[Golang Tip #48: Avoid Global Variables, Especially Mutable Ones.](#)

全局变量是我们放在函数或方法之外的变量，可供我们代码的任何部分使用和更改。



```
// Global variable for feature toggles
var featureConfig = loadFeatureConfig()

func processCheckout(userID string, cart Cart) {
    if featureConfig.NewCheckoutProcessEnabled {
        // Use the new checkout process
    } else {
        // Use the old checkout process
    }
}
```

现在，我并不是说所有全局变量都是坏消息，但它们带来的麻烦往往大于其价值。

原因如下（使用上面的代码示例）：

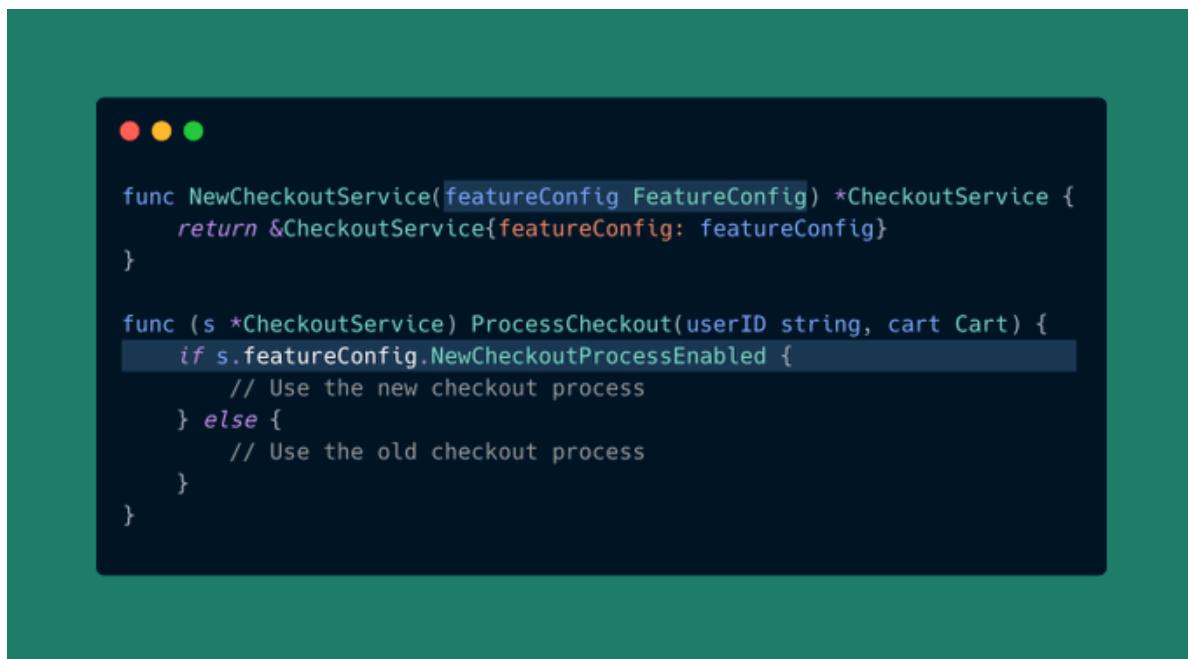
- **难以跟踪变化：**当代码的任何部分都可以改变 `featureConfig.NewCheckoutProcessEnabled` 时，识别它被改变的位置会很困难。

- **测试变得棘手**: 假设您正在测试新旧结帐流程。如果两个测试都涉及相同的全局 `featureConfig`，则您无法独立测试它们，因为其中一个测试会干扰另一个测试。
- **并发问题**: 当多个请求同时尝试读取或更改 `featureConfig` 时，可能会导致不一致（竞态条件）。

“那么，解决方案是什么？”

答案是依赖注入。

这是一种从外部满足对象需求的方法，而不是让它自己创建或从全局变量获取：



```
func NewCheckoutService(featureConfig FeatureConfig) *CheckoutService {
    return &CheckoutService{featureConfig: featureConfig}
}

func (s *CheckoutService) ProcessCheckout(userID string, cart Cart) {
    if s.featureConfig.NewCheckoutProcessEnabled {
        // Use the new checkout process
    } else {
        // Use the old checkout process
    }
}
```

是的，这种方法确实使事情变得有点复杂，但它也使得维护代码、测试代码和查找错误变得容易得多。

通过依赖注入，测试启用和禁用功能这两种场景变得非常简单：

```
func TestProcessCheckoutWithFeatureEnabled(t *testing.T) {
    featureConfig := FeatureConfig{NewCheckoutProcessEnabled: true}
    service := NewCheckoutService(featureConfig)
}

func TestProcessCheckoutWithFeatureDisabled(t *testing.T) {
    featureConfig := FeatureConfig{NewCheckoutProcessEnabled: false}
    service := NewCheckoutService(featureConfig)
}
```

但是如果您的全局变量不会改变，不需要测试并且必须这样工作，那么在这些情况下坚持使用全局变量可能会更好。

此外，如果您使用在运行时发生变化的全局变量，请确保使用同步技术（如互斥锁）来保证顺序。

简而言之，从全局状态转移到依赖注入可以让您的代码保持灵活性，并且不会过度依赖或紧密“耦合”。

Tip #49：赋予调用者决策权

原始链接: [Golang Tip #49: Give the Caller the Right to Make Decisions.](#)

想法是这样的：当你编写函数或包时，你必须决定：

- 如何管理错误，是打印日志还是触发 panic？
- 创建 goroutine 是否合适？
- 将上下文超时设置为 10 秒是个好主意吗？

关键原则是允许代码的使用者（调用者）拥有做出这些选择的控制权和职责，而不是在你的代码中为他们做出决定。

让我们看一些例子。

1. 处理错误

这是一个记录错误并将其返回给调用者的示例：

```
func DoSomething() error {
    // Attempt an operation
    if err != nil {
        log.Println("DoSomething's operation failed:", err)
        return err
    }
    return nil
}
```

与其在你的函数里触发 panic 或打印错误，不如只将错误发送给调用者。

这种方法可以让调用者找到最佳的处置方案，可以是记录错误，再试一次，或者在情况确实无法恢复时触发 panic。

“为什么不在我们的函数和调用者的函数里同时记录错误，以便更容易地追溯？”

这个想法与 [Tip #42：单次错误处理，减少噪音](#) 相关。

大体来说，我们让调用者根据自己的情况来确定错误的严重程度。

2. Goroutines

我见过一些用来运行后台任务的函数，像下图这样：

```
○ ○ ○

func fire() {
    go func() {
        // ... do something
    }()
}
```

当函数需要执行并发操作时，我们倾向于启动一个 goroutine。

但是，通常让调用者决定何时以及如何处理并发会更好，`go fire()`

上面的例子也缺乏控制，我们可能希望给予调用者权限来控制 goroutine 的生命周期。

3. 其他操作

这个原则不仅限于启动 goroutine 和错误处理，我之所以关注它们是因为它们与我们的语言密切相关。

它包括更多代码设计决策，它们都与允许调用者决定相关：

- 操作在超时前应等待多长时间
- 使用什么级别的日志记录
- 是否使用数据库事务

- 以及更多...

然而，就像软件开发中的许多技巧一样，一切都是权衡。你不会想用太多选项让调用者心智负担过重。

通过将关键决策交给调用者，您的代码将变得更具适应性、可重用性，并能够**适用不同的上下文**。

Tip #50 使结构体不可比较

原始链接: [Golang Tip #50: Make Structs Non-comparable](#)

在Go语言中,如果一个结构体中的每个字段都是可比较的,那么该结构体本身也是可比较的。这意味着你可以直接使用 `==` 和 `!=` 运算符来比较两个结构体:

```
○ ○ ○

type SimplePoint struct {
    X, Y float64
}

p1 := SimplePoint{1.0, 2.0}
p2 := SimplePoint{1.0, 2.0}

fmt.Println(p1 == p2) // True
```

但是,直接比较包含浮点数的结构体可能会有问题。理想情况下,浮点数值应该使用近似值进行比较。我们可能更希望团队成员使用自定义的 `.Equals` 方法进行比较:

○ ○ ○

```
func (p Point) Equals(other Point, tolerance float64) bool {  
    return math.Abs(p.X-other.X) < tolerance &&  
        math.Abs(p.Y-other.Y) < tolerance  
}
```

但让我们面对现实,使用 `p1 == p2` 是直接、快速的,对于不了解浮点数比较细微差别的任何人来说都是很诱人的。为了确保每个开发人员都使用您的比较方法,这里有一个零成本的策略:

○ ○ ○

```
type Point struct {  
    _ [0]func()  
    X, Y float64  
}
```

`[0]func()` 有3个特性:

- 非导出的: 对于你的结构体的使用者来说是隐藏的。
- 零宽度(或无成本): 因为长度为0, 所以这个数组在内存中不占用任何空间。
- 不可比较: `func()`是一个函数类型, 而函数在Go中是不可比较的。

试图直接使用 `==` 或 `!=` 比较这种结构体的两个实例将触发编译时错误: "invalid operation: a == b (struct containing [0]func() cannot be compared)"

但是, 请记住不要将[0]func()放在最后。

虽然它不占用空间,但它的位置仍然可能影响我们结构体的大小:

```
○ ○ ○

// 16 bytes
type Point struct {
    - [0]func()
    X, Y float64
}

// 24 bytes
type Point struct {
    X, Y float64
    - [0]func()
}
```

关于字段对齐(和填充)的进一步理解,可以参考以下两个资源:

- tips#41:按从大到小的顺序排列结构体中的字段。
(<https://colobu.com/gotips/041.html>)
- Go仓库中的Issue #9401: <https://github.com/golang/go/issues/9401>

Tip #51 避免使用init()

原始链接: [Golang Tip #51: Avoid using init\(\)](#)

`init()` 是一个特殊的函数, 它在主函数之前和全局变量初始化之后运行:

```
○○○

var precomputedValue float64

func init() {
    precomputedValue = math.Sqrt(2) * math.Pi
}

func main() {
    println("The precomputed value is:", precomputedValue)
}
```

