Frequently Asked Questions (FAQ)

由来

该项目的目的在于什么?

十年以来，主流的系统级编程语言并未出现过，但在这期间，计算环境已经发生了巨大的变化。以下是一些趋势：

- 计算机的速度变得极快，但软件开发还不够快。
- 在今天，依赖管理成为了软件开发中一个大的部分，但C传统语言的“头文件”与清晰的依赖分析——以及快速编译背道而驰。
- 对于像Java和C++那笨重的类型系统的反抗越来越多，这将人们推向了Python和JavaScript之类的动态类型语言。
- 流行的系统语言对于像垃圾回收与并行计算那种基本思想的支持并不算好。
- 多核计算机的出现产生了一些麻烦与混乱。

我们相信这值得重新尝试一种新的语言，一种并发的、带垃圾回收的、快速编译的语言。它需要满足以下几点：

- 它可以在一台计算机上用几秒钟的时间编译一个大型的Go程序。
- Go为软件构造提供了一种模型，它使依赖分析更加容易，且避免了大部分C风格include文件与库的开头。
- Go的类型系统没有层级，因此不需要在类型之间的关系定义上花费时间。此外，尽管Go是静态类型的，该语言也试图使类型感觉起来比典型的面向对象语言更轻量级。
- Go完全是垃圾回收型的语言，并为并发执行与通信提供了基本的支持。
- 按照其设计，Go打算为多核机器上系统软件的构造提供一种方法。

关于此问题的更多答案见 Go在Google：软件工程服务中的语言设计 一文。

该项目的状态如何?

Go在2009年11月10日成为了公共开源项目。在两年的积极设计与开发之后，应稳定性要求， Go 1于2012年3月28日发布。 Go 1包含语言规范、标准库与定制工具，它为创建可靠的产品、项目及出版物提供了稳定的基础。

随着其稳定性的确立，我们使用Go来开发程序、产品以及工具，而非积极地更改语言与库。实际上，Go 1的目的就是提供长期的稳定性。不向前兼容的更改将不会对任何Go 1点发行版进行。我们想通过我们所拥有的来了解Go未来的版本将看起来如何，而不是用语言阻碍前进的路。

当然，Go本身的开发将继续进行，但重点将在性能、可靠性、可移植性和新功能的添加上，例如提升对国际化的支持。

这在某一天可能会成为Go 2，但用不了几年，它就会被我们今天使用Go 1所学到的东西所影响。

名字的由来是什么？

对于Go的调试器，“Ogle”将会是个不错的名字。

吉祥物的由来是什么？

吉祥物与Logo由Renée French设计，她也设计了Plan 9的小兔子Glenda。 Gopher衍生自她在几年前为WFMU设计的一件T恤衫。其Logo与吉祥物以知识共享-署名3.0方式授权。

该项目的历史是什么？

2007年9月21日，Robert Griesemer、Rob Pike与Ken Thompson在白板上开始了对新语言目标的描绘。在几天之内，目标被制定成做事的计划，关于其未来的美妙想法便在此刻产生。其设计在与工作无关的平行时间中继续。到了2008年1月，Ken开始了编译器的工作，并在其上探索各种想法；它生成C代码并将其输出。到了年中，该语言成为了全职项目，拥有了充足的安排来尝试一个产品级编译器。在2008年5月时，Ian Taylor根据规范草案独自开始了Go GCC前端的工作。2008年年末， Russ Cox的加入帮助将该语言与库从原型变成了现实。

Go在2009年11月10日成为了公共开源项目。来自社区的许多人都可以贡献想法，参与讨论及编写代码。

你们为什么要创造新的语言？

Go在既有语言与环境下进行系统编程的挫折中诞生。编程变得太难，对语言的选择有一定的责任。我们必须在高效编译、高效执行或轻松编程之间选择其一，在同样主流的语言中，三者不能同时达到。程序员们通过转移到Python和JavaScript之类的动态类型语言，而非C++或一定程度上的Java上，来选择轻松在安全和效率之上。

Go试图成为结合解释型编程的轻松、动态类型语言的高效以及静态类型语言的安全的编译型语言。它也打算成为现代的，支持网络与多核计算的语言。要满足这些目标，需要解决一些语言上的问题：一个富有表达能力但轻量级的类型系统，并发与垃圾回收机制，严格的依赖规范等等。这些无法通过库或工具解决好，必须创造新的语言。

