目　录

并发

保持自己忙碌或做自己的工作

将并发性留给调用者

永远不要启动一个停止不了的 `goroutine`

介绍

版权申明

本文档转自：https://github.com/llitfkitfk/go-best-practice

指导原则

如果我要谈论任何编程语言的最佳实践，我需要一些方法来定义“什么是最佳”。如果你昨天来到我的主题演讲，你会看到 Go 团队负责人 Russ Cox 的这句话：

> Software engineering is what happens to programming when you add time and other programmers. (软件工程就是你和其他程序员花费时间在编程上所发生的事情。)
> — Russ Cox

Russ 作出了软件编程与软件工程的区分。 前者是你自己写的一个程序。 后者是很多人会随着时间的推移而开发的产品。 工程师们来来去去，团队会随着时间增长与缩小，需求会发生变化，功能会被添加，错误也会得到修复。 这是软件工程的本质。

我可能是这个房间里 Go 最早的用户之一，~~但要争辩说我的资历给我的看法更多是假的~~。相反，今天我要提的建议是基于我认为的 Go 语言本身的指导原则：

1. 简单性
2. 可读性
3. 生产力

> 注意:
> 你会注意到我没有说性能或并发。 有些语言比 Go 语言快一点，但它们肯定不像 Go 语言那么简单。 有些语言使并发成为他们的最高目标，但它们并不具有可读性及生产力。
> 性能和并发是重要的属性，但不如简单性，可读性和生产力那么重要。

简单性

我们为什么要追求简单？ 为什么 Go 语言程序的简单性很重要？

我们都曾遇到过这样的情况: “我不懂这段代码”，不是吗？ 我们都做过这样的项目:你害怕做出改变，因为你担心它会破坏程序的另一部分; 你不理解的部分，不知道如何修复。

这就是复杂性。 复杂性把可靠的软件中变成不可靠。 复杂性是杀死软件项目的罪魁祸首。

简单性是 Go 语言的最高目标。 无论我们编写什么程序，我们都应该同意这一点:它们很简单。

可读性

> Readability is essential for maintainability.
> (可读性对于可维护性是至关重要的。)
> — Mark Reinhold (2018 JVM 语言高层会议)

为什么 Go 语言的代码可读性是很重要的？我们为什么要争取可读性？

> Programs must be written for people to read, and only incidentally for machines to execute. (程序应该被写来让人们阅读，只是顺便为了机器执行。)
> — Hal Abelson 与 Gerald Sussman (计算机程序的结构与解释)

可读性很重要，因为所有软件不仅仅是 Go 语言程序，都是由人类编写的，供他人阅读。执行软件的计算机则是次要的。

代码的读取次数比写入次数多。一段代码在其生命周期内会被读取数百次，甚至数千次。

> The most important skill for a programmer is the ability to effectively communicate ideas. (程序员最重要的技能是有效沟通想法的能力。)
> — Gastón Jorquera [1]

可读性是能够理解程序正在做什么的关键。如果你无法理解程序正在做什么，那你希望如何维护它？如果软件无法维护，那么它将被重写;最后这可能是你的公司最后一次投资 Go 语言。

~~如果你正在为自己编写一个程序，也许它只需要运行一次，或者你是唯一一个曾经看过它的人，然后做任何对你有用的事。~~但是，如果是一个不止一个人会贡献编写的软件，或者在很长一段时间内需求、功能或者环境会改变，那么你的目标必须是你的程序可被维护。

编写可维护代码的第一步是确保代码可读。

生产力

> Design is the art of arranging code to work today, and be changeable forever. (设计是安排代码到工作的艺术，并且永远可变。)
> — Sandi Metz

我要强调的最后一个基本原则是生产力。开发人员的工作效率是一个庞大的主题，但归结为此; 你花多少时间做有用的工作，而不是等待你的工具或迷失在一个外国的代码库里。 Go 程序员应该觉得他们可以通过 Go 语言完成很多工作。

有人开玩笑说， Go 语言是在等待 C++ 语言程序编译时设计的。快速编译是 Go 语言的一个关键特性，也是吸引新开发人员的关键工具。虽然编译速度仍然是一个持久的战场，但可以说，在其他语言中需要几分钟的编译，在 Go 语言中只需几秒钟。这有助于 Go 语言开发人员感受到与使用动态语言的同行一样的高效，而且没有那些语言固有的可靠性问题。

对于开发人员生产力问题更为基础的是，Go 程序员意识到编写代码是为了阅读，因此将读代码的行为置于编写代码的行为之上。Go 语言甚至通过工具和自定义强制执行所有代码以特定样式格式化。这就消除了项目中学习特定格式的摩擦，并帮助发现错误，因为它们看起来不正确。

Go 程序员不会花费整天的时间来调试不可思议的编译错误。他们也不会将浪费时间在复杂的构建脚本或在生产中部署代码。最重要的是，他们不用花费时间来试图了解他们的同事所写的内容。

当他们说语言必须扩展时，Go 团队会谈论生产力。

标识符

我们要讨论的第一个主题是标识符。 标识符是一个用来表示名称的花哨单词; 变量的名称，函数的名称，方法的名称，类型的名称，包的名称等。

> Poor naming is symptomatic of poor design. (命名不佳是设计不佳的症状。)
> — Dave Cheney

鉴于 Go 语言的语法有限，我们为程序选择的名称对我们程序的可读性产生了非常大的影响。 可读性是良好代码的定义质量，因此选择好名称对于 Go 代码的可读性至关重要。

选择标识是为了清晰, 而不是简洁

Obvious code is important. What you can do in one line you should do in three.
(清晰的代码很重要。在一行可以做的你应当分三行做。(`if/else` 吗?))
— Ukiah Smith

Go 语言不是为了单行而优化的语言。 Go 语言不是为了最少行程序而优化的语言。我们没有优化源代码的大小，也没有优化输入所需的时间。

> Good naming is like a good joke. If you have to explain it, it’s not funny.
> (好的命名就像一个好笑话。如果你必须解释它，那就不好笑了。)
> — Dave Cheney

清晰的关键是在 Go 语言程序中我们选择的标识名称。让我们谈一谈所谓好的名字：

- **好的名字很简洁。** 好的名字不一定是最短的名字，但好的名字不会浪费在无关的东西上。好名字具有高的信噪比。
- **好的名字是描述性的。** 好的名字会描述变量或常量的应用，而不是它们的内容。好的名字应该描述函数的结果或方法的行为，而不是它们的操作。好的名字应该描述包的目的而非它的内容。描述东西越准确的名字就越好。
- **好的名字应该是可预测的。** 你能够从名字中推断出使用方式。~~这是选择描述性名称的功能，但它也遵循传统。~~这是 Go 程序员在谈到习惯用语时所谈论的内容。

让我们深入讨论以下这些属性。

标识符长度

有时候人们批评 Go 语言推荐短变量名的风格。正如 Rob Pike 所说，“ Go 程序员想要正确的长度的标识符”。 [1]

Andrew Gerrand 建议通过对某些事物使用更长的标识，向读者表明它们具有更高的重要性。

> The greater the distance between a name’s declaration and its uses, the longer the name should be. (名字的声明与其使用之间的距离越大，名字应该越长。)
> — Andrew Gerrand [2]

由此我们可以得出一些指导方针：

- 短变量名称在声明和上次使用之间的距离很短时效果很好。
- 长变量名称需要证明自己的合理性; 名称越长，需要提供的价值越高。冗长的名称与页面上的重量相比，信号量较小。
- 请勿在变量名称中包含类型名称。
- 常量应该描述它们持有的值，而不是该如何使用。
- 对于循环和分支使用单字母变量，参数和返回值使用单个字，函数和包级别声明使用多个单词
- 方法、接口和包使用单个词。
- 请记住，包的名称是调用者用来引用名称的一部分，因此要好好利用这一点。

我们来举个栗子:

```
type Person struct {
    Name string
    Age  int
}

// AverageAge returns the average age of people.
func AverageAge(people []Person) int {
    if len(people) == 0 {
        return 0
    }

    var count, sum int
    for _, p := range people {
        sum += p.Age
        count += 1
    }

    return sum / count
}
```

在此示例中，变量 `p` 的在第 `10` 行被声明并且也只在接下来的一行中被引用。 `p` 在执行函数期间存在时间很短。如果要了解 `p` 的作用只需阅读两行代码。

相比之下， `people` 在函数第 `7` 行参数中被声明。 `sum` 和 `count` 也是如此，他们用了更长的名字。读者必须查看更多的行数来定位它们，因此他们名字更为独特。

我可以选择 `s` 替代 `sum` 以及 `c` （或可能是 `n` ）替代 `count` ，但是这样做会将程序中的所有变量份量降低到同样的级别。我可以选择 `p` 来代替 `people` ，但是用什么来调用 `for ... range` 迭代变量。如果用 `person` 的话看起来很奇怪，因为循环迭代变量的生命时间很短，其名字的长度超出了它的值。

> 贴士:
> 与使用段落分解文档的方式一样用空行来分解函数。 在 `AverageAge` 中，按顺序共有三个操作。 第一个是前提条

> 件，检查 `people` 是否为空，第二个是 `sum` 和 `count` 的累积，最后是平均值的计算。

上下文是关键

重要的是要意识到关于命名的大多数建议都是需要考虑上下文的。 我想说这是一个原则，而不是一个规则。

两个标识符 `i` 和 `index` 之间有什么区别。 我们不能断定一个就比另一个好，例如

```
for index := 0; index < len(s); index++ {
    //
}
```

从根本上说，上面的代码更具有可读性

```
for i := 0; i < len(s); i++ {
    //
}
```

我认为它不是，因为就此事而论， `i` 和 `index` 的范围很大可能上仅限于 for 循环的主体，后者的额外冗长性(指 `index`)几乎没有增加对于程序的理解。

但是，哪些功能更具可读性？

```
func (s *SNMP) Fetch(oid []int, index int) (int, error)
```

或

```
func (s *SNMP) Fetch(o []int, i int) (int, error)
```

在此示例中， `oid` 是 `SNMP` 对象 `ID` 的缩写，因此将其缩短为 `o` 意味着程序员必须要将文档中常用符号转换为代码中较短的符号。 类似地将 `index` 替换成 `i` ，模糊了 `i` 所代表的含义，因为在 `SNMP` 消息中，每个 `OID` 的子值称为索引。