它通常用于准备一些全局变量, 但最好保持全局变量的数量尽量少。

[Golang Tip #48 避免使用全局变量, 尤其是可变变量](#)

我们的示例无法将 `precomputedValue` 设置为常量值, 因为它依赖于在运行时计算的值 `math.Sqrt(2)`。

但是, 有一种更好的方法:

```
○ ○ ○

// SOLUTION 1
var precomputedSqrtPi = calculateSqrt2TimesPi()

func calculateSqrt2TimesPi() float64 {
    return math.Sqrt(2) * math.Pi
}

// SOLUTION 2
var precomputedSqrtPi = math.Sqrt(2) * math.Pi
```

“现在，为什么我们应该避免使用 `init()`？”

1. 副作用

`init()` 可以更改全局状态或引起其他意外效果。

这意味着仅仅添加一个包到您的程序中就可能改变您的程序的行为方式，使您更难理解正在发生的事情。

2. 测试挑战

您的测试可能因与您实际测试的内容无关的原因而失败或通过，只是因为其中一个导入的 `init()` 函数执行了一些意外操作。

3. 团队合作

无论您是在一段时间后重新访问自己的代码，还是深入研究其他人的代码，记住或找到 `init()` 函数在您的程序行为中扮演一个角色可能会很棘手。

4. 全局变量

`init()` 经常设置全局变量。

但是，对全局变量要谨慎，因为它们可以从代码的任何地方访问并进行更改，这通常是您想要避免的。

只要可以，让您的包的使用者处理设置（只要对他们不会不方便）。

这样，一切都更加开放，且更容易从外部进行管理。

5. 如果您想使用它，随意

现在，我们并不是要强烈反对使用 `init` 函数。

如果您仍然想使用 `init()`，因为：

- 您不希望用户每次都要调用 `yourpackage.Init()`。
- 为您的包注册一些钩子以使其工作（而不是更改其他内容）。
- 您的包依赖于环境变量进行操作（这是许多标准库的常见情况）。
- 您的全局变量需要根据不同的构建标签进行变化。
- ...

因此，这里有一些指导原则：

- 避免外部调用（包括 I/O）。
- 避免启动 goroutines。
- 不要依赖其他包的 `init()` 顺序。
- 保持其他包的全局变量不变。
- 考虑将您的 `init` 函数放在调整它的全局变量附近。
- 在全局变量上方添加注释，指明涉及哪个文件的 `init`。

最重要的是，要保持确定性，您的 `init()` 函数无论运行多少次都产生相同的结果。

Tip #52 针对容器化环境 (Kubernetes、Docker等) 调整 **GOMAXPROCS**

原始链接: [Golang Tip #52: Adjusting GOMAXPROCS for Containerized Env \(Kubernetes, Docker, etc.\)](#)

什么是GOMAXPROCS?

默认情况下, Go可以并发执行高达10,000个线程, 但实际上并行运行的线程数量取决于一个关键设置: GOMAXPROCS。

GOMAXPROCS决定了可以同时运行用户级Go代码的系统线程数量上限 (注意是真正的并行执行, 而不仅仅是并发)。

它的默认值与操作系统的逻辑CPU核数是一致的 (可通过`runtime.NumCPU()`函数获取):

```
○ ○ ○

func main() {
    fmt.Println(runtime.GOMAXPROCS(0))
    fmt.Println(runtime.NumCPU())
}

// Output:
// 8
// 8
```

例如在我的8核MacOS上， 默认情况下Go可同时处理多达8个线程。

容器化环境（Docker和Kubernetes）运行Go程序

在诸如K8s这样的容器化环境下， 我们可以为每个容器设置CPU限制， 实际上是在告诉容器：“你最多能使用这么多CPU资源”

例如， 限制参数为250m意味着可以使用1/4个CPU， 而1则表示可以使用1个CPU。

然而Go默认没法自行识别容器配置的CPU资源限制。它依旧会根据宿主机上的CPU核心总数进行运算， 而非容器分配到的数量。

（宿主机或节点上可以运行多个pod）

结果就是， Go程序可能会尝试使用超过其被分配份额的 CPU 。

“难道不是使用更多 CPU 资源更佳吗？”

这背后有几个原因：

1. 上下文切换：当线程数量超过CPU核心数量时，操作系统会频繁在多个线程间切换。
2. 调度效率低：Go的调度器可能会创建出比实际CPU限制下可执行的更多的goroutine，从而导致CPU时间的争夺。
3. CPU密集型任务的使用效率欠佳：Go程序通常是CPU密集型的，这意味着当每个线程可以分配到一个独立的CPU上执行，而无需等待时，它们的表现最佳。

如果我们设置的GOMAXPROCS超过了分配给容器的CPU核心数，就会迫使Go运行时规划超过实际可用核心数的线程，导致CPU密集型任务的执行效率降低。

解决方案是什么？

对于那些想要“省心”的开发者，[uber-go/automaxprocs](https://github.com/uber-go/automaxprocs)可能是个不错的选择。这个库可以自动调整GOMAXPROCS以适配容器的CPU限制。

(如果你想了解uber-go/automaxprocs背后的原理，请告知我)

```
○ ○ ○

import _ "go.uber.org/automaxprocs"

func main() {
    // Your application logic here.
}
```

另外，如果你对deployment或pod规范有所了解，你可以直接在环境变量中设置GOMAXPROCS，以匹配CPU限制。

但我更倾向于从DevOps的角度讨论这个问题，建议尽量避免设置CPU限制，而应始终指定CPU请求（详情稍后说明），这一点不只是针对Go服务。

如果你的容器没有明确的CPU限制，这是一个值得深思的问题。

- [求你了，请停止在Kubernetes上使用CPU限制（更新版本）](#)
- [Kubernetes CPU限制和Go](#)

Tip #53：枚举从1开始用于分类，从0用于默认情况

原始链接：[Golang Tip #53: Enums start from 1 for categorization and 0 for default cases](#)

Go语言并不原生支持枚举类型，但许多开发者都熟知一种普遍的替代方案。

让我们通过一个例子来详细解释这个技巧：

```
○ ○ ○

type UserRole int

const (
    Admin UserRole = iota // Admin=0
    Editor                 // Editor=1
    Viewer                 // Viewer=2
)
```

若一个UserRole变量被声明而未初始化，其默认值为0，这可能无意中将其设置为管理员角色。

这与我们的设想背道而驰。

以下是一条实用准则：

- 从1开始枚举是一种策略，确保了零值（Go中数值类型变量的默认值）不会错误地代表一个有意义的状态。
- 当每个新创建的实例都自然而然地对应一个有意义的初始状态时，从0开始枚举是可取的。

一个更好的方案可能是这样的：

```
○ ○ ○

type UserRole int

const (
    Admin    UserRole = iota + 1 // Admin=1
    Editor               // Editor=2
    Viewer               // Viewer=3
)
```

或者你可以考虑另一个解决方案，即在发生错误的情况下，其中在发生错误的情况下，一个名为'Unknown'（未知）的角色作为默认值。

"我倾向于将默认值设为Viewer（观察者）是一个更好的选择。"

然而，情况并非总是如此。由于多种原因，我们需要考虑以下场景：

- 一个Editor（编辑者）由于默认值设置不当，可能被错误地赋予了Viewer（观察者）的角色，这可能引起潜在的逻辑错误。
- Viewer（观察者）角色可能会让某人拥有超出Editor（编辑者）角色所需的查看权限。对于只需要处理特定内容的编辑者来说，这可能并不适合，因为他们不需要像Viewer那样可以查看所有内容。

因此，我们使用0来检测代码中的异常，并防止可能出现的任何风险。

考虑以下例子以进一步说明：

- 应用模式（开发、测试、生产）
- 状态（成功、错误、待处理）
- 行为（登录、登出、购买）

在这些情况下，每个状态都具有同等的重要性，意味着没有一个默认或优先状态。

但当存在一个明确的默认状态时，创建一个值为零的枚举是完全合适的：

```
○ ○ ○

type ConnectionState int

const (
    Disconnected ConnectionState = iota
    Connecting
    Connected
    Failed
)
```

决定枚举值从0还是1开始，取决于我们的具体业务需求和安全性考虑。

Tip #54 仅在必要时为客户端定义 error(var Err = errors.New)

原始链接: [Golang Tips #54: Only define errors \(var Err = errors.New\) when it's necessary for your client](#)

一个在很多代码库中都有的常见误区是, 为每个逻辑错误定义一个error, 每个error都有一个高度描述性的名称和说明。

这种做法并不是总是必要的:



```
var (
    ErrPriceTooHigh = errors.New("sale: input price is too high")
    ErrPriceTooLow = errors.New("sale: input price is too low")
    ErrAlreadySale = errors.New("sale: item already in sale")
)

func Sale(price int) error {
    // ....
    if isPriceHigh(price) {
        return ErrPriceTooHigh
    }
}
```

这通常出现在处理我们的业务逻辑的时候, 开发人员想要控制每一个error, 但是这是多余的。

这里列举一些弊端 (以上述例子为例):

- 这对于维护者是一个负担, 他们必须记住或者查询每一个error的细节。
- 除了你自己之外没人知道这是冗余的, 甚至不久的将来你自己也会忘记。

- 你的客户端可能不需要知道这些错误，因为前端早已限定了输入价格从高到低的区间了。

只有当客户端绕过前端直接使用API时才会遇到这个error（这是一个我们不认可的行为）。

这个原则不仅仅适用于客户端-服务端通信中，还适用于内部代码。

例如，如果你无法向消息队列中投递一个消息时，别立即就创建一个ErrPublishMessage。很可能没有人会捕获这个error。

那么，这些场景下推荐的做法是什么呢？

当你的代码不需要客户端（不管是你的代码的一部分还是你写的库的外部用户）根据error类型的不同而采取不同的行为时，采取最简单的错误处理方法是最佳的：



```
func Sale(price int) error {
    // ....
    if isPriceHigh(price) {
        return errors.New("the price is too high")
    }
}
```

或者使用fmt.Errorf，它能让你利用动态数据格式化出一个错误类型。

这在需要包含上下文相关的信息时候特别有用。

```
func Sale(price int) error {
    // ....
    if isPriceHigh(price) {
        return fmt.Errorf("price (%v) is too high, cap (%v)", price, cap)
    }
}
```

当你的应用逻辑确实需要根据error类型的不同而要采取不同的行为时，例如：

- 根据error类型决定是否重试一个操作。
- 记录特定的error到日志中去。
- 通知用户他们的资金即将耗尽，或者显示一个充值弹窗。
- ...

