Go 风格指南

原文：https://google.github.io/styleguide/go

概述 | 风格指南 | 风格决策 | 最佳实践

风格原则

以下几条总体原则总结了如何编写可读的 Go 代码。以下为具有可读性的代码特征，按重要性排序：

1. 清晰：代码的目的和设计原理对读者来说是清楚的。
2. 简约：代码以最简单的方式来完成它的目的。
3. 简洁：代码具有很高的信噪比，即写出来的代码是有意义的，非可有可无的。
4. 可维护性：代码可以很容易地被维护。
5. 一致：代码与更广泛的 Google 代码库一致。

清晰

可读性的核心目标是写出对读者来说很清晰的代码。

清晰性主要是通过有效的命名、有用的注释和有效的代码组织来实现的。

清晰性要从读者的角度来看，而不是从代码的作者的角度来看，代码的易读性比易写性更重要。代码的清晰性有两个不同的方面：

- 该代码实际上在做什么？
- 为什么代码会这么做？

该代码实际上在做什么

Go 被设计得应该是可以比较直接地看到代码在做什么。在不确定的情况下或者读者可能需要先验知识才能理解代码的情况下，我们值得投入时间以使代码的目的对未来的读者更加明确。例如，它可能有助于：

- 使用更具描述性的变量名称
- 添加额外的评论
- 使用空白与注释来划分代码
- 将代码重构为独立的函数/方法，使其更加模块化

这里没有一个放之四海而皆准的方法，但在开发 Go 代码时，优先考虑清晰性是很重要的。

为什么代码会这么做

代码的基本原理通常由变量、函数、方法或包的名称充分传达。如果不是这样，添加注释是很重要的。当代码中包含读者可能不熟悉的细节时，“为什么？”就显得尤为重要，例如：

- 编程语言中的细微差别，例如，一个闭包将捕获一个循环变量，但闭包在许多行之外
- 业务逻辑的细微差别，例如，需要区分实际用户和虚假用户的访问控制检查

一个 API 可能需要小心翼翼才能正确使用。例如，由于性能原因，一段代码可能错综复杂，难以理解，或者一连串复杂的数学运算可能以一种意想不到的方式使用类型转换。在这些以及更多的情况下，附带的注释和文档对这些方面进行解释是很重要的，这样未来的维护者就不会犯错，读者也可以理解代码而不需要进行逆向工程。

同样重要的是，我们要意识到，一些基于清晰性考虑的尝试（如添加额外的注释），实际上会通过增加杂乱无章的内容、重述代码已经说过的内容、与代码相矛盾或增加维护负担来保持注释的最新性，以此来掩盖代码的目的。让代码自己说话（例如，通过代码中的名称本身进行描述），而不是添加多余的注释。通常情况下，注释最好是解释为什么要做某事，而不是解释代码在做什么。

Google 的代码库基本上是统一和一致的。通常情况下，那些比较突兀的代码（例如，应用一个不熟悉的模式）是基于充分的理由，通常是为了性能。保持这种特性很重要，可以让读者在阅读一段新的代码时清楚地知道他们应该把注意力放在哪里。

标准库中包含了许多这一原则发挥作用的例子。例如：

- 在 `package sort` 中的维护者注释
- 好的同一软件包中可运行的例子，这对用户（他们会查看 godoc）和维护者（他们作为测试的一部分运行）都有利
- `strings.Cut` 只有四行代码，但它们提高了 callsites 的清晰性和正确性

简约

你的 Go 代码对于使用、阅读和维护它的人来说应该是简单的。

Go 代码应该以最简单的方式编写，在行为和性能方面都能实现其目标。在 Google Go 代码库中，简单的代码：

- 从头至尾都易于阅读
- 不预设你已经知道它在做什么
- 不预设你能记住前面所有的代码
- 不含非必要的抽象层次
- 不含过于通用的命名
- 让读者清楚地了解到传值与决定的传播情况
- 有注释，解释为什么，而不是代码正在做什么，以避免未来的歧义
- 有独立的文档
- 包含有效的错误与失败用例测试
- 往往不是看起来“聪明”的代码

在代码的简单性和 API 使用的简单性之间可能会需要权衡。例如，让代码更复杂可能是值得的，这样 API 的终端用户可以更容易地正确调用 API。相反，把一些额外的工作留给 API 的终端用户也是值得的，这样代码就会保持简单和容易理解。

当代码需要复杂性时，应该有意地增加复杂性。如果需要额外的性能，或者一个特定的库或服务有多个不同的客户，这通常是必要的。复杂性可能是合理的，但它应该有相应的文档，以便客户和未来的维护者能够理解和驾驭这种复杂性。这应该用测试和例子来补充，以证明其正确的用法，特别是如果同时有一个“简单”和“复杂”的方法来使用代码。

这一原则并不意味着复杂的代码不能或不应该用 Go 编写，也不意味着 Go 代码不允许复杂。我们努力使代码库避免不必要的复杂性，因此当复杂性出现时，它表明有关的代码需要仔细理解和维护。理想情况下，应该有相应的注释来解释其中的道理，并指出应该注意的地方。在优化代码以提高性能时，经常会出现这种情况；这样做往往需要更复杂的方法，比如预先分配一个缓冲区并在整个 goroutine 生命周期内重复使用它。当维护者看到这种情况时，应该是一个线索，说明相关的代码是基于性能的关键考虑，这应该影响到未来修改时的谨慎。另一方面，如果不必要地使用，这种复杂性会给那些需要在未来阅读或修改代码的人带来负担。

如果代码非常复杂，但其目的应该是简单的，这往往是一个我们可以重新审视代码实现的信号，看看是否有更简单的方法来完成同样的事情。

最小化机制

如果有几种方法来表达同一个想法，最好选择使用最标准工具的方法。复杂的机制经常存在，但不应该无缘无故地使用。根据需要增加代码的复杂性是很容易的，而在发现没有必要的情况下删除现有的复杂性则要难得多。

1. 当足以满足你的使用情况时，争取使用一个核心语言结构（例如通道、切片、地图、循环或结构）
2. 如果没有，就在标准库中寻找一个工具（如 HTTP 客户端或模板引擎）
3. 最后，在引入新的依赖或创建自己的依赖之前，考虑 Google 代码库中是否有一个能够满足的核心库

例如，考虑生产代码包含一个绑定在变量上的标志，它的默认值必须在测试中被覆盖。除非打算测试程序的命令行界面本身（例如，用 `os/exec`），否则直接覆盖绑定的值比使用 `flag.Set` 更简单，因此更可取。

同样，如果一段代码需要检查集合成员的资格，一个布尔值的映射（例如，`map[string]bool`）通常就足够了。只有在需要更复杂的操作，不能使用 map 或过于复杂时，才应使用提供类似集合类型和功能的库。

简洁

简洁的 Go 代码具有很高的信噪比。它很容易分辨出相关的细节，而命名和结构则引导读者了解这些细节。

而有很多东西会常常会阻碍这些最突出的细节：

- 重复代码
- 外来的语法
- 含义不明的名称
- 不必要的抽象
- 空白

重复代码尤其容易掩盖每个相似代码之间的差异，需要读者直观地比较相似的代码行来发现变化。表驱动测试是一个很好的例子，这种机制可以简明地从每个重复的重要细节中找出共同的代码，但是选择哪些部分囊括在表中，会对表格的易懂程度产生影响。

在考虑多种结构代码的方式时，值得考虑哪种方式能使重要的细节最显著。

理解和使用常见的代码结构和规范对于保持高信噪比也很重要。例如，下面的代码块在错误处理中非常常见，读者可以很快理解这个代码块的目的。

```
// Good:
if err := doSomething(); err != nil {
    // ...
}
```

如果代码看起来非常相似但却有细微的不同，读者可能不会注意到这种变化。在这样的情况下，值得故意“提高”错误检查的信号，增加一个注释以引起关注。

```
// Good:
if err := doSomething(); err == nil { // if NO error
    // ...
}
```

可维护性

代码被编辑的次数比它写它的次数多得多。可读的代码不仅对试图了解其工作原理的读者有意义，而且对需要改写它的程序员也有意义，清晰性很关键。

可维护的代码：

- 容易让未来的程序员正确进行修改
- 拥有结构化的 API，使其能够优雅地增加
- 清楚代码预设条件，并选择映射到问题结构而不是代码结构的抽象
- 避免不必要的耦合，不包括不使用的功能
- 有一个全面的测试套件，以确保预期行为可控、重要逻辑正确，并且测试在失败的情况下提供清晰、可操作的诊断

当使用像接口和类型这样的抽象时，根据定义，它们会从使用的上下文中移除信息，因此必须确保它们提供足够的好处。当使用具体类型时，编辑器和 IDE 可以直接连接到方法定义并显示相应的文档，但在其他情况下只能参考接口定义。接口是一个强大的工具，但也是有代价的，因为维护者可能需要了解底层实现的具体细节才能正确使用接口，这必须在接口文档中或在调用现场进行解释。

可维护的代码还可以避免在容易忽视的地方隐藏重要的细节。例如，在下面的每一行代码中，是否有 `:` 字符对于理解这一行至关重要。

```
// Bad:
// 使用 = 而不是 := 可以完全改变这一行的含义
if user, err = db.UserByID(userID); err != nil {
    // ...
}
```

```
// Bad:
// 这行中间的 !  很容易错过
leap := (year%4 == 0) && (!(year%100 == 0) || (year%400 == 0))
```

这两种写法不能说错误，但都可以写得更明确，或者可以有一个附带的评论，提醒注意重要的行为。

```go
// Good:
u, err := db.UserByID(userID)
if err != nil {
    return fmt.Errorf("invalid origin user: %s", err)
}
user = u
```

```go
// Good:
// 公历闰年不仅仅是 year%4 == 0
// 查看 https://en.wikipedia.org/wiki/Leap_year#Algorithm.
var (
    leap4   = year%4 == 0
    leap100 = year%100 == 0
    leap400 = year%400 == 0
)
leap := leap4 && (!leap100 || leap400)
```

同样地，一个隐藏了关键逻辑或重要边界情况的辅助函数，可能会使未来的变化很容易地被误解。

易联想的名字是可维护代码的另一个特点。一个包的用户或一段代码的维护者应该能够联想到一个变量、方法或函数在特定情况下的名称。相同概念的函数参数和接收器名称通常应该共享相同的名称，这既是为了保持文档的可理解性，也是为了方便以最小的开销重构代码。

可维护的代码尽量减少其依赖性（包括隐性和显性）。对更少包的依赖意味着更少的代码行可以影响其行为。避免对内部或未记录的行为的依赖，使得代码在未来这些行为发生变化时，不太容易造成维护负担。

在考虑如何构造或编写代码时，值得花时间去思考代码可能随着时间的推移而演变的方式。如果一个给定的方法更有利于未来更容易和更安全的变化，这往往是一个很好的权衡，即使它意味着一个稍微复杂的设计。

一致

一致性的代码是指在更广泛的代码库中，在一个团队或包的范围内，甚至在一个文件中，看起来、感觉和行为都是类似的代码。

一致性的问题并不凌驾于上述的任何原则之上，但如果必须有所取舍，那往往有利于一致性的实现。

一个包内的一致性通常是最直接重要的一致性水平。如果同一个问题在一个包里有多种处理方式，或者同一个概念在一个文件里有很多名字，那就会非常不优雅。然而，即使这样，也不应该凌驾于文件的风格原则或全局一致性之上。

核心准则

这些准则收集了所有 Go 代码都应遵循的 Go 风格的最重要方面。我们希望这些原则在你被保障可读性的时候就已经学会并遵循了。这些不会经常改变，新增加内容也有较高准入门槛。

下面的准则是对 Effective Go 中建议的扩展，它为整个社区的 Go 代码提供了一个共同的基准线。

格式化

所有 Go 源文件必须符合 gofmt 工具所输出的格式。这个格式是由 Google 代码库中的预提交检查强制执行的。生成的代码通常也应该被格式化（例如，通过使用 `format.Source`），因为它也可以在代码搜索中浏览。

大小写混合

Go 源代码在编写包含多个字的名称时使用 `MixedCaps` 或 `mixedCaps`（驼峰大写）而不是下划线（蛇形大写）。

即使在其他语言中打破惯例，这也适用。例如，一个常量如果被导出，则为 `MaxLength`（而不是 `MAX_LENGTH`），如果未被导出，则为 `maxLength`（而不是 `max_length`）。

基于初始化大小写的考量，局部变量被认为是 不可导出的。

行长度

Go 源代码没有固定的行长度。如果觉得某一行太长，就应该对其进行重构而不是破坏。如果它已经很短了，那么就应该允许它继续增加。

不要在以下情况进行分行：

- 在缩进变化之前 (例如，函数声明、条件)
- 要使一个长的字符串（例如，一个 URL）适合于多个较短的行

命名

命名是艺术而不是科学。在 Go 中，名字往往比许多其他语言的名字短一些，但同样的一般准则也适用，名称应：

- 使用时不感到重复
- 将上下文考虑在内
- 不重复已经明确的概念

你可以在决定中找到关于命名的更具体的指导。

本地化一致性

如果风格指南对某一特定的风格点没有说明，作者可以自由选择他们喜欢的风格，除非相近的代码（通常在同一个文件或包内，但有时在一个团队或项目目录内）对这个问题采取了一致的立场。

有效的本地风格化考虑例子：

- 使用 `%s` or `%v` 来打印错误
- 使用缓冲通道来代替 mutexes

无效的本地化风格化考虑例子：

- 代码行长度的限制
- 使用基于断言的测试库

如果本地化风格与风格指南不一致，但对可读性的影响仅限于一个文件，它通常会在代码审查中浮出水面，而一致的修复将超出有关 CL 的范围。在这一点上，提交一个 bug 来跟踪修复是合适的。

如果一个改变会使现有的风格偏差变大，在更多的 API 表面暴露出来，扩大存在偏差的文件数量，或者引入一个实际的错误，那么局部一致性就不再是违反新代码风格指南的有效理由。在这些情况下，作者应该在同一 CL 中清理现有的代码库，在当前 CL 之前进行重构，或者找到一个至少不会使本地化问题恶化的替代方案。

Go 风格决策

原文：https://google.github.io/styleguide/go

概述 | 风格指南 | 风格决策 | 最佳实践

注意： 本文是 Google Go 风格 系列文档的一部分。本文档是 **规范性 (normative) 但不是强制规范 (canonical)**，并且从属于 Google 风格指南。请参阅概述获取更多详细信息。

关于

本文包含旨在统一和为 Go 可读性导师给出的建议提供标准指导、解释和示例的风格决策。

本文档**并不详尽**，且会随着时间的推移而增加。如果风格指南 与此处给出的建议相矛盾，**风格指南优先**，并且本文档应相应更新。

参见 关于 获取 Go 风格的全套文档。

以下部分已从样式决策移至指南的一部分：

- **混合大写字母 MixedCaps**: 参见 https://gocn.github.io/styleguide/docs/02-guide/#大小写混合
- **格式化 Formatting**: 参见 https://gocn.github.io/styleguide/docs/02-guide/#格式化
- **行长度 Line Length**: 参见 https://gocn.github.io/styleguide/docs/02-guide/#行长度

命名 Naming

有关命名的总体指导，请参阅核心风格指南 中的命名部分，以下部分对命名中的特定区域提供进一步的说明。

下划线 Underscores

Go 中的命名通常不应包含下划线。这个原则有三个例外：

1. 仅由生成代码导入的包名称可能包含下划线。有关如何选择多词包名称的更多详细信息，请参阅包名称。
2. `*_test.go` 文件中的测试、基准和示例函数名称可能包含下划线。
3. 与操作系统或 cgo 互操作的低级库可能会重用标识符，如 `syscall` 中所做的那样。在大多数代码库中，这预计是非常罕见的。

包名称 Package names

Go 包名称应该简短并且只包含小写字母。由多个单词组成的包名称应全部小写。例如，包 `tabwriter` 不应该命名为 `tabWriter`、`TabWriter` 或 `tab_writer`。

避免选择可能被常用局部变量遮蔽覆盖 的包名称。例如，`usercount` 是比 `count` 更好的包名，因为 `count` 是常用变量名。

Go 包名称不应该有下划线。如果你需要导入名称中确实有一个包（通常来自生成的或第三方代码），则必须在导入时将其重命名为适合在 Go 代码中使用的名称。

一个例外是仅由生成的代码导入的包名称可能包含下划线。具体例子包括：

- 对外部测试包使用 _test 后缀，例如集成测试
- 使用 `_test` 后缀作为 包级文档示例

避免使用无意义的包名称，例如 `util`、`utility`、`common`、`helper` 等。查看更多关于所谓的“实用程序包”。

当导入的包被重命名时（例如 `import foob "path/to/foo_go_proto"`），包的本地名称必须符合上述规则，因为本地名称决定了包中的符号在文件中的引用方式.如果给定的导入在多个文件中重命名，特别是在相同或附近的包中，则应尽可能使用相同的本地名称以保持一致性。

另请参阅：https://go.dev/blog/package-names

接收者命名 Receiver names

接收者 变量名必须满足：

- 短（通常是一两个字母的长度）
- 类型本身的缩写
- 始终如一地应用于该类型的每个接收者

长名称	更好命名
`func (tray Tray)`	`func (t Tray)`
`func (info *ResearchInfo)`	`func (ri *ResearchInfo)`
`func (this *ReportWriter)`	`func (w *ReportWriter)`
`func (self *Scanner)`	`func (s *Scanner)`

常量命名 Constant names

常量名称必须像 Go 中的所有其他名称一样使用 混合大写字母 MixedCaps。（导出 常量以大写字母开头，而未导出的常量以小写字母开头。）即使打破了其他语言的约定，这也是适用的。常量名称不应是其值的派生词，而应该解释值所表示的含义。

```
// Good:
const MaxPacketSize = 512

const (
    ExecuteBit = 1 << iota
    WriteBit
    ReadBit
)
```

不要使用非混合大写常量名称或带有 `K` 前缀的常量。

```
// Bad:
const MAX_PACKET_SIZE = 512
const kMaxBufferSize = 1024
const KMaxUsersPergroup = 500
```

根据它们的角色而不是它们的值来命名常量。如果一个常量除了它的值之外没有其他作用，那么就没有必要将它定义为一个常量。

```
// Bad:
const Twelve = 12

const (
    UserNameColumn = "username"
    GroupColumn    = "group"
)
```

缩写词 Initialisms

名称中的首字母缩略词或单独的首字母缩略词（例如，“URL”和“NATO”）应该具有相同的大小写。`URL` 应显示为 `URL` 或 `url`（如 `urlPony` 或 `URLPony`），绝不能显示为 `Url`。这也适用于 `ID` 是“identifier”的缩写； 写 `appID` 而不是 `appId`。

- 在具有多个首字母缩写的名称中（例如 `XMLAPI` 因为它包含 `XML` 和 `API`），给定首字母缩写中的每个字母都应该具有相同的大小写，但名称中的每个首字母缩写不需要具有相同的大小写。
- 在带有包含小写字母的首字母缩写的名称中（例如 `DDoS`、`iOS`、`gRPC`），首字母缩写应该像在标准中一样出现，除非你需要为了满足 导出 而更改第一个字母。在这些情况下，整个缩写词应该是相同的情况（例如 `ddos`、`IOS`、`GRPC`）。

缩写词	范围	正确	错误
XML API	Exported	`XMLAPI`	`XmlApi`, `XMLApi`, `XmlAPI`, `XMLapi`
XML API	Unexported	`xmlAPI`	`xmlapi`, `xmlApi`
iOS	Exported	`IOS`	`Ios`, `IoS`
iOS	Unexported	`iOS`	`ios`
gRPC	Exported	`GRPC`	`Grpc`
gRPC	Unexported	`gRPC`	`grpc`
DDoS	Exported	`DDoS`	`DDOS`, `Ddos`
DDoS	Unexported	`ddos`	`dDoS`, `dDOS`

Get 方法 Getters

函数和方法名称不应使用 `Get` 或 `get` 前缀，除非底层概念使用单词“get”（例如 HTTP GET）。此时，更应该直接以名词开头的名称，例如使用 `Counts` 而不是 `GetCounts`。

如果该函数涉及执行复杂的计算或执行远程调用，则可以使用 `Compute` 或 `Fetch` 等不同的词代替 `Get`，以使读者清楚函数调用可能需要时间，并有可能会阻塞或失败。

变量名 Variable names

一般的经验法则是，名称的长度应与其范围的大小成正比，并与其在该范围内使用的次数成反比。在文件范围内创建的变量可能需要多个单词，而单个内部块作用域内的变量可能是单个单词甚至只是一两个字符，以保持代码清晰并避免无关信息。

这是一条粗略的基础原则。这些数字准则不是严格的规则。要根据上下文、清晰 和[简洁](https://gocn.github.io/styleguide/docs/02-guide/#简洁）来进行判断。

- 小范围是执行一两个小操作的范围，比如 1-7 行。
- 中等范围是一些小的或一个大的操作，比如 8-15 行。
- 大范围是一个或几个大操作，比如 15-25 行。
- 非常大的范围是指超过一页（例如，超过 25 行）的任何内容。

在小范围内可能非常清楚的名称（例如，`c` 表示计数器）在较大范围内可能不够用，并且需要澄清以提示进一步了解其在代码中的用途。一个作用域中有很多变量，或者表示相似值或概念的变量，那就可能需要比作用域建议的采用更长的变量名称。

概念的特殊性也有助于保持变量名称的简洁。例如，假设只有一个数据库在使用，像 `db` 这样的短变量名通常可能保留给非常小的范围，即使范围非常大，也可能保持完全清晰。在这种情况下，根据范围的大小，单个词 `database` 可能是可接受的，但不是必需的，因为 `db` 是该词的一种非常常见的缩写，几乎没有其他解释。

局部变量的名称应该反映它包含的内容以及它在当前上下文中的使用方式，而不是值的来源。例如，通常情况下最佳局部变量名称与结构或协议缓冲区字段名称不同。

一般来说：

- 像 `count` 或 `options` 这样的单字名称是一个很好的起点。
- 可以添加其他词来消除相似名称的歧义，例如 `userCount` 和 `projectCount`。
- 不要简单地省略字母来节省打字时间。例如，`Sandbox` 优于 `Sbx`，特别是对于导出的名称。
- 大多数变量名可省略 类型和类似类型的词
  - 对于数字，`userCount` 是比 `numUsers` 或 `usersInt` 更好的名称。
  - 对于切片，`users` 是一个比 `userSlice` 更好的名字。
  - 如果范围内有两个版本的值，则包含类似类型的限定符是可以接受的，例如，你可能将输入存储在 `ageString` 中，并使用 `age` 作为解析值。
- 省略上下文 中清楚的单词。例如，在 UserCount 方法的实现中，名为 userCount 的局部变量可能是多余的；`count`、`users` 甚至 `c` 都具有可读性。

单字母变量名 Single-letter variable names

单字母变量名是可以减少重复 的有用工具，但也可能使代码变得不透明。将它们的使用限制在完整单词很明显以及它会重复出现以代替单字母变量的情况。

一般来说：

- 对于方法接收者变量，最好使用一个字母或两个字母的名称。

- 对常见类型使用熟悉的变量名通常很有帮助：
  - `r` 用于 `io.Reader` 或 `*http.Request`
  - `w` 用于 `io.Writer` 或 `http.ResponseWriter`
- 单字母标识符作为整数循环变量是可接受的，特别是对于索引（例如，`i`）和坐标（例如，`x` 和 `y`）。
- 当范围很短时，循环标识符使用缩写是可接受的，例如 `for _, n := range nodes { ... }`。

重复 Repetition

一段 Go 源代码应该避免不必要的重复。一个常见的情形是重复名称，其中通常包含不必要的单词或重复其上下文或类型。如果相同或相似的代码段在很近的地方多次出现，代码本身也可能是不必要的重复。