> 贴士: 在同一声明中长和短形式的参数不能混搭。

不要用变量类型命名变量

你不应该用变量的类型来命名你的变量，就像您不会将宠物命名为“狗”和“猫”。 出于同样的原因，您也不应在变量名字中包含类型的名字。

变量的名称应描述其内容，而不是内容的类型。 例如：

```
var usersMap map[string]*User
```

这个声明有什么好处？ 我们可以看到它是一个 `map` ，它与 `*User` 类型有关。 但是 `usersMap` 是一个 `map` ，而 Go 语言是一种静态类型的语言，如果没有定义变量,不会让我们意外地使用到它，因此 `Map` 后缀是多余的。

接下来，如果我们像这样来声明其他变量：

```
var (
    companiesMap map[string]*Company
    productsMap map[string]*Products
)
```

`usersMap` ， `companiesMap` 和 `productsMap` 三个 `map` 类型变量，所有映射字符串都是不同的类型。我们知道它们是 `map` ，我们也知道我们不能使用其中一个来代替另一个 - 如果我们在需要 `map[string]*User` 的地方尝试使用 `companiesMap` ，编译器将抛出错误异常。 在这种情况下，很明显变量中 `Map` 后缀并没有提高代码的清晰度，它只是增加了要输入的额外样板代码。

我的建议是避免使用任何类似变量类型的后缀。

> 贴士:
> 如果 `users` 的描述性都不够用，那么 `usersMap` 也不会。

此建议也适用于函数参数。 例如：

```
type Config struct {
    //
}

func WriteConfig(w io.Writer, config *Config)
```

命名 `*Config` 参数 `config` 是多余的。 我们知道它是 `*Config` 类型，就是这样。

在这种情况下，如果变量的生命周期足够短，请考虑使用 `conf` 或 `c` 。

如果有更多的 `*Config` ，那么将它们称为 `original` 和 `updated` 比 `conf1` 和 `conf2` 会更具描述性，因为前者不太可能被互相误解。

> 贴士:
> 不要让包名窃取好的变量名。
> 导入标识符的名称包括其包名称。 例如， `context` 包中的 `Context` 类型将被称为 `context.Context` 。 这使得无法将 `context` 用作包中的变量或类型。
>
> ```
> func WriteLog(context context.Context, message string)
> ```
>
> 上面的栗子将会编译出错。 这就是为什么 `context.Context` 类型的通常的本地声明是 `ctx` ，例如：

```
func WriteLog(ctx context.Context, message string)
```

使用一致的命名风格

一个好名字的另一个属性是它应该是可预测的。 在第一次遇到该名字时读者就能够理解名字的使用。 当他们遇到常见的名字时，他们应该能够认为自从他们上次看到它以来它没有改变意义。

例如，如果您的代码在处理数据库请确保每次出现参数时，它都具有相同的名称。 与其使用 `d * sql.DB` ， `dbase * sql.DB` ， `DB * sql.DB` 和 `database * sql.DB` 的组合，倒不如统一使用:

```
db *sql.DB
```

这样做使读者更为熟悉; 如果你看到 `db` ，你知道它就是 `*sql.DB` 并且它已经在本地声明或者由调用者为你提供。

类似地，对于方法接收器: 在该类型的每个方法上使用相同的接收者名称。 在这种类型的方法内部可以使读者更容易使用。

> 注意:
> Go 语言中的短接收者名称惯例与目前提供的建议不一致。 这只是早期做出的选择之一，已经成为首选的风格，就像使用 `CamelCase` 而不是 `snake_case` 一样。

> 贴士:
> Go 语言样式规定接收器具有单个字母名称或从其类型派生的首字母缩略词。 你可能会发现接收器的名称有时会与方法中参数的名称冲突。 在这种情况下，请考虑将参数名称命名稍长，并且不要忘记一致地使用此新参数名称。

最后，某些单字母变量传统上与循环和计数相关联。 例如， `i` ， `j` 和 `k` 通常是简单 `for` 循环的循环归纳变量。 `n` 通常与计数器或累加器相关联。 `v` 是通用编码函数中值的常用简写， `k` 通常用于 `map` 的键， `s` 通常用作字符串类型参数的简写。

与上面的 `db` 示例一样，程序员认为 `i` 是一个循环归纳变量。 如果确保 `i` 始终是循环变量，而且不在 `for` 循环之外的其他地方中使用。 当读者遇到一个名为 `i` 或 `j` 的变量时，他们知道循环就在附近。

> 贴士:
> 如果你发现自己有如此多的嵌套循环， `i` ， `j` 和 `k` 变量都无法满足时，这个时候可能就是需要将函数分解成更小的函数。

使用一致的声明样式

Go 至少有六种不同的方式来声明变量

- `var x int = 1`
- `var x = 1`
- `var x int; x = 1`
- `var x = int(1)`
- `x := 1`

我确信还有更多我没有想到的。 这可能是 Go 语言的设计师意识到的一个错误，但现在改变它为时已晚。 通过所有这些不同的方式来声明变量，我们如何避免每个 Go 程序员选择自己的风格？

我想就如何在程序中声明变量提出建议。 这是我尽可能使用的风格。

- **声明变量但没有初始化时，请使用** `var` 。 当声明变量稍后将在函数中初始化时，请使用 `var` 关键字。
  ```golang
  var players int // 0

var things []Thing // an empty slice of Things

var thing Thing // empty Thing struct
json.Unmarshall(reader, &thing)
```

`var` 表示此变量已被声明为指定类型的零值。 这也与使用 `var` 而不是短声明语法在包级别声明变量的要求一致 - 尽管我稍后会说你根本不应该使用包级变量。

* **在声明和初始化时，使用 `:=`。** 在同时声明和初始化变量时，也就是说我们不会将变量初始化为零值，我建议使用短变量声明。 这使得读者清楚地知道 `:=` 左侧的变量是初始化过的。

为了解释原因，让我们看看前面的例子，但这次是初始化每个变量：
```golang
var players int = 0

var things []Thing = nil

var thing *Thing = new(Thing)
json.Unmarshall(reader, thing)
```

在第一个和第三个例子中，因为在 Go 语言中没有从一种类型到另一种类型的自动转换；赋值运算符左侧的类型必须与右侧的类型相同。 编译器可以从右侧的类型推断出声明的变量的类型，上面的例子可以更简洁地写为：

```golang
var players = 0

var things []Thing = nil

var thing = new(Thing)
json.Unmarshall(reader, thing)
```

我们将 `players` 初始化为 `0` ，但这是多余的，因为 `0` 是 `players` 的零值。 因此，要明确地表示使用零值，我们将上面例子改写为:

```golang
var players int
```

第二个声明如何？ 我们不能省略类型而写作:

```
var things = nil
```

因为 `nil` 没有类型。 [2]相反，我们有一个选择，如果我们要使用切片的零值则写作:

```
var things []Thing
```

或者我们要创建一个有零元素的切片则写作:

```
var things = make([]Thing, 0)
```

如果我们想要后者那么这不是切片的零值，所以我们应该向读者说明我们通过使用简短的声明形式做出这个选择:

```
things := make([]Thing, 0)
```

这告诉读者我们已选择明确初始化事物。

下面是第三个声明，

```
var thing = new(Thing)
```

既是初始化了变量又引入了一些 Go 程序员不喜欢的 `new` 关键字的罕见用法。 如果我们用推荐地简短声明语法，那么就变成了:

```
thing := new(Thing)
```

这清楚地表明 `thing` 被初始化为 `new(Thing)` 的结果 - 一个指向 `Thing` 的指针 - 但依旧我们使用了 `new` 地罕见用法。 我们可以通过使用紧凑的文字结构初始化形式来解决这个问题，

```
thing := &Thing{}
```

与 `new(Thing)` 相同，这就是为什么一些 Go 程序员对重复感到不满。 然而，这意味着我们使用指向 `Thing{}` 的指针初始化了 `thing` ，也就是 `Thing` 的零值。

相反，我们应该认识到 `thing` 被声明为零值，并使用地址运算符将 `thing` 的地址传递给 `json.Unmarshall`

```
var thing Thing
json.Unmarshall(reader, &thing)
```

> 贴士:
> 当然，任何经验法则，都有例外。 例如，有时两个变量密切相关，这样写会很奇怪:
>
> ```
> var min int
> max := 1000
> ```
>
> 如果这样声明可能更具可读性

```
min, max := 0, 1000
```

综上所述：

在没有初始化的情况下声明变量时，请使用 `var` 语法。

声明并初始化变量时，请使用 `:=` 。

> 贴士:
> 使复杂的声明显而易见。
> 当事情变得复杂时，它看起来就会很复杂。例如
>
> ```
> var length uint32 = 0x80
> ```
>
> 这里 `length` 可能要与特定数字类型的库一起使用，并且 `length` 明确选择为 `uint32` 类型而不是短声明形式：
>
> ```
> length := uint32(0x80)
> ```
>
> 在第一个例子中，我故意违反了规则，使用 `var` 声明带有初始化变量的。 这个决定与我的常用的形式不同，这给读者一个线索,告诉他们一些不寻常的事情将会发生。

成为团队的合作者

我谈到了软件工程的目标，即编写可读及可维护的代码。 因此，您可能会将大部分职业生涯用于你不是唯一作者的项目。 我在这种情况下的建议是遵循项目自身风格。

在文件中间更改样式是不和谐的。 即使不是你喜欢的方式，对于维护而言一致性比你的个人偏好更有价值。 我的经验法则是：如果它通过了 `gofmt` ，那么通常不值得再做代码审查。

> 贴士:
> 如果要在代码库中进行重命名，请不要将其混合到另一个更改中。 如果有人使用 `git bisect` ，他们不想通过数千行重命名来查找您更改的代码。

注释

在我们继续讨论更大的项目之前，我想花几分钟时间谈论一下注释。

> Good code has lots of comments, bad code requires lots of comments.
> (好的代码有很多注释，坏代码需要很多注释。)
> — Dave Thomas and Andrew Hunt (The Pragmatic Programmer)

注释对 Go 语言程序的可读性非常重要。 注释应该做的三件事中的一件：

1. 注释应该解释其作用。
2. 注释应该解释其如何做的。
3. 注释应该解释其原因。

第一种形式是公共符号注释的理想选择：

```
// Open opens the named file for reading.
// If successful, methods on the returned file can be used for reading.
```

第二种形式非常适合在方法中注释：

```
// queue all dependant actions
var results []chan error
for _, dep := range a.Deps {
        results = append(results, execute(seen, dep))
}
```