那么对于这些场景定义error变量才是完美的。

Tip #55 使用空字段防止结构体无键字面量

原始链接: <https://twitter.com/func25/status/1770792643892519079>

这里的主要目的是确保当有人使用您的包并决定创建您的结构体的一个实例时，他们必须在其结构体字面量中使用**命名字段**。

假设我们有一个带有两个字段的结构体 Point: X 和 Y。

想象一下，您的库的用户像这样定义一个 Point (没有 X 和 Y)：

```
// package math
type Point struct {
    X float64
    Y float64
}

a := math.Point{1, 4}
```

当然，这种方法对于常见的 Point 结构体来说并不是问题，因为它通常只包含 X 和 Y。

但是，如果您计划为它添加更多字段，比如一个字符串 label 字段呢？

在这种情况下，用户没有适应这种变化的代码将触发编译时错误：“struct literal of type config.Point 中值太少”，并且它不是向后兼容的。

这里有一个策略来鼓励客户端显式定义 X 和 Y：向结构体中添加一个特殊的变量，该变量：

- 非导出。
- 是大小为零的字段。

大小为零的类型的常见选择是空结构体 (struct{}) 和长度为零的数组。

```
// package math
type Point struct {
    _ struct{}
    X float64
    Y float64
}

a := config.Point{X: 1, Y: 2}
```

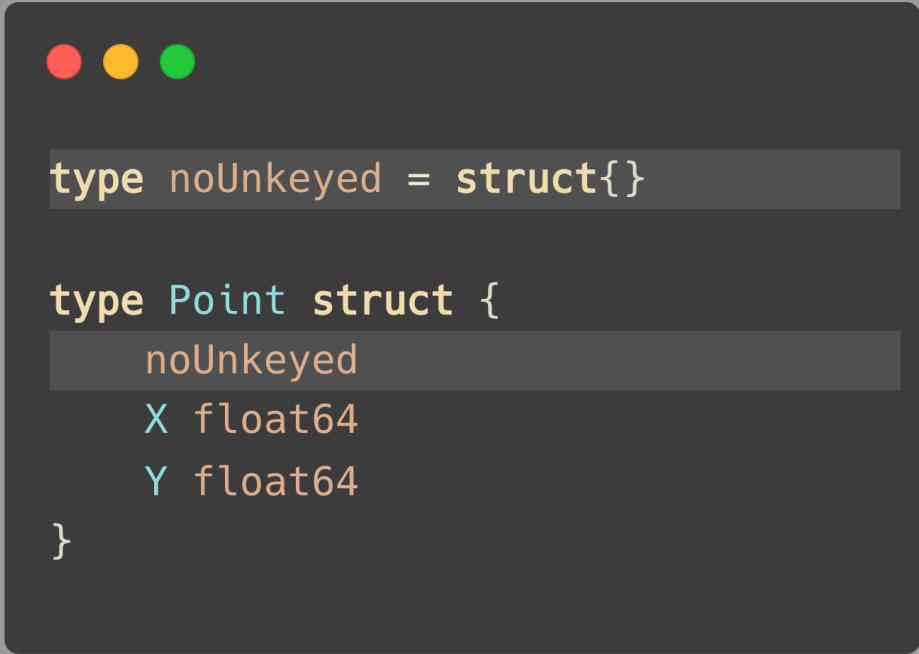
这个解决方案类似于说，“这个 Point 不仅包括 X 和 Y，将来还会有更多。”

“为什么是一个非导出的、零大小的字段？”

这是一个技巧，非导出的字段使得它在该包外部不可访问。零大小的字段不会增加结构体的内存大小。

如果您不喜欢 `_ struct{}` 语法，因为它在防止无键字面量方面的意图不够明确，可以考虑来自 [@nalesnikowydzem](#) 在提示 #50 中的建议 (<https://twitter.com/func25/status/1768621711929311620>)：

译者注：他的建议是定义一个类型并嵌入到你的结构体中。类似于标准库中的 `noCopy{}` 结构体。



```
type noUnkeyed = struct{ }

type Point struct {
    noUnkeyed
    X float64
    Y float64
}
```

依赖 linter 可能会捕获这个问题，但并非所有客户端都会使用 linter。

然而，在某些情况下，使用无键字面量可能是更受欢迎的，例如，在处理只包含 Key 和 Value 字段的映射元素时（例如 mongodb 中的 bson.E）。

如果确定这些不会改变，一致地指定 Key 和 Value 可能会降低代码的可读性。

Tip #56 简化接口并只要求你真正需要的东西

原始链接: [Golang Tip #56: Simplify interfaces and only ask for what you really need](#)

在Go中定义接口时, 请遵循以下技巧:

1. 只有在实际需要时才定义接口。 2. 接受接口并返回具体类型。 3. 将接口放在它们被使用的地方 (消费者), 而不是它们被创建的地方 (生产者)。 (Tip #18)

技巧3 实际上是基于技巧1的。

但是还有一个额外的技巧, 让我们遵循"2. 接受接口并返回具体类型。"

你能在下面的例子中发现一个问题吗?

```
type UserProfileManager interface {
    CreateUser(profile UserProfile) error
    UpdateUser(id string, profile UserProfile) error
    GetUser(id string) (UserProfile, error)
}

func LogUserDetails(manager UserProfileManager, userID string) error {
    user, err := manager.GetUser(userID)
    if err != nil {
        return err
    }

    log.Printf("User details: %+v", user)
    return nil
}
```

上述示例仅用于展示提示的目的，可能不符合最佳实践、命名规范等。

问题是，`LogUserDetails(...)` 函数仅仅需要 `GetUser` 方法，而不需要 `CreateUser` 或 `UpdateUser` 方法。

这种设置并不理想，因为它将函数与一个广泛的接口绑定在一起，这使得测试变得更加困难，降低了灵活性，并且可读性较差。

接口有助于实现抽象，但是**接口越庞大，它就变得越不抽象**。

“为什么这对测试来说是不利的？”

当我们进行测试时，我们不应该搞清楚输入接口的哪些方法被使用了，对吧？

而且，设置一个我们不需要的庞大的 Mock 对象也很麻烦。

那么，怎么做更好呢？

你可能已经猜到了，我们的函数应该请求一个**只有它需要的东西的接口**：

```
● ● ●

type UserGetter interface {
    GetUser(id string) (UserProfile, error)
}

func LogUserDetails(getter UserGetter, userID string) error {
    user, err := getter.GetUser(userID)
    if err != nil {
        return err
    }

    log.Printf("User details: %+v", user)
    return nil
}
```

任何符合 `UserManager` 的具体类型也同样应该符合 `UserGetter`。

这个简单的改变使得代码更易于测试，也更清晰。

Tip #57: Go中的标记枚举

原始链接: [Golang Tip #57: Flag Enums in Go](#)

在 Go 语言中，枚举的定义并不像某些其他语言那样直接明了。

反而，Go提供了一个巧妙的方式，通过使用一个包含iota关键词的常量群组来创造出具有枚举行为的结构。

我们已经做了基础的介绍；现在让我们来探讨枚举所面临的问题：

```
● ● ●

type BasicInfo int

const (
    IsBoolean  BasicInfo = iota + 1 // 1
    IsInteger                  // 2
    IsUnsigned                 // 3
    IsFloat                    // 4
    IsComplex                  // 5
    IsString                   // 6
    IsUntyped                  // 7
)
```

为了从 1 开始我们的枚举项，我们使用了 `iota + 1` 的方式。。

当我们需要为特殊情况保留零值，或者零值在我们的上下文中没有实际意义时，这种方法会非常有用，正如我们在技巧#53中所讨论的。

在上述例子中，如果一个类型同时可以是整数和无符号数该怎么办？

当然，我们可以定义一个新的类型，比如UnsignedInteger，但随着我们增加更多的组合类型，这种方法的可扩展性并不理想。

标记枚举 (Flag Enums)

在实际开发中，我们可能更倾向于使用位掩码或标志枚举。

这是一种使用枚举的新方法，它允许我们将多个状态或属性合并到单个变量中：

```
const (
    IsBoolean  BasicInfo = 1 << iota // 1
    IsInteger               // 2
    IsUnsigned              // 4
    IsFloat                 // 8
    IsComplex               // 16
    IsString                // 32
    IsUntyped              // 64
)
```

通过使用标志枚举，我们可以同时标记某个变量具有 IsInteger 和 IsUnsigned 属性，而无需为此创建新的类型。

我们利用位或运算符 `|` 表明`unsignedInteger`同时拥有这两种属性：

```
var unsignedInteger BasicInfo = IsInteger | IsUnsigned
```

为了增强代码的可读性，你可能会选择以下方式定义新的枚举：

```
const (
    IsBoolean  BasicInfo = 1 << iota // 1
    IsInteger              // 2
    IsUnsigned             // 4
    IsFloat                // 8
    IsComplex              // 16
    IsString               // 32
    IsUntyped              // 64

    IsOrdered   = IsInteger | IsFloat | IsString // 42
    IsNumeric   = IsInteger | IsFloat | IsComplex // 26
    IsConstType = IsBoolean | IsNumeric | IsString // 59
)
```

这段代码实际上是从 Go 语言的源代码中直接摘录的。

Tip #58 将互斥锁放在保护的数据附近

原始链接: [Golang Tip #58: Keep the mutex close to the data it's protecting](#)

将mutex紧密放置在它所保护的对象旁边是一个很好的做法, 这样做可以确保我们在完全知情的情况下锁定和解锁它。

让我们通过一些示例来看看这在实践中是如何工作的:

```
type UserSession struct {
    mu        sync.Mutex
    ID        string
    Preferences map[string]interface{}
    LastLogin  time.Time
    Profile    *UserProfile
    Cart       []string
    isLoggedIn bool
}
```

你认为互斥锁保护哪些字段?