重复命名可以有多种形式，包括：

包名 vs 可导出符号名 Package vs. exported symbol name

当命名导出的符号时，包的名称始终在包外可见，因此应减少或消除两者之间的冗余信息。如果一个包如果需要仅导出一种类型并且以包本身命名，则构造函数的规范名称是 `New`（如果需要的话）。

> **实例:** 重复的名称 -> 更好的名称
>
> - `widget.NewWidget` -> `widget.New`
> - `widget.NewWidgetWithName` -> `widget.NewWithName`
> - `db.LoadFromDatabase` -> `db.Load`
> - `goatteleportutil.CountGoatsTeleported` -> `gtutil.CountGoatsTeleported` or `goatteleport.Count`
> - `myteampb.MyTeamMethodRequest` -> `mtpb.MyTeamMethodRequest` or `myteampb.MethodRequest`

变量名 vs 类型 Variable name vs. type

编译器总是知道变量的类型，并且在大多数情况下，阅读者也可以通过变量的使用方式清楚地知道变量是什么类型。只有当一个变量的值在同一范围内出现两次时，才有需要明确变量的类型。

重复的名称	更好的名称
`var numUsers int`	`var users int`
`var nameString string`	`var name string`
`var primaryProject *Project`	`var primary *Project`

如果该值以多种形式出现，这可以通过额外的词（如 `raw` 和 `parsed`）或底层表示来澄清：

```
// Good:
limitStr := r.FormValue("limit")
limit, err := strconv.Atoi(limitStr)
// Good:
limitRaw := r.FormValue("limit")
limit, err := strconv.Atoi(limitRaw)
```

外部上下文 vs 本地名称 External context vs. local names

包含来自周围上下文信息的名称通常会产生额外的噪音，而没有任何好处。包名、方法名、类型名、函数名、导入路径，包含来自其上下文信息的名称。 Names that include information from their surrounding context often create extra noise without benefit. The package name, method name, type name, function name, import path, and even filename can all provide context that automatically qualifies all names within.

```
// Bad:
// In package "ads/targeting/revenue/reporting"
type AdsTargetingRevenueReport struct{}

func (p *Project) ProjectName() string
// Good:
// In package "ads/targeting/revenue/reporting"
type Report struct{}

func (p *Project) Name() string
// Bad:
// In package "sqldb"
type DBConnection struct{}
// Good:
// In package "sqldb"
type Connection struct{}
// Bad:
// In package "ads/targeting"
func Process(in *pb.FooProto) *Report {
    adsTargetingID := in.GetAdsTargetingID()
}
// Good:
// In package "ads/targeting"
func Process(in *pb.FooProto) *Report {
    id := in.GetAdsTargetingID()
}
```

重复通常应该在符号使用者的上下文中进行评估，而不是孤立地进行评估。例如，下面的代码有很多名称，在某些情况下可能没问题，但在上下文中是多余的：

```
// Bad:
func (db *DB) UserCount() (userCount int, err error) {
    var userCountInt64 int64
    if dbLoadError := db.LoadFromDatabase("count(distinct users)", &userCountInt64);
dbLoadError != nil {
        return 0, fmt.Errorf("failed to load user count: %s", dbLoadError)
    }
    userCount = int(userCountInt64)
    return userCount, nil
}
```

相反，在上下文和使用上信息是清楚的情况下，常常可以忽略：

```
// Good:
func (db *DB) UserCount() (int, error) {
    var count int64
    if err := db.Load("count(distinct users)", &count); err != nil {
        return 0, fmt.Errorf("failed to load user count: %s", err)
    }
    return int(count), nil
}
```

评论 Commentary

关于评论的约定（包括评论什么、使用什么风格、如何提供可运行的示例等）旨在支持阅读公共 API 文档的体验。有关详细信息，请参阅 Effective Go。

最佳实践文档关于 文档约定 的部分进一步讨论了这一点。

最佳实践：在开发和代码审查期间使用文档预览 查看文档和可运行示例是否有用并以你期望的方式呈现。

提示： Godoc 使用很少的特殊格式； 列表和代码片段通常应该缩进以避免换行。除缩进外，通常应避免装饰。

注释行长度 Comment line length

确保注释在即使在较窄的屏幕上的可读性。

当评论变得太长时，建议将其包装成多个单行评论。在可能的情况下，争取在 80 列宽的终端上阅读良好的注释，但这并不是硬性限制； Go 中的注释没有固定的行长度限制。例如，标准库经常选择根据标点符号来打断注释，这有时会使个别行更接近 60-70 个字符标记。

有很多现有代码的注释长度超过 80 个字符。本指南不应作为更改此类代码作为可读性审查的一部分的理由（请参阅一致），但鼓励团队作为其他重构的一部分，有机会时更新注释以遵循此指南。本指南的主要目标是确保所有 Go 可读性导师在提出建议时以及是否提出相同的建议。

有关评论的更多信息，请参阅此 来自 The Go Blog 的帖子。

```
# Good:
// This is a comment paragraph.
// The length of individual lines doesn't matter in Godoc;
// but the choice of wrapping makes it easy to read on narrow screens.
//
// Don't worry too much about the long URL:
// https://supercalifragilisticexpialidocious.example.com:8080/Animalia/Chordata/
Mammalia/Rodentia/Geomyoidea/Geomyidae/
//
// Similarly, if you have other information that is made awkward
// by too many line breaks, use your judgment and include a long line
// if it helps rather than hinders.
```

避免注释在小屏幕上重复换行，这是一种糟糕的阅读体验。

```
# Bad:
// This is a comment paragraph. The length of individual lines doesn't matter in
Godoc;
// but the choice of wrapping causes jagged lines on narrow screens or in
Critique,
// which can be annoying, especially when in a comment block that will wrap
repeatedly.
//
// Don't worry too much about the long URL:
// https://supercalifragilisticexpialidocious.example.com:8080/Animalia/Chordata/
Mammalia/Rodentia/Geomyoidea/Geomyidae/
```

文档注释 Doc comments

所有顶级导出名称都必须有文档注释，具有不明显行为或含义的未导出类型或函数声明也应如此。这些注释应该是完整句子，以所描述对象的名称开头。冠词（“a”、“an”、“the”）可以放在名字前面，使其读起来更自然。

```
// Good:
// A Request represents a request to run a command.
type Request struct { ...

// Encode writes the JSON encoding of req to w.
func Encode(w io.Writer, req *Request) { ...
```

文档注释出现在 Godoc 中，并通过 IDE 显示，因此应该为使用该包的任何人编写文档注释。

如果出现在结构中，文档注释适用于以下符号或字段组：

```
// Good:
// Options configure the group management service.
type Options struct {
    // General setup:
    Name  string
    Group *FooGroup

    // Dependencies:
    DB *sql.DB

    // Customization:
    LargeGroupThreshold int // optional; default: 10
    MinimumMembers      int // optional; default: 2
}
```

最佳实践：如果你对未导出的代码有文档注释，请遵循与导出代码相同的习惯（即，以未导出的名称开始注释）。这使得以后导出它变得容易，只需在注释和代码中用新导出的名称替换未导出的名称即可。

注释语句 Comment sentences

完整的注释应该像标准英语句子一样包含大写和标点符号。（作为一个例外，如果在其他方面很清楚，可以以非大写的标识符名称开始一个句子。这种情况最好只在段落的开头进行。）

作为句子片段的注释对标点符号或大小写没有此类要求。

文档注释 应始终是完整的句子，因此应始终大写和标点符号。简单的行尾注释（特别是对于结构字段）可以为假设字段名称是主语的简单短语。

```
// Good:
// A Server handles serving quotes from the collected works of Shakespeare.
type Server struct {
    // BaseDir points to the base directory under which Shakespeare's works are stored.
    //
    // The directory structure is expected to be the following:
    //   {BaseDir}/manifest.json
    //   {BaseDir}/{name}/{name}-part{number}.txt
    BaseDir string

    WelcomeMessage  string // displayed when user logs in
    ProtocolVersion string // checked against incoming requests
    PageLength      int    // lines per page when printing (optional; default: 20)
}
```

示例 Examples

包应该清楚地记录它们的预期用途。尝试提供一个可运行的例子； 示例出现在 Godoc 中。可运行示例属于测试文件，而不是生产源文件。请参阅此示例（Godoc，[source](https://cs.opensource.google/go/go/+/HEAD:src/time /example_test.go））。

如果无法提供可运行的示例，可以在代码注释中提供示例代码。与注释中的其他代码和命令行片段一样，它应该遵循标准格式约定。

命名的结果参数 Named result parameters

当有命名参数时，请考虑函数签名在 Godoc 中的显示方式。函数本身的名称和结果参数的类型通常要足够清楚。

```
// Good:
func (n *Node) Parent1() *Node
func (n *Node) Parent2() (*Node, error)
```

如果一个函数返回两个或多个相同类型的参数，添加名称会很有用。

```
// Good:
func (n *Node) Children() (left, right *Node, err error)
```

如果调用者必须对特定的结果参数采取行动，命名它们可以帮助暗示行动是什么：

```
// Good:
// WithTimeout returns a context that will be canceled no later than d duration
// from now.
//
// The caller must arrange for the returned cancel function to be called when
// the context is no longer needed to prevent a resource leak.
func WithTimeout(parent Context, d time.Duration) (ctx Context, cancel func())
```

在上面的代码中，取消是调用者必须执行的特定操作。但是，如果将结果参数单独写为 `(Context, func())`，“取消函数”的含义就不清楚了。

当名称产生 不必要的重复 时，不要使用命名结果参数。

```
// Bad:
func (n *Node) Parent1() (node *Node)
func (n *Node) Parent2() (node *Node, err error)
```

不要为了避免在函数内声明变量而使用命名结果参数。这种做法会导致不必要的冗长API，但收益只是微小的简洁性。

裸返回 仅在小函数中是可接受的。一旦它是一个中等大小的函数，就需要明确你的返回值。同样，不要仅仅因为可以裸返回就使用命名结果参数。清晰 总是比在你的函数中节省几行更重要。

如果必须在延迟闭包中更改结果参数的值，则命名结果参数始终是可以接受的。

> **提示：** 类型通常比函数签名中的名称更清晰。GoTip #38：作为命名类型的函数 演示了这一点。
>
> 在上面的 `WithTimeout` 中，代码使用了一个 `CancelFunc` 而不是结果参数列表中的原始 `func()`，并且几乎不需要做任何记录工作。

包注释

包注释必须出现在包内语句的上方，注释和包名称之间没有空行。例子：

```
// Good:
// Package math provides basic constants and mathematical functions.
//
// This package does not guarantee bit-identical results across architectures.
package math
```

每个包必须有一个包注释。如果一个包由多个文件组成，那么其中一个文件应该有包注释。

`main` 包的注释形式略有不同，其中 BUILD 文件中的 `go_binary` 规则的名称代替了包名。

```
// Good:
// The seed_generator command is a utility that generates a Finch seed file
// from a set of JSON study configs.
package main
```

只要二进制文件的名称与 BUILD 文件中所写的完全一致，其他风格的注释也是可以了。当二进制名称是第一个单词时，即使它与命令行调用的拼写不严格匹配，也需要将其大写。

```
// Good:
// Binary seed_generator ...
// Command seed_generator ...
// Program seed_generator ...
// The seed_generator command ...
// The seed_generator program ...
// Seed_generator ...
```

提示:

- 命令行调用示例和 API 用法可以是有用的文档。对于 Godoc 格式，缩进包含代码的注释行。
- 如果没有明显的 main 文件或者包注释特别长，可以将文档注释放在名为 doc.go 的文件中，只有注释和包子句。
- 可以使用多行注释代替多个单行注释。如果文档包含可能对从源文件复制和粘贴有用的部分，如示例命令行（用于二进制文件）和模板示例，这将非常有用。

  ```
  // Good:
  /*
  The seed_generator command is a utility that generates a Finch seed file
  from a set of JSON study configs.

      seed_generator *.json | base64 > finch-seed.base64
  */
  package template
  ```

- 供维护者使用且适用于整个文件的注释通常放在导入声明之后。这些不会出现在 Godoc 中，也不受上述包注释规则的约束。

导入

导入重命名

只有在为了避免与其他导入的名称冲突时，才使用重命名导入。（由此推论，好的包名称 不需要重命名。）如果发生名称冲突，最好重命名 最本地或特定于项目的导入。包的本地别名必须遵循包命名指南，包括禁止使用下划线和大写字母。

生成的 protocol buffer 包必须重命名以从其名称中删除下划线，并且它们的别名必须具有 `pb` 后缀。有关详细信息，请参阅 proto 和 stub 最佳实践。

```
// Good:
import (
    fspb "path/to/package/foo_service_go_proto"
)
```

导入的包名称没有有用的识别信息时（例如 `package v1`），应该重命名以包括以前的路径组件。重命名必须与导入相同包的其他本地文件一致，并且可以包括版本号。

注意： 最好重命名包以符合 好的包命名规则，但在 vendor 目录下的包通常是不可行的。

```
// Good:
import (
    core "github.com/kubernetes/api/core/v1"
    meta "github.com/kubernetes/apimachinery/pkg/apis/meta/v1beta1"
)
```

如果你需要导入一个名称与你要使用的公共局部变量名称（例如 `url`、`ssh`）冲突的包，并且你希望重命名该包，首选方法是使用 `pkg` 后缀（例如 `urlpkg`）。请注意，可以使用局部变量隐藏包； 仅当此类变量在范围内时仍需要使用此包时，才需要重命名。

导入分组

导入应分为两组：

- 标准库包
- 其他（项目和 vendor）包

```
// Good:
package main

import (
    "fmt"
    "hash/adler32"
    "os"

    "github.com/dsnet/compress/flate"
    "golang.org/x/text/encoding"
    "google.golang.org/protobuf/proto"
    foopb "myproj/foo/proto/proto"
    _ "myproj/rpc/protocols/dial"
    _ "myproj/security/auth/authhooks"
)
```

将导入项分成多个组是可以接受的，例如，如果你想要一个单独的组来重命名、导入仅为了特殊效果 或另一个特殊的导入组。

```
// Good:
package main

import (
    "fmt"
    "hash/adler32"
    "os"

    "github.com/dsnet/compress/flate"
    "golang.org/x/text/encoding"
    "google.golang.org/protobuf/proto"

    foopb "myproj/foo/proto/proto"

    _ "myproj/rpc/protocols/dial"
    _ "myproj/security/auth/authhooks"
)
```

注意： goimports 不支持维护可选组 - 超出标准库和 Google 导入之间强制分离所需的拆分。为了保持符合状态，额外的导入子组需要作者和审阅人的注意。

Google 程序有时也是 AppEngine 应用程序，应该有一个单独的组用于 AppEngine 导入。

Gofmt 负责按导入路径对每个组进行排序。但是，它不会自动将导入分成组。流行的 goimports 工具结合了 Gofmt 和导入管理，根据上述规则将导入进行分组。通过 goimports 来管理导入顺序是可行的，但随着文件的修改，其导入列表必须保持内部一致。

导入"空" (`import _`)

使用语法 `import _ "package"` 导入的包，称为副作用导入，只能在主包或需要它们的测试中导入。

此类软件包的一些示例包括：

- time/tzdata
- image/jpeg 在图像处理中的代码

避免在工具包中导入空白，即使工具包间接依赖于它们。将副作用导入限制到主包有助于控制依赖性，并使得编写依赖于不同导入的测试成为可能，而不会发生冲突或浪费构建成本。

以下是此规则的唯一例外情况：

- 你可以使用空白导入来绕过 nogo 静态检查器 中对不允许导入的检查。
- 你可以在使用 `//go:embed` 编译器指令的源文件中使用 embed 包的空白导入。

提示：如果生产环境中你创建的工具包间接依赖于副作用导入，请记录这里的预期用途。

导入 “.” (`import .`)

`import .` 形式是一种语言特性，它允许将从另一个包导出的标识符无条件地带到当前包中。有关更多信息，请参阅语言规范。

不要在 Google 代码库中使用此功能； 这使得更难判断功能来自何处。

```
// Bad:
package foo_test

import (
    "bar/testutil" // also imports "foo"
    . "foo"
)

var myThing = Bar() // Bar defined in package foo; no qualification needed.
// Good:
package foo_test

import (
    "bar/testutil" // also imports "foo"
    "foo"
)

var myThing = foo.Bar()
```

错误

返回错误

使用 `error` 表示函数可能会失败。按照惯例，`error` 是最后一个结果参数。

```
// Good:
func Good() error { /* ... */ }
```

返回 `nil` 错误是表示操作成功的惯用方式，否则表示可能会失败。如果函数返回错误，除非另有明确说明，否则调用者必须将所有非错误返回值视为未确定。通常来说，非错误返回值是它们的零值，但也不能直接这么假设。

```
// Good:
func GoodLookup() (*Result, error) {
    // ...
    if err != nil {
        return nil, err
    }
    return res, nil
}
```

返回错误的导出函数应使用 `error` 类型返回它们。具体的错误类型容易受到细微错误的影响：一个 `nil` 指针可以包装到接口中，从而就变成非 nil 值（参见 关于该主题的 Go FAQ 条目）。

```
// Bad:
func Bad() *os.PathError { /*...*/ }
```

提示：采用 `context.Context` 参数的函数通常应返回 `error`，以便调用者可以确定上下文是否在函数运行时被取消。

错误字符串

错误字符串不应大写（除非以导出名称、专有名词或首字母缩写词开头）并且不应以标点符号结尾。这是因为错误字符串通常在打印给用户之前出现在其他上下文中。

```
// Bad:
err := fmt.Errorf("Something bad happened.")
// Good:
err := fmt.Errorf("something bad happened")
```

另一方面，完整显示消息（日志记录、测试失败、API 响应或其他 UI）的样式视情况而定，但通常应大写首字母。

```
// Good:
log.Infof("Operation aborted: %v", err)
log.Errorf("Operation aborted: %v", err)
t.Errorf("Op(%q) failed unexpectedly; err=%v", args, err)
```

错误处理

遇到错误的代码应该慎重选择如何处理它。使用 _ 变量丢弃错误通常是不合适的。如果函数返回错误，请执行以下操作之一：

- 立即处理并解决错误
- 将错误返回给调用者
- 在特殊情况下，调用 `log.Fatal` 或（如绝对有必要）则调用 `panic`

注意： `log.Fatalf` 不是标准库日志。参见 [#logging]。

在极少数情况下适合忽略或丢弃错误（例如调用 `(*bytes.Buffer).Write` 被记录为永远不会失败），随附的注释应该解释为什么这是安全的。

```
// Good:
var b *bytes.Buffer

n, _ := b.Write(p) // never returns a non-nil error
```

关于错误处理的更多讨论和例子，请参见 Effective Go 和最佳实践。

In-band 错误

在 C 和类似语言中，函数通常会返回-1、null 或空字符串等值，以示错误或丢失结果。这就是所谓的 `In-band` 处理。

```
// Bad:
// Lookup returns the value for key or -1 if there is no mapping for key.
func Lookup(key string) int
```

未能检查 `In-band` 错误值会导致错误，并可能将 error 归于错误的功能。

```
// Bad:
// The following line returns an error that Parse failed for the input value,
// whereas the failure was that there is no mapping for missingKey.
return Parse(Lookup(missingKey))
```

Go对多重返回值的支持提供了一个更好的解决方案（见Effective Go关于多重返回的部分）。与其要求调用方检查 `In-band` 的错误值，函数更应该返回一个额外的值来表明返回值是否有效。这个返回值可以是一个错误，或者在不需要解释时是一个布尔值，并且应该是最终的返回值。

```
// Good:
// Lookup returns the value for key or ok=false if there is no mapping for key.
func Lookup(key string) (value string, ok bool)
```

这个 API 可以防止调用者错误地编写 `Parse(Lookup(key))`，从而导致编译时错误，因为 `Lookup(key)` 有两个返回值。

以这种方式返回错误，来构筑更强大和明确的错误处理。

```
// Good:
value, ok := Lookup(key)
if !ok {
    return fmt.Errorf("no value for %q", key)
}
return Parse(value)
```

一些标准库函数，如包 `strings` 中的函数，返回 `In-band` 错误值。这大大简化了字符串处理的代码，但代价是要求程序员更加勤奋。一般来说，Google 代码库中的 Go 代码应该为错误返回额外的值

缩进错误流程

在继续代码的其余部分之前处理错误。这提高了代码的可读性，使读者能够快速找到正常路径。这个逻辑同样适用于任何测试条件并以终端条件结束的代码块（例如，`return`、`panic`、`log.Fatal`）。

如果终止条件没有得到满足，运行的代码应该出现在 `if` 块之后，而不应该缩进到 `else` 子句中。

```
// Good:
if err != nil {
    // error handling
    return // or continue, etc.
}
// normal code
// Bad:
if err != nil {
    // error handling
} else {
    // normal code that looks abnormal due to indentation
}
```

> **提示：**如果你使用一个变量超过几行代码，通常不值得使用 `带有初始化的 if` 风格。在这种情况下，通常最好将声明移出，使用标准的 `if` 语句。

```
// Good:
x, err := f()
if err != nil {
  // error handling
  return
}
// lots of code that uses x
// across multiple lines
// Bad:
if x, err := f(); err != nil {
  // error handling
  return
} else {
  // lots of code that uses x
  // across multiple lines
}
```

更多细节见 Go Tip #1：视线和 TotT：通过减少嵌套降低代码的复杂性。

语言

字面格式化

Go 有一个非常强大的复合字面语法，用它可以在一个表达式中表达深度嵌套的复杂值。在可能的情况下，应该使用这种字面语法，而不是逐字段建值。字面意义的 `gofmt` 格式一般都很好，但有一些额外的规则可以使这些字面意义保持可读和可维护。

字段名称

对于在当前包之外定义的类型，结构体字面量通常应该指定**字段名**。

- 包括来自其他包的类型的字段名。

  ```
  // Good:
  good := otherpkg.Type{A: 42}
  ```

  结构中字段的位置和字段的完整集合（当字段名被省略时，这两者都是有必要搞清楚的）通常不被认为是结构的公共 API 的一部分；需要指定字段名以避免不必要的耦合。

  ```
  // Bad:
  // https://pkg.go.dev/encoding/csv#Reader
  r := csv.Reader{',', '#', 4, false, false, false, false}
  ```

  在小型、简单的结构中可以省略字段名，这些结构的组成和顺序都有文档证明是稳定的。

  ```
  // Good:
  okay := image.Point{42, 54}
  also := image.Point{X: 42, Y: 54}
  ```

- 对于包内类型，字段名是可选的。

```
// Good:
okay := Type{42}
also := internalType{4, 2}
```

如果能使代码更清晰，还是应该使用字段名，而且这样做是很常见的。例如，一个有大量字段的结构几乎都应该用字段名来初始化。

```
// Good:
okay := StructWithLotsOfFields{
  field1: 1,
  field2: "two",
  field3: 3.14,
  field4: true,
}
```

匹配的大括号

一对大括号的最后一半应该总是出现在一行中，其缩进量与开头的大括号相同。单行字词必然具有这个属性。当字面意义跨越多行时，保持这一属性可以使字面意义的括号匹配与函数和 `if` 语句等常见 Go 语法结构的括号匹配相同。

这方面最常见的错误是在多行结构字中把收尾括号与值放在同一行。在这种情况下，该行应以逗号结束，收尾括号应出现在下一行。

```
// Good:
good := []*Type{{Key: "value"}}
// Good:
good := []*Type{
    {Key: "multi"},
    {Key: "line"},
}
// Bad:
bad := []*Type{
    {Key: "multi"},
    {Key: "line"}}
// Bad:
bad := []*Type{
    {
        Key: "value"},
}
```