第三种形式是独一无二的，因为它不会取代前两种形式，但与此同时它并不能代替前两种形式。 此形式的注解用以解释代码的外部因素。 这些因素脱离上下文后通常很难理解，此注释的为了提供这种上下文。

```
return &v2.Cluster_CommonLbConfig{
    // Disable HealthyPanicThreshold
        HealthyPanicThreshold: &envoy_type.Percent{
            Value: 0,
        },
}
```

在此示例中，无法清楚地明白 `HealthyPanicThreshold` 设置为零百分比的效果。 需要注释 `0` 值将禁用 `panic` 阀值。

关于变量和常量的注释应描述其内容而非其目的

我之前谈过，变量或常量的名称应描述其目的。 向变量或常量添加注释时，该注释应描述变量内容，而不是变量目的。

```
const randomNumber = 6 // determined from an unbiased die
```

在此示例中，注释描述了为什么 `randomNumber` 被赋值为6，以及6来自哪里。 注释没有描述 `randomNumber` 的使用位置。 还有更多的栗子：

```
const (
    StatusContinue           = 100 // RFC 7231, 6.2.1
    StatusSwitchingProtocols = 101 // RFC 7231, 6.2.2
    StatusProcessing         = 102 // RFC 2518, 10.1

    StatusOK                 = 200 // RFC 7231, 6.3.1
```

在HTTP的上下文中，数字 `100` 被称为 `StatusContinue` ，如 RFC 7231 第 6.2.1 节中所定义。

> 贴士:
> 对于没有初始值的变量，注释应描述谁负责初始化此变量。
>
> ```
> // sizeCalculationDisabled indicates whether it is safe
> // to calculate Types' widths and alignments. See dowidth.
> var sizeCalculationDisabled bool
> ```
>
> 这里的注释让读者知道 `dowidth` 函数负责维护 `sizeCalculationDisabled` 的状态。
>
> **隐藏在众目睽睽下**
> 这个提示来自Kate Gregory[3]。有时你会发现一个更好的变量名称隐藏在注释中。
>
> ```
> // registry of SQL drivers
> var registry = make(map[string]*sql.Driver)
> ```
>
> 注释是由作者添加的，因为 `registry` 没有充分解释其目的 - 它是一个注册表，但注册的是什么？
>
> 通过将变量重命名为 `sqlDrivers` ，现在可以清楚地知道此变量的目的是保存SQL驱动程序。
>
> ```
> var sqlDrivers = make(map[string]*sql.Driver)
> ```
>
> 之前的注释就是多余的，可以删除。

公共符号始终要注释

`godoc` 是包的文档，所以应该始终为包中声明的每个公共符号 — 变量、常量、函数以及方法添加注释。

以下是 `Google Style` 指南中的两条规则:

- 任何既不明显也不简短的公共功能必须予以注释。
- 无论长度或复杂程度如何，对库中的任何函数都必须进行注释

  ```golang
  package ioutil

  // ReadAll reads from r until an error or EOF and returns the data it read.
  // A successful call returns err == nil, not err == EOF. Because ReadAll is
  // defined to read from src until EOF, it does not treat an EOF from Read
  // as an error to be reported.
  func ReadAll(r io.Reader) ([]byte, error)
  ```

这条规则有一个例外；您不需要注释实现接口的方法。 具体不要像下面这样做:

```golang
// Read implements the io.Reader interface
func (r *FileReader) Read(buf []byte) (int, error)
```

这个注释什么也没说。 它没有告诉你这个方法做了什么，更糟糕是它告诉你去看其他地方的文档。 在这种情况下，我建议完全删除该注释。

这是 `io` 包中的一个例子

```golang
// LimitReader returns a Reader that reads from r
// but stops with EOF after n bytes.
// The underlying implementation is a *LimitedReader.
func LimitReader(r Reader, n int64) Reader { return &LimitedReader{r, n} }

// A LimitedReader reads from R but limits the amount of
// data returned to just N bytes. Each call to Read
// updates N to reflect the new amount remaining.
// Read returns EOF when N <= 0 or when the underlying R returns EOF.
type LimitedReader struct {
    R Reader // underlying reader
    N int64  // max bytes remaining
}

func (l *LimitedReader) Read(p []byte) (n int, err error) {
    if l.N <= 0 {
        return 0, EOF
    }
    if int64(len(p)) > l.N {
        p = p[0:l.N]
    }
    n, err = l.R.Read(p)
    l.N -= int64(n)
    return
}
```

请注意， `LimitedReader` 的声明就在使用它的函数之前，而 `LimitedReader.Read` 的声明遵循 `LimitedReader` 本身的声明。 尽管 `LimitedReader.Read` 本身没有文档，但它清楚地表明它是 `io.Reader`

的一个实现。

> 贴士:
> 在编写函数之前，请编写描述函数的注释。 如果你发现很难写出注释，那么这就表明你将要编写的代码很难理解。

不要注释不好的代码，将它重写

> Don’t comment bad code — rewrite it
> — Brian Kernighan

粗劣的代码的注释高亮显示是不够的。 如果你遇到其中一条注释，则应提出问题，以提醒您稍后重构。 只要技术债务数额已知，它是可以忍受的。

标准库中的惯例是注意到它的人用 `TODO(username)` 的样式来注释。

```
// TODO(dfc) this is O(N^2), find a faster way to do this.
```

注释 `username` 不是该人承诺要解决该问题，但在解决问题时他们可能是最好的人选。 其他项目使用 `TODO` 与日期或问题编号来注释。

与其注释一段代码，不如重构它

> Good code is its own best documentation. As you’re about to add a comment, ask yourself, ‘How can I improve the code so that this comment isn’t needed?’ Improve the code and then document it to make it even clearer.
> 好的代码是最好的文档。 在即将添加注释时，请问下自己，“如何改进代码以便不需要此注释？’改进代码使其更清晰。
> — Steve McConnell

函数应该只做一件事。 如果你发现自己在注释一段与函数的其余部分无关的代码，请考虑将其提取到它自己的函数中。

除了更容易理解之外，较小的函数更易于隔离测试，将代码隔离到函数中，其名称可能是所需的所有文档。

包的设计

Write shy code - modules that don’t reveal anything unnecessary to other modules and that don’t rely on other modules’ implementations.
编写谨慎的代码 - 不向其他模块透露任何不必要的模块，并且不依赖于其他模块的实现。
— Dave Thomas

每个 Go 语言的包实际上都是它一个小小的 Go 语言程序。 正如函数或方法的实现对调用者而言并不重要一样，包的公共API-其函数、方法以及类型的实现对于调用者来说也并不重要。

一个好的 Go 语言包应该具有低程度的源码级耦合，这样，随着项目的增长，对一个包的更改不会跨代码库级联。 这些世界末日的重构严格限制了代码库的变化率以及在该代码库中工作的成员的生产率。

在本节中，我们将讨论如何设计包，包括包的名称，命名类型以及编写方法和函数的技巧。

一个好的包从它的名字开始

编写一个好的 Go 语言包从包的名称开始。将你的包名用一个词来描述它。

正如我在上一节中谈到变量的名称一样，包的名称也非常重要。我遵循的经验法则不是“我应该在这个包中放入什么类型的？”。相反，我要问是“该包提供的服务是什么？”通常这个问题的答案不是“这个包提供 `X` 类型”，而是“这个包提供 `HTTP` ”。

> 贴士:
> 以包所提供的内容来命名，而不是它包含的内容。

好的包名应该是唯一的。

在项目中，每个包名称应该是唯一的。包的名称应该描述其目的的建议很容易理解 - 如果你发现有两个包需要用相同名称，它可能是:

1. 包的名称太通用了。
2. 该包与另一个类似名称的包重叠了。在这种情况下，您应该检查你的设计，或考虑合并包。

避免使用类似 `base` 、`common` 或 `util` 的包名称

不好的包名的常见情况是 `utility` 包。这些包通常是随着时间的推移一些帮助程序和工具类的包。由于这些包包含各种不相关的功能，因此很难根据包提供的内容来描述它们。这通常会导致包的名称来自包含的内容 - `utilities` 。

像 `utils` 或 `helper` 这样的包名称通常出现在较大的项目中，这些项目已经开发了深层次包的结构，并且希望在不遇到导入循环的情况下共享 `helper` 函数。通过将 `utility` 程序函数提取到新的包中，导入循环会被破坏，但由于该包源于项目中的设计问题，因此其包名称不反映其目的，仅反映其为了打破导入循环。

我建议改进 `utils` 或 `helpers` 包的名称是分析它们的调用位置，如果可能的话，将相关的函数移动到调用者的包中。即使这涉及复制一些 `helper` 程序代码，这也比在两个程序包之间引入导入依赖项更好。

> [A little] duplication is far cheaper than the wrong abstraction.
> ([一点点]重复比错误的抽象的性价比高很多。)
> — Sandy Metz

在使用 `utility` 程序的情况下，最好选多个包，每个包专注于单个方面，而不是选单一的整体包。

> 贴士:
> 使用复数形式命名 `utility` 包。例如 `strings` 来处理字符串。

当两个或多个实现共有的功能或客户端和服务器的常见类型被重构为单独的包时，通常会找到名称类似于 `base` 或 `common` 的包。我相信解决方案是减少包的数量，将客户端，服务器和公共代码组合到一个以包的功能命名的包中。

例如， `net/http` 包没有 `client` 和 `server` 的分包，而是有一个 `client.go` 和 `server.go` 文件，每个文件都有各自的类型，还有一个 `transport.go` 文件，用于公共消息传输代码。

> 贴士:
> **标识符的名称包括其包名称。**
> 重要的是标识符的名称包括其包的名称。
>
> - 当由另一个包引用时， `net/http` 包中的 Get 函数变为 `http.Get` 。
> - 当导入到其他包中时， `strings` 包中的 `Reader` 类型变为 `strings.Reader` 。
> - `net` 包中的 `Error` 接口显然与网络错误有关。

尽早 `return` 而不是深度嵌套

由于 Go 语言的控制流不使用 `exception` ，因此不需要为 `try` 和 `catch` 块提供顶级结构而深度缩进代码。 Go 语言代码不是成功的路径越来越深地嵌套到右边，而是以一种风格编写，其中随着函数的进行，成功路径继续沿着屏幕向下移动。 我的朋友 Mat Ryer 将这种做法称为“视线”编码。@matryer/line-of-sight-in-code-186dd7cdea88"target="_blank">[4]