文章 Go 在 Google 中讨论了Go语言设计的其背景和动机，关于本FAQ中的许多为题，该文章提供了更多详情。

Go的前身是什么？

Go主要是C家族的（基本语法），从Pascal/Modula/Oberon家族引入了重要的东西（声明，包），加上一些由Tony Hoare的CSP所激发的语言的理念，例如Newsqueak与Limbo（并发）。然而，它是一个全新的语言。在各个方面上，该语言的设计都考虑到程序员做的事情以及如何去编程，至少是我们进行的那种编程。更实际，也就意味着更有趣。

其设计的指导原则是什么？

如今的编程包含了太多记账式的、重复的、文书式的工作。就像Dick Gabriel说的那样：“老程序读起来就像健谈的研究工作者与善于学习的书呆子同事之间平和的对话，而不像同编译器之间的争辩。谁会认为成熟必然带来杂乱？”这样的成熟是值得的——没有人想要回到老的语言——但它能更安静地被实现么？

Go试图在两种意义上减少文字的键入次数。贯穿其设计，我们试图减少混乱与复杂性。它没有前置声明与头文件；任何东西都只声明一次。初始化富有表现力，自动且易于使用。语法的关键字清晰而轻量。啰嗦的表达式（`foo.Foo* myFoo = new(foo.Foo)`）可使用 `:=` 声明并初始化结构，通过简单的类型推断来简化。也许最根本的是，这里没有类型层级：类型**就是**类型，无需说明它们之间的关系。这些简化允许Go无需牺牲成熟而富有表现力且易于理解。

另一个重要的原则是保持概念正交。方法可被任何类型实现，结构代表数据而接口代表抽象等等。正交性使一些东西相结合时发生的事情更易理解。

使用

Google是否在内部使用Go?

是的。现在有几个Go程序正部署在Google的内部产品中。一个公共的例子就是在 http://golang.org 后台支持的服务。它仅仅是在 Google应用引擎 上配置的产品中运行的文档服务。

其它例子包括用于大规模SQL安装的 Vitess 系统，以及Google的下载服务器 `dl.google.com`，它用于释放Chrome二进制文件和其它类似 `apt-get` 的大型可安装包。

Go程序能否链接C/C++程序?

现在有两种Go编译器实现，`gc`（`6g` 程序及其同类）和 `gccgo`。`Gc` 使用了一种不同的调用约定和连接器，因此只能与使用同样约定的C程序连接。现在只有这样的C编译器，而没有这样的C++编译器。 `gccgo` 为GCC的前端，可以小心地与GCC编译的C或C++程序连接。

cgo 程序为“外部函数接口”提供了一种机制，以允许从Go代码中安全地调用C库。SWIG为C++库扩展了这种能力。

Go是否支持Google的缓存协议?

一个单独的开源项目为此提供了必要的编译器插件与库。它可从 http://code.google.com/p/goprotobuf/ 获取。

我能否将Go主页翻译为其它语言?

完全可以。我们鼓励开发者将Go语言站点译成他们自己的语言。然而，如果你选择加入Google的Logo，或品牌化推广你的站点（它不会出现在golang.org上），你需要遵守 http://www.google.com/permissions/guidelines.html 上的指导方针。

设计

Unicode标识符如何?

从ASCII的限制中扩展标识符的空间对于我们是非常重要的。Go的规则——标识符字符必须是由Unicode定义的字母或数字——易于理解并实现，但也有限制。比如结合式字符就排除在设计之外。在一个标识符是什么的外部定义可被接受，且标识符的标准化定义可确保没有歧义之前，将结合式字符保持在混乱之外似乎更好些。因此我们有一条简单的规则可以在不破坏程序的情况下以后扩展，承认歧义性标识符的规则肯定会出现Bug，它可以避免此类Bug。

与此相关，由于可导出标识符必须以一个大写字母开始，根据定义，以"字母"创建的标识符在一些语言中不能被导出。目前，唯一的解决方案就是使用一些如 `X日本語` 这样的形式，这明显无法令人满意；我们在考虑其它的选项。大小写可视性规则无论如何都不会改变，因为这是我们最喜爱的Go特性之一。

为什么Go没有X特性?