它可能是"preferences", 也可能是"cart", "profile", 其他字段, 或者甚至是全部。

关键是要安排好你的结构体，以便我们一目了然，无需深入查看代码。

“但是你不是说要按照从大到小的顺序排列结构体的字段吗？”

你可能还记得 [Golang Tip #41：将结构体的字段按照从大到小的顺序排列。](#)

然而，如果我们阅读 Tip #41 的最后一部分，我添加了一个关于易于阅读与优化之间权衡的关键说明。

因此，我通常会使用空行将相关字段分组，而不是严格遵循大小顺序。

解决方案

再次看看我们的示例，答案就在提示的标题中：“将互斥锁放在保护的数据附近”。

但是要用空行将它们与其他字段分开：

```
type UserSession struct {
    ID           string
    LastLogin    time.Time
    isLoggedIn  bool

    mu          sync.Mutex
    Preferences map[string]interface{}
    Cart        []string

    Profile    *UserProfile
}
```

通过将互斥锁紧密放置在它们的上方，我们向团队（以及未来的自己）表明，这些字段"preferences"和"cart"需要以对多个 goroutine 安全的方式进行访问，使用互斥锁。

这个想法不仅适用于结构体。

它也可能适用于全局变量或只能一次调用的函数：

```
var (
    mu     sync.Mutex // Protects the global count.
    count  int
)

func IncrementCount() { ... }

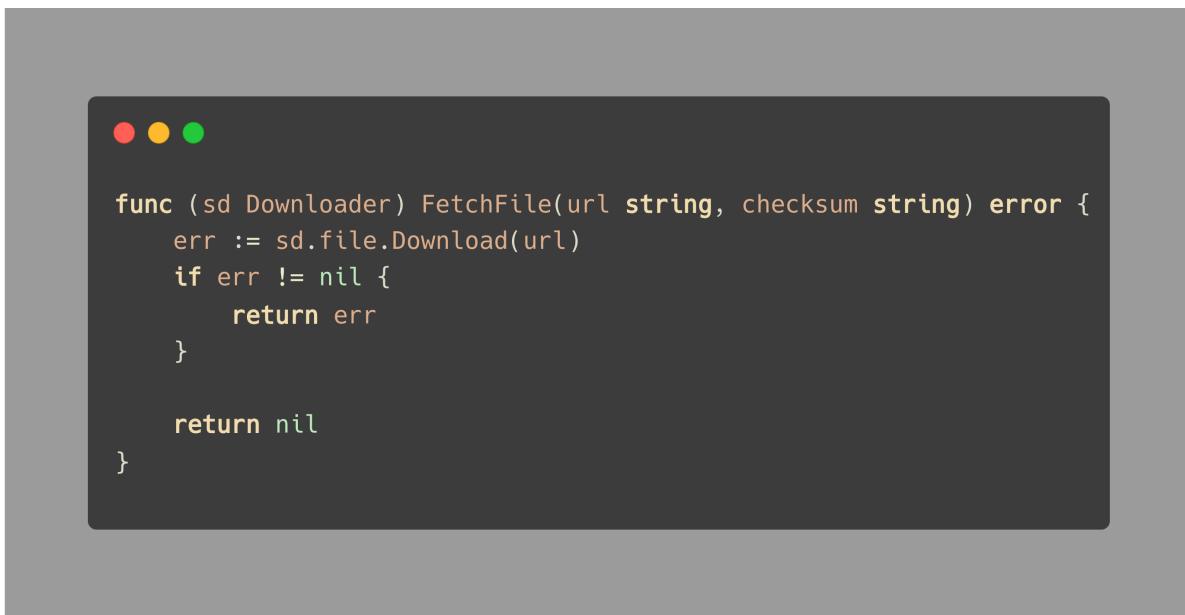
func GetCount() int { ... }
```

让我们不要担心关于全局变量的争论，这只是为了向你展示这个提示的含义。

Tip #59 如果不需要使用某个参数，删除它或是显式地忽略它

原始链接：[Golang Tip #59: If a parameter isn't needed, either drop it or ignore it on purpose](#)

在我们深入讨论这个技巧之前，让我们剖析下方例子中出现的问题：



```
func (sd Downloader) FetchFile(url string, checksum string) error {
    err := sd.file.Download(url)
    if err != nil {
        return err
    }

    return nil
}
```

在这个例子中，`FetchFile` 函数的参数是 `URL` 和一个文件的校验和 `checksum`。然而，我们只使用了 `URL` 来获取文件，没有使用到 `checksum`。

这个例子的问题在于，与局部变量不同，编译器不会告诉你是否忘记在函数中使用了一个参数。

因此，尽管校验和 `checksum` 理应用于检查文件是否完整，我们还是无法确定 `checksum` 是被意外遗漏了，还是被有意地省略了。

解决方法

我们有两种处理方式：

- 使用下划线 `_` 来故意忽略该参数。
- 删除未使用的参数。

让我们使用这个技巧来改进函数：



```
func (sd Downloader) FetchFile(url string, _ string) error {
    err := sd.file.Download(url)
    if err != nil {
        return err
    }

    return nil
}
```

通过将 `checksum` 参数替换为下划线 `_`，可以清晰地表示我们是故意省略这个参数的。根据不同的需求，我们可以使用多个下划线 `_` 来省略多个参数。

"为什么不直接删除这个参数呢？"

如前文所述，删除该参数是该问题的解决方法之一。

但是，有时我们必须遵循特定的模式，比如遵循接口定义或者特定的函数定义：

```
type FileDownloader interface {
    FetchFile(url string, checksum string) error
}
```

当然，如果在参数列表中使用了过多的下划线 `_`，这可能意味着我们的设计本身就存在着问题。

Tip #60 sync.Once是执行单次操作的最佳方式

原始链接: [Golang Tip #60: sync.Once is the best way to do things once](#)

当我们处理单例时,这种解决方案是很常见的。有时单例会带来问题,但在某些情况下,它们是可以接受的。我们将讨论:

- 问题所在
- 如何修复
- 一种误解
- sync.Once实际工作方式

假设我们有一个配置对象,需要在第一次被调用时进行一次设置,之后就可以共享使用。下面是一种简单的实现方式:

```
var instance *Config

func GetConfig() *Config {
    if instance == nil {
        instance = LoadConfig()
    }

    return instance
}
```

但是,当同时发生很多事情时,这种方式存在问题。如果在配置尚未设置时,许多goroutine试图同时获取配置,我们可能会多次执行 `LoadConfig()`。这并不是我们想

要的。现在,让我们看看sync.Once是如何提供帮助的:

```
var (
    once      sync.Once
    instance *Config
)

func GetConfig() *Config {
    once.Do(func() {
        instance = LoadConfig()
    })
}

return instance
}
```

我们创建一个设置配置并将其传递给 once.Do(f) 的函数。即使有很多goroutine同时调用 GetConfig() , sync.Once 也能确保我们传递给 .Do(..) 的设置函数只运行一次。

sync.Once不能被重复使用

关键是,如果我们试图再次使用sync.Once,并传递另一个函数,它将不起作用。就像这样:

- o .Do(f1)
- o .Do(f2)

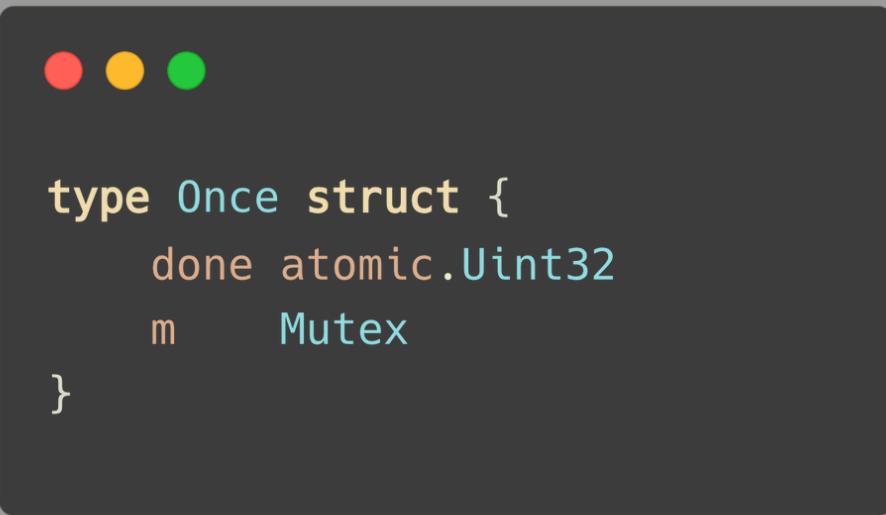
第二个函数f2将被忽略,尽管它与f1不同。

它是如何工作的?