这是通过使用 `guard clauses` 来实现的; 在进入函数时是具有断言前提条件的条件块。 这是一个来自 `bytes` 包的例子:

```
func (b *Buffer) UnreadRune() error {
    if b.lastRead <= opInvalid {
        return errors.New("bytes.Buffer: UnreadRune: previous operation was not a successful ReadRune")
    }
    if b.off >= int(b.lastRead) {
        b.off -= int(b.lastRead)
    }
    b.lastRead = opInvalid
    return nil
}
```

进入 `UnreadRune` 后，将检查 `b.lastRead` 的状态，如果之前的操作不是 `ReadRune` ，则会立即返回错误。之后，函数的其余部分继续进行 `b.lastRead` 大于 `opInvalid` 的断言。

与没有 `guard clause` 的相同函数进行比较，

```
func (b *Buffer) UnreadRune() error {
    if b.lastRead > opInvalid {
        if b.off >= int(b.lastRead) {
            b.off -= int(b.lastRead)
        }
        b.lastRead = opInvalid
        return nil
    }
    return errors.New("bytes.Buffer: UnreadRune: previous operation was not a successful ReadRune")
}
```

最常见的执行成功的情况是嵌套在第一个if条件内，成功的退出条件是 `return nil` ，而且必须通过仔细匹配大括号来发现。 函数的最后一行是返回一个错误，并且被调用者必须追溯到匹配的左括号，以了解何时执行到此点。

对于读者和维护程序员来说，这更容易出错，因此 Go 语言更喜欢使用 `guard clauses` 并尽早返回错误。

让零值更有用

假设变量没有初始化，每个变量声明都会自动初始化为与零内存的内容相匹配的值。 这就是零值。 值的类型决定了其零值; 对于数字类型，它为 `0` ，对于指针类型为 `nil` ， `slices` 、 `map` 和 `channel` 同样是 `nil` 。

始终设置变量为已知默认值的属性对于程序的安全性和正确性非常重要，并且可以使 Go 语言程序更简单、更紧凑。 这就是 Go 程序员所说的“给你的结构一个有用的零值”。

对于 `sync.Mutex` 类型。 `sync.Mutex` 包含两个未公开的整数字段，它们用来表示互斥锁的内部状态。 每当声明 `sync.Mutex` 时，其字段会被设置为 `0` 初始值。 `sync.Mutex` 利用此属性来编写，使该类型可直接使用而无需初始化。

```
type MyInt struct {
    mu  sync.Mutex
    val int
}

func main() {
    var i MyInt

    // i.mu is usable without explicit initialisation.
    i.mu.Lock()
    i.val++
    i.mu.Unlock()
}
```

另一个利用零值的类型是 `bytes.Buffer` 。您可以声明 `bytes.Buffer` 然后就直接写入而无需初始化。

```
func main() {
    var b bytes.Buffer
    b.WriteString("Hello, world!\n")
    io.Copy(os.Stdout, &b)
}
```

切片的一个有用属性是它们的零值 `nil` 。如果我们看一下切片运行时 `header` 的定义就不难理解:

```
type slice struct {
        array *[...]T // pointer to the underlying array
        len   int
        cap   int
}
```

此结构的零值意味着 `len` 和 `cap` 的值为 `0` ，而 `array` （指向保存切片的内容数组的指针）将为 `nil` 。这意味着你不需要 `make` 切片，你只需声明它即可。

```
func main() {
    // s := make([]string, 0)
    // s := []string{}
    var s []string

    s = append(s, "Hello")
    s = append(s, "world")
    fmt.Println(strings.Join(s, " "))
}
```

> 注意:
> `var s []string` 类似于它上面的两条注释行，但并不完全相同。值为 `nil` 的切片与具有零长度的切片就可以来相互比较。以下代码将输出 `false` 。
>
> ```
> func main() {
>     var s1 = []string{}
>     var s2 []string
>     fmt.Println(reflect.DeepEqual(s1, s2))
> }
> ```

`nil pointers` - 未初始化的指针变量的一个有用属性是你可以在具有 `nil` 值的类型上调用方法。它可以简单地用于提供默认值。

```
type Config struct {
    path string
}

func (c *Config) Path() string {
    if c == nil {
        return "/usr/home"
    }
    return c.path
}

func main() {
    var c1 *Config
    var c2 = &Config{
        path: "/export",
    }
    fmt.Println(c1.Path(), c2.Path())
}
```

避免包级别状态

编写可维护程序的关键是它们应该是松散耦合的 - 对一个程序包的更改应该很少影响另一个不直接依赖于第一个程序包的程序包。

在 Go 语言中有两种很好的方法可以实现松散耦合

1. 使用接口来描述函数或方法所需的行为。
2. 避免使用全局状态。

在 Go 语言中，我们可以在函数或方法范围以及包范围内声明变量。当变量是公共的时，给定一个以大写字母开头的标识符，那么它的范围对于整个程序来说实际上是全局的 - 任何包都可以随时观察该变量的类型和内容。

可变全局状态引入程序的独立部分之间的紧密耦合，因为全局变量成为程序中每个函数的不可见参数！如果该变量的类型发生更改，则可以破坏依赖于全局变量的任何函数。如果程序的另一部分更改了该变量，则可以破坏依赖于全局变量状态的任何函数。

如果要减少全局变量所带来的耦合，

1. 将相关变量作为字段移动到需要它们的结构上。
2. 使用接口来减少行为与实现之间的耦合。

项目结构

我们来谈谈如何将包组合到项目中。 通常一个项目是一个 `git` 仓库，但在未来 Go 语言开发人员会交替地使用 `module` 和 `project` 。

就像一个包，每个项目都应该有一个明确的目的。 如果你的项目是一个库，它应该提供一件事，比如 `XML` 解析或记录。 您应该避免在一个包实现多个目的，这将有助于避免成为 `common` 库。

> 贴士:
> 据我的经验， `common` 库最终会与其最大的调用者紧密相连，在没有升级该库与最大调用者的情况下是很难修复的，还会带来了许多无关的更改以及API破坏。

如果你的项目是应用程序，如 `Web` 应用程序， `Kubernetes` 控制器等，那么项目中可能有一个或多个 `main` 程序包。 例如，我编写的 `Kubernetes` 控制器有一个 `cmd/contour` 包，既可以作为部署到 `Kubernetes` 集群的服务器，也可以作为调试目的的客户端。

考虑更少，更大的包

对于从其他语言过渡到 Go 语言的程序员来说，我倾向于在代码审查中提到的一件事是他们会过度使用包。

Go 语言没有提供有关可见性的详细方法; Java有 `public` 、 `protected` 、 `private` 以及隐式 `default` 的访问修饰符。 没有 `C++` 的 `friend` 类概念。

在 Go 语言中，我们只有两个访问修饰符， `public` 和 `private` ，由标识符的第一个字母的大小写表示。 如果标识符是公共的，则其名称以大写字母开头，该标识符可用于任何其他 Go 语言包的引用。

> 注意:
> 你可能会听到人们说 `exported` 与 `not exported` ,跟 `public` 和 `private` 是同义词。

鉴于包的符号的访问有限控件，Go 程序员应遵循哪些实践来避免创建过于复杂的包层次结构？

> 贴士:
> 除 `cmd/` 和 `internal/` 之外的每个包都应包含一些源代码。

我的建议是选择更少，更大的包。 你应该做的是不创建新的程序包。 这将导致太多类型被公开，为你的包创建一个宽而浅的 API。

以下部分将更为详细地探讨这一建议。

> 贴士:
> 来自 `Java` ？
> 如果您来自 `Java` 或 `C#` ，请考虑这一经验法则 – `Java` 包相当于单个 `.go` 源文件。 - Go 语言包相当于整个 `Maven` 模块或 `.NET` 程序集。

通过 `import` 语句将代码排列到文件中

如果你按照包提供的内容来安排你的程序包，是否需要对 Go 包中的文件也执行相同的操作？什么时候应该将 `.go` 文件拆分成多个文件？什么时候应该考虑整合 `.go` 文件？

以下是我的经验法则：

- 开始时使用一个 `.go` 文件。为该文件指定与文件夹名称相同的名称。例如: `package http` 应放在名为 `http` 的目录中名为 `http.go` 的文件中。
- 随着包的增长，您可能决定将各种职责任务拆分为不同的文件。例如： `messages.go` 包含 `Request` 和 `Response` 类型， `client.go` 包含 `Client` 类型， `server.go` 包含 `Server` 类型。
- 如果你的文件中 `import` 的声明类似，请考虑将它们组合起来。或者确定 `import` 集之间的差异并移动它们。
- 不同的文件应该负责包的不同区域。 `messages.go` 可能负责网络的 `HTTP` 请求和响应， `http.go` 可能包含底层网络处理逻辑， `client.go` 和 `server.go` 实现 `HTTP` 业务逻辑请求的实现或路由等等。

> 贴士: 首选名词为源文件命名。

> 注意:
> Go编译器并行编译每个包。 在一个包中，编译器并行编译每个函数（方法只是 Go 语言中函数的另一种写法）。 更改包中代码的布局不会影响编译时间。

优先内部测试再到外部测试

`go tool` 支持在两个地方编写 `testing` 包测试。假设你的包名为 `http2` ，您可以编写 `http2_test.go` 文件并使用包 `http2` 声明。这样做会编译 `http2_test.go` 中的代码，就像它是 `http2` 包的一部分一样。这就是内部测试。

`go tool` 还支持一个特殊的包声明，以 `test` 为结尾，即 `package http_test` 。这允许你的测试文件与代码一起存放在同一个包中，但是当编译时这些测试不是包的代码的一部分，它们存在于自己的包中。就像调用另一个包的代码一样来编写测试。这被称为外部测试。

我建议在编写单元测试时使用内部测试。这样你就可以直接测试每个函数或方法，避免外部测试干扰。

但是，你应该将 `Example` 测试函数放在外部测试文件中。这确保了在 `godoc` 中查看时，示例具有适当的包名前缀并且可以轻松地进行复制粘贴。

> 贴士:
> `避免复杂的包层次结构，抵制应用分类法`
> Go 语言包的层次结构对于 `go tool` 没有任何意义除了下一节要说的。 例如， `net/http` 包不是一个子包或者 `net` 包的子包。
>
> 如果在项目中创建了不包含 `.go` 文件的中间目录，则可能无法遵循此建议。