任何语言都会包含新奇的特性，也会省略掉一些人最喜爱的特性。Go的设计着眼于编程的快乐，编译的素的，概念的正交以及一些必须支持的特性，例如并发机制和垃圾回收机制。你最喜欢的特性可能由于不合适而缺失了，因为它影响了编译速度或设计的清晰度，或因为它会使根本的系统模型变得太复杂。

若Go因为缺失了特性 *X* 而烦扰到您了，请原谅我们。我们建议您研究一下Go所拥有的特性，您可能会发现它们以有趣的方式弥补了 *X* 的缺失。

为什么Go没有泛型?

泛型可能会在某个时刻加入。我们对其并不感到紧急，尽管我们明白一些程序员会是这样。

泛型是方便的，但它们也同时付出了类型系统与运行时的复杂性代价。尽管我们还在继续思索着，但还未找到拥有与其复杂度相称价值的设计。与此同时，Go内建的映射与切片，加上使用空接口来构造容器（带显式拆箱）的能力，意味着如果顺利的话，在某些情况下，它可以写出泛型所能做到的代码。

这保留为一个开放性问题。

为什么Go没有异常处理?

我们相信用 `try-catch-finally` 习语那样的控制结构连接成的异常，其结果就是令人费解的代码。它往往也会怂恿程序员标注太多普通的错误，诸如打开文件失败之类的作为异常。

Go采用了一种不同的方法。对于朴素的错误处理，Go的多值返回使错误易于报告而无需重载返回值。 一个典型的错误类型，配合Go的其它特性，使错误处理变得愉快而与众不同。

Go也拥有内建函数的配合来标记出真正的异常状况并从中恢复。该恢复机制只会在函数的错误状态解除之后，作为它的一部分执行，这足以处理灾难而无需格外的控制结构。如果使用得当，就能产生清晰的错误处理代码。

详情见Defer、Panic、与Recover一文。

为什么Go没有断言?

Go不提供断言。它们无疑是很方便的，但我们的经验是，程序员们会使用它们作为依靠，以避免考虑适当的错误处理和报告。适当的错误处理意味着服务器在非致命错误后可以继续运行，而不会彻底崩溃。适当的错误报告意味着错误更加直接了当，最关键的一点是，它能将程序员从解释大型崩溃的跟踪中拯救出来。精确的错误是极其重要的，尤其在程序员们从不熟悉的代码中发现错误时。

我们明白这是一个争论的焦点。Go语言和库中的一些东西不同于现代的实践，只不过是因为我们觉得偶尔尝试下不同的方法是值得的。

为什么以CSP思想来构建并发?

并发和多线程编程以其困难著称。我们相信一部分原因是因为复杂的设计，例如pthreads；一部分是因为过于强调低级的细节，例如互斥、条件变量以及内存屏障。更高级的接口可简化代码，尽管像互斥这类的东西仍然存在。

Hoare的通信序列过程（即CSP）为并发提供了高级的语言支持，它是最成功的模型之一。Go的并发原语来自该家族树不同的部分，它最主要的贡献就是将强大的信道概念作为第一类对象。从这些早期语言中得到的经验显现出CSP模型很适合用作过程式语言的框架。

为什么使用Go程而非线程?

Go程是让使发易于使用的一部分。这个想法已经存在了一段时间，它是将独立执行的函数——协程——多路复用到一组线程上。当协程被阻塞，如通过调用一个阻塞的系统调用时，运行时会在相同的操作系统线程上自动将其它的协程转移到一个不同的，可运行的，不会被阻塞的线程上。重点是程序员不会看见。结果，我们称之为Go程，可以非常廉价：除非它们在在长期运行的系统调用上花费了大量的时间，否则它们只会花费比栈多一点的内存，那只有几KB而已。

为了使栈很小，Go的运行时使用了分段式栈。一个新创建的Go程给定几KB，这几乎总是足够的。当它不够时，运行时会自动地分配（并释放）扩展片段。每个函数调用平均需要大概三条廉价的指令。这实际上是在相同的地址空间中创建了成百上千的Go程。如果Go程是线程的话，系统资源会更快地耗尽。

为什么映射操作不定义为原子性的?