Cuddled 大括号

只有在以下两种情况下，才允许在大括号之间为切片和数组丢弃空格（又称 ""cuddling"）。

- 缩进匹配
- 内部值也是字面意义或原语构建者（即不是变量或其他表达式）

```
// Good:
good := []*Type{
    { // Not cuddled
        Field: "value",
    },
    {
        Field: "value",
    },
}
// Good:
good := []*Type{{ // Cuddled correctly
    Field: "value",
}, {
    Field: "value",
}}
// Good:
good := []*Type{
    first, // Can't be cuddled
    {Field: "second"},
}
// Good:
okay := []*pb.Type{pb.Type_builder{
    Field: "first", // Proto Builders may be cuddled to save vertical space
}.Build(), pb.Type_builder{
    Field: "second",
}.Build()}
// Bad:
bad := []*Type{
    first,
    {
        Field: "second",
    }}
```

重复的类型名称

重复的类型名称可以从 slice 和 map 字面上省略，这对减少杂乱是有帮助的。明确重复类型名称的一个合理场合，当在你的项目中处理一个不常见的复杂类型时，当重复的类型名称在一行上却相隔很远的时候，可以提醒读者的上下文。

```
// Good:
good := []*Type{
    {A: 42},
    {A: 43},
}
// Bad:
repetitive := []*Type{
    &Type{A: 42},
    &Type{A: 43},
}
// Good:
good := map[Type1]*Type2{
    {A: 1}: {B: 2},
    {A: 3}: {B: 4},
}
// Bad:
repetitive := map[Type1]*Type2{
    Type1{A: 1}: &Type2{B: 2},
    Type1{A: 3}: &Type2{B: 4},
}
```

提示：如果你想删除结构字中重复的类型名称，可以运行`gofmt -s`。

零值字段

零值字段可以从结构字段中省略，但不能因此而失去`清晰`这个风格原则。

设计良好的 API 经常采用零值结构来提高可读性。例如，从下面的结构中省略三个零值字段，可以使人们注意到正在指定的唯一选项。

```
// Bad:
import (
  "github.com/golang/leveldb"
  "github.com/golang/leveldb/db"
)

ldb := leveldb.Open("/my/table", &db.Options{
    BlockSize int: 1<<16,
    ErrorIfDBExists: true,

    // These fields all have their zero values.
    BlockRestartInterval: 0,
    Comparer: nil,
    Compression: nil,
    FileSystem: nil,
    FilterPolicy: nil,
    MaxOpenFiles: 0,
    WriteBufferSize: 0,
    VerifyChecksums: false,
})
// Good:
import (
  "github.com/golang/leveldb"
  "github.com/golang/leveldb/db"
)

ldb := leveldb.Open("/my/table", &db.Options{
    BlockSize int: 1<<16,
    ErrorIfDBExists: true,
})
```

表驱动的测试中的结构经常受益于显式字段名，特别是当测试结构不是琐碎的时候。这允许作者在有关字段与测试用例无关时完全省略零值字段。例如，成功的测试案例应该省略任何与错误或失败相关的字段。在零值对于理解测试用例是必要的情况下，例如测试零或 `nil` 输入，应该指定字段名。

简明

```
tests := []struct {
    input      string
    wantPieces []string
    wantErr    error
}{
    {
        input:      "1.2.3.4",
        wantPieces: []string{"1", "2", "3", "4"},
    },
    {
        input:   "hostname",
        wantErr: ErrBadHostname,
    },
}
```

明确

```
tests := []struct {
    input    string
    wantIPv4 bool
    wantIPv6 bool
    wantErr  bool
}{
    {
        input:    "1.2.3.4",
        wantIPv4: true,
        wantIPv6: false,
    },
    {
        input:    "1:2::3:4",
        wantIPv4: false,
        wantIPv6: true,
    },
    {
        input:    "hostname",
        wantIPv4: false,
        wantIPv6: false,
        wantErr:  true,
    },
}
```

Nil 切片

在大多数情况下，`nil`和空切片之间没有功能上的区别。像`len`和`cap`这样的内置函数在`nil`片上的表现与预期相同。

```
// Good:
import "fmt"

var s []int         // nil

fmt.Println(s)      // []
fmt.Println(len(s)) // 0
fmt.Println(cap(s)) // 0
for range s {...}   // no-op

s = append(s, 42)
fmt.Println(s)      // [42]
```

如果你声明一个空切片作为局部变量（特别是如果它可以成为返回值的来源），最好选择 nil 初始化，以减少调用者的错误风险

```
// Good:
var t []string
// Bad:
t := []string{}
```

不要创建强迫调用者区分 nil 和空切片的 API。

```
// Good:
// Ping pings its targets.
// Returns hosts that successfully responded.
func Ping(hosts []string) ([]string, error) { ... }
// Bad:
// Ping pings its targets and returns a list of hosts
// that successfully responded. Can be empty if the input was empty.
// nil signifies that a system error occurred.
func Ping(hosts []string) []string { ... }
```

在设计接口时，避免区分 `nil` 切片和非 `nil` 的零长度切片，因为这可能导致微妙的编程错误。这通常是通过使用 `len` 来检查是否为空，而不是 `==nil` 来实现的。

这个实现同时将 `nil` 和零长度的切片视为 “空”。

```
// Good:
// describeInts describes s with the given prefix, unless s is empty.
func describeInts(prefix string, s []int) {
    if len(s) == 0 {
        return
    }
    fmt.Println(prefix, s)
}
```

而不是依靠二者的区别作为 API 的一部分：

```
// Bad:
func maybeInts() []int { /* ... */ }

// describeInts describes s with the given prefix; pass nil to skip completely.
func describeInts(prefix string, s []int) {
  // The behavior of this function unintentionally changes depending on what
  // maybeInts() returns in 'empty' cases (nil or []int{}).
  if s == nil {
    return
  }
  fmt.Println(prefix, s)
}

describeInts("Here are some ints:", maybeInts())
```

详见 in-band 错误.

缩进的混乱

如果断行会使其余的行与缩进的代码块对齐，则应避免引入断行。如果这是不可避免的，请留下一个空间，将代码块中的代码与包线分开。

```
// Bad:
if longCondition1 && longCondition2 &&
    // Conditions 3 and 4 have the same indentation as the code within the if.
    longCondition3 && longCondition4 {
    log.Info("all conditions met")
}
```

具体准则和例子见以下章节：

- 函数格式化
- Conditionals and loops
- Literal formatting

函数格式化

函数定义或方法声明的签名应该保持在一行，以避免缩进的混乱。

函数参数列表可以成为 Go 源文件中最长的几行。然而，它们在缩进的变化之前，因此很难以不使后续行看起来像函数体的一部分的混乱方式来断行。

```
// Bad:
func (r *SomeType) SomeLongFunctionName(foo1, foo2, foo3 string,
    foo4, foo5, foo6 int) {
    foo7 := bar(foo1)
    // ...
}
```

参见最佳实践，了解一些缩短函数调用的选择，否则这些函数会有很多参数。

```
// Good:
good := foo.Call(long, CallOptions{
    Names:   list,
    Of:      of,
    The:     parameters,
    Func:    all,
    Args:    on,
    Now:     separate,
    Visible: lines,
})
// Bad:
bad := foo.Call(
    long,
    list,
    of,
    parameters,
    all,
    on,
    separate,
    lines,
)
```

通过分解局部变量，通常可以缩短行数。

```
// Good:
local := helper(some, parameters, here)
good := foo.Call(list, of, parameters, local)
```

类似地，函数和方法调用不应该仅仅由于行的长度而进行换行。

```
// Good:
good := foo.Call(long, list, of, parameters, all, on, one, line)
// Bad:
bad := foo.Call(long, list, of, parameters,
    with, arbitrary, line, breaks)
```

不要为特定的函数参数添加注释。相反，使用 option struct 或在函数文档中添加更多细节。

```
// Good:
good := server.New(ctx, server.Options{Port: 42})
// Bad:
bad := server.New(
    ctx,
    42, // Port
)
```

如果调用参数确实长得令人很难受，那么就应该考虑重构：

```
// Good:
// Sometimes variadic arguments can be factored out
replacements := []string{
    "from", "to", // related values can be formatted adjacent to one another
    "source", "dest",
    "original", "new",
}

// Use the replacement struct as inputs to NewReplacer.
replacer := strings.NewReplacer(replacements...)
```

当 API 无法更改或本地调用是不频繁的（无论调用是否太长），在有助于理解本次调用的前提下，那么是始终允许添加换行符的。

```
// Good:
canvas.RenderCube(cube,
    x0, y0, z0,
    x0, y0, z1,
    x0, y1, z0,
    x0, y1, z1,
    x1, y0, z0,
    x1, y0, z1,
    x1, y1, z0,
    x1, y1, z1,
)
```

请注意，上面示例中的行没有在特定的列边界处换行，而是根据坐标三元组进行分组。

函数中的长字符串不应该因为行的长度而被破坏。对于包含此类字符串的函数，可以在字符串格式之后添加换行符，并且可以在下一行或后续行中提供参数。最好根据输入的语义分组来决定换行符应该放在哪里，而不是单纯基于行长。

```
// Good:
log.Warningf("Database key (%q, %d, %q) incompatible in transaction started by (%q, %d,
%q)",
    currentCustomer, currentOffset, currentKey,
    txCustomer, txOffset, txKey)
// Bad:
log.Warningf("Database key (%q, %d, %q) incompatible in"+
    " transaction started by (%q, %d, %q)",
    currentCustomer, currentOffset, currentKey, txCustomer,
    txOffset, txKey)
```

条件和循环

`if` 语句不应换行； 多行 `if` 子句的形式会出现 缩进混乱带来的困扰。

```
// Bad:
// The second if statement is aligned with the code within the if block, causing
// indentation confusion.
if db.CurrentStatusIs(db.InTransaction) &&
    db.ValuesEqual(db.TransactionKey(), row.Key()) {
    return db.Errorf(db.TransactionError, "query failed: row (%v): key does not match
transaction key", row)
}
```

如果不需要短路 (short-circuit) 行为，可以直接提取布尔操作数：

```
// Good:
inTransaction := db.CurrentStatusIs(db.InTransaction)
keysMatch := db.ValuesEqual(db.TransactionKey(), row.Key())
if inTransaction && keysMatch {
    return db.Error(db.TransactionError, "query failed: row (%v): key does not match
transaction key", row)
}
```

尤其注意，在条件已经重复的情况下，很可能还是有可以提取的局部变量：

```
// Good:
uid := user.GetUniqueUserID()
if db.UserIsAdmin(uid) || db.UserHasPermission(uid, perms.ViewServerConfig) ||
db.UserHasPermission(uid, perms.CreateGroup) {
    // ...
}
// Bad:
if db.UserIsAdmin(user.GetUniqueUserID()) ||
db.UserHasPermission(user.GetUniqueUserID(), perms.ViewServerConfig) ||
db.UserHasPermission(user.GetUniqueUserID(), perms.CreateGroup) {
    // ...
}
```

包含闭包或多行结构文字的 `if` 语句应确保 大括号匹配 以避免 缩进混淆。

```
// Good:
if err := db.RunInTransaction(func(tx *db.TX) error {
    return tx.Execute(userUpdate, x, y, z)
}); err != nil {
    return fmt.Errorf("user update failed: %s", err)
}
// Good:
if _, err := client.Update(ctx, &upb.UserUpdateRequest{
    ID:   userID,
    User: user,
}); err != nil {
    return fmt.Errorf("user update failed: %s", err)
}
```

同样，不要尝试在 `for` 语句中人为的插入换行符。如果没有优雅的重构方式，是可以允许单纯的较长的行：

```
// Good:
for i, max := 0, collection.Size(); i < max && !collection.HasPendingWriters(); i++ {
    // ...
}
```

但是，通常可以优化为：

```
// Good:
for i, max := 0, collection.Size(); i < max; i++ {
    if collection.HasPendingWriters() {
        break
    }
    // ...
}
```

`switch` 和 `case` 语句都应始终保持在一行：

```
// Good:
switch good := db.TransactionStatus(); good {
case db.TransactionStarting, db.TransactionActive, db.TransactionWaiting:
    // ...
case db.TransactionCommitted, db.NoTransaction:
    // ...
default:
    // ...
}
// Bad:
switch bad := db.TransactionStatus(); bad {
case db.TransactionStarting,
    db.TransactionActive,
    db.TransactionWaiting:
    // ...
case db.TransactionCommitted,
    db.NoTransaction:
    // ...
default:
    // ...
}
```

如果行太长，将所有大小写缩进并用空行分隔以避免缩进混淆：

```
// Good:
switch db.TransactionStatus() {
case
    db.TransactionStarting,
    db.TransactionActive,
    db.TransactionWaiting,
    db.TransactionCommitted:

    // ...
case db.NoTransaction:
    // ...
default:
    // ...
}
```

在将变量比较的条件中，变量值放在等号运算符的左侧：

```
// Good:
if result == "foo" {
  // ...
}
```

不要采用常量在前的表达含糊的写法 (尤达条件式)

```
// Bad:
if "foo" == result {
  // ...
}
```

复制

为了避免意外的别名和类似的错误，从另一个包复制结构时要小心。例如 `sync.Mutex` 是不能复制的同步对象。

`bytes.Buffer` 类型包含一个 `[]byte` 切片和切片可以引用的小数组，这是为了对小字符串的优化。如果你复制一个 `Buffer`，复制的切片会指向原始切片中的数组，从而在后续方法调用产生意外的效果。

一般来说，如果类型的方法与指针类型 `*T` 相关联，不要复制类型为 `T` 的值。

```
// Bad:
b1 := bytes.Buffer{}
b2 := b1
```

调用值接收者的方法可以隐藏拷贝。当你编写 API 时，如果你的结构包含不应复制的字段，你通常应该采用并返回指针类型。

如此是可接受的:

```
// Good:
type Record struct {
  buf bytes.Buffer
  // other fields omitted
}

func New() *Record {...}

func (r *Record) Process(...) {...}

func Consumer(r *Record) {...}
```

但下面这种通常是错误的:

```
// Bad:
type Record struct {
  buf bytes.Buffer
  // other fields omitted
}

func (r Record) Process(...) {...} // Makes a copy of r.buf

func Consumer(r Record) {...} // Makes a copy of r.buf
```

这一指南同样也适用于 `sync.Mutex` 复制的情况。

不要 panic

不要使用 `panic` 进行正常的错误处理。相反，使用 `error` 和多个返回值。请参阅 关于错误的有效 Go 部分。

在 `package main` 和初始化代码中，考虑 `log.Exit` 中应该终止程序的错误（例如，无效配置），因为在许多这些情况下，堆栈跟踪对阅读者没有帮助。请注意 `log.Exit` 中调用了 `os.Exit`，此时所有 `defer` 函数都将不会运行。

对于那些表示“不可能”出现的条件错误、命名错误，应该在代码评审、测试期间发现，函数应合理地返回错误或调用 [`log.Fatal`](https://pkg.go.dev/github.com/golang/glog#Fatal)。

注意： `log.Fatalf` 不是标准库日志。请参阅 [#logging]。

Must 类函数

用于在失败时停止程序的辅助函数应遵循命名约定“MustXYZ”（或“mustXYZ”）。一般来说，它们应该只在程序启动的早期被调用，而不是在像用户输入时，此时更应该首选 `error` 处理。

这类方式，通常只在[包初始化时]（https://golang.org/ref/spec#Package_initialization）进行包级变量初始化的函数常见（例如 template.Must 和 regexp.MustCompile)。

```
// Good:
func MustParse(version string) *Version {
    v, err := Parse(version)
    if err != nil {
        log.Fatalf("MustParse(%q) = _, %v", version, err)
    }
    return v
}

// Package level "constant". If we wanted to use `Parse`, we would have had to
// set the value in `init`.
var DefaultVersion = MustParse("1.2.3")
```

相同的约定也可用于仅停止当前测试的情况（使用 `t.Fatal`）。这样在创建测试时通常很方便的，例如在表驱动测试 的结构字段中，作为返回错误的函数是不能直接复制给结构字段的。

```
// Good:
func mustMarshalAny(t *testing.T, m proto.Message) *anypb.Any {
  t.Helper()
  any, err := anypb.New(m)
  if err != nil {
    t.Fatalf("MustMarshalAny(t, m) = %v; want %v", err, nil)
  }
  return any
}

func TestCreateObject(t *testing.T) {
  tests := []struct{
    desc string
    data *anypb.Any
  }{
    {
      desc: "my test case",
      // Creating values directly within table driven test cases.
      data: mustMarshalAny(t, mypb.Object{}),
    },
    // ...
  }
  // ...
}
```

在这两种情况下，这种模式的价值在于可以在“值”上下文中调用。不应在难以确保捕获错误的地方或应检查错误的上下文中调用这些程序（如，在许多请求处理程序中）。对于常量输入，这允许测试确保“必须”的参数格式正确，对于非常量的输入，它允许测试验证错误是否正确处理或传播。

在测试中使用 `Must` 函数的地方，通常应该 标记为测试辅助函数 并调用 `t.Fatal`（请参阅测试辅助函数中的错误处理来了解使用它的更多注意事项）。

当有可能通过 普通错误处理 处理时，就不应该使用 `Must` 类函数：

```
// Bad:
func Version(o *servicepb.Object) (*version.Version, error) {
    // Return error instead of using Must functions.
    v := version.MustParse(o.GetVersionString())
    return dealiasVersion(v)
}
```

Goroutine 生命周期

当你生成 goroutines 时，要明确它们何时或是否退出。

Goroutines 可以在阻塞通道发送或接收出现泄漏。垃圾收集器不会终止一个 goroutine，即使它被阻塞的通道已经不可用。

即使 goroutine 没有泄漏，在不再需要时仍处于运行状态也会导致其他微妙且难以诊断的问题。向已关闭的通道上发送会导致panic。

```
// Bad:
ch := make(chan int)
ch <- 42
close(ch)
ch <- 13 // panic
```

“在结果已经不需要之后”修改仍在使用的入参可能会导致数据竞争。运行任意长时间的 goroutine 会导致不可预测的内存占用。

并发代码的编写应该让 goroutine 生命周期非常明显。通常，这意味着在与同步相关的代码限制的函数范围内，将逻辑分解为 同步函数。如果并发性仍然不明显，那么文档说明 goroutine 在何时、为何退出就很重要。

遵循上下文使用最佳实践的代码通常有助于明确这一点，其通常使用 `context.Context` 进行管理：

```
// Good:
func (w *Worker) Run(ctx context.Context) error {
    // ...
    for item := range w.q {
        // process 至少在ctx取消时会返回
        go process(ctx, item)
    }
    // ...
}
```

上面还有其他使用通道的情况，例如 `chan struct{}`、同步变量、条件变量 等等。重要的部分是 goroutine 的 结束对于后续维护者来说是显而易见的。

相比之下，以下代码不关心其衍生的 goroutine 何时完成：

```
// Bad:
func (w *Worker) Run() {
    // ...
    for item := range w.q {
        // process returns when it finishes, if ever, possibly not cleanly
        // handling a state transition or termination of the Go program itself.
        go process(item)
    }
    // ...
}
```

这段代码看起来还行，但有几个潜在的问题：

- 该代码在生产中可能有未定义的行为，即使操作系统已经释放资源，程序也可能没有完全干净地结束
- 由于代码的不确定生命周期，代码难以进行有效的测试
- 代码可能会出现资源泄漏，如上所述

更多可阅读：

- 永远不要在不知道它将如何停止的情况下启动 goroutine
- 重新思考经典并发模式：幻灯片，视频
- Go 程序何时结束

接口

Go 接口通常属于_使用_接口类型值的包，而不是_实现_接口类型的包。实现包应该返回具体的（通常是指针或结构）类型。这样就可以将新方法添加到实现中，而无需进行大量重构。有关详细信息，请参阅 GoTip #49：接受接口、返回具体类型。

不要从使用 API 导出接口的 test double 实现。相反，应设计可以使用 实际实现 的公共 API 进行测试。有关详细信息，请参阅 GoTip #42：为测试编写存根。即使在使用实现不可行的情况下，也没有必要引入一个完全覆盖类型所有方法的接口；消费者可以创建一个只包含它需要的方法的接口，如 GoTip #78: Minimal Viable Interfaces 中所示。

要测试使用 Stubby RPC 客户端的包，请使用真实的客户端连接。如果无法在测试中运行真实服务器，Google 的内部做法是使用内部 rpctest 包（即将推出！）获得与本地 [test double] 的真实客户端连接。

在使用之前不要定义接口（参见 TotT: Code Health: Eliminate YAGNI Smells）。如果没有实际的使用示例，就很难判断一个接口是否必要，更不用说它应该包含哪些方法了。

如果不需要传递不同的类型，则不要使用接口类型作为参数。

不要导出不需要开放的接口。

TODO: 写一个关于接口的更深入的文档并在这里链接到它。

```
// Good:
package consumer // consumer.go

type Thinger interface { Thing() bool }

func Foo(t Thinger) string { ... }
// Good:
package consumer // consumer_test.go

type fakeThinger struct{ ... }
func (t fakeThinger) Thing() bool { ... }
...
if Foo(fakeThinger{...}) == "x" { ... }
// Bad:
package producer

type Thinger interface { Thing() bool }

type defaultThinger struct{ ... }
func (t defaultThinger) Thing() bool { ... }

func NewThinger() Thinger { return defaultThinger{ ... } }
// Good:
package producer

type Thinger struct{ ... }
func (t Thinger) Thing() bool { ... }

func NewThinger() Thinger { return Thinger{ ... } }
```

泛型

在满足业务需求时，泛型（正式名称为“类型参数”）才应该被使用。在许多应用程序中，使用现有语言特性中传统方式（切片、映射、接口等）也可以正常工作，而不会增加复杂性，因此请注意不要过早使用。请参阅关于 最小机制 的讨论。

在引入使用泛型的导出 API 时，请确保对其进行适当的记录。强烈鼓励包含可运行的 示例。

不要仅仅因为你正在实现一个算法或不关心元素类型的数据结构而使用泛型。如果在实践中只有一种类型可以使用，那么首先让您的代码在该类型上工作，而不使用泛型。与其删除不必要的抽象，稍后为其添加多态将更简单。

不要使用泛型来发明领域特定语言 (DSL)。特别是，不要引入可能会给阅读者带来沉重负担的错误处理框架。相反，更应该使用 错误处理 做法。对于测试，要特别小心引入 断言库 或框架，尤其是很少发现失败case的。

一般来说：

- 写代码，不要去设计类型。来自 Robert Griesemer 和 Ian Lance Taylor 的 GopherCon 演讲。
- 如果你有几种类型共享一个统一接口，请考虑使用该接口对解决方案进行建模。这种情况可能不需要泛型。
- 否则，不要依赖 `any` 类型和过多的 类型断言的情况，应考虑泛型。

更多也可以参考：

- 在 Go 中使用泛型，Ian Lance Taylor 的演讲
- Go 网页上的泛型教程

参数值传递

不要为了节省几个字节而将指针作为函数参数传递。如果一个函数在整个过程中只将参数 `x` 处理为 `*x`，那么不应该采用指针。常见的例子包括传递一个指向字符串的指针（`*string`）或一个指向接口值的指针（`*io.Reader`）。在这两种情况下，值本身都是固定大小的，可以直接传递。

此建议不适用于大型结构体，甚至可能会增加大小的小型结构。特别是，`pb` 消息通常应该通过指针而不是值来处理。指针类型满足 `proto.Message` 接口（被 `proto.Marshal`、`protocmp.Transform` 等接受），并且协议缓冲区消息可能非常大，并且随着时间的推移通常会变得更大。

接收者类型

方法接收者 和常规函数参数一样，也可以使用值或指针传递。选择哪个应该基于该方法应该属于哪个[方法集]（https://golang.org/ref/spec#Method_sets）。