使用 `internal` 包来减少公共API

如果项目包含多个包，可能有一些公共的函数，这些函数旨在供项目中的其他包使用，但不打算成为项目的公共API的一部分。如果你发现是这种情况，那么 `go tool` 会识别一个特殊的文件夹名称 - 而非包名称 - `internal/` 可用于放置对项目公开的代码，但对其他项目是私有的。

要创建此类包，请将其放在名为 `internal/` 的目录中，或者放在名为 `internal/` 的目录的子目录中。 当 `go` 命令在其路径中看到导入包含 `internal` 的包时，它会验证执行导入的包是否位于 `internal` 目录。

例如， `.../a/b/c/internal/d/e/f` 的包只能通过以 `.../a/b/c/` 为根目录的代码被导入。 它无法通过 `.../a/b/g` 或任何其他仓库中的代码导入。[5]

保持 `main` 包内容尽可能的少

`main` 函数和 `main` 包的内容应尽可能少。 这是因为 `main.main` 充当单例; 程序中只能有一个 `main` 函数，包括 `tests` 。

因为 `main.main` 是一个单例，假设 `main` 函数中需要执行很多事情，`main.main` 只会在 `main.main` 或 `main.init` 中调用它们并且只调用一次。 这使得为 `main.main` 编写代码测试变得很困难，因此你应该将所有业务逻辑从 `main` 函数中移出，最好是从 `main` 包中移出。

> 贴士:
> `main` 应该做解析 `flags` ，开启数据库连接、开启日志等，然后将执行交给更高一级的对象。

API 设计

我今天要给出的最后一条建议是设计，我认为也是最重要的。

到目前为止我提出的所有建议都是建议。 这些是我尝试编写 Go 语言的方式，但我不打算在代码审查中拼命推广。

但是，在审查 API 时，我就不会那么宽容了。 这是因为到目前为止我所谈论的所有内容都是可以修复而且不会破坏向后兼容性；它们在很大程度上是实现的细节。

当涉及到软件包的公共 API 时，在初始设计中投入大量精力是值得的，因为稍后更改该设计对于已经使用 API 的人来说会是破坏性的。

设计难以被误用的 API

> APIs should be easy to use and hard to misuse.
> (API 应该易于使用且难以被误用)
> — Josh Bloch [3]

如果你从这个演讲中带走任何东西，那应该是 Josh Bloch 的建议。 如果一个 API 很难用于简单的事情，那么 API 的每次调用都会很复杂。 当 API 的实际调用很复杂时，它就会便得不那么明显，而且会更容易被忽视。

警惕采用几个相同类型参数的函数

简单，但难以正确使用的 API 是采用两个或更多相同类型参数的 API。 让我们比较两个函数签名：

```
func Max(a, b int) int
func CopyFile(to, from string) error
```

这两个函数有什么区别？ 显然，一个返回两个数字最大的那个，另一个是复制文件，但这不重要。

```
Max(8, 10) // 10
Max(10, 8) // 10
```

`Max` 是可交换的；参数的顺序无关紧要。 无论是 8 比 10 还是 10 比 8，最大的都是 10。

但是，却不适用于 `CopyFile` 。

```
CopyFile("/tmp/backup", "presentation.md")
CopyFile("presentation.md", "/tmp/backup")
```

这些声明中哪一个备份了 `presentation.md` ，哪一个用上周的版本覆盖了 `presentation.md` ？ 没有文档，你无法分辨。 如果没有查阅文档，代码审查员也无法知道你写对了顺序。

一种可能的解决方案是引入一个 `helper` 类型，它会负责如何正确地调用 `CopyFile` 。

```
type Source string

func (src Source) CopyTo(dest string) error {
    return CopyFile(dest, string(src))
}

func main() {
    var from Source = "presentation.md"
    from.CopyTo("/tmp/backup")
}
```

通过这种方式， `CopyFile` 总是能被正确调用 - 还可以通过单元测试 - 并且可以被设置为私有，进一步降低了误用的可能性。

> 贴士: 具有多个相同类型参数的API难以正确使用。

为其默认用例设计 API

几年前，我就对 `functional options` [7] 进行过讨论[6]，使 API 更易用于默认用例。

本演讲的主旨是你应该为常见用例设计 API。 另一方面， API 不应要求调用者提供他们不在乎参数。

不鼓励使用 `nil` 作为参数

本章开始时我建议是不要强迫提供给 API 的调用者他们不在乎的参数。 这就是我要说的为默认用例设计 API。

这是 `net/http` 包中的一个例子

```
package http

// ListenAndServe listens on the TCP network address addr and then calls
// Serve with handler to handle requests on incoming connections.
// Accepted connections are configured to enable TCP keep-alives.
//
// The handler is typically nil, in which case the DefaultServeMux is used.
//
// ListenAndServe always returns a non-nil error.
func ListenAndServe(addr string, handler Handler) error {
```

`ListenAndServe` 有两个参数，一个用于监听传入连接的 `TCP` 地址，另一个用于处理 `HTTP` 请求的 `http.Handler` 。 `Serve` 允许第二个参数为 `nil` ，需要注意的是调用者通常会传递 `nil` ，表示他们想要使用 `http.DefaultServeMux` 作为隐含参数。

现在， `Serve` 的调用者有两种方式可以做同样的事情。

```
http.ListenAndServe("0.0.0.0:8080", nil)
http.ListenAndServe("0.0.0.0:8080", http.DefaultServeMux)
```

两者完全相同。

这种 `nil` 行为是病毒式的。 `http` 包也有一个 `http.Serve` 帮助类，你可以合理地想象一下 `ListenAndServe` 是这样构建的

```
func ListenAndServe(addr string, handler Handler) error {
    l, err := net.Listen("tcp", addr)
    if err != nil {
        return err
    }
    defer l.Close()
    return Serve(l, handler)
}
```

因为 `ListenAndServe` 允许调用者为第二个参数传递 `nil` ，所以 `http.Serve` 也支持这种行为。 事实上， `http.Serve` 实现了如果 `handler` 是 `nil` ，使用 `DefaultServeMux` 的逻辑。 参数可为 `nil` 可能会导致调用者认为他们可以为两个参数都使用 `nil` 。 像下面这样:

```
http.Serve(nil, nil)
```

会导致 `panic` 。

> 贴士:
> 不要在同一个函数签名中混合使用可为 `nil` 和不能为 `nil` 的参数。

`http.ListenAndServe` 的作者试图在常见情况下让使用 API 的用户更轻松些，但很可能会让该程序包更难以被安全地使用。

使用 `DefaultServeMux` 或使用 `nil` 没有什么区别。

```
const root = http.Dir("/htdocs")
http.Handle("/", http.FileServer(root))
http.ListenAndServe("0.0.0.0:8080", nil)
```

对比

```
const root = http.Dir("/htdocs")
http.Handle("/", http.FileServer(root))
http.ListenAndServe("0.0.0.0:8080", http.DefaultServeMux)
```

这种混乱值得拯救吗？

```
const root = http.Dir("/htdocs")
mux := http.NewServeMux()
http.Handle("/", http.FileServer(root))
http.ListenAndServe("0.0.0.0:8080", mux)
```

> 贴士: 认真考虑 `helper` 函数会节省不少时间。 清晰要比简洁好。

> 贴士:
> `避免公共 API 使用测试参数`
> 避免在公开的 API 上使用仅在测试范围上不同的值。 相反，使用 `Public wrappers` 隐藏这些参数，使用辅助方式来设置测试范围中的属性。

首选可变参数函数而非 `[]T` 参数

编写一个带有切片参数的函数或方法是很常见的。

```
func ShutdownVMs(ids []string) error
```

这只是我编的一个例子，但它与我所写的很多代码相同。 这里的问题是他们假设他们会被调用于多个条目。 但是很多时候这些类型的函数只用一个参数调用，为了满足函数参数的要求，它必须打包到一个切片内。

另外，因为 `ids` 参数是切片，所以你可以将一个空切片或 `nil` 传递给该函数，编译也没什么错误。 但是这会增加额外的测试负载，因为你应该涵盖这些情况在测试中。

举一个这类 API 的例子，最近我重构了一条逻辑，要求我设置一些额外的字段，如果一组参数中至少有一个非零。 逻辑看起来像这样：

```
if svc.MaxConnections > 0 || svc.MaxPendingRequests > 0 || svc.MaxRequests > 0 || svc.MaxRetries > 0 {
    // apply the non zero parameters
}
```

由于 `if` 语句变得很长，我想将签出的逻辑拉入其自己的函数中。 这就是我提出的：

```
// anyPostive indicates if any value is greater than zero.
func anyPositive(values ...int) bool {
    for _, v := range values {
        if v > 0 {
            return true
        }
    }
    return false
}
```

这就能够向读者明确内部块的执行条件：

```
if anyPositive(svc.MaxConnections, svc.MaxPendingRequests, svc.MaxRequests, svc.MaxRetries) {
        // apply the non zero parameters
}
```

但是 `anyPositive` 还存在一个问题，有人可能会这样调用它:

```
if anyPositive() { ... }
```

在这种情况下， `anyPositive` 将返回 `false` ，因为它不会执行迭代而是立即返回 `false` 。对比起如果 `anyPositive` 在没有传递参数时返回 `true` ，这还不算世界上最糟糕的事情。

然而，如果我们可以更改 `anyPositive` 的签名以强制调用者应该传递至少一个参数，那会更好。我们可以通过组合正常和可变参数来做到这一点，如下所示：

```
// anyPostive indicates if any value is greater than zero.
func anyPositive(first int, rest ...int) bool {
    if first > 0 {
        return true
    }
    for _, v := range rest {
        if v > 0 {
            return true
        }
    }
    return false
}
```

现在不能使用少于一个参数来调用 `anyPositive` 。

让函数定义它们所需的行为

假设我需要编写一个将 `Document` 结构保存到磁盘的函数的任务。

```
// Save writes the contents of doc to the file f.
func Save(f *os.File, doc *Document) error
```

我可以指定这个函数 `Save` ，它将 `*os.File` 作为写入 `Document` 的目标。但这样做会有一些问题

`Save` 的签名排除了将数据写入网络位置的选项。假设网络存储可能在以后成为需求，则此功能的签名必须改变，从而影响其所有调用者。

`Save` 测试起来也很麻烦，因为它直接操作磁盘上的文件。因此，为了验证其操作，测试时必须在写入文件后再读取该文件的内容。

而且我必须确保 `f` 被写入临时位置并且随后要将其删除。

`*os.File` 还定义了许多与 `Save` 无关的方法，比如读取目录并检查路径是否是符号链接。 如果 `Save` 函数的签名只用 `*os.File` 的相关内容，那将会很有用。