一个sync.Once会跟踪两件事:

- 一个原子计数器(或标志),有0和1两个值。
- 一个用于保护慢路径的互斥锁。

我们将稍后讨论所谓的快路径和慢路径,但先让我们看看sync.Once是如何构建的:



```
type Once struct {
    done atomic.Uint32
    m     Mutex
}
```

快速路径

当调用 `Do(f)` 时, 它首先查看原子计数器。如果计数器为0, 则表示函数尚未执行。

这个快速通道的存在是为了当函数已经执行过时, 后续的调用可以快速跳过并无需等待。

慢速路径 若计数器不为0, 则会触发慢速路径。sync.Once 会进入慢速模式并调用 `doSlow(f)` :

```
func (o *Once) doSlow(f func()) {
    o.m.Lock()
    defer o.m.Unlock()

    if o.done.Load() == 0 {
        defer o.done.Store(1)
        f()
    }
}
```

译者注：这里作者说错了，计数器为0 (`o.done.Load() == 0`) 才会进入慢速路径

- `o.m.Lock()`：通过互斥锁确保同一时间只有一个goroutine能够执行接下来的步骤。
- `o.done.Load() == 0`：获得锁之后，再次检查计数器以确保在此期间没有其他goroutine抢先执行。
- `o.done.Store(1)`：为了确保我们知道函数已完成，在函数执行完毕后更新计数器。

“为什么要设置快速路径和慢速路径？”

我们采用快速路径和慢速路径是因为我们希望建立既能尽可能快速又能必要时确保安全的机制。

慢速路径仅仅是用于初始化阶段，这个阶段很短暂。一旦初始化完成，每次调用 `.Do` 都会变得很快，这对长远性能而言是有益的。

Tip #61 使用内置锁的类型 (`sync.Mutex` 嵌入)

原始链接: [Golang Tip #61: Making a Type with Built-In Locking \(`sync.Mutex` embedding\)](#)

写代码时, 我们经常需要让程序的多个部分同时被访问, 为此我们通常使用 `sync.Mutex` 来确保安全性。

下面是一种常见的做法:



```
type MyStruct struct {
    mu sync.Mutex

    // other fields
}

func (s *MyStruct) DoSomething() {
    s.mu.Lock()
    defer s.mu.Unlock()

    // critical section
}
```

但这种解决方案会使我们的代码中充斥着 `.mu.Lock()` 和 `.mu.Unlock()` 的调用。

为了简化代码，我们可以将 `sync.Mutex` 直接嵌入到结构体中。现在我们只需在结构体本身上调用 `Lock` 和 `Unlock`：

```
● ● ●  
  
type MyStruct struct {  
    sync.Mutex  
    // other fields  
}  
  
func (s *MyStruct) DoSomething() {  
    s.Lock()  
    defer s.Unlock()  
  
    // ...  
}
```

但请注意：如果 `MyStruct` 是公共的（以大写字母开头），直接添加 `sync.Mutex` 会使其 `Lock` 和 `Unlock` 方法也变为公共方法。

因此，对于不打算在包外共享的类型，这可能是一个更好的选择。

通用技巧

在查看一些想法和在线资源时，我发现了这个巧妙的方法。

要创建一个已准备好进行锁定的类型，您可以使用通用类型：

```
type Lockable[T any] struct {
    sync.Mutex
    value T
}

func (l *Lockable[T]) Get() T {
    l.Lock()
    defer l.Unlock()

    return l.value
}

func (l *Lockable[T]) Set(v T) {
    l.Lock()
    defer l.Unlock()

    l.value = v
}
```

(事实证明，我们无法将 `T` 嵌入到 `Lockable` 中。)

通过这种方式，我们可以保护任何类型不被同时用于太多地方。您可以直接使用它，或者创建一个新类型：



```
// directly use
func main() {
    var safeUser Lockable[User]
    safeUser.Set(...)
}

// new lockable type
type LockableUser Lockable[User]
```

Tip #62 context.Value不是我们的朋友

原始链接: [Golang Tip #62: context.Value is not our friend](#)

让我们谈谈 Go 中的一个糟糕模式, 特别是对于那些刚接触 Go 语言的人 (包括我自己)。

随着我们的 Go 应用程序变得越来越大, 我们经常发现自己需要在多个函数之间共享数据:

```
func A(ctx context.Context, transactionID string) {
    payment := db.GetPayment(ctx, transactionID)
    ctx = context.WithValue(ctx, "payment", payment)

    B(ctx)
}

func B(ctx context.Context) {
    // ...
    C(ctx)
}

func C(ctx context.Context) {
    payment, ok := ctx.Value("payment").(*Payment)
    // ...
}
```

如图中示例，函数 A 可能会从数据库中获取付款记录 `payment` 并将其添加到上下文 `context` 中，然后经过多次调用传递后，才在函数 C 中被读取使用。

这看起来很好，因为：

- 它让我们跳过将数据传递给不需要它的函数（例如函数 B）的步骤。
 - 它允许我们在 `context` 中存储和读取所有需要的数据。
 - 无需添加额外的函数参数。
-

“为什么不直接从函数 A 调用函数 C？”

存在很多场景会导致链式调用；例如，函数 C 是函数 B 中逻辑的一部分，需要函数 B 中的一些数据做为参数。

那么使用 `context` 传递数据的问题出在哪里呢？

以下就是我们遇到的一些问题：

- 我们放弃了 Go 在编译期间提供的类型检查安全性。
- 我们将数据放入黑盒中并希望稍后再获取它，但一周过后你可能就真的要像瞎子那样的去黑盒中搜索它了。
- 放入 `context` 中会使得 `payment` 数据看似是可选的，然而实际上它很重要。

在我看来，使用 `ctx.Value` 最大的问题是它太隐式了。数据被隐藏在 `context` 中，没有任何清晰的跟踪。

使用隐式可能不是一个坏主意，但却很难成为一个好主意。

...

所以，我们什么时候应该使用 `context.Value()`？

我建议尽可能的避免它。然而，Go 官方文档确实提到它对于“跨 API 边界 和 进程之间 传递 请求相关的值”很有用。

所以您可能会考虑使用 `context` 来跟踪与请求相关的特定数据，例如：

- 开始时间
 - 调用者的 IP
 - Trace 和 span ID
 - 被调用的 HTTP 路由
 - ...
-

“ payment 不是请求相关数据的一部分吗？”

很明显，我们的函数无需读取请求体即可处理 payment。如果调用链中的大多数函数都需要 payment 数据，那么显式传递它会更好。

综上，我会避免在上下文中传递业务数据。

Tip #63 避免使用time.Sleep(), 它不能被context感知且无法被中断

原始链接: [Golang Tip Tip #63: Avoid time.Sleep\(\), it's not context-aware and can't be interrupted.](#)

使用 `time.Sleep` 可能看起来很有用, 但它不能被context 感知, 也无法被中断。

例如, 如果我们的应用程序正在关闭, 我们无法向正在休眠的函数发送信号。一旦它醒来, 它会开始执行一些逻辑, 但在工作时可能会被中断。

我自己也犯过这个错误, 并将其作为学习的一点:

```
func doJob() {
    for ;; time.Sleep(5 * time.Second) {
        // some work
    }
}

func doJob() {
    for {
        // some work
        time.Sleep(5 * time.Second)
    }
}
```

这些循环执行一些工作，然后使用 `time.Sleep` 暂停 5 秒钟，然后继续执行。

但是，这些循环无法通过 `context cancel` 来停止，如果我们需要快速停止一切，这是一个问题。

让我们传递 `context` 以使我们的工作考虑到 `context cancel`：

```
func doWork(ctx context.Context) {
    for {
        select {
        case <-ctx.Done():
            return
        default:
            // doWork(ctx)
            time.Sleep(5 * time.Second)
        }
    }
}
```

这样做更好一些；虽然有点冗长，但我们的工作现在尊重上下文，例如：

- 1. -> doWork -> sleep -> shutdown -> ctx.Done() -> 结束。
- 2. -> doWork -> shutdown -> sleep -> ctx.Done() -> 结束。

(我们可以将 `doWork(ctx)` 放在 `for` 循环的正下方，以减少嵌套。)

但是，我们仍然需要等待 5 秒钟，或者根据作业延迟的情况可能更长。

使用**time**包作为信号

现在，我们正在讨论的解决方案使用了**time**包，但有一些微妙之处。

第一个修复涉及 `time.After`，在循环中如下所示：

```
for {
    //... Our intended operation

    select {
        case <-time.After(5 * time.Second):
        case <-ctx.Done():
            return
    }
}
```

这种方法简单而直接，但并不完美。

- 我们每次都在分配一个新的channel。
- Go 社区指出，`time.After` 可能会导致短期内内存泄漏。

如果我们的函数在倒计时之前因为 `ctx.Done()` 而结束，那么 `time.After` 将一直存在，直到时间到期。

第二个修复可能更冗长，但旨在解决这个问题：

```
delay := time.NewTimer(5 * time.Second)

for {
    // ... Our intended operations

    select {
        case <-delay.C:
            _ = delay.Reset(5 * time.Second)
        case <-ctx.Done():
            if !delay.Stop() {
                <-delay.C
            }
            return
    }
}
```

我们设置了一个计时器，当它到期时，我们重新启动它。

当context完成时，重要的是我们停止计时器以避免泄漏，可以使用与上面相同的解决方案，或者使用 `defer delay.Stop()`。

最后的解决方案并不完美

将我们的工作直接放在 `for` 循环下面可以减少代码的嵌套。但这并不能确保计时器第一次就是正确的。

您可能希望将我们预期的逻辑放在 'case <- delay.C' 下面。

Tip #64 让main()函数更清晰并且易于测试

原始链接: [Make main\(\) clean and testable](#)

通常, 我们会在 main() 函数中执行许多不同的任务, 例如:

- 设置环境、JSON 配置
- 连接到数据库或 Redis 缓存
- 创建与消息队列的连接或与其他服务进行链接
- ...

当出现问题时, 我们通常会使用 log.Fatal 来立即停止程序 (我自己通常避免使用 panic() 或 os.Exit(.)) :

```
func main() {
    conf, err := config.Fetch()
    if err != nil {
        log.Fatal("Failed to fetch config:", err)
    }

    if err := connectDB(); err != nil {
        log.Fatal("Failed to connect to database:", err)
    }

    // Additional logic that might also call log.Fatal...
}
```

如果您正在这样做，这并不一定是一个坏的解决方案，尤其是如果你不需要想我一样精细的控制。

但是，如果我们想要改进它呢？以下是一些目标：

- 立即使用`log.Fatal`停止程序，这意味着那些应该在代码完成后执行的函数（`defer`函数）都不会执行。
- 我希望能够更改参数（例如`os.Args`）或者修改安装程序的环境，以便在测试时覆盖到不同的场景。
- 我不想在我们遇到的每个错误都使用`log.Fatal()`
- 我想决定应用程序如何结束，是报错退出（退出代码 1）还是没有任何问题的退出（退出代码 0）。

此外，如果您还没有这样做，请查看[Golang Tip #43: Gracefully Shut Down Your Application](#)。

因此，让我们将所有的设置工作从 `main` 函数中移出：

```
○ ○ ○

func main() {
    if err := run(os.Args[1:]); err != nil {
        log.Fatal(err)
    }
}

func run(args []string) error {
    conf, err := config.Fetch(args)
    if err != nil {
        return fmt.Errorf("failed to fetch config: %w", err)
    }

    db, err := connectDB(conf.DatabaseURL)
    if err != nil {
        return fmt.Errorf("failed to connect to database: %w", err)
    }
    defer db.Close() // This will now be called when run() exits.

    // ...

    return nil
}
```

现在，我们已经实现了所有的目标，main()函数现在只是替run函数把东西准备好。

我们可以用不同的参数集来测试“run函数”部分是如何独立工作的。

我们可能想要做的不仅仅只是返回一个错误 (error)，比如返回一个特定的退出代码 (errCode)。这样，我们可以使用os.Exit和这个退出代码来结束程序。

○ ○ ○

```
func run(args []string) (errCode int, err error) {  
    // ...  
}
```

我们让主函数来处理错误，这样我们就不用为每种情况写一个log了。

Tip #66 在fmt.Errorf中简化你的错误信息

原始链接: [Golang Tip #66: Simplify Your Error Messages in fmt.Errorf](#)

在Go语言中, 当我们处理错误时, 提供足够的详细信息以便了解问题的具体原因是非常重要的。

之前有一个提示, 即 Go Tips#38, 专门讨论了这个问题。我们可以通过访问以下链接获取更多上下文信息: [tips#38](#)

(感谢 `@thedenisnikulin` 提供的额外建议, 我们可以进一步改进错误处理方式。)

现在, 我们应该熟悉使用 `fmt.Errorf` 函数以及 `%w` 来包裹 (wrap) 错误的做法:

```
○○○  
if err != nil {  
    return fmt.Errorf("failed to open file %s: %w", filename, err)  
}
```

通常情况下, 我们可能会得到一个像这样的长错误消息:

"error while crawling: can't retrieve log: failed to open file server-logs.txt: file not exist."