正确性胜过速度或简单性。 在某些情况下是必须使用指针的。在其他情况下，如果你对代码的增长方式没有很好的了解，请为大类型或考虑未来适用性上选择指针，并为简单的的数据使用值。

下面的列表更详细地说明了每个案例：

- 如果接收者是一个切片并且该方法没有重新切片或重新分配切片，应使用值而不是指针。

```
// Good:
type Buffer []byte

func (b Buffer) Len() int { return len(b) }
```

- 如果方法需要修改接收者，应使用指针。

```
// Good:
type Counter int

func (c *Counter) Inc() { *c++ }

// See https://pkg.go.dev/container/heap.
type Queue []Item

func (q *Queue) Push(x Item) { *q = append([]Item{x}, *q...) }
```

- 如果接收者包含 不能被安全复制的 字段, 应使用指针接收者。常见的例子是 `sync.Mutex` 和其他同步类型。

```
// Good:
type Counter struct {
    mu    sync.Mutex
    total int
}

func (c *Counter) Inc() {
    c.mu.Lock()
    defer c.mu.Unlock()
    c.total++
}
```

提示： 检查类型是否可被安全复制的相关信息可参考 Godoc。

- 如果接收者是“大”结构或数组，则指针接收者可能更有效。传递结构相当于将其所有字段或元素作为参数传递给方法。如果这看起来太大而无法按值传递，那么指针是一个不错的选择。
- 对于将调用修改接收者的其他函数，而这些修改对此方法不可见，请使用值类型； 否则使用指针。
- 如果接收者是一个结构或数组，其元素中的任何一个都是指向可能发生变化的东西的指针，那么更应该指针接收者以使阅读者清楚地了解可变性的意图。

```
// Good:
type Counter struct {
    m *Metric
}

func (c *Counter) Inc() {
    c.m.Add(1)
}
```

- 如果接收者是内置类型，例如整数或字符串，不需要修改，使用值。

```
// Good:
type User string

func (u User) String() { return string(u) }
```

- 接收者是 `map`，`function` 或 `channel`，使用值类型，而不是指针。

```
// Good:
// See https://pkg.go.dev/net/http#Header.
type Header map[string][]string

func (h Header) Add(key, value string) { /* omitted */ }
```

- 如果接收器是一个“小”数组或结构，它自然是一个没有可变字段和指针，那么值接收者通常是正确的选择。

```
// Good:
// See https://pkg.go.dev/time#Time.
type Time struct { /* omitted */ }

func (t Time) Add(d Duration) Time { /* omitted */ }
```

- 如有疑问，请使用指针接收者。

作为一般准则，最好将类型的方法设为全部指针方法或全部值方法。

注意： 关于是否值或指针的函数是否会影响性能，存在很多错误信息。编译器可以选择将指针传递到堆栈上的值以及复制堆栈上的值，但在大多数情况下，这些考虑不应超过代码的可读性和正确性。当性能确实很重要时，重要的是在确定一种方法优于另一种方法之前，用一个实际的基准来描述这两种方法。

`switch` 和 `break`

不要在 `switch` 子句末尾使用没有目标标签的 `break` 语句； 它们是多余的。与 C 和 Java 不同，Go 中的 `switch` 子句会自动中断，并且需要 `fallthrough` 语句来实现 C 风格的行为。如果你想阐明空子句的目的，请使用注释而不是 `break`。

```
// Good:
switch x {
case "A", "B":
    buf.WriteString(x)
case "C":
    // handled outside of the switch statement
default:
    return fmt.Errorf("unknown value: %q", x)
}
// Bad:
switch x {
case "A", "B":
    buf.WriteString(x)
    break // this break is redundant
case "C":
    break // this break is redundant
default:
    return fmt.Errorf("unknown value: %q", x)
}
```

> **Note:** If a `switch` clause is within a `for` loop, using `break` within `switch` does not exit the enclosing `for` loop.
>
> ```
> for {
>   switch x {
>   case "A":
>      break // exits the switch, not the loop
>   }
> }
> ```
>
> To escape the enclosing loop, use a label on the `for` statement:
>
> ```
> loop:
>   for {
>     switch x {
>     case "A":
>        break loop // exits the loop
>     }
>   }
> ```

同步函数

同步函数直接返回它们的结果，并在返回之前完成所有回调或通道操作。首选同步函数而不是异步函数。

同步函数使 goroutine 在调用中保持本地化。这有助于推理它们的生命周期，并避免泄漏和数据竞争。同步函数也更容易测试，因为调用者可以传递输入并检查输出，而无需轮询或同步。

如有必要，调用者可以通过在单独的 goroutine 中调用函数来添加并发性。然而，在调用方移除不必要的并发是相当困难的（有时是不可能的）。

也可以看看：

- “重新思考经典并发模式”，Bryan Mills 的演讲：幻灯片，[视频](https://www.youtube.com/ 看?v=5zXAHh5tJqQ)

类型别名

使用_类型定义_，`type T1 T2`，定义一个新类型。 使用 类型别名，`type T1 = T2` 来引用现有类型而不定义新类型。 类型别名很少见； 它们的主要用途是帮助将包迁移到新的源代码位置。不要在不需要时使用类型别名。

使用 %q

Go 的格式函数（`fmt.Printf` 等）有一个 `%q` 动词，它在双引号内打印字符串。

```
// Good:
fmt.Printf("value %q looks like English text", someText)
```

更应该使用 `%q` 而不是使用 `%s` 手动执行等效操作：

```
// Bad:
fmt.Printf("value \"%s\" looks like English text", someText)
// Avoid manually wrapping strings with single-quotes too:
fmt.Printf("value '%s' looks like English text", someText)
```

建议在供人使用的输出中使用 `%q`，其输入值可能为空或包含控制字符。可能很难注意到一个无声的空字符串，但是 `""` 就这样清楚地突出了。

使用 any

Go 1.18 将 `any` 类型作为 别名 引入到 `interface{}`。因为它是一个别名，所以 `any` 在许多情况下等同于 `interface{}`，而在其他情况下，它可以通过显式转换轻松互换。在新代码中应使用 `any`。

通用库

Flags

Google 代码库中的 Go 程序使用 标准 `flag` 包 的内部变体。它具有类似的接口，但与 Google 内部系统的互操作性很好。Go 二进制文件中的标志名称应该更应该使用下划线来分隔单词，尽管包含标志值的变量应该遵循标准的 Go 名称样式（混合大写字母）。具体来说，标志名称应该是蛇形命名，变量名称应该是驼峰命名。

```
// Good:
var (
    pollInterval = flag.Duration("poll_interval", time.Minute, "Interval to use for
polling.")
)
// Bad:
var (
    poll_interval = flag.Int("pollIntervalSeconds", 60, "Interval to use for polling in
seconds.")
)
```

Flags只能在 `package main` 或等效项中定义。

通用包应该使用 Go API 进行配置，而不是通过命令行界面进行配置；不要让导入库导出新标志作为副作用。也就是说，更倾向于显式的函数参数或结构字段分配，或者低频和严格审查的全局变量导出。在需要打破此规则的极少数情况下，标志名称必须清楚地表明它配置的包。

如果你的标志是全局变量，在导入部分之后，将它们放在 `var` 组中。

关于使用子命令创建 complex CLI (https://gocn.github.io/styleguide/docs/04-best-practices/#自定义日志级别) (https://gocn.github.io/styleguide/docs/04-best-practices/#自定义日志级别) 的最佳实践还有其他讨论。

也可以看看：

- 本周提示 #45：避免标记，尤其是在库代码中
- Go Tip #10：配置结构和标志
- Go Tip #80：依赖注入原则

日志

Google 代码库中的 Go 程序使用 标准 `log` 包 的变体。它具有类似但功能更强大的interface，并且可以与 Google 内部系统进行良好的互操作。该库的开源版本可通过 package `glog` 获得，开源 Google 项目可能会使用它，但本指南指的是它始终作为“日志”。

注意： 对于异常的程序退出，这个库使用 `log.Fatal` 通过堆栈跟踪中止，使用 `log.Exit` 在没有堆栈跟踪的情况下停止。标准库中没有 `log.Panic` 函数。

提示： `log.Info(v)` 等价于 `log.Infof("%v", v)`，其他日志级别也是如此。当你没有格式化要做时，首选非格式化版本。

也可以看看：

- 记录错误 和 自定义详细日志级别
- 何时以及如何使用日志包停止程序

上下文

`context.Context` 类型的值携带跨 API 和进程边界的安全凭证、跟踪信息、截止日期和取消信号。与 Google 代码库中使用线程本地存储的 C++ 和 Java 不同，Go 程序在整个函数调用链中显式地传递上下文，从传入的 RPC 和 HTTP 请求到传出请求。

当传递给函数或方法时，`context.Context` 始终是第一个参数。

```
func F(ctx context.Context /* other arguments */) {}
```

例外情况是：

- 在 HTTP 处理程序中，上下文来自 `req.Context()`。
- 在流式 RPC 方法中，上下文来自流。

  使用 gRPC 流的代码从生成的服务器类型中的 `Context()` 方法访问上下文，该方法实现了 `grpc.ServerStream`。请参阅 https://grpc.io/docs/languages/go/generated-code/。
- 在入口函数（此类函数的示例见下文）中，使用 `context.Background()`。
  - 在二进制目标中：`main`
  - 在通用代码和库中：`init`
  - 在测试中：`TestXXX`、`BenchmarkXXX`、`FuzzXXX`

> **注意**：调用链中间的代码很少需要使用 `context.Background()` 创建自己的基本上下文。更应该从调用者那里获取上下文，除非它是错误的上下文。
>
> 你可能会遇到服务库（在 Google 的 Go 服务框架中实现 Stubby、gRPC 或 HTTP），它们为每个请求构建一个新的上下文对象。这些上下文立即填充来自传入请求的信息，因此当传递给请求处理程序时，上下文的附加值已从客户端调用者通过网络边界传播给它。此外，这些上下文的生命周期仅限于请求的生命周期：当请求完成时，上下文将被取消。
>
> 除非你正在实现一个服务器框架，否则你不应该在库代码中使用 `context.Background()` 创建上下文。相反，如果有可用的现有上下文，则更应该使用下面提到的上下文分离。如果你认为在入口点函数之外确实需要 `context.Background()`，请在提交实现之前与 Google Go 风格的邮件列表讨论它。
> `context.Context` 在函数中首先出现的约定也适用于测试辅助函数。

```
// Good:
func readTestFile(ctx context.Context, t *testing.T, path string) string {}
```

不要将上下文成员添加到结构类型。相反，为需要传递它的类型的每个方法添加一个上下文参数。一个例外是其签名必须与标准库或 Google 无法控制的第三方库中的接口匹配的方法。这种情况非常罕见，应该在实施和可读性审查之前与 Google Go 风格的邮件列表讨论。

Google 代码库中必须产生可以在取消父上下文后运行的后台操作的代码可以使用内部包进行分离。关注 https://github.com/golang/go/issues/40221 讨论开源替代方案。

由于上下文是不可变的，因此可以将相同的上下文传递给共享相同截止日期、取消信号、凭据、父跟踪等的多个调用。

更多参见：

- 上下文和结构

自定义上下文

不要在函数签名中创建自定义上下文类型或使用上下文以外的接口。这条规定没有例外。

想象一下，如果每个团队都有一个自定义上下文。对于包 P 和 Q 的所有对，从包 P 到包 Q 的每个函数调用都必须确定如何将“PContext”转换为“QContext”。这对开发者来说是不切实际且容易出错的，并且它会进行自动重构 添加上下文参数几乎是不可能的。

如果你要传递应用程序数据，请将其放入参数、接收器、全局变量中，或者如果它确实属于那里，则放入 Context 值中。创建自己的 Context 类型是不可接受的，因为它破坏了 Go 团队使 Go 程序在生产中正常工作的能力。

crypto/rand

不要使用包 `math/rand` 来生成密钥，即使是一次性的。如果未生成随机种子，则生成器是完全可预测的。用 `time.Nanoseconds()` 生成种子，也只有几位熵。相反，请使用 `crypto/rand` ，如果需要文本，请打印为十六进制或 base64。

```
// Good:
import (
    "crypto/rand"
    // "encoding/base64"
    // "encoding/hex"
    "fmt"

    // ...
)

func Key() string {
    buf := make([]byte, 16)
    if _, err := rand.Read(buf); err != nil {
        log.Fatalf("Out of randomness, should never happen: %v", err)
    }
    return fmt.Sprintf("%x", buf)
    // or hex.EncodeToString(buf)
    // or base64.StdEncoding.EncodeToString(buf)
}
```

有用的测试失败

应该可以在不读取测试代码的情况下诊断测试失败。测试失败应当显示详细有用的消息说明：

- 是什么导致了失败
- 哪些输入导致错误
- 实际结果
- 预期的结果

下面概述了实现这一目标的具体约定。

断言库

不要创建“断言库”作为测试辅助函数。

断言库是试图在测试中结合验证和生成失败消息的库（尽管同样的陷阱也可能适用于其他测试辅助函数）。有关测试辅助函数和断言库之间区别的更多信息，请参阅 最佳实践。

```
// Bad:
var obj BlogPost

assert.IsNotNil(t, "obj", obj)
assert.StringEq(t, "obj.Type", obj.Type, "blogPost")
assert.IntEq(t, "obj.Comments", obj.Comments, 2)
assert.StringNotEq(t, "obj.Body", obj.Body, "")
```

断言库倾向于提前停止测试（如果 `assert` 调用 `t.Fatalf` 或 `panic`）或省略有关测试正确的相关信息：

```
// Bad:
package assert

func IsNotNil(t *testing.T, name string, val interface{}) {
    if val == nil {
        t.Fatalf("data %s = nil, want not nil", name)
    }
}

func StringEq(t *testing.T, name, got, want string) {
    if got != want {
        t.Fatalf("data %s = %q, want %q", name, got, want)
    }
}
```

复杂的断言函数通常不提供 有用的失败消息 和存在于测试函数中的上下文。太多的断言函数和库会导致开发人员体验支离破碎：我应该使用哪个断言库，它应该发出什么样的输出格式等问题？ 碎片化会产生不必要的混乱，特别是对于负责修复潜在下游破坏的库维护者和大规模更改的作者。与其创建用于测试的特定领域语言，不如使用 Go 本身。

断言库通常会排除比较和相等检查。更应该使用标准库，例如 `cmp` 和 `fmt` 修改为：

```
// Good:
var got BlogPost

want := BlogPost{
    Comments: 2,
    Body:     "Hello, world!",
}

if !cmp.Equal(got, want) {
    t.Errorf("blog post = %v, want = %v", got, want)
}
```

对于更多特定于域的比较助手，更应该返回一个可以在测试失败消息中使用的值或错误，而不是传递 `*testing.T` 并调用其错误报告方法：

```
// Good:
func postLength(p BlogPost) int { return len(p.Body) }

func TestBlogPost_VeritableRant(t *testing.T) {
    post := BlogPost{Body: "I am Gunnery Sergeant Hartman, your senior drill
instructor."}

    if got, want := postLength(post), 60; got != want {
        t.Errorf("length of post = %v, want %v", got, want)
    }
}
```

最佳实践： 如果 `postLength` 很重要，直接测试它是有意义的，独立于调用它的其他函数测试。

也可以看看：

- 等值比较和差异
- 打印差异
- 有关测试辅助函数和断言助手之间区别的更多信息，请参阅最佳实践

标识出函数

在大多数测试中，失败消息应该包括失败的函数的名称，即使从测试函数的名称中看起来很明显。具体来说，你的失败信息应该是 `YourFunc(%v) = %v, want %v` 而不仅仅是 `got %v, want %v`。

标识出输入

在大多数测试中，失败消息应该包括功能输入（如果它们很短）。如果输入的相关属性不明显（例如，因为输入很大或不透明），你应该使用对正在测试的内容的描述来命名测试用例，并将描述作为错误消息的一部分打印出来。

Got before want

测试输出应包括函数在打印预期值之前返回的实际值。打印测试输出的标准格式是 `YourFunc(%v) = %v, want %v`。在你会写“实际”和“预期”的地方，更应该分别使用“get”和“want”这两个词。

对于差异，方向性不太明显，因此包含一个有助于解释失败的关键是很重要的。请参阅 关于打印差异的部分。无论你在失败消息中使用哪种 diff 顺序，都应将其明确指示为失败消息的一部分，因为现有代码的顺序不一致。

全结构比较

如果你的函数返回一个结构体（或任何具有多个字段的数据类型，例如切片、数组和映射），请避免编写执行手动编码的结构体逐个字段比较的测试代码。相反，构建期望函数返回的数据，并使用 深度比较 直接进行比较。

注意： 如果你的数据包含模糊测试意图的不相关字段，则这不适用。

如果你的结构比较时需要近似相等（或等效类型的语义），或者它包含无法比较相等的字段（例如，如果其中一个字段是 `io.Reader`），请调整 `cmp. Diff` 或 `cmp.Equal` 与 `cmpopts` 选项比较，例如 `cmpopts.IgnoreInterfaces` 可能满足你的需求（示例）。

如果你的函数返回多个返回值，则无需在比较它们之前将它们包装在结构中。只需单独比较返回值并打印它们。

```
// Good:
val, multi, tail, err := strconv.UnquoteChar(`\"Fran & Freddie's Diner\"`, '"')
if err != nil {
  t.Fatalf(...)
}
if val != `"` {
  t.Errorf(...)
}
if multi {
  t.Errorf(...)
}
if tail != `Fran & Freddie's Diner"` {
  t.Errorf(...)
}
```

比较稳定的结果

避免比较那些可能依赖于非自有包输出稳定性的结果。相反，测试应该在语义相关的信息上进行比较，这些信息是稳定的，并能抵抗依赖关系的变化。对于返回格式化字符串或序列化字节的功能，一般来说，假设输出是稳定的是不安全的。

例如，`json.Marshal` 可以改变（并且在过去已经改变）它所输出的特定字节。如果 `json` 包改变了它序列化字节的方式，在 JSON 字符串上执行字符串相等的测试可能会中断。相反，一个更强大的测试将解析 JSON 字符串的内容，并确保它在语义上等同于一些预期的数据结构。

测试继续进行

测试应该尽可能地持续下去，即使是在失败之后，以便在一次运行中打印出所有的失败检查。这样一来，正在修复失败测试的开发人员就不必在修复每个错误后重新运行测试来寻找下一个错误。

更倾向于调用 `t.Error` 而不是 `t.Fatal` 来报告不匹配。当比较一个函数输出的几个不同属性时，对每一个比较都使用 `t.Error`。

调用 `t.Fatal` 主要用于报告一个意外的错误情况，当后续的比较失败是没有意义的。

对于表驱动的测试，考虑使用子测试，使用 `t.Fatal` 而不是 `t.Error` 和 `continue`。参见[GoTip #25: Subtests: Making Your Tests Lean]（https://gocn.github.io/styleguide/docs/01-overview/#gotip）。

最佳实践：关于何时应使用 `t.Fatal` 的更多讨论，见最佳实践。

等值比较和差异

`==` 操作符使用语言定义的比较来评估相等性。标量值 (数字、布尔运算等) 根据其值进行比较, 但只有一些结构和接口可以用这种方式进行比较。指针的比较是基于它们是否指向同一个变量，而不是基于它们所指向的值是否相等。

对于类似切片这种情况下，`==` 是不能正确处理比较的，`cmp` 包则可以用于比较更复杂的数据结构。使用 `cmp.Equal` 进行等价比较，使用 `cmp.Diff` 获得对象之间可供人类阅读的差异。

```
// Good:
want := &Doc{
    Type:     "blogPost",
    Comments: 2,
    Body:     "This is the post body.",
    Authors:  []string{"isaac", "albert", "emmy"},
}
if !cmp.Equal(got, want) {
    t.Errorf("AddPost() = %+v, want %+v", got, want)
}
```

作为一个通用的比较库，`cmp` 可能不知道如何比较某些类型。例如，它只能在传递 `protocmp.Transform` 选项时比较 `protobuf` 的信息。

```
// Good:
if diff := cmp.Diff(want, got, protocmp.Transform()); diff != "" {
    t.Errorf("Foo() returned unexpected difference in protobuf messages (-want +got):
%s", diff)
}
```

虽然 `cmp` 包不是Go标准库的一部分，但它是由Go团队维护的，随着时间的推移应该会产生稳定的相等结果。它是用户可配置的，应该可以满足大多数的比较需求。

现有的代码可能会使用以下旧的库，为了保持一致性，可以继续使用它们。

- `pretty` 产生美观的差异报告。然而，它非常谨慎地认为具有相同视觉表现的数值是相等的。特别注意，`pretty` 不区分nil切片和空切片之间的差异，对具有相同字段的不同接口实现也不敏感，而且有可能使用嵌套图作为与结构值比较的基础。在产生差异之前，它还会将整个值序列化为一个字符串，因此对于比较大的值来说不是一个好的选择。默认情况下，它比较的是未导出的字段，这使得它对你的依赖关系中实现细节的变化很敏感。由于这个原因，在protobuf信息上使用 `pretty` 是不合适的。

在新的代码中更倾向于使用 `cmp`，值得考虑更新旧的代码，在实际可行的情况下使用 `cmp`。

旧的代码可以使用标准库中的 `reflect.DeepEqual` 函数来比较复杂的结构。`reflect.DeepEqual` 不应该被用来检查等值比较，因为它对未导出的字段和其他实现细节的变化很敏感。使用 `reflect.DeepEqual` 的代码应该更新为上述库之一。

注意： `cmp` 包是为测试而设计的，而不是用于生产。因此，当它怀疑一个比较被错误地执行时，它可能会panic，以向用户提供如何改进测试的指导，使其不那么脆弱。鉴于cmp的恐慌倾向，它不适合在生产中使用的代码，因为虚假的panic可能是致命的。

详细程度

传统的失败信息，适用于大多数Go测试，是 `YourFunc(%v) = %v, want %v`。然而，有些情况可能需要更多或更少的细节。

- 进行复杂交互的测试也应该描述交互。例如，如果同一个 `YourFunc` 被调用了好几次，那么要确定哪个调用未通过测试。如果知道系统的任何额外状态是很重要的，那么在失败输出中应包括这些（或者至少在日志中）。
- 如果数据是一个复杂的结构，在消息中只描述重要的部分是可以接受的，但不要过分掩盖数据。
- 设置失败不需要同样水平的细节。如果一个测试辅助函数填充了一个Spanner表，但Spanner却坏了，你可能不需要包括你要存储在数据库中的测试输入。`t.Fatalf("Setup: Failed to set up test database: %s", err)` 通常足以解决这个问题。

提示：应该在开发过程中触发失败。审查失败信息是什么样子的，维护者是否能有效地处理失败。

有一些技术可以清晰地再现测试输入和输出：

- 当打印字符串数据时，`%q` 通常是有用的以强调该值的重要性，并更容易发现坏值。
- 当打印（小）结构时，`%+v` 可能比 `%v` 更有用。
- 当验证较大的值失败时，打印差异可以使人们更容易理解失败的原因。

打印差异

如果你的函数返回较大的输出，那么当你的测试失败时，阅读失败信息的人很难发现其中的差异。与其同时打印返回值和想要的值，不如做一个差异。

为了计算这些值的差异，`cmp.Diff` 是首选，特别是对于新的测试和新的代码，但也可以使用其他工具。关于每个函数的优点和缺点的指导，见类型的等值。

- `cmp.Diff`
- `pretty.Compare`

你可以使用 `diff` 包来比较多行字符串或字符串的列表。你可以把它作为其他类型的比较的构建块。

在失败信息中添加一些文字，解释差异的方向。

- 当你使用 `cmp`，`pretty` 和 `diff` 包时，类似 `diff (-want +got)` 的东西很好（如果把 `(want, got)` 传递给函数），因为你添加到格式字符串中的 `-` 和 `+` 将与实际出现在diff行开头的 `-` 和 `+` 匹配。如果把 `(got, want)` 传给函数，正确的键将是 `(-got +want)`。
- `messagediff` 包使用不同的输出格式，所以当你使用它时，`diff (want -> got)` 的信息是合适的（如果把 `(want, got)` 传给函数），因为箭头的方向将与 “修改 “行中箭头的方向一致。