我们能做什么 ？

```
// Save writes the contents of doc to the supplied
// ReadWriterCloser.
func Save(rwc io.ReadWriteCloser, doc *Document) error
```

使用 `io.ReadWriteCloser` ，我们可以应用接口隔离原则来重新定义 `Save` 以获取更通用文件形式。

通过此更改，任何实现 `io.ReadWriteCloser` 接口的类型都可以替换以前的 `*os.File` 。

这使 `Save` 在其应用程序中更广泛，并向 `Save` 的调用者阐明 `*os.File` 类型的哪些方法与其操作有关。

而且， `Save` 的作者也不可以在 `*os.File` 上调用那些不相关的方法，因为它隐藏在 `io.ReadWriteCloser` 接口后面。

但我们可以进一步采用接口隔离原则。

首先，如果 `Save` 遵循单一功能原则，它不可能读取它刚刚写入的文件来验证其内容 - 这应该是另一段代码的功能。

```
// Save writes the contents of doc to the supplied
// WriteCloser.
func Save(wc io.WriteCloser, doc *Document) error
```

因此，我们可以将我们传递给 `Save` 的接口的规范缩小到只写和关闭。

其次，通过向 `Save` 提供一个关闭其流的机制，使其看起来仍然像一个文件，这就提出了在什么情况下关闭 `wc` 的问题。

可能 `Save` 会无条件地调用 `Close` ，或者在成功的情况下调用 `Close` 。

这给 `Save` 的调用者带来了问题，因为它可能希望在写入文档后将其他数据写入流。

```
// Save writes the contents of doc to the supplied
// Writer.
```

```
func Save(w io.Writer, doc *Document) error
```

一个更好的解决方案是重新定义 `Save` 仅使用 `io.Writer` ，它只负责将数据写入流。

将接口隔离原则应用于我们的 `Save` 功能，同时，就需求而言，得出了最具体的一个函数 - 它只需要一个可写的东西 - 并且它的功能最通用，现在我们可以使用 `Save` 将我们的数据保存到实现 `io.Writer` 的任何事物中。

[译注: 不理解设计原则部分的同学可以阅读 Dave 大神的另一篇《Go 语言 SOLID 设计》]

错误处理

通过消除错误来消除错误处理

如果你昨天在我的演讲中，我谈到了改进错误处理的提案。但是你知道有什么比改进错误处理的语法更好吗？那就是根本不需要处理错误。

> 注意：
> 我不是说“删除你的错误处理”。我的建议是，修改你的代码，这样就不用处理错误了。

本节从 John Ousterhout 最近的著作“软件设计哲学”[9]中汲取灵感。该书的其中一章是“定义不存在的错误”。我们将尝试将此建议应用于 Go 语言。

计算行数

让我们编写一个函数来计算文件中的行数。

```
func CountLines(r io.Reader) (int, error) {
    var (
        br    = bufio.NewReader(r)
        lines int
        err   error
    )

    for {
        _, err = br.ReadString('\n')
        lines++
        if err != nil {
            break
        }
    }

    if err != io.EOF {
        return 0, err
    }
    return lines, nil
}
```

由于我们遵循前面部分的建议，`CountLines` 需要一个 `io.Reader`，而不是一个 `*File`；它的任务是调用者为我们想要计算的内容提供 `io.Reader`。

我们构造一个 `bufio.Reader`，然后在一个循环中调用 `ReadString` 方法，递增计数器直到我们到达文件的末尾，然后我们返回读取的行数。

至少这是我们想要编写的代码，但是这个函数由于需要错误处理而变得更加复杂。 例如，有这样一个奇怪的结构：

```
_, err = br.ReadString('\n')
lines++
if err != nil {
    break
}
```

我们在检查错误之前增加了行数，这样做看起来很奇怪。

我们必须以这种方式编写它的原因是，如果在遇到换行符之前就读到文件结束，则 `ReadString` 将返回错误。如果文件中没有换行符，同样会出现这种情况。

为了解决这个问题，我们重新排列逻辑增来加行数，然后查看是否需要退出循环。

> 注意:
> 这个逻辑仍然不完美，你能发现错误吗？

但是我们还没有完成检查错误。当 `ReadString` 到达文件末尾时，预期它会返回 `io.EOF` 。 `ReadString` 需要某种方式在没有什么可读时来停止。因此，在我们将错误返回给 `CountLine` 的调用者之前，我们需要检查错误是否是 `io.EOF` ，如果不是将其错误返回，否则我们返回 `nil` 说一切正常。

我认为这是 Russ Cox 观察到错误处理可能会模糊函数操作的一个很好的例子。我们来看一个改进的版本。

```
func CountLines(r io.Reader) (int, error) {
    sc := bufio.NewScanner(r)
    lines := 0

    for sc.Scan() {
        lines++
    }
    return lines, sc.Err()
}
```

这个改进的版本从 `bufio.Reader` 切换到 `bufio.Scanner` 。

在 `bufio.Scanner` 内部使用 `bufio.Reader` ，但它添加了一个很好的抽象层，它有助于通过隐藏 `CountLines` 的操作来消除错误处理。

> 注意:
> `bufio.Scanner` 可以扫描任何模式，但默认情况下它会查找换行符。

如果扫描程序匹配了一行文本并且没有遇到错误，则 `sc.Scan()` 方法返回 `true` 。因此，只有当扫描仪的缓冲区中有一行文本时，才会调用 `for` 循环的主体。这意味着我们修改后的 `CountLines` 正确处理没有换行符的情况，并且还处理文件为空的情况。

其次，当 `sc.Scan` 在遇到错误时返回 `false` ，我们的 `for` 循环将在到达文件结尾或遇到错误时退出。 `bufio.Scanner` 类型会记住遇到的第一个错误，一旦我们使用 `sc.Err()` 方法退出循环，我们就可以获取该错误。

最后， `sc.Err()` 负责处理 `io.EOF` 并在达到文件末尾时将其转换为 `nil` ，而不会遇到其他错误。

> 贴士:
> 当遇到难以忍受的错误处理时，请尝试将某些操作提取到辅助程序类型中。

WriteResponse

我的第二个例子受到了 `Errors are values` 博客文章[10]的启发。

在本章前面我们已经看过处理打开、写入和关闭文件的示例。错误处理是存在的，但是接收范围内的，因为操作可以封装在诸如 `ioutil.ReadFile` 和 `ioutil.WriteFile` 之类的辅助程序中。但是，在处理底层网络协议时，有必要使用 `I/O` 原始的错误处理来直接构建响应，这样就可能会变得重复。看一下构建 `HTTP` 响应的 `HTTP` 服务器的这个片段。

```
type Header struct {
    Key, Value string
}

type Status struct {
    Code   int
    Reason string
}
```

```
func WriteResponse(w io.Writer, st Status, headers []Header, body io.Reader) error {
    _, err := fmt.Fprintf(w, "HTTP/1.1 %d %s\r\n", st.Code, st.Reason)
    if err != nil {
        return err
    }

    for _, h := range headers {
        _, err := fmt.Fprintf(w, "%s: %s\r\n", h.Key, h.Value)
        if err != nil {
            return err
        }
    }

    if _, err := fmt.Fprint(w, "\r\n"); err != nil {
        return err
    }

    _, err = io.Copy(w, body)
    return err
}
```

首先，我们使用 `fmt.Fprintf` 构造状态码并检查错误。 然后对于每个标题，我们写入键值对，每次都检查错误。 最后，我们使用额外的 `\r\n` 终止标题部分，检查错误之后将响应主体复制到客户端。 最后，虽然我们不需要检查 `io.Copy` 中的错误，但我们需要将 `io.Copy` 返回的两个返回值形式转换为 `WriteResponse` 的单个返回值。

这里很多重复性的工作。 我们可以通过引入一个包装器类型 `errWriter` 来使其更容易。

`errWriter` 实现 `io.Writer` 接口，因此可用于包装现有的 `io.Writer` 。 `errWriter` 写入传递给其底层 `writer` ，直到检测到错误。 从此时起，它会丢弃任何写入并返回先前的错误。

```
type errWriter struct {
    io.Writer
    err error
}

func (e *errWriter) Write(buf []byte) (int, error) {
    if e.err != nil {
        return 0, e.err
    }
    var n int
    n, e.err = e.Writer.Write(buf)
    return n, nil
}

func WriteResponse(w io.Writer, st Status, headers []Header, body io.Reader) error {
    ew := &errWriter{Writer: w}
    fmt.Fprintf(ew, "HTTP/1.1 %d %s\r\n", st.Code, st.Reason)

    for _, h := range headers {
        fmt.Fprintf(ew, "%s: %s\r\n", h.Key, h.Value)
    }

    fmt.Fprint(ew, "\r\n")
    io.Copy(ew, body)
    return ew.err
}
```

将 `errWriter` 应用于 `WriteResponse` 可以显着提高代码的清晰度。 每个操作不再需要自己做错误检查。 通过检查 `ew.err` 字段，将错误报告移动到函数末尾，从而避免转换从 `io.Copy` 的两个返回值。

错误只处理一次

最后，我想提一下你应该只处理错误一次。 处理错误意味着检查错误值并做出单一决定。

```
// WriteAll writes the contents of buf to the supplied writer.
func WriteAll(w io.Writer, buf []byte) {
        w.Write(buf)
}
```

如果你做出的决定少于一个，则忽略该错误。 正如我们在这里看到的那样， `w.WriteAll` 的错误被丢弃。

但是，针对单个错误做出多个决策也是有问题的。 以下是我经常遇到的代码。

```
func WriteAll(w io.Writer, buf []byte) error {
    _, err := w.Write(buf)
    if err != nil {
        log.Println("unable to write:", err) // annotated error goes to log file
        return err                           // unannotated error returned to caller
    }
    return nil
}
```

在此示例中，如果在 `w.Write` 期间发生错误，则会写入日志文件，注明错误发生的文件与行数，并且错误也会返回给调用者，调用者可能会记录该错误并将其返回到上一级，一直回到程序的顶部。