这个消息虽然清晰, 但比实际所需冗长, 因为它重复了一些诸如"error while"、"failed to"这样的短语, 而我们知道我们在处理错误时, 这些前置词汇是可以省略的。

因此，这里有一种更好的做法：

```
○○○  
  
if err != nil {  
    return fmt.Errorf("open file %s: %w", filename, err)  
}
```

注意到不同之处了吗？

相比于冗长的消息，我们得到的是更简洁、更直接的内容：“crawling: retrieve log: open file server-logs.txt: file not exist.””

这条消息仍然清楚地告诉你错误发生时正在进行的操作，并且更容易阅读。

因此，在Go语言中编写错误消息时，要保持简短，重点突出未能成功执行的动作。

Tip #65 使用泛型返回指针

原始链接: [Golang Tip #65: Returning Pointers Made Easy with Generics.](#)

给那些经常需要指向某个值的指针的 Go 程序员的一个小提示。

以前, 你可能做过类似的事情:

```
○ ○ ○  
// standard way  
result := getData()  
ptr := &result
```

或者, 你可能用一个小技巧在一行内完成所有操作:

○ ○ ○

```
// or, for a one-liner, using an inline anonymous
function
ptr := func(t Data) *Data { return &t }(getData())
```

这种方法很好用，但稍微有点长，特别是当你经常使用不同类型数据的时候。

现在，让我们看看使用泛型的更新、更简单的方法：

○ ○ ○

```
func Ptr[T any](v T) *T {
    return &v
}
```

通过这个小函数，你可以为任何类型的值创建一个指针，而不必反复编写相同的代码。

只需将值传给 `Ptr`，就能得到所需的指针：

```
○ ○ ○  
timePtr := Ptr(time.Now())  
  
intPtr := Ptr(42)  
  
stringPtr := Ptr("gopher")
```

这样使你的代码更加简洁，避免了重复编写相同的代码。

你不再需要写单独的函数，也不需要为每种类型手动处理指针。这可以让你的代码保持简单，专注于重要的事情。

Tip #67 如何处理长函数签名

原始链接: [Golang Tip #67: How to deal with long function signatures](#)

当你在 Go 中处理具有许多参数、长名称、接收器、多个返回结果的函数签名时，可能会遇到类似这样的情况：



```
func SendEmail(subj string, recip string, body string,
attachmentPaths []string, cc []string, bcc []string, replyTo string)
error
```

在不破坏代码流程（从上到下）的情况下，有几种解决方法：

1. 长参数可能表明函数的功能超出了应有的范围，考虑将其拆分为较小的函数。
2. 如果任何参数是可选的，请考虑使用可选的结构体或变长函数。这一技术在我之前分享的tips中介绍过 (Tips#22)。
3. 如果参数是必需的，仍然可以将它们分组到一个结构体中并进行验证，必要时抛出错误。
4. 使用仍然清晰且描述准确的较短名称。
5. 具有相同类型的参数可以在类型前声明一次。

“但我仍然想保留 4 或 5 个参数；我不想每次都创建一个新的结构体。”

语义换行

更清晰的解决方案是根据它们的语义关系将一组参数放在自己的一行上：

○ ○ ○

```
func SendEmail(  
    subj, receip, body, replyTo string,  
    attachmentPaths, cc, bcc []string,  
) error {  
    // ...  
}
```

尽管由于多行而稍显冗长，但它使我们需要阅读的所有内容都在视线范围内。

Tip #68 使用deadcode工具来找到和删除无用的函数

原始链接: [Golang Tip #68: Use the deadcode tool to find and remove unused functions](#)

我们有时会有一些未使用的代码, 我们称之为“死代码”。

这可能会让您在编写代码时感到困难, 因为我们不确定是否可以删除它或更改它。

幸运的是, 有一个名为 `deadcode` 的工具, 可以帮助我们找到这些未使用的函数。我们可以使用以下命令进行安装:

```
○○○  
$ go install golang.org/x/tools/cmd/deadcode@latest
```

我将其放在这里, 以便您可以复制: `go install
http://golang.org/x/tools/cmd/deadcode@latest`

然后, 使用以下简短的命令运行它:

```
○○○

$ deadcode .

internal/params.go:34:6: unreachable func: WithUserRanking
internal/wallet.go:50:35: unreachable func: Transaction.CollectionName
internal/utilx/randomx/string.go:23:6: unreachable func: RandomAlphabet
```

运行后，它将显示未使用的函数及其在代码中的位置。

如果您想知道为什么使用某个函数，我们可以使用 `-whylive` 标志，该工具将告诉我们该函数是如何连接其它代码。

根据我的经验，有时该工具可能不准确。因此，使用此工具来帮助您决定可以删除哪些代码。

其工作原理（简化）：

- 该工具首先阅读所有代码，检查类型。
- 它将 `main()` 和任何 `init()` 标识为起始点。
- 从这些起始点开始，`deadcode` 查看直接调用的函数，它列出正在使用的函数。
- 然后，它检查通过接口间接调用的函数。
- 该工具跟踪转换为接口的任何类型，因为这些类型的方法可能会被间接调用。
- 在工具完成分析后，不在此列表中的任何函数都被视为“死代码”，这意味着该函数与主路径运行的代码没有连接。

您可以在此处阅读有关查找和删除死代码的更多信息：

<https://go.dev/blog/deadcode>

Tip #69 通过errgroup管理多个goroutine

原始链接: [Golang Tip #69: Manage multiple goroutines with errgroup](#)

当我们处理一堆 goroutine 时, 要处理错误并确保它们之间良好协同工作可能会有些困难。

您可能知道 `sync.WaitGroup`, 对吗? 但是有一个名为 **errgroup** 的包可以更轻松地处理这个问题:

○ ○ ○

```
$ go get -u golang.org/x/sync
```

```
func main() {
    urls := []string{
        "http://example.com",
        "http://example.net",
    }
```

```
    var g errgroup.Group
```

```
    for _, url := range urls {
        g.Go(func() error {
            return fetch(url)
        })
    }
```

```
    if err := g.Wait(); err != nil {
        fmt.Printf("Fetch error: %v\n", err)
    }
}
```

在我们讨论的示例中，我们获取了 2 个页面，并使用 `g.Wait()` 等待它们。

`errgroup` 是一个旨在帮助我们管理多个 goroutine 并处理它们在执行过程中抛出任何错误的工具。

这里有三个概念：

- 要并发运行任务, 请使用 `g.Go()`` 启动一个 goroutine 并传递一个函数。
- 使用 `g.Wait()`` 等待所有 goroutine 完成, 它会返回第一个发生的错误, 而不是所有错误。
- `errgroup` 与 `context` 很搭配。

通过使用 `errgroup.WithContext()`, 如果发生错误, `context` 将被取消。

内部实现机制:

1. `Group` 结构使用了以下组合:

- `sync.WaitGroup` 用于等待所有 goroutine 完成
- `sync.Once` 确保以线程安全的方式捕获第一个错误
- 一个信号量 `chan` 用于控制同时运行的 goroutine 数量 (我们甚至可以使用 `errg.SetLimit()` 设置限制)

```
○ ○ ○  
type Group struct {  
    cancel func(error)  
  
    wg sync.WaitGroup  
  
    sem chan token  
  
    errOnce sync.Once  
    err      error  
}  
}
```

2. `errg.done()` 是一个标记 goroutine 完成的辅助函数。

它会减少活动 goroutine 计数 (如果设置了限制), 并通知 WaitGroup 该 goroutine 已完成:

○ ○ ○

```
func (g *Group) done() {
    if g.sem != nil {
        <-g.sem
    }
    g.wg.Done()
}
```

3. 它将错误处理集中在一个地方。使用 `errOnce`，只记录第一个错误，然后触发停止并由 `Wait` 返回。

○ ○ ○

```
if err := f(); err != nil {  
    g.errOnce.Do(func() {  
        g.err = err  
        if g.cancel != nil {  
            g.cancel(g.err)  
        }  
    })  
}
```

此外，使用 goroutine 并不总是最佳选择，特别是如果任务很快。有时，一个接一个地执行它们可能更好。

Tip #70 实现一个感知context的sleep 函数

原始链接: [Golang Tip #70: Implement a context-aware sleep function.](#)

常规 `time.Sleep()`不关心 `context`。如果您将其设置为暂停 5 分钟，无论如何，它都会暂停整个过程。

即使我们尝试在 `context` 中使用它，就像下面的例子一样，它也要等到 10 秒后才会停止：

```
○○○

func Job(ctx context.Context) {
    for {
        select {
        case <-ctx.Done():
            return
        default:
            // ...
            time.Sleep(10 * time.Second)
        }
    }
}
```