差异将跨越多行，所以应该在打印差异之前打印一个新行。

测试错误语义

当一个单元测试执行字符串比较或使用 `cmp` 来检查特定的输入是否返回特定种类的错误时，你可能会发现，如果这些错误信息在将来被重新修改，你的测试就会很脆弱。因为这有可能将你的单元测试变成一个变化检测器（参见 TotT: Change-Detector Tests Considered Harmful），不要使用字符串比较来检查你的函数返回什么类型的错误。然而，允许使用字符串比较来检查来自被测包的错误信息是否满足某些属性，例如，它是否包括参数名称。

Go中的错误值通常有一个用于人眼的部分和一个用于语义控制流的部分。测试应该力求只测试可以可靠观察到的语义信息，而不是显示用于人类调试的信息，因为这往往会在未来发生变化。关于构建具有语义的错误的指导，请参见关于错误的最佳实践。如果语义信息不充分的错误来自于你无法控制的依赖关系，请考虑针对所有者提交一个错误，以帮助改进API，而不是依靠解析错误信息。

在单元测试中，通常只关心错误是否发生。如果是这样，那么在你预期发生错误时，只测试错误是否为非零就足够了。如果你想测试错误在语义上与其他错误相匹配，那么可以考虑使用 `cmp` 与 `cmpopts.EquateErrors`。

> **注意：**如果一个测试使用了 `cmpopts.EquateErrors`，但是它所有的 `wantErr` 值都是 `nil` 或者 `cmpopts.AnyError`，那么使用 `cmp` 是不必要的。简化代码，使 want 字段改为 `bool` 类型，然后就可以用 `!=` 进行简单的比较。

```
// Good:
gotErr := f(test.input) != nil
if gotErr != test.wantErr {
    t.Errorf("f(%q) returned err = %v, want error presence = %v", test.input, gotErr,
test.wantErr)
}
```

另请参阅 GoTip #13：设计用于检查的错误。

测试结构

子测试

标准的 Go 测试库提供了一种工具来 定义子测试。这允许在设置和清理、控制并行性和测试过滤方面具有灵活性。子测试可能很有用（特别是对于表驱动测试），但使用它们不是强制性的。另请参阅 https://blog.golang.org/subtests。

子测试不应该依赖于其他 case 的执行来获得成功或初始状态，因为子测试应该能够使用 `go test -run` 标志或使用 Bazel 测试过滤器 表达式。

子测试名称

命名子测试，使其在测试输出中可读，并且在命令行上对测试过滤的用户有用。当你使用“t.Run”创建子测试时，第一个参数用作测试的描述性名称。为了确保测试结果对于阅读日志的人来说是清晰的，请选择在转义后仍然有用且可读的子测试名称。将子测试名称视为函数标识符而不是散文描述。测试运行程序用下划线替换空格，并转义非打印字符。如果你的测试数据受益于更长的描述，请考虑将描述放在单独的字段中（可能使用“t.Log”或与失败消息一起打印）。

可以使用 Go 测试运行器 或 Bazel 测试过滤器 的标志单独运行子测试，选择易于输入的描述性名称。

> **警告：**斜杠字符在子测试名称中特别不友好，因为它们具有[测试过滤器的特殊含义](https://blog.golang.org/subtests#:~:text=Perhaps 一位匹配任何测试）。

```
# Bad:
# Assuming TestTime and t.Run("America/New_York", ...)
bazel test :mytest --test_filter="Time/New_York"    # Runs nothing!
bazel test :mytest --test_filter="Time//New_York"   # Correct, but awkward.
```

要识别函数的输入，将它们包含在测试的失败消息中，它们不会被测试执行者所忽略。

```
// Good:
func TestTranslate(t *testing.T) {
    data := []struct {
        name, desc, srcLang, dstLang, srcText, wantDstText string
    }{
        {
            name:        "hu=en_bug-1234",
            desc:        "regression test following bug 1234. contact: cleese",
            srcLang:     "hu",
            srcText:     "cigarettát és egy öngyújtót kérek",
            dstLang:     "en",
            wantDstText: "cigarettes and a lighter please",
        }, // ...
    }
    for _, d := range data {
        t.Run(d.name, func(t *testing.T) {
            got := Translate(d.srcLang, d.dstLang, d.srcText)
            if got != d.wantDstText {
                t.Errorf("%s
Translate(%q, %q, %q) = %q, want %q",
                    d.desc, d.srcLang, d.dstLang, d.srcText, got, d.wantDstText)
            }
        })
    }
}
```

以下是一些要避免的事情的例子：

```
// Bad:
// Too wordy.
t.Run("check that there is no mention of scratched records or hovercrafts", ...)
// Slashes cause problems on the command line.
t.Run("AM/PM confusion", ...)
```

表驱动测试

当许多不同的测试用例可以使用相似的测试逻辑进行测试时，使用表驱动测试。

- 测试函数的实际输出是否等于预期输出时。例如，许多 `fmt.Sprintf` 的测试 或下面的最小片段。
- 测试函数的输出是否始终符合同一组不变量时。例如，测试 `net.Dial`。

这是从标准“字符串”库复制的表驱动测试的最小结构。如果需要，你可以使用不同的名称，将测试切片移动到测试函数中，或者添加额外的工具，例如子测试或设置和清理函数。始终牢记有用的测试失败。

```
// Good:
var compareTests = []struct {
    a, b string
    i    int
}{
    {"", "", 0},
    {"a", "", 1},
    {"", "a", -1},
    {"abc", "abc", 0},
    {"ab", "abc", -1},
    {"abc", "ab", 1},
    {"x", "ab", 1},
    {"ab", "x", -1},
    {"x", "a", 1},
    {"b", "x", -1},
    // test runtime·memeq's chunked implementation
    {"abcdefgh", "abcdefgh", 0},
    {"abcdefghi", "abcdefghi", 0},
    {"abcdefghi", "abcdefghj", -1},
}

func TestCompare(t *testing.T) {
    for _, tt := range compareTests {
        cmp := Compare(tt.a, tt.b)
        if cmp != tt.i {
            t.Errorf(`Compare(%q, %q) = %v`, tt.a, tt.b, cmp)
        }
    }
}
```

注意：上面这个例子中的失败消息满足了识别函数和识别输入。无需用数字标识行。

当需要使用与其他测试用例不同的逻辑来检查某些测试用例时，编写多个测试函数更为合适，如 GoTip #50: Disjoint Table Tests。当表中的每个条目都有自己不同的条件逻辑来检查每个输出的输入时，测试代码的逻辑可能会变得难以理解。如果测试用例具有不同的逻辑但设置相同，则单个测试函数中的一系列子测试 可能有意义。

你可以将表驱动测试与多个测试函数结合起来。例如，当测试函数的输出与预期输出完全匹配并且函数为无效输入返回非零错误时，编写两个单独的表驱动测试函数是最好的方法：一个用于正常的非错误输出，一个用于错误输出。

数据驱动的测试用例

表测试行有时会变得复杂，行值指示测试用例内的条件行为。测试用例之间重复的额外清晰度对于可读性是必要的。

```
// Good:
type decodeCase struct {
    name   string
    input  string
    output string
    err    error
}

func TestDecode(t *testing.T) {
    // setupCodex is slow as it creates a real Codex for the test.
    codex := setupCodex(t)

    var tests []decodeCase // rows omitted for brevity

    for _, test := range tests {
        t.Run(test.name, func(t *testing.T) {
            output, err := Decode(test.input, codex)
            if got, want := output, test.output; got != want {
                t.Errorf("Decode(%q) = %v, want %v", test.input, got, want)
            }
            if got, want := err, test.err; !cmp.Equal(got, want) {
                t.Errorf("Decode(%q) err %q, want %q", test.input, got, want)
            }
        })
    }
}

func TestDecodeWithFake(t *testing.T) {
    // A fakeCodex is a fast approximation of a real Codex.
    codex := newFakeCodex()

    var tests []decodeCase // rows omitted for brevity

    for _, test := range tests {
        t.Run(test.name, func(t *testing.T) {
            output, err := Decode(test.input, codex)
            if got, want := output, test.output; got != want {
                t.Errorf("Decode(%q) = %v, want %v", test.input, got, want)
            }
            if got, want := err, test.err; !cmp.Equal(got, want) {
                t.Errorf("Decode(%q) err %q, want %q", test.input, got, want)
            }
        })
    }
}
```

在下面的反例中，请注意在case设置中区分每个测试案例使用哪种类型的 `Codex` 是多么困难。（突出显示的部分与 TotT：数据驱动陷阱！ 的建议相冲突。）

```
// Bad:
type decodeCase struct {
  name   string
  input  string
  codex  testCodex
  output string
  err    error
}

type testCodex int

const (
  fake testCodex = iota
  prod
)

func TestDecode(t *testing.T) {
  var tests []decodeCase // rows omitted for brevity

  for _, test := tests {
    t.Run(test.name, func(t *testing.T) {
      var codex Codex
      switch test.codex {
      case fake:
        codex = newFakeCodex()
      case prod:
        codex = setupCodex(t)
      default:
        t.Fatalf("unknown codex type: %v", codex)
      }
      output, err := Decode(test.input, codex)
      if got, want := output, test.output; got != want {
        t.Errorf("Decode(%q) = %q, want %q", test.input, got, want)
      }
      if got, want := err, test.err; !cmp.Equal(got, want) {
        t.Errorf("Decode(%q) err %q, want %q", test.input, got, want)
      }
    })
  }
}
```

标识行

不要使用测试表中的测试索引来代替命名测试或打印输入。没有人愿意通过你的测试表并计算条目以找出哪个测试用例失败。

```
// Bad:
tests := []struct {
    input, want string
}{
    {"hello", "HELLO"},
    {"wORld", "WORLD"},
}
for i, d := range tests {
    if strings.ToUpper(d.input) != d.want {
        t.Errorf("failed on case #%d", i)
    }
}
```

在你的测试结构中添加测试描述，并将其与失败信息一起打印。当使用子测试时，你的子测试名称应能有效识别行。

重要的是：即使`t.Run`对输出和执行有一定的范围，你必须始终识别输入。表的测试行名称必须遵循子测试命名的指导。

测试辅助函数

一个测试辅助函数是一个执行设置或清理任务的函数。所有发生在测试辅助函数中的故障都被认为是环境的故障（而不是被测代码的故障）–例如，当一个测试数据库不能被启动，因为在这台机器上没有更多的空闲端口。

如果你传递一个`*testing.T`，调用`t.Helper`，将测试辅助函数中的故障归结到调用助手的那一行。这个参数应该在context参数之后，如果有的话，在任何其他参数之前。

```
// Good:
func TestSomeFunction(t *testing.T) {
    golden := readFile(t, "testdata/golden-result.txt")
    // ... tests against golden ...
}

// readFile returns the contents of a data file.
// It must only be called from the same goroutine as started the test.
func readFile(t *testing.T, filename string) string {
    t.Helper()
    contents, err := runfiles.ReadFile(filename)
    if err != nil {
        t.Fatal(err)
    }
    return string(contents)
}
```

当这种模式掩盖了测试失败和导致失败的条件之间的联系时，请不要使用这种模式。具体来说，关于断言库的指导仍然适用，`t.Helper`不应该被用来实现这种库。

提示：更多关于测试辅助函数和断言助手的区别，请参见最佳实践。

虽然上面提到的是`*testing.T`，但大部分建议对基准和模糊帮助器来说都是一样的。

测试包

同一包内的测试

测试可以和被测试的代码定义在同一个包里。

要在同一个包中编写测试

- 将测试放在一个 `foo_test.go` 文件中
- 在测试文件中使用 `package foo`。
- 不要明确地导入要测试的包

```
# Good:
go_library(
    name = "foo",
    srcs = ["foo.go"],
    deps = [
        ...
    ],
)

go_test(
    name = "foo_test",
    size = "small",
    srcs = ["foo_test.go"],
    library = ":foo",
    deps = [
        ...
    ],
)
```

同一个包中的测试可以访问包中未导出的标识符。这可以实现更好的测试覆盖率和更简洁的测试。注意在测试中声明的任何examples都不会有用户在他们的代码中需要的包名。

不同包中的测试

将测试定义在与被测代码相同的包中并不总是合适的，甚至不可能。在这种情况下，使用一个带有`_test`后缀的包名。这是对包名的“不使用下划线"规则的一个例外。比如说。

- 如果一个集成测试没有一个它明显属于的库

  ```
  // Good:
  package gmailintegration_test

  import "testing"
  ```

- 如果在同一软件包中定义测试会导致循环依赖性

```
// Good:
package fireworks_test

import (
  "fireworks"
  "fireworkstestutil" // fireworkstestutil also imports fireworks
)
```

使用 `testing` 包

Go标准库提供了 `testing` 包。这是Google代码库中唯一允许用于Go代码的测试框架。特别是断言库和第三方测试框架是不允许的。

`testing` 包为编写好的测试提供了最小但完整的功能集。

- 顶层测试
- 基准
- 可运行的例子
- 子测试
- 记录
- 失败和致命的失败

这些设计是为了与核心语言特性如复合字面和带有初始化的if语句语法协同工作，使测试作者能够编写[清晰、可读、可维护的测试]。

非决策性的

风格指南不能列举所有事项的正面规定，也不能列举所有它不提供意见的事项。也就是说，这里有几件可读性社区以前争论过但没有达成共识的事情。

- **零值的局部变量初始化。** `var i int` 和 `i := 0` 是等同的。参见初始化最佳实践。
- **空的复合字面与 `new` 或 `make`。** `&File{}` 和 `new(File)` 是等同的。`map[string]bool{}` 和 `make(map[string]bool)` 也是如此。参见复合声明最佳实践。
- **got/want参数在cmp.Diff调用中的排序。** 要有本地一致性，并在你的失败信息中包括一个图例。
- **`errors.New` 与 `fmt.Errorf` 在非格式化字符串上的对比。** `errors.New("foo")` 和 `fmt.Errorf("foo")` 可以互换使用。

如果有特殊情况，它们又出现了，可读性导师可能会做一个可选的注释，但一般来说，作者可以自由选择他们在特定情况下喜欢的风格。

当然，如果风格指南中没有涉及的东西确实需要更多的讨论，欢迎在具体的审查中，或者在内部留言板上提出来。

最佳实践

原文：https://google.github.io/styleguide/go

概述 | 风格指南 | 风格决策 | 最佳实践

注意： 本文是 Google Go 风格 系列文档的一部分。本文档是 **规范性 (normative) 但不是强制规范 (canonical)**，并且从属于 Google 风格指南。请参阅概述获取更多详细信息。

关于

本文件记录了**关于如何更好地应用 Go 风格指南的指导意见**。该指导旨在解决经常出现的通用问题，但不一定适用于所有情况。在可能的情况下，我们讨论了多种替代方法，以及决定何时该用和何时不该用这些方法的考虑因素。

查看概述来获取完整的风格指导文档

命名

函数和方法名称

避免重复

在为一个函数或方法选择名称时，要考虑该名称将被阅读的环境。请考虑以下建议，以避免在调用地点出现过多的重复：

- 以下内容一般可以从函数和方法名称中省略。
  - 输入和输出的类型（当没有冲突的时候）
  - 方法的接收器的类型
  - 一个输入或输出是否是一个指针
- 对于函数，不要重复软件包的名称。

```go
// Bad:
package yamlconfig

func ParseYAMLConfig(input string) (*Config, error)
```

```go
// Good:
package yamlconfig

func Parse(input string) (*Config, error)
```

- 对于方法不要重复方法接收器的名称。

```go
// Bad:
func (c *Config) WriteConfigTo(w io.Writer) (int64, error)
```

```go
// Good:
func (c *Config) WriteTo(w io.Writer) (int64, error)
```

- 不要重复传参的变量名称

```
// Bad:
func OverrideFirstWithSecond(dest, source *Config) error
```

```
// Good:
func Override(dest, source *Config) error
```

- 不要重复返回值的名称和类型

```
// Bad:
func TransformYAMLToJSON(input *Config) *jsonconfig.Config
```

```
// Good:
func Transform(input *Config) *jsonconfig.Config
```

当有必要区分类似名称的函数时，包含额外信息是可以接受的。

```
// Good:
func (c *Config) WriteTextTo(w io.Writer) (int64, error)
func (c *Config) WriteBinaryTo(w io.Writer) (int64, error)
```

命名约定

在为函数和方法选择名称时，还有一些常见的约定：

- 有返回结果的函数使用类名词的名称。

```
// Good:
func (c *Config) JobName(key string) (value string, ok bool)
```

这方面的一个推论是，函数和方法名称应该避免使用前缀 `Get`。

```
// Bad:
func (c *Config) GetJobName(key string) (value string, ok bool)
```

- 做事的函数被赋予类似动词的名称。

```
// Good:
func (c *Config) WriteDetail(w io.Writer) (int64, error)
```

- 只因所涉及的类型而不同，而功能相同的函数在名称的末尾带上类型的名称。

```
// Good:
func ParseInt(input string) (int, error)
func ParseInt64(input string) (int64, error)
func AppendInt(buf []byte, value int) []byte
func AppendInt64(buf []byte, value int64) []byte
```

如果有一个明确的“主要“版本，该版本的名称中可以省略类型：

```
// Good:
func (c *Config) Marshal() ([]byte, error)
func (c *Config) MarshalText() (string, error)
```

测试替身包和类型

有几个原则你可以应用于命名包和类型，提供测试辅助函数，特别是测试替身。一个测试替身可以是一个桩、假的、模拟的或间谍的。 这些例子大多使用打桩。如果你的代码使用假的或其他类型的测试替身，请相应地更新你的名字。 假设你有一个重点突出的包，提供与此类似的生产代码：

```
package creditcard

import (
    "errors"

    "path/to/money"
)

// ErrDeclined indicates that the issuer declines the charge.
var ErrDeclined = errors.New("creditcard: declined")

// Card contains information about a credit card, such as its issuer,
// expiration, and limit.
type Card struct {
    // omitted
}

// Service allows you to perform operations with credit cards against external
// payment processor vendors like charge, authorize, reimburse, and subscribe.
type Service struct {
    // omitted
}

func (s *Service) Charge(c *Card, amount money.Money) error { /* omitted */ }
```

创建测试辅助包

假设你想创建一个包，包含另一个包的测试替身。在这个例子中我们将使用 `package creditcard`（来自上面）。 一种方法是在生产包的基础上引入一个新的 Go 包进行测试。一个安全的选择是在原来的包名后面加上 `test` 这个词（“creditcard” + “test”）。

```
// Good:
package creditcardtest
```

除非另有明确说明，以下各节中的所有例子都在 `package creditcardtest` 中。

简单案例

你想为 `Service` 添加一组测试替身。因为 `Card` 是一个有效的哑巴数据类型，类似于协议缓冲区的消息，它在测试中不需要特殊处理，所以不需要替身。如果你预计只有一种类型（如`Service’）的测试替身，你可以采取一种简洁的方法来命名替身。

```go
// Good:
import (
    "path/to/creditcard"
    "path/to/money"
)

// Stub stubs creditcard.Service and provides no behavior of its own.
type Stub struct{}

func (Stub) Charge(*creditcard.Card, money.Money) error { return nil }
```

严格来说，这比像 `StubService` 或非常差的 `StubCreditCardService` 这样的命名选择要好，因为基础包的名字和它的域类型意味着 `creditcardtest.Stub` 是什么。 最后，如果该包是用 Bazel 构建的，确保该包的新 `go_library` 规则被标记为 `testonly`。

```
# Good:
go_library(
    name = "creditcardtest",
    srcs = ["creditcardtest.go"],
    deps = [
        ":creditcard",
        ":money",
    ],
    testonly = True,
)
```

上述方法是常规的，其他工程师也会很容易理解。

另见:

- Go Tip #42: 为测试便编写桩

多重测试替身行为

当一种桩不够用时（例如，你还需要一种总是失败的桩），我们建议根据它们所模拟的行为来命名桩。这里我们把 `Stub` 改名为 `AlwaysCharges`，并引入一个新的桩，叫做 `AlwaysDeclines`：

```go
// Good:
// AlwaysCharges stubs creditcard.Service and simulates success.
type AlwaysCharges struct{}

func (AlwaysCharges) Charge(*creditcard.Card, money.Money) error { return nil }

// AlwaysDeclines stubs creditcard.Service and simulates declined charges.
type AlwaysDeclines struct{}

func (AlwaysDeclines) Charge(*creditcard.Card, money.Money) error {
    return creditcard.ErrDeclined
}
```

多种类型的多重替身

但现在假设 `package creditcard` 包含多个值得创建替身的类型，如下面的 `Service` 和 `StoredValue`：

```
package creditcard

type Service struct {
    // omitted
}

type Card struct {
    // omitted
}

// StoredValue manages customer credit balances.  This applies when returned
// merchandise is credited to a customer's local account instead of processed
// by the credit issuer.  For this reason, it is implemented as a separate
// service.
type StoredValue struct {
    // omitted
}

func (s *StoredValue) Credit(c *Card, amount money.Money) error { /* omitted */ }
```

In this case, more explicit test double naming is sensible:

在这种情况下，更明确的测试替身命名是明智的：

```
// Good:
type StubService struct{}

func (StubService) Charge(*creditcard.Card, money.Money) error { return nil }

type StubStoredValue struct{}

func (StubStoredValue) Credit(*creditcard.Card, money.Money) error { return nil }
```

测试中的局部变量

当你的测试中的变量引用替身时，要根据上下文选择一个能最清楚地区分替身和其他生产类型的名称。考虑一下你要测试的一些生产代码：

```
package payment

import (
    "path/to/creditcard"
    "path/to/money"
)

type CreditCard interface {
    Charge(*creditcard.Card, money.Money) error
}

type Processor struct {
    CC CreditCard
}

var ErrBadInstrument = errors.New("payment: instrument is invalid or expired")

func (p *Processor) Process(c *creditcard.Card, amount money.Money) error {
    if c.Expired() {
        return ErrBadInstrument
    }
    return p.CC.Charge(c, amount)
}
```

在测试中，一个被称为“间谍“的 `CreditCard` 的测试替身与生产类型并列，所以给名字加前缀可以提高清晰度。

```
// Good:
package payment

import "path/to/creditcardtest"

func TestProcessor(t *testing.T) {
    var spyCC creditcardtest.Spy

    proc := &Processor{CC: spyCC}

    // declarations omitted: card and amount
    if err := proc.Process(card, amount); err != nil {
        t.Errorf("proc.Process(card, amount) = %v, want %v", got, want)
    }

    charges := []creditcardtest.Charge{
        {Card: card, Amount: amount},
    }

    if got, want := spyCC.Charges, charges; !cmp.Equal(got, want) {
        t.Errorf("spyCC.Charges = %v, want %v", got, want)
    }
}
```

这比没有前缀的名字更清楚。

```go
// Bad:
package payment

import "path/to/creditcardtest"

func TestProcessor(t *testing.T) {
    var cc creditcardtest.Spy

    proc := &Processor{CC: cc}

    // declarations omitted: card and amount
    if err := proc.Process(card, amount); err != nil {
        t.Errorf("proc.Process(card, amount) = %v, want %v", got, want)
    }

    charges := []creditcardtest.Charge{
        {Card: card, Amount: amount},
    }

    if got, want := cc.Charges, charges; !cmp.Equal(got, want) {
        t.Errorf("cc.Charges = %v, want %v", got, want)
    }
}
```