经过长时间的讨论，决定了映射的典型使用无需从多Go程中安全地访问，在那些情况下，映射可能是一些大型数据结构的一部分或已经同步的计算。所以要求所有映射操作抓取互斥会减慢大部分程序并添加一些安全性。这并不是个容易的决定，然而，这也就意味着不受控制的映射访问会使程序崩溃。

该语言并不排除原子性映射的更新，在需要时，例如在部署一个不信任的程序，该实现可以互锁映射访问。

你们会接受我对语言的修改么?

人们经常会向该语言提出改进建议——邮件列表中包含了此类讨论的丰富历史——但只有极少的修改会被接受。

尽管Go是个开源项目，但该语言和库受兼容性保证 以确保更改不回破坏既有的程序。若你的建议违反了Go 1规范，不论它是否值得，我们都不会接受。Go将来的主版本可能不会与Go 1兼容，但我们还不打算讨论这意味着什么。

即使你的建议与Go 1规范兼容，它也可能不符合Go设计目标的精神。文章 **Go在Google：软件工程服务中的语言设计** 解释了Go的起源机器设计背后的动机。

类型

Go是面向对象的语言吗？

既是也不是。尽管Go拥有类型和方法，也允许面向对象风格的编程，但它没有类型层级。在Go中“接口”的概念提供了不同的方法，我们相信它易于使用且在某些方面更通用。也有一些在其它类型中嵌入类型的方法，来提供类似（而非完全相同）的东西进行子类化。此外，Go中的方法比C++或Java中的更通用：它们可被定义为任何种类的数据。甚至是像普通的“未装箱”整数这样的内建类型。它们并不受结构（类）的限制。

此外，类型层级的缺失也使Go中的“对象”感觉起来比C++或Java的更轻量级。

我如何获得方法的动态分配？

拥有动态分配方法的唯一途径就是通过接口。结构或其它混合类型的方法总是静态地确定。

为什么没有类型继承？

面向对象编程，至少在最著名的语言中，涉及了太多在类型之间关系的讨论，关系总是可以自动地推断出来。Go则使用了一种不同的方法。

不像需要程序员提前声明两个类型的关联，在Go中类型会自动满足任何接口，以此实现其方法的子集。除了减少记账式编程外，这种方法拥有真正的优势。类型可立刻满足一些接口，而没有传统多重继承的复杂性。接口可以非常轻量——带一个甚至零个方法的接口能够表达一个有用的概念。若出现了新的想法，或为了测试目的，接口其实可以在以后添加——而无需注释掉原来的类型。由于在类型和接口之间没有明确的关系，也就无需管理或讨论类型层级。

用这些思想来构造一些类似于类型安全的Unix管道是可能的。例如，看看 `fmt.Fprintf` 如何能将格式化打印到任何输出而不只是文件，或 `bufio` 包如何能从文件I/O中完全分离，或 `image` 包如何生成已压缩的图像文件。所有这些想法都来源于一个单一的接口（`io.Writer`），都由一个单一的方法来表现（`Write`）。而这只是表面文章。Go的接口在如何组织程序方面有着深刻的影响。

它需要一段时间来适应，但这种隐式的类型依赖是Go中最具生产力的东西之一。

为什么 `len` 是函数而非方法？

我们讨论过这个问题，但显然在实践中将 `len` 及与其相关的功能实现为函数比较好，而且这不会复杂化关于基本类型接口（在Go类型意义上）的问题。

为什么Go不支持方法和操作符的重载?

若方法分配无需很好地进行类型匹配，该方法即会被简化。其它语言的经验告诉我们，拥有名字相同但签名不同的多种方法偶尔是有用的，但它也会在实践中造成混乱和不确定。在Go的类型系统中，只通过名字进行匹配以及类型的一致性需求是主要的简化决策。

至于操作符重载，它似乎并不能比绝对必要的东西提供更多便利。此外，没有它事情会变得更简单。

为什么Go没有 "implements" 声明?

Go的类型通过实现接口的方法来满足该接口，仅此而已。这个性质允许定义接口并使用，而无需修改已有的代码。它使用一种结构类型来帮助关系的分离并改进代码可重用性，并使它更容易在为代码开发而出现的模式上构建。接口的语义学是Go的灵活、轻量感的主要原因之一。

更多详情见类型继承的问题。

我如何保证我的类型满足某个接口?