调用者可能正在做同样的事情

```
func WriteConfig(w io.Writer, conf *Config) error {
    buf, err := json.Marshal(conf)
    if err != nil {
        log.Printf("could not marshal config: %v", err)
        return err
    }
    if err := WriteAll(w, buf); err != nil {
        log.Println("could not write config: %v", err)
        return err
    }
    return nil
}
```

因此你在日志文件中得到一堆重复的内容，

```
unable to write: io.EOF
could not write config: io.EOF
```

但在程序的顶部，虽然得到了原始错误，但没有相关内容。

```
err := WriteConfig(f, &conf)
fmt.Println(err) // io.EOF
```

我想深入研究这一点，因为作为个人偏好，我并没有看到 `logging` 和返回的问题。

```
func WriteConfig(w io.Writer, conf *Config) error {
    buf, err := json.Marshal(conf)
    if err != nil {
        log.Printf("could not marshal config: %v", err)
        // oops, forgot to return
    }
    if err := WriteAll(w, buf); err != nil {
        log.Println("could not write config: %v", err)
        return err
    }
    return nil
}
```

很多问题是程序员忘记从错误中返回。正如我们之前谈到的那样，Go 语言风格是使用 `guard clauses` 以及检查前提条件作为函数进展并提前返回。

在这个例子中，作者检查了错误，记录了它，但忘了返回。这就引起了一个微妙的错误。

Go 语言中的错误处理规定，如果出现错误，你不能对其他返回值的内容做出任何假设。由于 `JSON` 解析失败，`buf` 的内容未知，可能它什么都没有，但更糟的是它可能包含解析的 `JSON` 片段部分。

由于程序员在检查并记录错误后忘记返回，因此损坏的缓冲区将传递给 `WriteAll`，这可能会成功，因此配置文件将被错误地写入。但是，该函数会正常返回，并且发生问题的唯一日志行是有关 `JSON` 解析错误，而与写入配置失败有关。

为错误添加相关内容

发生错误的原因是作者试图在错误消息中添加 `context`。他们试图给自己留下一些线索，指出错误的根源。

让我们看看使用 `fmt.Errorf` 的另一种方式。

```
func WriteConfig(w io.Writer, conf *Config) error {
    buf, err := json.Marshal(conf)
    if err != nil {
        return fmt.Errorf("could not marshal config: %v", err)
    }
    if err := WriteAll(w, buf); err != nil {
        return fmt.Errorf("could not write config: %v", err)
    }
    return nil
}

func WriteAll(w io.Writer, buf []byte) error {
    _, err := w.Write(buf)
    if err != nil {
        return fmt.Errorf("write failed: %v", err)
    }
    return nil
}
```

通过将注释与返回的错误组合起来，就更难以忘记错误的返回来避免意外继续。

如果写入文件时发生 `I/O` 错误，则 `error` 的 `Error()` 方法会报告以下类似的内容;

```
could not write config: write failed: input/output error
```

使用 `github.com/pkg/errors` 包装 `errors`

`fmt.Errorf` 模式适用于注释错误 `message`，但这样做的代价是模糊了原始错误的类型。 我认为将错误视为不透明值对于松散耦合的软件非常重要，因此如果你使用错误值做的唯一事情是原始错误的类型应该无关紧要的面孔

1. 检查它是否为 `nil`。
2. 输出或记录它。

但是在某些情况下，我认为它们并不常见，您需要恢复原始错误。 在这种情况下，使用类似我的 `errors` 包来注释这样的错误，如下

```
func ReadFile(path string) ([]byte, error) {
    f, err := os.Open(path)
    if err != nil {
        return nil, errors.Wrap(err, "open failed")
    }
    defer f.Close()

    buf, err := ioutil.ReadAll(f)
    if err != nil {
        return nil, errors.Wrap(err, "read failed")
    }
    return buf, nil
}

func ReadConfig() ([]byte, error) {
    home := os.Getenv("HOME")
    config, err := ReadFile(filepath.Join(home, ".settings.xml"))
    return config, errors.WithMessage(err, "could not read config")
}

func main() {
    _, err := ReadConfig()
    if err != nil {
        fmt.Println(err)
        os.Exit(1)
    }
}
```

现在报告的错误就是 `K&D` [11]样式错误，

```
could not read config: open failed: open /Users/dfc/.settings.xml: no such file or directory
```

并且错误值保留对原始原因的引用。

```
func main() {
    _, err := ReadConfig()
    if err != nil {
        fmt.Printf("original error: %T %v\n", errors.Cause(err), errors.Cause(err))
        fmt.Printf("stack trace:\n%+v\n", err)
        os.Exit(1)
    }
}
```

因此，你可以恢复原始错误并打印堆栈跟踪;

```
original error: *os.PathError open /Users/dfc/.settings.xml: no such file or directory
stack trace:
open /Users/dfc/.settings.xml: no such file or directory
```

```
open failed
main.ReadFile
        /Users/dfc/devel/practical-go/src/errors/readfile2.go:16
main.ReadConfig
        /Users/dfc/devel/practical-go/src/errors/readfile2.go:29
main.main
        /Users/dfc/devel/practical-go/src/errors/readfile2.go:35
runtime.main
        /Users/dfc/go/src/runtime/proc.go:201
runtime.goexit
        /Users/dfc/go/src/runtime/asm_amd64.s:1333
could not read config
```

使用 `errors` 包，你可以以人和机器都可检查的方式向错误值添加上下文。 如果昨天你来听我的演讲，你会知道这个库在被移植到即将发布的 Go 语言版本的标准库中。

并发

由于 Go 语言的并发功能，经常被选作项目编程语言。 Go 语言团队已经竭尽全力以廉价（在硬件资源方面）和高性能来实现并发，但是 Go 语言的并发功能也可以被用来编写性能不高同时也不太可靠的代码。在结尾，我想留下一些建议，以避免 Go 语言的并发功能带来的一些陷阱。

Go 语言以 `channels` 以及 `select` 和 `go` 语句来支持并发。如果你已经从书籍或培训课程中正式学习了 Go 语言，你可能已经注意到并发部分始终是这些课程的最后一部分。这个研讨会也没有什么不同，我选择最后覆盖并发，好像它是 Go 程序员应该掌握的常规技能的额外补充。

这里有一个二分法; Go 语言的最大特点是简单、轻量级的并发模型。作为一种产品，我们的语言几乎只推广这个功能。另一方面，有一种说法认为并发使用起来实际上并不容易，否则作者不会把它作为他们书中的最后一章，我们也不会遗憾地来回顾其形成过程。

本节讨论了 Go 语言的并发功能的“坑”。

保持自己忙碌或做自己的工作

这个程序有什么问题？

```
package main

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

func main() {
    http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
        fmt.Fprintln(w, "Hello, GopherCon SG")
    })
    go func() {
        if err := http.ListenAndServe(":8080", nil); err != nil {
            log.Fatal(err)
        }
    }()

    for {
    }
}
```

该程序实现了我们的预期，它提供简单的 Web 服务。 然而，它同时也做了其他事情，它在无限循环中浪费 CPU 资源。 这是因为 `main` 的最后一行上的 `for {}` 将阻塞 `main goroutine` ，因为它不执行任何 IO、等待锁定、发送或接收通道数据或以其他方式与调度器通信。

由于 Go 语言运行时主要是协同调度，该程序将在单个 CPU 上做无效地旋转，并可能最终实时锁定。

我们如何解决这个问题？ 这是一个建议。

```
package main

import (
    "fmt"
    "log"
    "net/http"
    "runtime"
)

func main() {
    http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
        fmt.Fprintln(w, "Hello, GopherCon SG")
    })
    go func() {
        if err := http.ListenAndServe(":8080", nil); err != nil {
            log.Fatal(err)
        }
    }()

    for {
        runtime.Gosched()
    }
}
```

这看起来很愚蠢，但这是我看过的一种常见解决方案。 这是不了解潜在问题的症状。

现在，如果你有更多的经验，你可能会写这样的东西。

```
package main

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

func main() {
    http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
        fmt.Fprintln(w, "Hello, GopherCon SG")
    })
    go func() {
        if err := http.ListenAndServe(":8080", nil); err != nil {
            log.Fatal(err)
        }
    }()

    select {}
}
```

空的 `select` 语句将永远阻塞。 这是一个有用的属性，因为现在我们不再调用 `runtime.GoSched()` 而耗费整个 CPU。 但是这也只是治疗了症状，而不是病根。

我想向你提出另一种你可能在用的解决方案。 与其在 `goroutine` 中运行 `http.ListenAndServe`，会给我们留下处理 `main goroutine` 的问题，不如在 `main goroutine` 本身上运行 `http.ListenAndServe`。

> 贴士:
> 如果 Go 语言程序的 `main.main` 函数返回，无论程序在一段时间内启动的其他 `goroutine` 在做什么，Go 语言程序会无条件地退出。

```
package main

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

func main() {
    http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
        fmt.Fprintln(w, "Hello, GopherCon SG")
    })
    if err := http.ListenAndServe(":8080", nil); err != nil {
        log.Fatal(err)
    }
}
```

所以这是我的第一条建议：如果你的 `goroutine` 在得到另一个结果之前无法取得进展，那么让自己完成此工作而不是委托给其他 `goroutine` 会更简单。

这通常会消除将结果从 `goroutine` 返回到其启动程序所需的大量状态跟踪和通道操作。

> 贴士:
> 许多 Go 程序员过度使用 `goroutine` ，特别是刚开始时。与生活中的所有事情一样，适度是成功的关键。

将并发性留给调用者

以下两个 API 有什么区别？

```
// ListDirectory returns the contents of dir.
func ListDirectory(dir string) ([]string, error)
```

```
// ListDirectory returns a channel over which
// directory entries will be published. When the list
// of entries is exhausted, the channel will be closed.
func ListDirectory(dir string) chan string
```

首先，最明显的不同: 第一个示例将目录读入切片然后返回整个切片，如果出错则返回错误。这是同步发生的， `ListDirectory` 的调用者会阻塞，直到读取了所有目录条目。根据目录的大小，这可能需要很长时间，并且可能会分配大量内存来构建目录条目。

让我们看看第二个例子。 这个示例更像是 Go 语言风格， `ListDirectory` 返回一个通道，通过该通道传递目录条目。当通道关闭时，表明没有更多目录条目。由于在 `ListDirectory` 返回后发生了通道的填充， `ListDirectory` 可能会启动一个 `goroutine` 来填充通道。