我们在Tips #63 中讨论了解决方案：[避免 time.Sleep \(\)](#)

我们之前讨论的解决方案是可行的，但是一遍又一遍地写就有点麻烦了。

因此，让我们制作一个更加用户友好的版本，仍然让我们使用 Sleep() 函数，但尊重 context 被取消的时间。我们可以创建一个“假”睡眠函数，如果 context 告诉它停止，它就会停止：

```
○ ○ ○

func Sleep(ctx context.Context, duration time.Duration) error {
    timer := time.NewTimer(duration)
    defer timer.Stop()

    select {
    case <-ctx.Done():
        return ctx.Err()
    case <-timer.C:
        return nil
    }
}
```

<https://go.dev/play/p/FErMIDKoulb>

这样，我们可以在 Go 中暂停代码，但如果有东西告诉 context 停止，Sleep将提前结束：

```
○ ○ ○

func Job(ctx context.Context) {
    for {
        select {
        case <-ctx.Done():
            return
        default:
            // ...
            timex.Sleep(ctx, 10 * time.Second)
        }
    }
}
```

“等等，你为什么不处理sleep()的错误呢？”

嗯，我们通常不需要。

大多数时候，当 `context` 被取消时，与 `sleep` 功能无关。这通常是因为程序中有一个更大的问题，其中包括 `sleep` 部分。

例如，如果代码中 `sleep()` 之后有步骤，并且它们被设置为监听上下文，那么如果 `context` 被取消，它们也会停止。

因此，让我们的 `sleep` 时间更短更加重要。

○ ○ ○

```
func Sleep(ctx context.Context, duration time.Duration) {
    timer := time.NewTimer(duration)
    defer timer.Stop()

    select {
    case <-ctx.Done():
        return
    case <-timer.C:
        return
    }
}
```

译者注：评论区里面提到了一个 stackoverflow 的讨论，[How can I sleep with responsive context cancelation?](#) 感兴趣的朋友可以看看。

Golang Tip #71: 用泛型让 sync.Pool 类型安全

什么是 sync.Pool?

原文链接: [Golang Tip #71: sync.Pool, make it typed-safe with generics.](#)

在讲 sync.Pool 之前, 对于那些不熟悉 sync.Pool 的人来说, 它是 Go 的标准库的一个特性, 用于重用对象。

它可以减少内存分配的数量, 有利于性能。想象一把每秒射出数百发子弹的枪, 为每一发子弹在内存中分配新空间是很浪费的。

相反, 你可以有大约 100 颗子弹容量的池子, 并从池子中重复使用它们:

```
○ ○ ○

var bulletPool = sync.Pool{
    New: func() interface{} {
        return new(Bullet) // Create a new Bullet.
    },
}

func fireBullet() {
    bullet := bulletPool.Get().(*Bullet)
    bullet.Reset()

    // ...

    // When the bullet is no longer needed:
    bulletPool.Put(bullet)
}
```

但是，需要注意以下几点：

- sync.Pool没有固定的大小，所以我们可以无限制地添加和检索条目
- 在我们把对象放回池子后，需要忘掉它，它可能会被清除或GC
- 对象可能有状态，我们应该在放入池中之前或之后从池中检索时清除或者重置它的状态

类型安全的池

现在让我们来讨论如何使 sync.Pool 变得类型安全。

我们上面使用空接口(interface{})来存储和检索条目。在类型安全的版本中，我们封装一下这个过程：

```
○ ○ ○

type Pool[T any] struct {
    internal *sync.Pool
}

func NewPool[T any](newF func() T) Pool[T] {
    return Pool[T]{
        internal: &sync.Pool{
            New: func() interface{}
            return newF()
        },
        },
    }
}
```

像上面那样，我们创建了一个与特定类型T相关联的池，但在内部仍然是用的interface{}

这个方案来自于我们如何以类型安全的方式处理获取和放置数据：

○ ○ ○

```
func (p Pool[T]) Get() T {
    return p.internal.Get().(T)
}

func (p Pool[T]) Put(t T) {
    p.internal.Put(t)
}
```

我们像之前一样从池中取出数据，但是我们将接口转换为类型 T，而不进行错误检查。

“我们为什么不检查转换过程中的错误呢？”

泛型已经为我们确保了类型，所以我们不需要担心转换失败。sync.Pool 始终只包含类型 T 的实例，因此在正常情况下，断言 p.internal.Get(T) 不会引发 panic。

实例代码参见：<https://go.dev/play/p/N3suxuK-yCp>

Tip #72 使用strings.EqualFold进行忽略大小写的字符串比较

原始链接: [Case-Insensitive string comparison with strings.EqualFold](#)

当你需要比较字符串且不关心字母是大写还是小写时, 你可能会想到:

- 使用strings.ToLower()将两个字符串都转换为小写;
- 或者使用strings.ToUpper()将它们都转换为大写。

然后检查它们是否相同:

```
○○○  
if strings.ToLower(str1) == strings.ToLower(str2) {  
    // Strings are the same, ignoring case  
}
```

但是, 在 Go 中有一种更好的比较字符串的方法, 而不必担心大小写。

就是使用strings.EqualFold:

○ ○ ○

```
if strings.EqualFold(str1, str2) {  
    // Strings are the same, ignoring case  
}
```

我们选择strings.EqualFold函数不仅仅是因为它更短，而是因为它专门用于忽略大小写的比较，所以它会做的很好。

“strings.EqualFold函数比使用ToLower后再进行比较更快吗？”

是的，这是在Go中忽略大小写比较字符串时的常用方法：

○ ○ ○

BenchmarkToLower-8	29140400	40.67	ns/op	8	B/op	1	allocs/op
BenchmarkEqualFold-8	208766617	5.718	ns/op	0	B/op	0	allocs/op

大小写折叠不仅仅只是将字母变成大写字母或者小写字母

它会小心处理 Unicode 的细节，确保对所有语言都有效，而不仅仅是简单的英语（或 ASCII? ）字符：

○ ○ ○

```
strings.EqualFold("Σ", "σ")           // true
strings.EqualFold("RESUMÉ", "résumé") // true
```

简而言之，它会起始于：

- **快速路径**：快速检查 ASCII 字符，逐个字符查看每个字符。
- **慢速路径**：如果在任何字符串中发现 Unicode 字符，则切换到详细的 Unicode 比较。

请注意，你仍然可以在上面的示例中使用 strings.ToLower 或 strings.ToUpper，但这关系到速度和使你的代码易于阅读。

有时候即使 strings.EqualFold 也是不够用的，例如：

```
s1 := "Résumé" // Normal 'é'
s2 := "resume\u0301" // 'e' 后跟一个组合重音符
```

在这种情况下，仅使用 strings.EqualFold 将不够，因为这些字符看起来相同但编码不同。

要正确比较这些字符串，您将需要其他方法来处理字符串，参考
<http://golang.org/x/text/nocode/norm>

Tip #73 用stringer工具给枚举类型实现String()方法

原始链接: [Implement String\(\) for enum with the stringer tool](#)

你是否注意到，在你用Go语言打印中打印持续时间的时候，比如：
fmt.Println(time.Second)，它显示为"1s"而不是"1000000000"，尽管
time.Duration的底层类型是int64类型。

这是因为time.Duration类型有一个String()方法，使其以一种更易于理解的方式打印
出来的来。

这个方法就是fmt.Stringer接口的一部分。

```
○ ○ ○

package fmt

type Stringer interface {
    String() string
}

---  
fmt.Println(time.Second) // "1s"
```

为了让我们自定义的类型也同样清晰，我们也可以添加一个String()方法。

对于枚举类型，我们通常使用数字，但是我们也希望打印出的内容更易于阅读。

我们可以写一个带有switch语句的函数来完成这件事。

```
func (h HeroType) String() string {
    switch h {
    case HeroTypeTank:
        return "Tank"
    case HeroTypeAssassin:
        return "Assassin"
    case HeroTypeMage:
        return "Mage"
    }

    return ""
}
```

然而这可能是一份额外的工作。

如果我们更改了枚举值并忘记去更新此函数，可能会导致问题。

幸运的是，Go有一个stringer工具，这是一个命令行工具，可以自动为我们创建String()方法：

○ ○ ○

```
$ go install golang.org/x/tools/cmd/stringer@latest  
$ stringer -type HeroType
```

我们是否需要为不同包中的每一个单独的类型都执行一次这个命令吗？

这就是"go generate"派上用场的地方了。

我们只需在我们的代码中添加一个特殊的注释"go generate"将调用 stringer工具，并为我们创建String()方法：

```
○ ○ ○

//go:generate stringer -type=HeroType
type HeroType int

const (
    HeroTypeTank HeroType = iota + 1
    HeroTypeAssassin
    HeroTypeMage
)

---  
$ go generate ./...
```

我们可以将这一行注释放在同一个包的任何地方，但我更喜欢将其放在对应的枚举类型上方。

有一些选项可以更改 String() 的工作方式：

-trimprefix: 删除名称的前缀

如果我们有一个 HeroTypeTank 枚举值，它通常会显示为 "HeroTypeTank"。如果我们将 -trimprefix 设置为 "HeroType"，它将显示为 "Tank"。

```
//go:generate stringer -type=HeroType -trimprefix=HeroType
```

-linecomment: 设置一个完全不同的枚举值名称，只需要在枚举值后面的加上注释。

```
HeroTypeAssassin // Something
//go:generate stringer -type=HeroType -linecomment
```

Tip #74 使 `time.Duration` 清晰易懂

原始链接: [Golang Tip #74: Make `time.Duration` clear and easy to understand](#)

在编码里处理时间间隔的时候, 你会经常使用 `time.Duration` 类型。一个常见的问题可能会让代码变得混乱, 那就是当你使用秒时:

```
○ ○ ○  
const warmUpSeconds int = 10  
  
// later in the code  
time.Sleep(time.Duration(warmUpSeconds) * time.Second)
```

在 Go 中, 这通常不是常规的做法, 但是如果我们出于某种原因需要使用秒, 我们可以使用一个无类型的常量:

○ ○ ○

```
const warmUpSeconds = 10

// later in the code
time.Sleep(warmUpSeconds * time.Second)
```