你可以通过尝试赋值来要求编译器检查类型 `T` 是否实现了接口 `I`：

```
type T struct{}
var _ I = T{}   // 确认T是否实现了I。
```

若 `T` 未实现 `I`，则错误会在编译时捕获。

如果你希望接口的使用者显式地声明它们实现了它，你可以将一个带描述性名称的方法添加到该接口的方法集中：

```
type Fooer interface {
    Foo()
    ImplementsFooer()
}
```

然后类型必须实现 `ImplementsFooer` 方法成为 `Fooer`， godoc的输出中清晰地记录了事实和通告。

```
type Bar struct{}
func (b Bar) ImplementsFooer() {}
func (b Bar) Foo() {}
```

大部分代码无需使用这类约束，因为它们限制了实用程序的接口思想。但有时候，它们也需要解决相似接口之间的歧义。

为什么类型T不满足Equal接口?

考虑以下简单的接口，它表示一个可以将自身与另一个值进行比较的对象：

```
type Equaler interface {
    Equal(Equaler) bool
}
```

以及此类型 T：

```
type T int
func (t T) Equal(u T) bool { return t == u } // does not satisfy Equaler
```

不像在一些多态类型系统中类似的情况，T 并未实现 Equaler。 T.Equal 的实参类型为 T，而非字面上所需要的类型 Equalar。

在Go中，类型系统并不提升 Equal 的实参，那是程序员的责任，就以下类型 T2 所示，它实现了 Equaler：

```
type T2 int
func (t T2) Equal(u Equaler) bool { return t == u.(T2) }  // 满足Equaler
```

即使它不像其它的类型系统也好，因为在Go中*任何*满足 Equaler 的类型都能作为实参传至 T2.Equal，并在运行时我们必须检查该实参是否为 T2类型。一些语言将其安排在编译时以保证做到这一点。

一个相关的例子是另一种情形：

```
type Opener interface {
   Open() Reader
}

func (t T3) Open() *os.File
```

在Go中，T3 并不满足 Opener，尽管它在另一种语言中可能满足。

在相同情况下，Go的类型系统确实为程序员做的更少，子类型化的缺乏使关于接口满足的规则非常容易制订：函数的名字和签名完全就是那些接口吗？Go的规则也容易高效地实现。我们感觉这些效益抵消了自动类型提升的缺失。Go在某天应当采取一些泛型的形式，我们期望会有一些方式来表达这些例子的想法，且也拥有静态检查。

我能否将[]T转换为[]interface{}?

不能直接转换，因为它们在内存中的表示并不相同。必须单独地将元素复制到目标切片。下面的例子将 int 切片转换为 interface{} 切片：

```
t := []int{1, 2, 3, 4}
```

```
s := make([]interface{}, len(t))
for i, v := range t {
    s[i] = v
}
```

为什么我的nil错误值不等于nil?

在底层，接口作为两个元素实现：一个类型和一个值。该值被称为接口的动态值，它是一个任意的具体值，而该接口的类型则为该值的类型。对于 int 值3，一个接口值示意性地包含(int, 3)。

只有在内部值和类型都未设置时(nil, nil)，一个接口的值才为 nil。特别是，一个 nil 接口将总是拥有一个 nil 类型。若我们在一个接口值中存储一个 *int 类型的指针，则内部类型将为 *int，无论该指针的值是什么：(*int, nil)。因此，这样的接口值会是非 nil 的，*即使在该指针的内部为* nil。

这种情况会让人迷惑，而且当 nil 值存储在接口值内部时这种情况总是发生，例如错误返回：

```
func returnsError() error {
        var p *MyError = nil
        if bad() {
                p = ErrBad
        }
        return p // 将总是返回一个非nil错误。
}
```

如果一切顺利，该函数会返回一个 nil 的 p，因此该返回值为拥有(*MyError, nil)的 error 接口值。这也就意味着如果调用者将返回的错误与 nil 相比较，它将总是看上去有错误，即便没有什么坏事发生。要向调用者返回一个适当的 nil error，该函数必须返回一个显式的 nil：

```
func returnsError() error {
        if bad() {
                return ErrBad
        }
        return nil
}
```

这对于总是在签名中使用 error 类型返回错误（正如我们上面做的）而非像 *MyError 这样具体类型的函数来说是个不错的主意，它可以帮助确保错误被正确地创建。例如，即使 os.Open 返回一个 error，若非 nil 的话，它总是具体的类型 *os.PathError。

对于那些描述，无论接口是否被使用，相似的情形都会出现。只要记住，如果任何具体的值已被存储在接口中，该接口就不为 nil。更多信息请访问反射法则。

为什么没有像C那样的无标签联合?