阴影

注意：本解释使用了两个非正式的术语，*stomping* 和 *shadowing*。它们不是 Go 语言规范中的正式概念。像许多编程语言一样，Go 有可变的变量：对一个变量的赋值会改变其值。

```go
// Good:
func abs(i int) int {
    if i < 0 {
        i *= -1
    }
    return i
}
```

当使用短变量声明与 `:=` 操作符时，在某些情况下不会创建一个新的变量。我们可以把这称为 *stomping*。当不再需要原来的值时，这样做是可以的。

```
// Good:
// innerHandler is a helper for some request handler, which itself issues
// requests to other backends.
func (s *Server) innerHandler(ctx context.Context, req *pb.MyRequest) *pb.MyResponse {
    // Unconditionally cap the deadline for this part of request handling.
    ctx, cancel := context.WithTimeout(ctx, 3*time.Second)
    defer cancel()
    ctxlog.Info("Capped deadline in inner request")

    // Code here no longer has access to the original context.
    // This is good style if when first writing this, you anticipate
    // that even as the code grows, no operation legitimately should
    // use the (possibly unbounded) original context that the caller provided.

    // ...
}
```

不过要小心在新的作用域中使用短的变量声明 `:` 这将引入一个新的变量。我们可以把这称为对原始变量的“阴影”。块结束后的代码指的是原来的。这里是一个有条件缩短期限的错误尝试:

```
// Bad:
func (s *Server) innerHandler(ctx context.Context, req *pb.MyRequest) *pb.MyResponse {
    // Attempt to conditionally cap the deadline.
    if *shortenDeadlines {
        ctx, cancel := context.WithTimeout(ctx, 3*time.Second)
        defer cancel()
        ctxlog.Info(ctx, "Capped deadline in inner request")
    }

    // BUG: "ctx" here again means the context that the caller provided.
    // The above buggy code compiled because both ctx and cancel
    // were used inside the if statement.

    // ...
}
```

正确版本的代码应该是:

```
// Good:
func (s *Server) innerHandler(ctx context.Context, req *pb.MyRequest) *pb.MyResponse {
    if *shortenDeadlines {
        var cancel func()
        // Note the use of simple assignment, = and not :=.
        ctx, cancel = context.WithTimeout(ctx, 3*time.Second)
        defer cancel()
        ctxlog.Info(ctx, "Capped deadline in inner request")
    }
    // ...
}
```

在我们称之为 stomping 的情况下，因为没有新的变量，所以被分配的类型必须与原始变量的类型相匹配。有了 shadowing，一个全新的实体被引入，所以它可以有不同的类型。有意的影子可以是一种有用的做法，但如果能提高清晰度，你总是可以使用一个新的名字。

除了非常小的范围之外，使用与标准包同名的变量并不是一个好主意，因为这使得该包的自由函数和值无法访问。相反，当为你的包挑选名字时，要避免使用那些可能需要导入重命名或在客户端造成阴影的其他好的变量名称。

```
// Bad:
func LongFunction() {
    url := "https://example.com/"
    // Oops, now we can't use net/url in code below.
}
```

工具包

Go 包在 `package` 声明中指定了一个名称，与导入路径分开。包的名称比路径更重要，因为它的可读性。

Go 包的名字应该是与包所提供的内容相关。将包命名为 `util`、`helper`、`common` 或类似的名字通常是一个糟糕的选择（但它可以作为名字的一部分）。没有信息的名字会使代码更难读，而且如果使用的范围太广，很容易造成不必要的导入冲突。

相反，考虑一下调用站会是什么样子。

```
// Good:
db := spannertest.NewDatabaseFromFile(...)

_, err := f.Seek(0, io.SeekStart)

b := elliptic.Marshal(curve, x, y)
```

即使不知道导入列表，你也能大致知道这些东西的作用（`cloud.google.com/go/spanner/spannertest`，`io`，和 `crypto/elliptic`）。如果名称不那么集中，这些可能是：

```
// Bad:
db := test.NewDatabaseFromFile(...)

_, err := f.Seek(0, common.SeekStart)

b := helper.Marshal(curve, x, y)
```

包大小

如果你在问自己，你的 Go 包应该有多大，是把相关的类型放在同一个包里，还是把它们分成不同的包，可以从关于包名的 Go 博文开始。尽管帖子的标题是这样的，但它并不仅仅是关于命名的。它包含了一些有用的提示并引用了一些有用的文章和讲座。

这里有一些其他的考虑和说明。

用户在一页中看到包的 godoc，任何由包提供的类型导出的方法都按其类型分组。Godoc 还将构造函数与它们返回的类型一起分组。如果_客户端代码_可能需要两个不同类型的值来相互作用，那么把它们放在同一个包里对用户来说可能很方便。

包内的代码可以访问包内未导出的标识符。如果你有几个相关的类型，它们的_实现_是紧密耦合的，把它们放在同一个包里可以让你实现这种耦合，而无需用这些细节污染公共 API。

综上所述，把你的整个项目放在一个包里可能会使这个包变得太大。当一个东西在概念上是不同的，给它一个自己的小包可以使它更容易使用。客户端所知道的包的短名称和导出的类型名称一起构成了一个有意义的标识符：例如 `bytes.Buffer`，`ring.New`。博文有更多的例子。

Go 风格在文件大小方面很灵活，因为维护者可以将包内的代码从一个文件移到另一个文件而不影响调用者。但作为一般准则：通常情况下，一个文件有几千行，或者有许多小文件，都不是一个好主意。没有像其他一些语言那样的 “一个类型，一个文件 “的约定。作为一个经验法则，文件应该足够集中，以便维护者可以知道哪个文件包含了什么东西，而且文件应该足够小，以便一旦有了这些东西，就很容易找到。标准库经常将大型包分割成几个源文件，将相关的代码按文件分组。`bytes` 包的源代码就是一个很好的例子。有很长包文档的包可以选择专门的一个文件，称为 `doc.go`，其中有包文档，包声明，而没有其他内容，但这不是必须的。

在 Google 代码库和使用 Bazel 的项目中，Go 代码的目录布局与开源 Go 项目不同：你可以在一个目录中拥有多个 `go_library` 目标。如果你期望在未来将你的项目开源，那么给每个包提供自己的目录是一个很好的理由。

另见：

- 测试替身包

导入

Protos and stubs

由于其跨语言的特性，Proto 库导入的处理方式与标准 Go 导入不同。重命名的 proto 导入的约定是基于生成包的规则：

- `pb` 后缀一般用于 `go_proto_library` 规则。
- `grpc` 后缀一般用于 `go_grpc_library` 规则。

一般来说，使用简短的一个或两个字母的前缀：

```
// Good:
import (
    fspb "path/to/package/foo_service_go_proto"
    fsgrpc "path/to/package/foo_service_go_grpc"
)
```

如果一个包只使用一个 proto，或者该包与该 proto 紧密相连，那么前缀可以省略：

import (pb “path/to/package/foo_service_go_proto” grpc “path/to/package/foo_service_go_grpc”)

如果 proto 中的符号是通用的，或者没有很好的自我描述，或者用首字母缩写来缩短包的名称是不明确的，那么一个简短的词就可以作为前缀：

```
// Good:
import (
    mapspb "path/to/package/maps_go_proto"
)
```

在这种情况下，如果有关的代码还没有明确与地图相关，那么 `mapspb.Address` 可能比 `mpb.Address` 更清楚。

导入顺序

导入通常按顺序分为以下两（或更多）块。

1. 标准库导入（例如，`"fmt"`）。
2. 导入（例如，"/path/to/somelib"）。
3. （可选）Protobuf 导入（例如，`fpb "path/to/foo_go_proto"`）。
4. （可选）无使用导入（例如，`_ "path/to/package"`）

如果一个文件没有上述可选类别组，相关的导入将包含在项目倒入组中。

任何清晰和容易理解的导入分组通常都是好的。例如，一个团队可以选择将 gRPC 导入与 protobuf 导入分开分组。

> **注意：** 对于只维护两个强制组的代码 (一个组用于标准库，一个组用于所有其他的导入)，`goimports` 工具产生的输出与这个指南一致。 然而，`goimports` 并不了解强制性组以外的组；可选组很容易被该工具废止。当使用可选的组别时，作者和审稿人都需要注意，以确保组别符合要求。 两种方法都可以，但不要让进口部分处于不一致的、部分分组的状态。

错误处理

在 Go 中，错误就是价值；它们由代码创造，也由代码消耗。错误可以是：

- 转化为诊断信息，显示给程序员看
- 由维护者使用
- 向终端用户解释

错误信息也显示在各种不同的渠道，包括日志信息、错误转储和渲染的 UI。

处理（产生或消耗）错误的代码应该刻意这样做。忽略或盲目地传播错误的返回值可能是很诱人的。然而，值得注意的是，调用框架中的当前函数是否被定位为最有效地处理该错误。这是一个很大的话题，很难给出明确的建议。请使用你自己的判断，但要记住以下的考虑。

- 当创建一个错误值时，决定是否给它任何结构。
- 当处理一个错误时，考虑添加信息，这些信息你有，但调用者和/或被调用者可能没有。
- 也请参见关于错误记录的指导。

虽然忽略一个错误通常是不合适的，但一个合理的例外是在协调相关操作时，通常只有第一个错误是有用的。包 `errgroup` 为一组操作提供了一个方便的抽象，这些操作都可以作为一个组失败或被取消。

也请参见：

- Effective Go on errors
- Go 博客关于错误的文章
- `errors` 包
- `upspin.io/errors` 包
- GoTip #89: 何时使用规范的状态码作为错误?
- GoTip #48: 错误的哨兵值
- GoTip #13: 设计用于检查的错误

错误结构

如果调用者需要询问错误（例如，区分不同的错误条件），请给出错误值结构，这样可以通过编程完成，而不是让调用者进行字符串匹配。这个建议适用于生产代码，也适用于关心不同错误条件的测试。

最简单的结构化错误是无参数的全局值。

```
type Animal string

var (
    // ErrDuplicate occurs if this animal has already been seen.
    ErrDuplicate = errors.New("duplicate")

    // ErrMarsupial occurs because we're allergic to marsupials outside Australia.
    // Sorry.
    ErrMarsupial = errors.New("marsupials are not supported")
)

func pet(animal Animal) error {
    switch {
    case seen[animal]:
        return ErrDuplicate
    case marsupial(animal):
        return ErrMarsupial
    }
    seen[animal] = true
    // ...
    return nil
}
```

调用者可以简单地将函数返回的错误值与已知的错误值之一进行比较：

```
// Good:
func handlePet(...) {
    switch err := process(an); err {
    case ErrDuplicate:
        return fmt.Errorf("feed %q: %v", an, err)
    case ErrMarsupial:
        // Try to recover with a friend instead.
        alternate = an.BackupAnimal()
        return handlePet(..., alternate, ...)
    }
}
```

上面使用了哨兵值，其中误差必须等于（在`==`的意义上）预期值。这在很多情况下是完全足够的。如果`process`返回包装好的错误（下面讨论），你可以使用`errors.Is`。

```
// Good:
func handlePet(...) {
    switch err := process(an); {
    case errors.Is(err, ErrDuplicate):
        return fmt.Errorf("feed %q: %v", an, err)
    case errors.Is(err, ErrMarsupial):
        // ...
    }
}
```

不要试图根据字符串的形式来区分错误。(参见 Go Tip #13: 设计用于检查的错误以了解更多信息）。

```
// Bad:
func handlePet(...) {
    err := process(an)
    if regexp.MatchString(`duplicate`, err.Error()) {...}
    if regexp.MatchString(`marsupial`, err.Error()) {...}
}
```

如果错误中有调用者需要的额外信息，最好是以结构化方式呈现。例如，`os.PathError` 类型的记录是将失败操作的路径名，放在调用者可以轻松访问的结构域中。

其他错误结构可以酌情使用，例如一个包含错误代码和细节字符串的项目结构。Package `status` 是一种常见的封装方式；如果你选择这种方式（你没有义务这么做），请使用 规范错误码。参见 Go Tip #89: 何时使用规范的状态码作为错误 以了解使用状态码是否是正确的选择。

为错误添加信息

任何返回错误的函数都应该努力使错误值变得有用。通常情况下，该函数处于一个调用链的中间，并且只是在传播它所调用的其他函数的错误（甚至可能来自另一个包）。这里有机会用额外的信息来注解错误，但程序员应该确保错误中有足够的信息，而不添加重复的或不相关的细节。如果你不确定，可以尝试在开发过程中触发错误条件：这是一个很好的方法来评估错误的观察者（无论是人类还是代码）最终会得到什么。

习惯和良好的文档有帮助。例如，标准包`os`宣传其错误包含路径信息，当它可用时。这是一种有用的风格，因为得到错误的调用者不需要用他们已经提供了失败的函数的信息来注释它。

```
// Good:
if err := os.Open("settings.txt"); err != nil {
    return err
}

// Output:
//
// open settings.txt: no such file or directory
```

如果对错误的_意义_有什么有趣的说法，当然可以加入。只需考虑调用链的哪一层最适合理解这个含义。

```
// Good:
if err := os.Open("settings.txt"); err != nil {
    // We convey the significance of this error to us. Note that the current
    // function might perform more than one file operation that can fail, so
    // these annotations can also serve to disambiguate to the caller what went
    // wrong.
    return fmt.Errorf("launch codes unavailable: %v", err)
}

// Output:
//
// launch codes unavailable: open settings.txt: no such file or directory
```

与这里的冗余信息形成鲜明对比：

```
// Bad:
if err := os.Open("settings.txt"); err != nil {
    return fmt.Errorf("could not open settings.txt: %w", err)
}

// Output:
//
// could not open settings.txt: open settings.txt: no such file or directory
```

当添加信息到一个传播的错误时，你可以包裹错误或提出一个新的错误。用 `fmt.Errorf` 中的 `%w` 动词来包装错误，允许调用者访问原始错误的数据。这在某些时候是非常有用的，但在其他情况下，这些细节对调用者来说是误导或不感兴趣的。更多信息请参见关于错误包装的博文。包裹错误也以一种不明显的方式扩展了你的包的 API 表面，如果你改变了你的包的实现细节，这可能会导致破坏。

最好避免使用 `%w`，除非你也记录（并有测试来验证）你所暴露的基本错误。如果你不期望你的调用者调用 `errors.Unwrap`，`errors.Is` 等等，就不要费心使用 `%w`。

同样的概念适用于结构化错误，如 `*status.Status`（见规范错误码）。例如，如果你的服务器向后端发送畸形的请求，并收到一个 `InvalidArgument` 错误码，这个代码不应该传播给客户端，假设客户端没有做错。相反，应该向客户端返回一个 `内部` 的规范码。

然而，注解错误有助于自动日志系统保留错误的状态有效载荷。例如，在一个内部函数中注释错误是合适的：

```
// Good:
func (s *Server) internalFunction(ctx context.Context) error {
    // ...
    if err != nil {
        return fmt.Errorf("couldn't find remote file: %w", err)
    }
}
```

直接位于系统边界的代码（通常是RPC、IPC、存储等之类的）应该使用规范的错误空间报告错误。这里的代码有责任处理特定领域的错误，并以规范的方式表示它们。比如说：

```
// Bad:
func (*FortuneTeller) SuggestFortune(context.Context, *pb.SuggestionRequest)
(*pb.SuggestionResponse, error) {
    // ...
    if err != nil {
        return nil, fmt.Errorf("couldn't find remote file: %w", err)
    }
}
```

```
// Good:
import (
    "google.golang.org/grpc/codes"
    "google.golang.org/grpc/status"
)
func (*FortuneTeller) SuggestFortune(context.Context, *pb.SuggestionRequest)
(*pb.SuggestionResponse, error) {
    // ...
    if err != nil {
        // Or use fmt.Errorf with the %w verb if deliberately wrapping an
        // error which the caller is meant to unwrap.
        return nil, status.Errorf(codes.Internal, "couldn't find fortune database",
status.ErrInternal)
    }
}
```

错误中的 %w 的位置

倾向于将`%w`放在错误字符串的末尾。

错误可以用`%w`动词来包装，或者把它们放在一个实现`Unwrap()`错误的结构化错误中（例如：`fs.PathError`）。

被包裹的错误形成错误链：每一层新的包裹都会在错误链的前面增加一个新的条目。错误链可以用`Unwrap()error`方法进行遍历。比如说：

```
err1 := fmt.Errorf("err1")
err2 := fmt.Errorf("err2: %w", err1)
err3 := fmt.Errorf("err3: %w", err2)
```

这就形成了一个错误链的形式，

```
flowchart LR
  err3 == err3 wraps err2 ==> err2;
  err2 == err2 wraps err1 ==> err1;
```

不管 `%w` 动词放在哪里，返回的错误总是代表错误链的前面，而 `%w` 是下一个子节点。同样，`Unwrap()error` 总是从最新的错误到最旧的错误穿越错误链。

然而，`%w` 动词的位置会影响错误链是从最新到最旧，从最旧到最新，还是两者都不影响：

```
// Good:
err1 := fmt.Errorf("err1")
err2 := fmt.Errorf("err2: %w", err1)
err3 := fmt.Errorf("err3: %w", err2)
fmt.Println(err3) // err3: err2: err1
// err3 is a newest-to-oldest error chain, that prints newest-to-oldest.
```

```
// Bad:
err1 := fmt.Errorf("err1")
err2 := fmt.Errorf("%w: err2", err1)
err3 := fmt.Errorf("%w: err3", err2)
fmt.Println(err3) // err1: err2: err3
// err3 is a newest-to-oldest error chain, that prints oldest-to-newest.
```

```
// Bad:
err1 := fmt.Errorf("err1")
err2 := fmt.Errorf("err2-1 %w err2-2", err1)
err3 := fmt.Errorf("err3-1 %w err3-2", err2)
fmt.Println(err3) // err3-1 err2-1 err1 err2-2 err3-2
// err3 is a newest-to-oldest error chain, that neither prints newest-to-oldest
// nor oldest-to-newest.
```

因此，为了使错误文本反映错误链结构，最好将 `%w` 动词放在最后，形式为 `[...]: %w`。

错误日志

函数有时需要告诉外部系统一个错误，而不把它传播给其调用者。在这里，日志是一个明显的选择；但要注意记录错误的内容和方式。

- 像好的测试失败信息一样，日志信息应该清楚地表达出错的原因，并通过包括相关信息来帮助维护者诊断问题。
- 避免重复。如果你返回一个错误，通常最好不要自己记录，而是让调用者处理。调用者可以选择记录错误，也可以使用 `rate.sometimes` 限制记录的速度。其他选择包括尝试恢复，甚至是停止程序。在任何情况下，让调用者控制有助于避免日志垃圾。 然而，这种方法的缺点是，任何日志都是用调用者的行座标写的。
- 对 PII 要小心。许多日志汇不是敏感的终端用户信息的合适目的地。
- 尽量少使用 `log.Error`。ERROR 级别的日志会导致刷新，比低级别的日志更昂贵。这可能会对你的代码产生严重的性能影响。当决定错误级别还是警告级别时，考虑最佳实践，即错误级别的消息应该是可操作的，而不是比警告“更严重”。
- 在谷歌内部，我们有监控系统，可以设置更有效的警报，而不是写到日志文件，希望有人注意到它。这与标准库 `expvar` 包类似，但不完全相同。

自定义日志级别

使用日志分级（`log.V`）对你有利。分级的日志对开发和追踪很有用。建立一个关于粗略程度的约定是有帮助的。比如说。

- 在 `V(1)` 写少量的额外信息
- 在 `V(2)` 中跟踪更多信息
- 在 `V(3)` 中倾倒大量的内部状态。

为了尽量减少粗略记录的成本，你应该确保即使在 `log.V` 关闭的情况下也不要意外地调用昂贵的函数。`log.V` 提供两个 API。更方便的那个带有这种意外支出的风险。如有疑问，请使用稍显粗略的风格。

```
// Good:
for _, sql := range queries {
  log.V(1).Infof("Handling %v", sql)
  if log.V(2) {
    log.Infof("Handling %v", sql.Explain())
  }
  sql.Run(...)
}
```

```
// Bad:
// sql.Explain called even when this log is not printed.
log.V(2).Infof("Handling %v", sql.Explain())
```

程序初始化

程序的初始化错误（如坏的标志和配置）应该向上传播到 `main`，它应该调用 `log.Exit`，并说明如何修复错误。在这些情况下，一般不应使用 `log.Fatal`，因为指向检查的堆栈跟踪不可能像人工生成的可操作信息那样有用。

程序检查和 panic

正如反对 panic 的决定中所述，标准的错误处理应该围绕错误返回值进行结构化。库应该倾向于向调用者返回错误，而不是中止程序，特别是对于瞬时错误。

偶尔有必要对一个不变量进行一致性检查，如果违反了这个不变量，就终止程序。一般来说，只有当不变量检查失败意味着内部状态已经无法恢复时，才会这样做。在谷歌代码库中，最可靠的方法是调用 `log.Fatal`。在这种情况下使用 `panic` 是不可靠的，因为延迟函数有可能会出现死锁或进一步破坏内部或外部状态。

同样，抵制恢复 panic 以避免崩溃的诱惑，因为这样做可能导致传播损坏的状态。你离 panic 越远，你对程序的状态就越不了解，它可能持有锁或其他资源。然后，程序可以发展出其他意想不到的故障模式，使问题更加难以诊断。与其试图在代码中处理意外的 panic，不如使用监控工具来浮现出意外的故障，并高度优先修复相关的错误。

注意：标准的 `net/http` 服务器违反了这个建议，从请求处理程序中恢复 panic。有经验的 Go 工程师的共识是，这是一个历史性的错误。如果你对其他语言的应用服务器的日志进行采样，通常会发现有大量的堆栈轨迹没有被处理。在你的服务器中应避免这种陷阱。

何时 panic

标准库对 API 的误用感到 panic。例如，`reflect` 在许多情况下，如果一个值的访问方式表明它被误读，就会发出 panic。这类似于对核心语言错误的 panic，如访问一个越界的 slice 元素。代码审查和测试应该发现这样的错误，这些错误预计不会出现在生产代码中。这些 panic 作为不依赖库的不变性检查，因为标准库不能访问谷歌代码库使用的 levelled `log` 包。

另一种情况下，panic 可能是有用的，尽管不常见，是作为一个包的内部实现细节，在调用链中总是有一个匹配的恢复。解析器和类似的深度嵌套、紧密耦合的内部函数组可以从这种设计中受益，其中管道错误返回增加了复杂性而没有价值。这种设计的关键属性是，这些 panic 永远不允许跨越包的边界，不构成包的 API 的一部分。这通常是通过一个顶层的延迟恢复来实现的，它将传播的 panic 转化为公共 API 表面的返回错误。

当编译器无法识别不可到达的代码时，例如使用像 `log.Fatal` 这样不会返回的函数时，也会使用 Panic：

```
// Good:
func answer(i int) string {
    switch i {
    case 42:
        return "yup"
    case 54:
        return "base 13, huh"
    default:
        log.Fatalf("Sorry, %d is not the answer.", i)
        panic("unreachable")
    }
}
```

不要在标志被解析之前调用 `log` 函数。如果你必须在 `init` 函数中死亡，可以接受用 panic 来代替日志调用。

文档

公约

本节是对决策文件的注释部分的补充。 以熟悉的风格记录的 Go 代码比那些错误记录或根本没有记录的代码更容易阅读，更不容易被误用。可运行的实例会出现在 Godoc 和代码搜索中，是解释如何使用你的代码的绝佳方式。