这些常量很灵活，因为它们不会固定在某一种类型上，它们会适应于任何类型。

这里有一个例子，它是正确的，但可能更清晰一些：

○ ○ ○

```
const refreshDuration = 168 * time.Hour
```

看到 $168 * \text{time.Hour}$ 可能不会立即告诉我们这是1周。为了让我们的表述更加清楚，我们可以这样设置：

○ ○ ○

```
// A clearer alternative:  
const refreshDuration = 7 * 24 * time.Hour
```

现在，当有人阅读我们的代码时，他们可以立即看到 `refreshDuration` 是1周，而不必考虑这是多少小时。

Tip #75 使用singleflight优化多次调用

原始链接: [Golang Tip #75: Optimize multiple calls with singleflight.](#)

假设您有一个从网络获取数据或执行 I/O 的函数, 大约需要 3 秒钟:

```
○ ○ ○

func FetchExpensiveData() (int64, error) {
    time.Sleep(3 * time.Second)
    return time.Now().Unix() / 10, nil
}
```

上述函数会在 10 秒后发出一个不同的数字。

- 现在, 如果您连续调用这个函数 3 次, 最终总共需要等待约 9 秒钟。
- 如果您决定使用 3 个 goroutines, 总等待时间可能会降至 3 秒左右, 但您仍然要运行该函数 3 次才能得到相同的结果。(~99%)

这就是 singleflight 软件包发挥作用的地方, 它非常有用。您可以在 <http://golang.org/x/sync/singleflight...> 找到它的详细信息。

这个软件包可以帮助我们只运行一次函数, 无论在 3 秒钟内调用多少次, 它都能返回一个可靠的结果。

```
○ ○ ○

var group singleflight.Group

func UsingSingleFlight(key string) {
    v, _, _ := group.Do(key, func() (interface{}, error) {
        return FetchExpensiveData()
    })
    fmt.Println(v)
}
```

这对于优化耗时较长或耗费大量资源的函数来说非常有用。下面是它的工作原理：我们首先创建一个 `singleflight.Group` 对象。

然后，我们将获取昂贵数据的函数传递到该对象的 `group.Do()` 方法中。`group.Do` 返回: (`result any, err error, shared bool`)，其中的 "shared" (共享) 部分非常简单，它只是表示结果是否在多次调用中共享。

基本上就是这样

"key" 参数的作用是什么？

`key` 本质上是请求的标识符。

当多个请求具有相同的关键字时，`singleflight` 就会意识到它们请求的是同一件事。

您可以再这里看到实际操作: <https://go.dev/play/p/dxQrznx7m4>

使用这种方法，如果同一函数被同时调用多次，则只对该函数进行一次实际调用。然后，所有调用者共享这一次调用的结果。

Tip #76 函数调用的结果回传

原始链接: [Golang Tip #76: Result forwarding in function call](#)

当我刚开始使用go语言的时候，我发现有一个概念比较棘手：函数调用结果的回传。

一个函数返回多个值是很常见的，通常是一个result附带一个error，看看我们在结尾是如何处理这个processResult(result)的：

```
○ ○ ○

func doSomething() (int, error) {
    ...
    return result, nil
}

func main() {
    result, err := doSomething()
    if err != nil {
        ...
    }

    processResult(result)
}
```

如果processResult函数的传入参数正好适配doSomthing的返回值，我们可以将结果直接传递给processResult，这样看起来十分整洁。

```
○ ○ ○

func main() {
    processResult(doSomething())
}

func processResult(int, error) {}
```

现在这里有一个例子：我们获取一个result和一个error，并且我们需要基于此给客户端响应一个状态码。

通常情况下，你可能会看到很多controllers或者API层的函数都在做着同样的事情。

它们接收 (result, error) 这样的返回值，如果有错误的话会返回400状态码，如果没有错误就返回200状态码。

○ ○ ○

```
func GetResult(api *API) {
    result, err := Response()
    if err != nil {
        api.JSON(http.StatusBadRequest, err)
        return
    }
    api.JSON(http.StatusOK, result)
}
```

我们可以通过创造一个基于函数调用的标准方法来处理这些响应，来简化我们的代码，而不是在多个函数里重复的使用以上的模式。

```
○ ○ ○

func GetResult(api *API) {
    api.JSONWithStatus(Response())
}

func (api *API) JSONWithStatus(result any, err error) {
    if err != nil {
        api.JSON(http.StatusBadRequest, err)
        return
    }
    api.JSON(http.StatusOK, result)
}
```

这个技巧与我在之前提到的“must”函数结合使用时效果非常好。具体详见：[Golang Tip #44: Intentionally Stop with Must Functions](#)

我还有一个小的通用的辅助的函数，如果没有error的时候它会返回结果，否则就会停止运行。

```
func Must[T any](result T, err error) T { ... }
```

Tip #77 带缓冲的 channel 作为信号量来限制 goroutine 执行

原始链接: [Golang Tip #77: Buffered channels as semaphores to limit goroutine execution](#)

当我们想要管理有多少 goroutine 可以同时访问一个资源时, 使用信号量是一种可靠的方法。

我们可以使用带有缓冲的 channel 来创建信号量, channel 的大小决定了可以同时运行多少个 goroutine:

```
○ ○ ○  
semaphore := make(chan struct{}, numTokens)
```

接下来:

- 一个 goroutine 将一个值发送到 channel 中, 占据一个槽位
- 在完成任务后移除该值, 从而为另一个 goroutine 释放该槽位

```
○ ○ ○

semaphore := make(chan struct{}, 3)

var wg sync.WaitGroup
wg.Add(10)

for i := 0; i < 10; i++ {
    go func(id int) {
        defer wg.Done()
        semaphore <- struct{}{} // Acquire a token.
        ...
        <-semaphore // Release the token.
    }(i)
}

wg.Wait()
```

在这个例子中：

- `wg.Add(10)` 指我们准备用 10 个 goroutine 来完成所有工作
- `make(chan struct{}, 3)` 设置了一个只允许 3 并发的有缓冲 channel 作为信号量

如果想更简洁的实现，我们可以考虑创建一个 `Semaphore` 类型来处理信号量相关的操作：

○ ○ ○

```
type Semaphore chan struct{}

func NewSemaphore(max int) Semaphore {
    return make(chan struct{}, max)
}

func (s Semaphore) Acquire() {
    s <- struct{}{}
}

func (s Semaphore) Release() {
    <-s
}
```

使用这个自定义的 Semaphore 类型简化了对资源访问的控制：

○ ○ ○

```
func doSomething(semaphore *Semaphore) {
    semaphore.Acquire()
    defer semaphore.Release()

    ...
}
```

此外，<http://golang.org/x/sync/semaphore> 包中提供了一个信号量的实现，是加权信号量，加权信号量允许一个 goroutine 占用多个槽位，适用于每个 goroutine 资源消耗不同的情况。例如，管理数据库的连接池时，某些操作可能需要一次使用多个连接。

Tip #78 非阻塞 channel 发送技巧

原始链接: <https://twitter.com/func25/status/1780207297991782834>

当我们向channel发送数据时, 通常会等待接收方准备好接收数据:



但有时我们并不想等待。例如, 如果您学习了前面关于使用 semaphore 的小技巧, 我们就可以使用 `TryAcquire() bool` 函数, 如果所有令牌都已被占用, 它就会立即返回。

如果您错过了上一条Tip 请看 [Tip #77: Buffered channels as semaphores to limit goroutine execution](#)

以 `errgroup` 为例, 其内部使用了简单的信号量机制来管理 goroutine 的数量。

现在, 如果信号量已满, 我们希望它在无法启动时立即返回 `false`, 就像 `mutex.TryLock()` 一样。

让我们看看 `errgroup` 如何处理这个问题:

○ ○ ○

```
func (g *Group) TryGo(f func() error) bool {
    if g.sem != nil {
        select {
            case g.sem <- token{}:
            default:
                return false
        }
    }
    ...
}
```

关注 if 条件中的 select{} 语句。通常，select{} 用于等待来自多个通道操作的值，但这里的使用方式很特别：

- `case g.sem <- token{}:`：这一行试图向信号通道 `g.sem` 发送一个标记。如果有空格（意味着没有达到上限），则标记发送成功，这部分代码执行。
- `default:`：如果 `g.sem` 通道已满，则选择这种情况。

如果其他情况尚未准备就绪，选择语句中的默认情况会立即运行。

在这种情况下，它会返回**false**，通知我们函数没有启动新的 goroutine，因为我们已经达到了之前设置的活动 goroutine 的最大数量。

Tip #79 如果做了不寻常的事，请说明原因

原始链接: [Golang Tip #79: If doing something unusual, comment why](#)

我已经审查了一段时间的代码，并意识到最常见且最令人沮丧的问题之一是对不寻常代码的选择缺乏注释。

当我们使用Go语言足够长的时间后，我们明白大部分代码应该遵循常用的习惯用法和社区实践。

但是，有时我们需要打破这些规范，例如：

```
○ ○ ○  
user, _ := fetchUser(id)
```

忽略错误或返回的结果，这行代码可能会让审查者或其他团队成员感到困惑或惊讶。

为什么忽略这个错误并不清楚。或许这个错误并不关键，即使没有有效的用户，功能也能继续执行，但为什么会这样呢？

以下是如何澄清这个问题的方法：

```
○ ○ ○  
  
// Ignore error; nil user handled in subsequent logic  
user, _ := fetchUser(id)
```

这条注释迅速告诉读者，忽略错误是一个故意的决定，因为后续代码可以适当地处理一个为nil的用户。

再次强调，不应该忽略错误；这只是一个例子。至少，应该记录下这个错误。

还有一个：

```
○ ○ ○  
  
user, err := fetchUser(id)  
if err == nil {  
    user = createUser()  
    ...  
}
```

如果我们不突出显示那行不同寻常的代码，就很难看出哪里令人惊讶。

因此，这条注释不仅解释了为什么我们选择使用'err == nil'，还通知了团队成员：“嘿，这是不寻常的代码，请看一下。”

这是不寻常的，并且很容易被忽视，因为不寻常的细节太细微了，不容易被发现。

因此，当你以不同的方式做事时，一定要加上注释。