无标签联合会违反Go的内存安全保证。

为什么没有变体类型?

变体类型，亦称为代数类型，它提供了一种方法来指定一个值可以获得其它类型集中的一个类型，但仅限于那些类型。在系统编程中一个常见的例子是指定了一个错误，具体来说，一个网络错误、一个安全性错误或一个应用错误，并允许调用者通过检查该错误的类型来辨别错误的根源。另一个例子是在语法树中的每个节点可以为不同的类型：声明、语句、赋值等等。

我们考虑过将变体类型添加到Go中，但经过讨论后决定远离它们，因为它们以混乱的方式与接口重叠。如果变体类型的元素是它们自己的接口会发生什么?

此外，变体类型从事的一些工作已被该语言所覆盖。上面有关错误的例子很容易使用接口值来表达，以此控制错误及类型转换来辨别状况。关于语法树的例子也可以这么做，尽管不太优雅。

值

为什么Go不提供隐式数值转换?

在C中数值类型之间的自动转换所造成的混乱超过了它的便利。一个表达式什么时候是无符号的?这个值有多大?它可以被覆盖吗?该结果可被移植，独立于它所执行的机器吗?这也使编译器陷入了麻烦；“一般的算数转换”不容易实现且在跨架构时不一致。出于可移植性的原因，我们决定在代码中付出一些显式转换的代价，来使事情变得清晰而直接。而在Go中对常量的定义——无符号和大小注解的任意精度的值——对事情有却重大的改善。

一个相关的细节是，不像在C中，`int` 和 `int64` 是不同的类型，即使 `int` 是一个64位的类型。`int` 类型是一般的，如果你关心一个整数占多少位，Go会鼓励你搞清楚它。

常量一文探讨了此话题的更多细节。

为什么映射是内建的?

同样的理由是：它们是如此地强大和重要的数据结构，提供了一种有句法上支持的卓越的实现，使编程更加惬意。我们相信Go的映射实现足够健壮以服务于绝大多数的使用。若一个具体的应用可从定制的实现中收益，那就可以写一个，但它将在语法上不那么方便，这似乎是个合理的权衡。

为什么映射不允许将切片作为键?

映射查找需要一个相等性操作符，而切片并未实现它。它们不能实现相等性，因为相等性没有在这种类型上很好地定义，这里有多个因素，涉及到浅层次与深层次之间的比较，指针与值之间的比较，如何处理递归类型等等。我们可能会重新审视这个问题——并为切片实现相等性，而不会使任何已存在的程序失效——但切片的相等性意味着什么还没有一个清晰的概念，现在最简单的方法就是远离它。

不像之前的发布版，在Go 1中为结构和数组的相等性下了定义，因此这样的类型可被用作映射的键。然而，切片仍然没有相等性的定义。

为什么映射、切片和信道是引用，而数组是值?

这个话题有很长的历史。在早期，映射和在语法上是指针，它不可能通过声明或使用非指针来实例化。此外，我们也曾在数组该如何工作的问题上挣扎。最后我们认为，指针和值的严格分离会使语言更难用。将这些类型的行为修改成对关联的引用，共享数据结构解决了这些问题。这种修改向该语言加入了一些令人遗憾的复杂性，但它们在可用性上有更大的效果：当它被引入时，Go成为了一门更高效，更舒适的语言。

编写代码

如何将库文档化?

有一个用Go编写的程序 `godoc` 它可以从源码中提取包文档。它可以在命令行或Web中使用。一个例子就是它运行在http://golang.org/pkg/上。实际上，`godoc` 已经在http://golang.org/上实现了完整的网站。

有没有Go编程风格指南?