参数和配置

不是每个参数都必须在文档中列举出来。这适用于

- 函数和方法参数
- 结构字段
- API 的选项

将易出错或不明显的字段和参数记录下来，说说它们为什么有趣。

在下面的片段中，突出显示的注释对读者来说没有增加什么有用的信息：

```
// Bad:
// Sprintf formats according to a format specifier and returns the resulting
// string.
//
// format is the format, and data is the interpolation data.
func Sprintf(format string, data ...interface{}) string
```

然而，这个片段展示了一个与之前类似的代码场景，其中评论反而说明了一些不明显或对读者有实质性帮助的东西：

```
// Good:
// Sprintf formats according to a format specifier and returns the resulting
// string.
//
// The provided data is used to interpolate the format string. If the data does
// not match the expected format verbs or the amount of data does not satisfy
// the format specification, the function will inline warnings about formatting
// errors into the output string as described by the Format errors section
// above.
func Sprintf(format string, data ...interface{}) string
```

在选择文档的内容和深度时，要考虑到你可能的受众。维护者、新加入团队的人、外部用户，甚至是六个月后的你，可能会与你第一次来写文档时的想法略有不同的信息。

也请参见。

- GoTip #41: 识别函数调用参数
- GoTip #51: 配置的模式

上下文

这意味着取消一个上下文参数会中断提供给它的函数。如果该函数可以返回一个错误，习惯上是 `ctx.Err()`。

这个事实不需要重述：

```
// Bad:
// Run executes the worker's run loop.
//
// The method will process work until the context is cancelled and accordingly
// returns an error.
func (Worker) Run(ctx context.Context) error
```

因为这句话是隐含的，所以下面的说法更好：

```
// Good:
// Run executes the worker's run loop.
func (Worker) Run(ctx context.Context) error
```

如果上下文行为是不同的或不明显的，应该明确地记录下来：

- 如果函数在取消上下文时返回一个除 `ctx.Err()` 以外的错误：

```
// Good:
// Run executes the worker's run loop.
//
// If the context is cancelled, Run returns a nil error.
func (Worker) Run(ctx context.Context) error
```

- 如果该功能有其他机制，可能会中断它或影响其生命周期：

```
// Good:
// Run executes the worker's run loop.
//
// Run processes work until the context is cancelled or Stop is called.
// Context cancellation is handled asynchronously internally: run may return
// before all work has stopped. The Stop method is synchronous and waits
// until all operations from the run loop finish. Use Stop for graceful
// shutdown.
func (Worker) Run(ctx context.Context) error

func (Worker) Stop()
```

- 如果该函数对上下文的生命周期、脉络或附加值有特殊期望：

```
// Good:
// NewReceiver starts receiving messages sent to the specified queue.
// The context should not have a deadline.
func NewReceiver(ctx context.Context) *Receiver

// Principal returns a human-readable name of the party who made the call.
// The context must have a value attached to it from security.NewContext.
func Principal(ctx context.Context) (name string, ok bool)
```

警告：避免设计对其调用者提出这种要求（比如上下文没有截止日期）的API。以上只是一个例子，说明在无法避免的情况下该如何记录，而不是对该模式的认可。

并发

Go 用户认为概念上的只读操作对于并发使用是安全的，不需要额外的同步。

在这个 Godoc 中，关于并发性的额外说明可以安全地删除：

```
// Len returns the number of bytes of the unread portion of the buffer;
// b.Len() == len(b.Bytes()).
//
// It is safe to be called concurrently by multiple goroutines.
func (*Buffer) Len() int
```

然而，变异操作并不被认为对并发使用是安全的，需要用户考虑同步化。 同样地，这里可以安全地删除关于并发的额外注释：

```
// Grow grows the buffer's capacity.
//
// It is not safe to be called concurrently by multiple goroutines.
func (*Buffer) Grow(n int)
```

强烈鼓励在以下情况下提供文档：

- 目前还不清楚该操作是只读的还是变异的。

```
// Good:
package lrucache

// Lookup returns the data associated with the key from the cache.
//
// This operation is not safe for concurrent use.
func (*Cache) Lookup(key string) (data []byte, ok bool)
```

  为什么？在查找密钥时，缓存命中会在内部突变一个 LRU 缓存。这一点是如何实现的，对所有的读者来说可能并不明显。

- 同步是由 API 提供的

```
// Good:
package fortune_go_proto

// NewFortuneTellerClient returns an *rpc.Client for the FortuneTeller service.
// It is safe for simultaneous use by multiple goroutines.
func NewFortuneTellerClient(cc *rpc.ClientConn) *FortuneTellerClient
```

  为什么？Stubby 提供了同步性。

  注意：如果 API 是一个类型，并且 API 完整地提供了同步，传统上只有类型定义记录了语义。

- 该 API 消费用户实现的接口类型，并且该接口的消费者有特殊的并发性要求：

```
// Good:
package health

// A Watcher reports the health of some entity (usually a backen service).
//
// Watcher methods are safe for simultaneous use by multiple goroutines.
type Watcher interface {
    // Watch sends true on the passed-in channel when the Watcher's
    // status has changed.
    Watch(changed chan<- bool) (unwatch func())

    // Health returns nil if the entity being watched is healthy, or a
    // non-nil error explaining why the entity is not healthy.
    Health() error
}
```

  为什么？一个 API 是否能被多个 goroutines 安全使用是其契约的一部分。

清理

记录 API 的任何明确的清理要求。否则，调用者不会正确使用 API，导致资源泄漏和其他可能的错误。

调出由调用者决定的清理工作：

```
// Good:
// NewTicker returns a new Ticker containing a channel that will send the
// current time on the channel after each tick.
//
// Call Stop to release the Ticker's associated resources when done.
func NewTicker(d Duration) *Ticker

func (*Ticker) Stop()
```

如果有可能不清楚如何清理资源，请解释如何清理：

```
// Good:
// Get issues a GET to the specified URL.
//
// When err is nil, resp always contains a non-nil resp.Body.
// Caller should close resp.Body when done reading from it.
//
//    resp, err := http.Get("http://example.com/")
//    if err != nil {
//        // handle error
//    }
//    defer resp.Body.Close()
//    body, err := io.ReadAll(resp.Body)
func (c *Client) Get(url string) (resp *Response, err error)
```

预览

Go 的特点是有一个文档服务器。建议在代码审查前和审查过程中预览你的代码产生的文档。这有助于验证 godoc 格式化是否正确呈现。

Godoc 格式化

Godoc为格式化文档提供了一些特定的语法。

- 段落之间需要有一个空行：

  ```
  // Good:
  // LoadConfig reads a configuration out of the named file.
  //
  // See some/shortlink for config file format details.
  ```

- 测试文件可以包含可运行的例子，这些例子出现在 godoc 中相应的文档后面：

```
// Good:
func ExampleConfig_WriteTo() {
  cfg := &Config{
    Name: "example",
  }
  if err := cfg.WriteTo(os.Stdout); err != nil {
    log.Exitf("Failed to write config: %s", err)
  }
  // Output:
  // {
  //   "name": "example"
  // }
}
```

- 缩进的行数再加上两个空格，就可以将它们逐字排开：

```
// Good:
// Update runs the function in an atomic transaction.
//
// This is typically used with an anonymous TransactionFunc:
//
//   if err := db.Update(func(state *State) { state.Foo = bar }); err != nil {
//     //...
//   }
```

然而，请注意，把代码放在可运行的例子中，而不是把它放在注释中，往往会更合适。 这种逐字格式化可以用于非godoc原生的格式化，如列表和表格：

```
// Good:
// LoadConfig reads a configuration out of the named file.
//
// LoadConfig treats the following keys in special ways:
//   "import" will make this configuration inherit from the named file.
//   "env" if present will be populated with the system environment.
```

- 一行以大写字母开始，除括号和逗号外不含标点符号，后面是另一个段落，其格式为标题：

```
// Good:
// The following line is formatted as a heading.
//
// Using headings
//
// Headings come with autogenerated anchor tags for easy linking.
```

信号增强

有时一行代码看起来很普通，但实际上并不普通。这方面最好的例子之一是`err == nil`的检查（因为`err != nil`更常见）。下面的两个条件检查很难区分：

```
// Good:
if err := doSomething(); err != nil {
    // ...
}
```

```
// Bad:
if err := doSomething(); err == nil {
    // ...
}
```

你可以通过添加评论来“提高”条件的信号：

```
// Good:
if err := doSomething(); err == nil { // if NO error
    // ...
}
```

该评论提请注意条件的不同。

变量声明

初始化

为了保持一致性，当用非零值初始化一个新的变量时，首选`:=`而不是`var`。

```
// Good:
i := 42
// Bad:
var i = 42
```

非指针式零值

下面的声明使用零值：

```
// Good:
var (
    coords Point
    magic  [4]byte
    primes []int
)
```

当你想要传递一个空值**以供以后使用**时，你应该使用零值声明。使用带有显式初始化的复合字面会显得很笨重：

```
// Bad:
var (
    coords = Point{X: 0, Y: 0}
    magic  = [4]byte{0, 0, 0, 0}
    primes = []int(nil)
)
```

零值声明的一个常见应用是当使用一个变量作为反序列化时的输出：

```
// Good:
var coords Point
if err := json.Unmarshal(data, &coords); err != nil {
```

在你的结构体中，如果你需要一个不得复制的锁或其他字段，可以将其设为值类型以利用零值初始化。这确实意味着，现在必须通过指针而不是值来传递包含的类型。该类型的方法必须采用指针接收器。

```
// Good:
type Counter struct {
    // This field does not have to be "*sync.Mutex". However,
    // users must now pass *Counter objects between themselves, not Counter.
    mu   sync.Mutex
    data map[string]int64
}

// Note this must be a pointer receiver to prevent copying.
func (c *Counter) IncrementBy(name string, n int64)
```

对复合体（如结构体和数组）的局部变量使用值类型是可以接受的，即使它们包含这种不可复制的字段。然而，如果复合体是由函数返回的，或者如果对它的所有访问最终都需要获取一个地址，那么最好在一开始就将变量声明为指针类型。同样地，protobufs 也应该被声明为指针类型。

```
// Good:
func NewCounter(name string) *Counter {
    c := new(Counter) // "&Counter{}" is also fine.
    registerCounter(name, c)
    return c
}

var myMsg = new(pb.Bar) // or "&pb.Bar{}".
```

这是因为 `*pb.Something` 满足 `proto.Message` 而 `pb.Something` 不满足。

```
// Bad:
func NewCounter(name string) *Counter {
    var c Counter
    registerCounter(name, &c)
    return &c
}

var myMsg = pb.Bar{}
```

> **重要的是：** Map 类型在被修改之前必须明确地初始化。然而，从零值 Map 中读取是完全可以的。 对于 map 和 slice 类型，如果代码对性能特别敏感，并且你事先知道大小，请参见 size hints 部分。

复合字面量

以下是复合字面量的声明：

```
// Good:
var (
    coords   = Point{X: x, Y: y}
    magic    = [4]byte{'I', 'W', 'A', 'D'}
    primes   = []int{2, 3, 5, 7, 11}
    captains = map[string]string{"Kirk": "James Tiberius", "Picard": "Jean-Luc"}
)
```

当你知道初始元素或成员时，你应该使用复合字面量来声明一个值。

相比之下，与[零值初始化]相比，使用复合字面量声明空或无成员值可能会在视觉上产生噪音

当你需要一个指向零值的指针时，你有两个选择：空复合字面和 `new`。两者都很好，但是 `new` 关键字可以提醒读者，如果需要一个非零值，这个复合字面量将不起作用：

```
// Good:
var (
  buf = new(bytes.Buffer) // non-empty Buffers are initialized with constructors.
  msg = new(pb.Message) // non-empty proto messages are initialized with builders or by
setting fields one by one.
)
```

size 提示

以下是利用 size 提示来预分配容量的声明方式：

```
// Good:
var (
    // Preferred buffer size for target filesystem: st_blksize.
    buf = make([]byte, 131072)
    // Typically process up to 8-10 elements per run (16 is a safe assumption).
    q = make([]Node, 0, 16)
    // Each shard processes shardSize (typically 32000+) elements.
    seen = make(map[string]bool, shardSize)
)
```

根据对代码及其集成的经验分析，对创建性能敏感和资源高效的代码，size 提示和预分配是重要的步骤。

大多数代码不需要 size 提示或预分配，可以允许运行时根据需要增长 slice 或 map。当最终大小已知时，预分配是可以接受的（例如，在 slice 或 map 之间转换时），但这不是一个可读性要求，而且在少数情况下可能不值得这样做。

警告：预先分配比你需要的更多的内存，会在队列中浪费内存，甚至损害性能。如有疑问，请参阅 GoTip #3: Benchmarking Go Code 并默认为零初始化或复合字面量声明。

Channel 方向

尽可能地指定 Channel 方向。

```
// Good:
// sum computes the sum of all of the values. It reads from the channel until
// the channel is closed.
func sum(values <-chan int) int {
    // ...
}
```

这可以防止在没有规范的情况下可能出现的随意编码错误。

```
// Bad:
func sum(values chan int) (out int) {
    for v := range values {
        out += v
    }
    // values must already be closed for this code to be reachable, which means
    // a second close triggers a panic.
    close(values)
}
```

当方向被指定时，编译器会捕捉到像这样的简单错误。它还有助于向类型传达一种所有权的措施。 也请看 Bryan Mills 的演讲 “重新思考经典的并发模式”。PPT链接 视频链接。

函数参数列表

不要让一个函数的签名变得太长。当一个函数中的参数越多，单个参数的作用就越不明确，同一类型的相邻参数就越容易混淆。有大量参数的函数不容易被记住，在调用点也更难读懂。

在设计 API 时，可以考虑将一个签名越来越复杂的高配置函数分割成几个更简单的函数。如果有必要的话，这些函数可以共享一个（未导出的）实现。

当一个函数需要许多输入时，可以考虑为一些参数引入一个 option 模式，或者采用更高级的变体选项技术。选择哪种策略的主要考虑因素应该是函数调用在所有预期的使用情况下看起来如何。

下面的建议主要适用于导出的 API，它比未导出的 API 的标准要高。这些技术对于你的用例可能是不必要的。使用你的判断，并平衡清晰性和最小机制的原则。

也请参见。Go技巧#24：使用特定案例的结构

option 模式

option 模式是一种结构类型，它收集了一个函数或方法的部分或全部参数，然后作为最后一个参数传递给该函数或方法。(该结构只有在导出的函数中使用时，才应该导出)。

使用 option 模式有很多好处。

- 结构体字面量包括每个参数的字段和值，这使得它们可以自己作为文档，并且更难被交换。
- 不相关的或 “默认 “的字段可以被省略。
- 调用者可以共享 option 模式，并编写帮助程序对其进行操作。
- 与函数参数相比，结构体提供了更清晰的每个字段的文档。
- option 模式可以随着时间的推移而增长，而不会影响到调用点。

下面是一个可以改进的函数的例子：

```
// Bad:
func EnableReplication(ctx context.Context, config *replicator.Config, primaryRegions,
readonlyRegions []string, replicateExisting, overwritePolicies bool, replicationInterval
time.Duration, copyWorkers int, healthWatcher health.Watcher) {
    // ...
}
```

上面的函数可以用一个 option 模式重写如下：

```
// Good:
type ReplicationOptions struct {
    Config              *replicator.Config
    PrimaryRegions      []string
    ReadonlyRegions     []string
    ReplicateExisting   bool
    OverwritePolicies   bool
    ReplicationInterval time.Duration
    CopyWorkers         int
    HealthWatcher       health.Watcher
}

func EnableReplication(ctx context.Context, opts ReplicationOptions) {
    // ...
}
```

然后，该函数可以在不同的包中被调用：

```
// Good:
func foo(ctx context.Context) {
    // Complex call:
    storage.EnableReplication(ctx, storage.ReplicationOptions{
        Config:              config,
        PrimaryRegions:      []string{"us-east1", "us-central2", "us-west3"},
        ReadonlyRegions:     []string{"us-east5", "us-central6"},
        OverwritePolicies:   true,
        ReplicationInterval: 1 * time.Hour,
        CopyWorkers:         100,
        HealthWatcher:       watcher,
    })

    // Simple call:
    storage.EnableReplication(ctx, storage.ReplicationOptions{
        Config:         config,
        PrimaryRegions: []string{"us-east1", "us-central2", "us-west3"},
    })
}
```

注意：option 模式中从不包含上下文。

当遇到以下某些情况时，通常首选 option 模式：

- 所有调用者都需要指定一个或多个选项。

- 大量的调用者需要提供许多选项。
- 用户要调用的多个函数之间共享这些选项。

可变 option 模式

使用可变 option 模式，可以创建导出的函数，其返回的闭包可以传递给函数的variadic（`...`）参数。该函数将选项的值作为其参数（如果有的话），而返回的闭包接受一个可变的引用（通常是一个指向结构体类型的指针），该引用将根据输入进行更新。

使用可变 option 模式可以提供很多好处。

- 当不需要配置时，选项在调用点不占用空间。
- 选项仍然是值，所以调用者可以共享它们，编写帮助程序，并积累它们。
- 选项可以接受多个参数（例如：`cartesian.Translate(dx, dy int) TransformOption`）。
- 选项函数可以返回一个命名的类型，以便在 godoc 中把选项组合起来。
- 包可以允许（或阻止）第三方包定义（或不定义）自己的选项。

注意：使用可变 option 模式需要大量的额外代码（见下面的例子），所以只有在好处大于坏处时才可以使用。

下面是一个可以改进的功能的例子：

```
// Bad:
func EnableReplication(ctx context.Context, config *placer.Config, primaryCells,
readonlyCells []string, replicateExisting, overwritePolicies bool, replicationInterval
time.Duration, copyWorkers int, healthWatcher health.Watcher) {
  ...
}
```

上面的例子可以用可变 option 模式重写如下：

```
// Good:
type replicationOptions struct {
    readonlyCells       []string
    replicateExisting   bool
    overwritePolicies   bool
    replicationInterval time.Duration
    copyWorkers         int
    healthWatcher       health.Watcher
}

// A ReplicationOption configures EnableReplication.
type ReplicationOption func(*replicationOptions)

// ReadonlyCells adds additional cells that should additionally
// contain read-only replicas of the data.
//
// Passing this option multiple times will add additional
// read-only cells.
//
// Default: none
func ReadonlyCells(cells ...string) ReplicationOption {
    return func(opts *replicationOptions) {
        opts.readonlyCells = append(opts.readonlyCells, cells...)
    }
}

// ReplicateExisting controls whether files that already exist in the
// primary cells will be replicated.  Otherwise, only newly-added
// files will be candidates for replication.
//
// Passing this option again will overwrite earlier values.
//
// Default: false
func ReplicateExisting(enabled bool) ReplicationOption {
    return func(opts *replicationOptions) {
        opts.replicateExisting = enabled
    }
}

// ... other options ...

// DefaultReplicationOptions control the default values before
// applying options passed to EnableReplication.
var DefaultReplicationOptions = []ReplicationOption{
    OverwritePolicies(true),
    ReplicationInterval(12 * time.Hour),
    CopyWorkers(10),
}

func EnableReplication(ctx context.Context, config *placer.Config, primaryCells
[]string, opts ...ReplicationOption) {
    var options replicationOptions
    for _, opt := range DefaultReplicationOptions {
        opt(&options)
    }
    for _, opt := range opts {
        opt(&options)
    }
}
```

然后，该函数可以在不同的包中被调用：

```
// Good:
func foo(ctx context.Context) {
    // Complex call:
    storage.EnableReplication(ctx, config, []string{"po", "is", "ea"},
        storage.ReadonlyCells("ix", "gg"),
        storage.OverwritePolicies(true),
        storage.ReplicationInterval(1*time.Hour),
        storage.CopyWorkers(100),
        storage.HealthWatcher(watcher),
    )

    // Simple call:
    storage.EnableReplication(ctx, config, []string{"po", "is", "ea"})
}
```

当遇到很多以下情况时，首选可变 option 模式：

- 大多数调用者不需要指定任何选项。
- 大多数选项不经常使用。
- 有大量的选项。
- 选项需要参数。
- 选项可能会失败或设置错误（在这种情况下，选项函数会返回一个`错误'）。
- 选项需要大量的文档，在一个结构中很难容纳。
- 用户或其他软件包可以提供自定义选项。

这种风格的选项应该接受参数，而不是在命名中标识来表示它们的价值；后者会使参数的动态组合变得更加困难。例如，二进制设置应该接受一个布尔值（例如，`rpc.FailFast(enable bool)`比`rpc.EnableFailFast()`更合适）。枚举的选项应该接受一个枚举的常数（例如`log.Format(log.Capacitor)`比`log.CapacitorFormat()`更好）。另一种方法使那些必须以编程方式选择传递哪些选项的用户更加困难；这种用户被迫改变参数的实际组成，而不是改变参数到选项。不要假设所有的用户都会知道全部的选项。

一般来说，option 应该被按顺序处理。如果有冲突或者一个非累积的选项被多次传递，将应用最后一个参数。

在这种模式下，选项函数的参数通常是未导出的，以限制选项只在包本身内定义。这是一个很好的默认值，尽管有时允许其他包定义选项也是合适的。

参见 Rob Pike 的原始博文和 Dave Cheney 的演讲，以更深入地了解这些选项的使用方法。

复杂的命令行界面

一些程序希望为用户提供丰富的命令行界面，包括子命令。例如，`kubectl create`，`kubectl run`，以及许多其他的子命令都是由程序`kubectl`提供。至少有以下常用的库可以实现这个目的。

如果你没有偏好或者其他考虑因素相同，推荐使用 subcommands，因为它是最简单的，而且容易正确使用。然而，如果你需要它所不提供的不同功能，请挑选其他选项之一。

- **cobra**
  - 习惯标志：getopt
  - 在谷歌代码库之外很常见。
  - 许多额外的功能。
  - 使用中的陷阱（见下文）。
- **subcommands**
  - 习惯标志：Go
  - 简单且易于正确使用。
  - 如果你不需要额外的功能，推荐使用。

警告：cobra 命令函数应该使用 `cmd.Context()` 来获取上下文，而不是用 `context.Background` 来创建自己的根上下文。使用子命令包的代码已经将正确的上下文作为一个函数参数接收。

你不需要把每个子命令放在一个单独的包中，而且通常也没有必要这样做。应用与任何 Go 代码库相同的关于包边界的考虑。如果你的代码既可以作为库也可以作为二进制文件使用，通常将 CLI 代码和库分开是有好处的，使 CLI 只是其客户端中的一个。(这不是专门针对有子命令的CLI的，但在此提及，因为它经常出现。)

测试

把测试留给 `Test` 函数

Go 区分了 “测试辅助函数 “和 “断言辅助函数”。

- **测试辅助函数**就是做设置或清理任务的函数。所有发生在测试辅助函数中的故障都被认为是环境的故障（而不是来自被测试的代码）–例如，当一个测试数据库不能被启动，因为在这台机器上没有更多的空闲端口。对于这样的函数，调用 `t.Helper` 通常是合适的，将其标记为测试辅助函数。参见 测试辅助函数的错误处理 了解更多细节。
- **断言辅助函数**是检查系统正确性的函数，如果没有达到预期，则测试失败。断言辅助函数在 Go 中不被认为是常见用法。

测试的目的是报告被测试代码的通过/失败情况。测试失败的理想场所是在 `Test` 函数本身，因为这样可以确保失败信息和测试逻辑是清晰的。

随着你的测试代码的增长，可能有必要将一些功能分解到独立的函数中。标准的软件工程考虑仍然适用，因为_测试代码仍然是代码_。如果这些功能不与测试框架交互，那么所有的常规规则都适用。然而，当通用代码与框架交互时，必须注意避免常见的陷阱，这些陷阱会导致语焉不详的失败信息和不可维护的测试。