> 注意:
> 第二个版本实际上不必使用 Go 协程; 它可以分配一个足以保存所有目录条目而不阻塞的通道，填充通道，关闭它，然后将通道返回给调用者。但这样做不太现实，因为会消耗大量内存来缓冲通道中的所有结果。

通道版本的 `ListDirectory` 还有两个问题：

- 通过使用关闭通道作为没有其他项目要处理的信号，在中途遇到了错误时， `ListDirectory` 无法告诉调用者通过通道返回的项目集是否完整。调用者无法区分空目录和读取目录的错误。两者都导致从 `ListDirectory` 返回的通道立即关闭。
- 调用者必须持续从通道中读取，直到它被关闭，因为这是调用者知道此通道的是否停止的唯一方式。这是对 `ListDirectory` 使用的严重限制，即使可能已经收到了它想要的答案，调用者也必须花时间从通道中读取。就中型到大型目录的内存使用而言，它可能更有效，但这种方法并不比原始的基于切片的方法快。

以上两种实现所带来的问题的解决方案是使用回调，该回调是在执行时在每个目录条目的上下文中调用函数。

```
func ListDirectory(dir string, fn func(string))
```

毫不奇怪，这就是 `filepath.WalkDir` 函数的工作方式。

> 贴士:
> 如果你的函数启动了 `goroutine` ，你必须为调用者提供一种明确停止 `goroutine` 的方法。 把异步执行函数的决定留给该函数的调用者通常会更容易些。

永远不要启动一个停止不了的 `goroutine`

前面的例子显示当一个任务时没有必要时使用 `goroutine` 。但使用 Go 语言的原因之一是该语言提供的并发功能。实际上，很多情况下你希望利用硬件中可用的并行性。为此，你必须使用 `goroutines` 。

这个简单的应用程序在两个不同的端口上提供 `http` 服务，端口 `8080` 用于应用程序服务，端口 `8001` 用于访问 `/debug/pprof` 终端。

```
package main

import (
    "fmt"
    "net/http"
    _ "net/http/pprof"
)

func main() {
    mux := http.NewServeMux()
    mux.HandleFunc("/", func(resp http.ResponseWriter, req *http.Request) {
        fmt.Fprintln(resp, "Hello, QCon!")
    })
    go http.ListenAndServe("127.0.0.1:8001", http.DefaultServeMux) // debug
    http.ListenAndServe("0.0.0.0:8080", mux)                       // app traffic
}
```

虽然这个程序不是很复杂，但它代表了真实应用程序的基础。

该应用程序存在一些问题，因为它随着应用程序的增长而显露出来，所以我们现在来解决其中的一些问题。

```
func serveApp() {
    mux := http.NewServeMux()
    mux.HandleFunc("/", func(resp http.ResponseWriter, req *http.Request) {
        fmt.Fprintln(resp, "Hello, QCon!")
    })
    http.ListenAndServe("0.0.0.0:8080", mux)
}

func serveDebug() {
    http.ListenAndServe("127.0.0.1:8001", http.DefaultServeMux)
}

func main() {
    go serveDebug()
    serveApp()
}
```

通过将 `serveApp` 和 `serveDebug` 处理程序分解成为它们自己的函数，我们将它们与 `main.main` 分离。 也遵循了上面的建议，并确保 `serveApp` 和 `serveDebug` 将它们的并发性留给调用者。

但是这个程序存在一些可操作性问题。 如果 `serveApp` 返回，那么 `main.main` 将返回，导致程序关闭并由你使用的进程管理器来重新启动。

> 贴士:
> 正如 Go 语言中的函数将并发性留给调用者一样，应用程序应该将监视其状态和检测是否重启的工作留给另外的程序来做。 不要让你的应用程序负责重新启动自己，最好从应用程序外部处理该过程。

然而， `serveDebug` 是在一个单独的 `goroutine` 中运行的，返回后该 `goroutine` 将退出，而程序的其余部分继续。 由于 `/debug` 处理程序已停止工作很久，因此操作人员不会很高兴发现他们无法在你的应用程序中获取统计信息。

我们想要确保的是，如果任何负责提供此应用程序的 `goroutine` 停止，我们将关闭该应用程序。

```
func serveApp() {
    mux := http.NewServeMux()
    mux.HandleFunc("/", func(resp http.ResponseWriter, req *http.Request) {
        fmt.Fprintln(resp, "Hello, QCon!")
    })
    if err := http.ListenAndServe("0.0.0.0:8080", mux); err != nil {
        log.Fatal(err)
    }
}

func serveDebug() {
    if err := http.ListenAndServe("127.0.0.1:8001", http.DefaultServeMux); err != nil {
        log.Fatal(err)
    }
}

func main() {
    go serveDebug()
    go serveApp()
    select {}
}
```

现在 `serverApp` 和 `serveDebug` 检查从 `ListenAndServe` 返回的错误，并在需要时调用 `log.Fatal` 。因为两个处理程序都在 `goroutine` 中运行，所以我们将 `main goroutine` 停在 `select{}` 中。

这种方法存在许多问题：

1. 如果 `ListenAndServer` 返回 `nil` 错误，则不会调用 `log.Fatal` ，并且该端口上的 HTTP 服务将在不停止应用程序的情况下关闭。
2. `log.Fatal` 调用 `os.Exit` ，它将无条件地退出程序; `defer` 不会被调用，其他 `goroutines` 也不会被通知关闭，程序就停止了。 这使得编写这些函数的测试变得困难。

> 贴士:
> 只在 `main.main` 或 `init` 函数中的使用 `log.Fatal` 。

我们真正想要的是任何错误发送回 `goroutine` 的调用者，以便它可以知道 `goroutine` 停止的原因，可以干净地关闭程序进程。

```
func serveApp() error {
    mux := http.NewServeMux()
    mux.HandleFunc("/", func(resp http.ResponseWriter, req *http.Request) {
        fmt.Fprintln(resp, "Hello, QCon!")
    })
    return http.ListenAndServe("0.0.0.0:8080", mux)
}

func serveDebug() error {
    return http.ListenAndServe("127.0.0.1:8001", http.DefaultServeMux)
}

func main() {
    done := make(chan error, 2)
```

```
    go func() {
        done <- serveDebug()
    }()
    go func() {
        done <- serveApp()
    }()

    for i := 0; i < cap(done); i++ {
        if err := <-done; err != nil {
            fmt.Println("error: %v", err)
        }
    }
}
```

我们可以使用通道来收集 `goroutine` 的返回状态。通道的大小等于我们想要管理的 `goroutine` 的数量，这样发送到 `done` 通道就不会阻塞，因为这会阻止 `goroutine` 的关闭，导致它泄漏。

由于没有办法安全地关闭 `done` 通道，我们不能使用 `for range` 来循环通道直到获取所有 `goroutine` 发来的报告，而是循环我们开启的多个 `goroutine`，即通道的容量。

现在我们有办法等待每个 `goroutine` 干净地退出并记录他们遇到的错误。所需要的只是一种从第一个 `goroutine` 转发关闭信号到其他 `goroutine` 的方法。

事实证明，要求 `http.Server` 关闭是有点牵扯的，所以我将这个逻辑转给辅助函数。`serve` 助手使用一个地址和 `http.Handler`，类似于 `http.ListenAndServe`，还有一个 `stop` 通道，我们用它来触发 `Shutdown` 方法。

```
func serve(addr string, handler http.Handler, stop <-chan struct{}) error {
    s := http.Server{
        Addr:    addr,
        Handler: handler,
    }

    go func() {
        <-stop // wait for stop signal
        s.Shutdown(context.Background())
    }()

    return s.ListenAndServe()
}

func serveApp(stop <-chan struct{}) error {
    mux := http.NewServeMux()
    mux.HandleFunc("/", func(resp http.ResponseWriter, req *http.Request) {
        fmt.Fprintln(resp, "Hello, QCon!")
    })
    return serve("0.0.0.0:8080", mux, stop)
}

func serveDebug(stop <-chan struct{}) error {
    return serve("127.0.0.1:8001", http.DefaultServeMux, stop)
}

func main() {
    done := make(chan error, 2)
    stop := make(chan struct{})
    go func() {
        done <- serveDebug(stop)
    }()
    go func() {
```

```
        done <- serveApp(stop)
    }()

    var stopped bool
    for i := 0; i < cap(done); i++ {
        if err := <-done; err != nil {
            fmt.Println("error: %v", err)
        }
        if !stopped {
            stopped = true
            close(stop)
        }
    }
}
```

现在，每次我们在 `done` 通道上收到一个值时，我们关闭 `stop` 通道，这会导致在该通道上等待的所有 `goroutine` 关闭其 `http.Server` 。这反过来将导致其余所有的 `ListenAndServe` `goroutines` 返回。一旦我们开启的所有 `goroutine` 都停止了， `main.main` 就会返回并且进程会干净地停止。

> 贴士:
> 自己编写这种逻辑是重复而微妙的。 参考下这个包: https://github.com/heptio/workgroup，它会为你完成大部分工作。

> *引用: *
>
> 1. https://gaston.life/books/effective-programming/
>
> 2. https://talks.golang.org/2014/names.slide#4
>
> 3. https://www.infoq.com/articles/API-Design-Joshua-Bloch
>
> 1. https://www.lysator.liu.se/c/pikestyle.html
>
> 2. https://speakerdeck.com/campoy/understanding-nil
>
> 3. https://www.youtube.com/watch?v=Ic2y6w8lMPA
>
> 4. @matryer/line-of-sight-in-code-186dd7cdea88"target="_blank"">https://medium.com/@matryer/line-of-sight-in-code-186dd7cdea88
>
> 5. https://golang.org/doc/go1.4#internalpackages
>
> 6. https://dave.cheney.net/2014/10/17/functional-options-for-friendly-apis
>
> 7. https://commandcenter.blogspot.com/2014/01/self-referential-functions-and-design.html
>
> 8. https://dave.cheney.net/2016/04/27/dont-just-check-errors-handle-them-gracefully
>
> 9. https://www.amazon.com/Philosophy-Software-Design-John-Ousterhout/dp/1732102201
>
> 10. https://blog.golang.org/errors-are-values
>
> 11. http://www.gopl.io/

原文链接：Practical Go: Real world advice for writing maintainable Go programs

- *如有翻译有误或者不理解的地方，请评论指正*
- *待更新的译注之后会做进一步修改翻译*
- *翻译：田浩*
- *邮箱：llitfkitfk@gmail.com*