如果许多独立的测试用例需要相同的验证逻辑，请以下列方式之一安排测试，而不是使用断言辅助函数或复杂的验证函数。

- 在 `Test` 函数中内联逻辑（包括验证和失败），即使它是重复的。这在简单的情况下效果最好。
- 如果输入是类似的，可以考虑把它们统一到一个表格驱动的测试，同时在循环中保持逻辑的内联。这有助于避免重复，同时在 “测试 “中保持验证和失败。

- 如果有多个调用者需要相同的验证功能，但表格测试不适合（通常是因为输入不够简单或验证需要作为操作序列的一部分），安排验证功能，使其返回一个值（通常是一个“错误”），而不是接受一个“testing.T “参数并使用它来让测试失败。在`测试`中使用逻辑来决定是否失败，并提供有用的测试失败。你也可以创建测试辅助函数，以剔除常见的模板设置代码。

最后一点中概述的设计保持了正交性。例如，`cmp` 包 不是为了测试失败而设计的，而是为了比较（和差异）值。因此，它不需要知道进行比较的上下文，因为调用者可以提供这个。如果你的普通测试代码为你的数据类型提供了一个`cmp.Transformer`，这通常是最简单的设计。对于其他的验证，可以考虑返回一个`error`值。

```
// Good:
// polygonCmp returns a cmp.Option that equates s2 geometry objects up to
// some small floating-point error.
func polygonCmp() cmp.Option {
    return cmp.Options{
        cmp.Transformer("polygon", func(p *s2.Polygon) []*s2.Loop { return p.Loops() }),
        cmp.Transformer("loop", func(l *s2.Loop) []s2.Point { return l.Vertices() }),
        cmpopts.EquateApprox(0.00000001, 0),
        cmpopts.EquateEmpty(),
    }
}

func TestFenceposts(t *testing.T) {
    // This is a test for a fictional function, Fenceposts, which draws a fence
    // around some Place object. The details are not important, except that
    // the result is some object that has s2 geometry (github.com/golang/geo/s2)
    got := Fencepost(tomsDiner, 1*meter)
    if diff := cmp.Diff(want, got, polygonCmp()); diff != "" {
        t.Errorf("Fencepost(tomsDiner, 1m) returned unexpected diff (-want+got):
%v", diff)
    }
}

func FuzzFencepost(f *testing.F) {
    // Fuzz test (https://go.dev/doc/fuzz) for the same.

    f.Add(tomsDiner, 1*meter)
    f.Add(school, 3*meter)

    f.Fuzz(func(t *testing.T, geo Place, padding Length) {
        got := Fencepost(geo, padding)
        // Simple reference implementation: not used in prod, but easy to
        // reasonable and therefore useful to check against in random tests.
        reference := slowFencepost(geo, padding)

        // In the fuzz test, inputs and outputs can be large so don't
        // bother with printing a diff. cmp.Equal is enough.
        if !cmp.Equal(got, reference, polygonCmp()) {
            t.Errorf("Fencepost returned wrong placement")
        }
    })
}
```

`polygonCmp`函数对它的调用方式是不可知的；它不接受具体的输入类型，也不规定在两个对象不匹配的情况下该怎么做。因此，更多的调用者可以使用它。

注意：在测试辅助函数和普通库代码之间有一个类比。库中的代码通常应该不 panic，除非在极少数情况下；从测试中调用的代码不应该停止测试，除非继续进行没有意义。

设计可扩展的验证 API

风格指南中关于测试的大部分建议都是关于测试你自己的代码。本节是关于如何为其他人提供设施来测试他们编写的代码，以确保它符合你的库的要求。

验收测试

这种测试被称为验收测试。这种测试的前提是，使用测试的人不知道测试中发生的每一个细节；他们只是把输入交给测试机构来完成。这可以被认为是一种控制反转的形式。

在典型的Go测试中，测试函数控制着程序流程，无断言和测试函数指南鼓励你保持这种方式。本节解释了如何以符合 Go 风格的方式来编写对这些测试的支持。

在深入探讨如何做之前，请看下面摘录的`io/fs`中的一个例子：

```
type FS interface {
    Open(name string) (File, error)
}
```

虽然存在众所周知的`fs.FS`的实现，但 Go 开发者可能会被期望编写一个。为了帮助验证用户实现的`fs.FS`是否正确，在`testing/fstest`中提供了一个通用库，名为`fstest.TestFS`。这个 API 将实现作为一个黑箱来处理，以确保它维护了`io/fs`契约的最基本部分。

撰写验收测试

现在我们知道了什么是验收测试以及为什么要使用验收测试，让我们来探讨为`package chess`建立一个验收测试，这是一个用于模拟国际象棋游戏的包。`chess`的用户应该实现`chess.Player`接口。这些实现是我们要验证的主要内容。我们的验收测试关注的是棋手的实现是否走了合法的棋，而不是这些棋是否聪明。

1.为验证行为创建一个新的包，习惯上命名为，在包名后面加上 “test “一词（例如：`chesstest`）。

2.创建执行验证的函数，接受被测试的实现作为参数，并对其进行练习：

```
// ExercisePlayer tests a Player implementation in a single turn on a board.
// The board itself is spot checked for sensibility and correctness.
//
// It returns a nil error if the player makes a correct move in the context
// of the provided board. Otherwise ExercisePlayer returns one of this
// package's errors to indicate how and why the player failed the
// validation.
func ExercisePlayer(b *chess.Board, p chess.Player) error
```

测试应该注意哪些不变式被破坏，以及如何破坏。你的设计可以选择两种失败报告的原则：

- **快速失败**：一旦实现违反了一个不变式，就返回一个错误。

这是最简单的方法，如果预计验收测试会快速执行，那么它的效果很好。简单的错误 sentinels 和自定义类型在这里可以很容易地使用，反过来说，这使得测试验收测试变得很容易。

```
for color, army := range b.Armies {
    // The king should never leave the board, because the game ends at
    // checkmate.
    if army.King == nil {
        return &MissingPieceError{Color: color, Piece: chess.King}
    }
}
```

- **集合所有的失败**：收集所有的失败，并报告它们。 这种方法类似于 keep going 的指导，如果验收测试预计会执行得很慢，这种方法可能更可取。

  你如何聚集故障，应该由你是否想让用户或你自己有能力询问个别故障（例如，为你测试你的验收测试）来决定的。下面演示了使用一个自定义错误类型，聚合错误。

```
var badMoves []error

move := p.Move()
if putsOwnKingIntoCheck(b, move) {
    badMoves = append(badMoves, PutsSelfIntoCheckError{Move: move})
}

if len(badMoves) > 0 {
    return SimulationError{BadMoves: badMoves}
}
return nil
```

验收测试应该遵守 keep going 的指导，不调用 `t.Fatal`，除非测试检测到被测试系统中的不变量损坏。例如，`t.Fatal` 应该保留给特殊情况，如设置失败，像往常一样：

```
func ExerciseGame(t *testing.T, cfg *Config, p chess.Player) error {
    t.Helper()

    if cfg.Simulation == Modem {
        conn, err := modempool.Allocate()
        if err != nil {
            t.Fatalf("no modem for the opponent could be provisioned: %v", err)
        }
        t.Cleanup(func() { modempool.Return(conn) })
    }
    // Run acceptance test (a whole game).
}
```

这种技术可以帮助你创建简明、规范的验证。但不要试图用它来绕过断言指南。 最终产品应该以类似这样的形式提供给终端用户：

```
// Good:
package deepblue_test

import (
    "chesstest"
    "deepblue"
)

func TestAcceptance(t *testing.T) {
    player := deepblue.New()
    err := chesstest.ExerciseGame(t, chesstest.SimpleGame, player)
    if err != nil {
        t.Errorf("deepblue player failed acceptance test: %v", err)
    }
}
```

使用真正的传输工具

当测试组件集成时，特别是 HTTP 或 RPC 被用作组件之间的底层传输时，最好使用真正的底层传输来连接到测试版本的后端。

例如，假设你要测试的代码（有时被称为“被测系统“或SUT）与实现长期运行操作 API 的后端交互。为了测试你的 SUT，使用一个真正的 OperationsClient，它连接到 OperationsServer 的替身测试上。

为了确保测试代码贴切于生产环境，相对于使用手工实现的客户端，我们更加推荐使用生产的客户端和专用的测试服务器来模拟生产环境的复杂性。

提示：在可能的情况下，使用由被测服务的作者提供的测试库。

`t.Error` vs. `t.Fatal`

正如在执行策略中讨论的一样, 测试过程不应该在遇到问题的地方中断。

然而，有些情况需要终止当前测试。当某些测试需要标记失败时，调用t.Fatal是合适的，特别是在 使用测试辅助函数时，没有它，你就不能运行测试的其余部分。在表格驱动的测试中，t.Fatal适合于在测试在进入循环之前整个测试函数标记为失败状态。它只会影响整个测试列表中被标记为失败的测试函数不能继续向前推进, 而不会影响其他的测试函数, 所以, 错误报告应该像下面这样:

- 如果你不使用 `t.Run子` 测试，使用 `t.Error`，后面跟一个 `continue` 语句，继续到下一个测试项。
- 如果你使用子测试（并且你在调用 `t.Run` 时），使用 `t.Fatal`，结束当前子测试，并允许你的测试用例进入下一个子测试。

警告：调用和 `t.Fatal` 和类似函数并不总是安全的。更多细节在这里。

在测试辅助函数中处理错误

注意：本节讨论的测试辅助函数是 Go 使用的术语：这些函数用于准备测试环境和清理测试现场，而不是普通的断言设施。更多的讨论请参见 test functions 部分。

由测试辅助函数执行的操作有时会失败。例如，设置一个带有文件的目录涉及到 I/O，这可能会失败。当测试辅助函数失败时，它们的失败往往标志着测试不能继续，因为一个设置的前提条件失败了。当这种情况发生时，最好在辅助函数中调用一个 `Fatal` 函数：

```
// Good:
func mustAddGameAssets(t *testing.T, dir string) {
    t.Helper()
    if err := os.WriteFile(path.Join(dir, "pak0.pak"), pak0, 0644); err != nil {
        t.Fatalf("Setup failed: could not write pak0 asset: %v", err)
    }
    if err := os.WriteFile(path.Join(dir, "pak1.pak"), pak1, 0644); err != nil {
        t.Fatalf("Setup failed: could not write pak1 asset: %v", err)
    }
}
```

这就使调用测试辅助函数返回错误给测试本身更清晰：

```
// Bad:
func addGameAssets(t *testing.T, dir string) error {
    t.Helper()
    if err := os.WriteFile(path.Join(d, "pak0.pak"), pak0, 0644); err != nil {
        return err
    }
    if err := os.WriteFile(path.Join(d, "pak1.pak"), pak1, 0644); err != nil {
        return err
    }
    return nil
}
```

警告： 调用和 `t.Fatal` 类似函数并不总是安全的。点击这里查看更多细节。

失败信息应该包括对错误的详细描述信息。这一点很重要，因为你可能会向许多用户提供测试 API，特别是在测试辅助函数中产生错误的场景增多时。用户应该知道在哪里，以及为什么产生错误。

提示： Go 1.14 引入了一个 `t.Cleanup` 函数，可以用来注册清理函数，在你的测试完成后运行。该函数也适用于测试辅助函数。参见 GoTip #4: Cleaning Up Your Tests 以获得简化测试辅助程序的指导。

下面是一个名为 `paint_test.go` 的虚构文件中的片段，演示了 `(*testing.T).Helper` 如何影响 Go 测试中的失败报告：

```go
package paint_test

import (
    "fmt"
    "testing"
)

func paint(color string) error {
    return fmt.Errorf("no %q paint today", color)
}

func badSetup(t *testing.T) {
    // This should call t.Helper, but doesn't.
    if err := paint("taupe"); err != nil {
        t.Fatalf("could not paint the house under test: %v", err) // line 15
    }
}

func mustGoodSetup(t *testing.T) {
    t.Helper()
    if err := paint("lilac"); err != nil {
        t.Fatalf("could not paint the house under test: %v", err)
    }
}

func TestBad(t *testing.T) {
    badSetup(t)
    // ...
}

func TestGood(t *testing.T) {
    mustGoodSetup(t) // line 32
    // ...
}
```

下面是运行该输出的一个例子。请注意突出显示的文本和它的不同之处：

```
=== RUN   TestBad
    paint_test.go:15: could not paint the house under test: no "taupe" paint today
--- FAIL: TestBad (0.00s)
=== RUN   TestGood
    paint_test.go:32: could not paint the house under test: no "lilac" paint today
--- FAIL: TestGood (0.00s)
FAIL
```

`paint_test.go:15` 的错误是指在 `badSetup` 中失败的设置函数的那一行：`t.Fatalf("could not paint the house under test: %v", err)` 而 `paint_test.go:32` 指的是在 `TestGood` 中失败的那一行测试：`goodSetup(t)`。

正确地使用 `(*testing.T).Helper` 可以更好地归纳出失败的位置，当：

- 辅助函数数量增加
- 在辅助函数中使用其他的辅助函数

- 测试函数使用辅助函数的数量增加

提示：如果一个辅助函数调用 `(*testing.T).Error` 或 `(*testing.T).Fatal`，在格式字符串中提供一些上下文，以帮助确定出错的原因。

提示：如果一个辅助函数没有做任何事情会导致测试失败，那么它就不需要调用 `t.Helper`。通过从函数参数列表中删除 `t` 来简化其签名。

不要从独立的 goroutines 中调用 `t.Fatal`

正如 package testing 中记载的那样，在测试函数或子测试函数之外的任何 goroutine 中调用 `t.FailNow`，`t.Fatal` 等都是不正确的。如果你的测试启动了新的 goroutine，它们不能从这些 goroutine 内部调用这些函数。

测试辅助函数通常不会从新的 goroutine 发出失败信号，因此它们使用 `t.Fatal` 是完全正确的。如果有疑问，可以调用 t.Error 并返回。

```
// Good:
func TestRevEngine(t *testing.T) {
    engine, err := Start()
    if err != nil {
        t.Fatalf("Engine failed to start: %v", err)
    }

    num := 11
    var wg sync.WaitGroup
    wg.Add(num)
    for i := 0; i < num; i++ {
        go func() {
            defer wg.Done()
            if err := engine.Vroom(); err != nil {
                // This cannot be t.Fatalf.
                t.Errorf("No vroom left on engine: %v", err)
                return
            }
            if rpm := engine.Tachometer(); rpm > 1e6 {
                t.Errorf("Inconceivable engine rate: %d", rpm)
            }
        }()
    }
    wg.Wait()

    if seen := engine.NumVrooms(); seen != num {
        t.Errorf("engine.NumVrooms() = %d, want %d", seen, num)
    }
}
```

在测试或子测试中添加 `t.Parallel` 并不会使调用 `t.Fatal` 变得不安全。

当所有对 `testing` API 的调用都在 test function 中时，通常很容易发现不正确的用法，因为 `go` 关键字是显而易见的。传递 `testing.T` 参数会使跟踪这种用法更加困难。通常，传递这些参数的原因是为了引入一个测试辅助函数，而这些测试辅助函数不应该依赖于被测系统。因此，如果一个测试辅助函数注册了一个致命的测试失败，它可以而且应该从测试的 goroutine 中这样做。

对结构字使用字段标签

在表格驱动的测试中，最好为每个测试用例指定密钥。当测试用例覆盖了大量的垂直空间（例如，超过 20-30 行），当有相同类型的相邻字段，以及当你希望省略具有零值的字段时，这是有帮助的。比如说：

```
// Good:
tests := []struct {
    foo     *pb.Foo
    bar     *pb.Bar
    want    string
}{
    {
        foo: pb.Foo_builder{
            Name: "foo",
            // ...
        }.Build(),
        bar: pb.Bar_builder{
            Name: "bar",
            // ...
        }.Build(),
        want: "result",
    },
}
```

保持设置代码在特定的测试范围内

在可能的情况下，资源和依赖关系的设置应该尽可能地与具体的测试用例密切相关。例如，给定一个设置函数：

```
// mustLoadDataSet loads a data set for the tests.
//
// This example is very simple and easy to read. Often realistic setup is more
// complex, error-prone, and potentially slow.
func mustLoadDataset(t *testing.T) []byte {
    t.Helper()
    data, err := os.ReadFile("path/to/your/project/testdata/dataset")

    if err != nil {
        t.Fatalf("could not load dataset: %v", err)
    }
    return data
}
```

在需要的测试函数中明确调用 `mustLoadDataset`：

```go
// Good:
func TestParseData(t *testing.T) {
    data := mustLoadDataset(t)
    parsed, err := ParseData(data)
    if err != nil {
        t.Fatalf("unexpected error parsing data: %v", err)
    }
    want := &DataTable{ /* ... */ }
    if got := parsed; !cmp.Equal(got, want) {
        t.Errorf("ParseData(data) = %v, want %v", got, want)
    }
}

func TestListContents(t *testing.T) {
    data := mustLoadDataset(t)
    contents, err := ListContents(data)
    if err != nil {
        t.Fatalf("unexpected error listing contents: %v", err)
    }
    want := []string{ /* ... */ }
    if got := contents; !cmp.Equal(got, want) {
        t.Errorf("ListContents(data) = %v, want %v", got, want)
    }
}

func TestRegression682831(t *testing.T) {
    if got, want := guessOS("zpc79.example.com"), "grhat"; got != want {
        t.Errorf(`guessOS("zpc79.example.com") = %q, want %q`, got, want)
    }
}
```

测试函数`TestRegression682831`不使用数据集，因此不调用`mustLoadDataset`，这可能会很慢且容易失败：

```
// Bad:
var dataset []byte

func TestParseData(t *testing.T) {
    // As documented above without calling mustLoadDataset directly.
}

func TestListContents(t *testing.T) {
    // As documented above without calling mustLoadDataset directly.
}

func TestRegression682831(t *testing.T) {
    if got, want := guessOS("zpc79.example.com"), "grhat"; got != want {
        t.Errorf(`guessOS("zpc79.example.com") = %q, want %q`, got, want)
    }
}

func init() {
    dataset = mustLoadDataset()
}
```

用户希望在与其他函数隔离的情况下运行一个函数，不应受到这些因素的影响：

```
# No reason for this to perform the expensive initialization.
$ go test -run TestRegression682831
```

何时使用自定义的 `TestMain` 入口点

如果**包中的所有测试**都需要共同设置，并且**设置需要拆解**，你可以使用自定义测试主入口。如果测试用例所需的资源的设置特别昂贵，而且成本应该被摊销，就会发生这种情况。通常情况下，你在这一点上已经从测试套件中提取了任何无关的测试。它通常只用于功能测试。

使用自定义的 `TestMain` **不应该是你的首选**，因为要正确使用它，必须要有足够的谨慎。首先考虑摊销普通测试设置部分的解决方案或普通的测试辅助函数是否足以满足你的需求。

```go
// Good:
var db *sql.DB

func TestInsert(t *testing.T) { /* omitted */ }

func TestSelect(t *testing.T) { /* omitted */ }

func TestUpdate(t *testing.T) { /* omitted */ }

func TestDelete(t *testing.T) { /* omitted */ }

// runMain sets up the test dependencies and eventually executes the tests.
// It is defined as a separate function to enable the setup stages to clearly
// defer their teardown steps.
func runMain(ctx context.Context, m *testing.M) (code int, err error) {
    ctx, cancel := context.WithCancel(ctx)
    defer cancel()

    d, err := setupDatabase(ctx)
    if err != nil {
        return 0, err
    }
    defer d.Close() // Expressly clean up database.
    db = d          // db is defined as a package-level variable.

    // m.Run() executes the regular, user-defined test functions.
    // Any defer statements that have been made will be run after m.Run()
    // completes.
    return m.Run(), nil
}

func TestMain(m *testing.M) {
    code, err := runMain(context.Background(), m)
    if err != nil {
        // Failure messages should be written to STDERR, which log.Fatal uses.
        log.Fatal(err)
    }
    // NOTE: defer statements do not run past here due to os.Exit
    //       terminating the process.
    os.Exit(code)
}
```

理想情况下，一个测试用例在自身的调用和其他测试用例之间是密封的。

至少要确保单个测试用例重置他们所修改的任何全局状态，如果他们已经这样做了（例如，如果测试是与外部数据库一起工作）。

摊销共同测试设置

如果普通设置中存在以下情况，使用 `sync.Once` 可能是合适的，尽管不是必须的。

- 它很昂贵。
- 它只适用于某些测试。

- 它不需要拆解。

```
// Good:
var dataset struct {
    once sync.Once
    data []byte
    err  error
}

func mustLoadDataset(t *testing.T) []byte {
    t.Helper()
    dataset.once.Do(func() {
        data, err := os.ReadFile("path/to/your/project/testdata/dataset")
        // dataset is defined as a package-level variable.
        dataset.data = data
        dataset.err = err
    })
    if err := dataset.err; err != nil {
        t.Fatalf("could not load dataset: %v", err)
    }
    return dataset.data
}
```

当`mustLoadDataset`被用于多个测试函数时，其成本被摊销：

```
// Good:
func TestParseData(t *testing.T) {
    data := mustLoadDataset(t)

    // As documented above.
}

func TestListContents(t *testing.T) {
    data := mustLoadDataset(t)

    // As documented above.
}

func TestRegression682831(t *testing.T) {
    if got, want := guessOS("zpc79.example.com"), "grhat"; got != want {
        t.Errorf(`guessOS("zpc79.example.com") = %q, want %q`, got, want)
    }
}
```

普通拆解之所以棘手，是因为没有统一的地方来注册清理线程。如果设置函数（本例中为`loadDataset`）依赖于上下文，`sync.Once`可能会有问题。这是因为对设置函数的两次调用中的第二次需要等待第一次调用完成后再返回。这段等待时间不容易做到尊重上下文的取消。

字符串拼接

在 Go 中，有几种字符串拼接的方法。包括下面几种例子：

- “+“运算符
- `fmt.Sprintf`
- `strings.Builder`
- `text/template`
- `safehtml/template`

尽管选择哪种方法没有一刀切的规则，但下面的指南概述了在什么情况下哪种方法是首选。

简单情况下，首选 “+”

当连接几个字符串时，更愿意使用 “+"。这种方法在语法上是最简单的，不需要导入包。

```
// Good:
key := "projectid: " + p
```

格式化时首选 `fmt.Sprintf`

当建立一个带有格式化的复杂字符串时，倾向于使用 `fmt.Sprintf`。使用许多 “+“运算符可能会掩盖最终的结果。

```
// Good:
str := fmt.Sprintf("%s [%s:%d]-> %s", src, qos, mtu, dst)
```

```
// Bad:
bad := src.String() + " [" + qos.String() + ":" + strconv.Itoa(mtu) + "]-> " +
dst.String()
```

最佳做法：当构建字符串操作的输出是一个 `io.Writer` 时，不要用 `fmt.Sprintf` 构建一个临时字符串，只是为了把它发送给 Writer。相反，使用 `fmt.Fprintf` 来直接向 Writer 发送。

当格式化更加复杂时，请酌情选择 `text/template` 或 `safehtml/template`。

倾向于使用 `strings.Builder` 来零散地构建一个字符串

在逐位建立字符串时，更倾向于使用 `strings.Builder`。`strings.Builder` 需要摊销的线性时间，而 “+“和 `fmt.Sprintf` 在连续调用以形成一个较大的字符串时需要二次时间。

```
// Good:
b := new(strings.Builder)
for i, d := range digitsOfPi {
    fmt.Fprintf(b, "the %d digit of pi is: %d
", i, d)
}
str := b.String()
```

注意。更多的讨论，请参见 GoTip #29: 高效地构建字符串。

常量字符串

在构建常量、多行字符串变量时，倾向于使用反引号（`）。

```
// Good:
usage := `Usage:

custom_tool [args]`
```

```
// Bad:
usage := "" +
  "Usage:
" +
  "
" +
  "custom_tool [args]"
```