最后，可能有少量的规则来指导像命名、布局以及文件组织这类的事情。文档高效Go编程包含了一些风格建议。更直接的说，程序 `gofmt` 是一个美观打印工具，它的目的在于强制实施布局规则，它可以取代要解释该做什么和不该做什么的纲要所有代码仓库中的Go代码都运行过了 `gofmt`。

文档 Go 代码审核评注 收集了 Go 的习惯用法，程序员们经常忽视它们。它是一份便于审核人员审核Go项目代码的引用资料。

我如何向Go库提交补丁?

库的源码在 `go/src/pkg` 中。如果你想进行重大的更改，请在动手前在邮件列表中讨论。

关于如何进行的更多信息请访问为Go项目做贡献。

为什么“go get”在克隆代码仓库时使用HTTPS?

公司通常只允许标准TCP端口80（HTTP）和443（HTTPS）的向外通行，而封闭其它端口的向外通行，包括TCP端口9418（git）和TCP端口22（SSH）。当使用HTTPS代替HTTP时，`git` 默认会强制执行证书认证来提供保护，防止中间人的窃听和篡改攻击。因此为了安全， `go get` 命令就使用了HTTPS。

如果你更喜欢 `git` 用你现有的密钥通过SSH来推送更改，那也很容易。对于GitHub，试试下面的解决方案：

- 在期望的目录下手动克隆代码仓库：

```
$ cd $GOPATH/src/github.com/username
$ git clone git@github.com:username/package.git
```

- 将一下两行添加到 `~/.gitconfig` 中，强制 `git push` 使用 SSH 协议：

```
[url "git@github.com:"]
        pushInsteadOf = https://github.com/
```

我如何通过“go get”来管理包的版本？

“go get”并没有明确的包版本概念。版本是最主要的复杂性来源，尤其是在大型代码的基础上，我们并没有适合所有Go用户的万能解决方案。“go get”和更大的Go工具链，仅能为包提供不同的导入路径来隔离它们。例如，尽管标准库的 `html/template` 和 `text/template` 都是“template包”，但它们能够共存。下面为包的作者和用户给出一些意见和建议。

公用的包应为它们的演化尽量保持向下兼容性。Go 1 兼容性方针 是个不错的参考：不要移除可导出的名称，鼓励标记出复合字面，等等。若需要不同的功能，请添加一个新的名字，而非更改旧的名字。若需要完全打破兼容性，请用新的导入路径创建一个新的包。

若您使用的是外部提供的包，并担心它会以意想不到的方式改变，最简单的解决方案就是把它复制到你的本地仓库中。（这是Google内部采用的方法。）将该副本存储在一个新的导入路径中，以此来标识出它是个本地的副本。例如，你可以将“original.com/pkg”复制成“you.com/external/original.com/pkg”。 Keith Rarick的goven就是个帮你自动处理它的工具。

指针与分配

函数形参在什么时候传值？

和所有C家族中的语言一样，Go中的所有懂喜都通过值来传递。也就是说，函数总是会获得向它传递的东西的一份副本，就好像有一个赋值语句向它的形参赋值。例如，将一个 `int` 值传入一个函数就会创建该 `int` 值的一份副本，而传入一个指针值则会创建该指针的一份副本，而不是它所指向的数据。（关于它如何影响方法接收器的讨论见下一节。）

映射和切片值的行为就像指针一样：它们就是包含指向基本映射或切片数据的指针的描述符。复制一个映射或切片会创建一个存储在接口值中的东西的一个副本。若该接口值保存了一个结构，复制该接口值则会创建一个该结构的副本。若该接口值保存了一个指针，复制该接口值则会创建一个该指针的副本，而且同样不是它所指向的数据。

我应在何时使用接口指针？

几乎用不着。接口值指针极其罕见，复杂情况涉及掩饰接口值的类型以延迟求值。

不过有一个常见的错误，就是将接口值指针传递给一个期望接受接口的函数。编译器会抱怨这种错误，但这种情况仍然会产生混淆，因为有时为了满足一个接口，指针是必要的。这种洞察力就在于，尽管常量类型的指针可以满足一个接口，但还有一个例外：**接口指针永远无法满足接口**。

考虑此变量声明：

```
var w io.Writer
```

打印函数 fmt.Fprintf 将其第一个实参作为接口值，该值满足像 io.Writer 这类实现了标准 Write 方法的接口。因此我们可以这样写：

```
fmt.Fprintf(w, "hello, world\n")
```

If however we pass the address of w, the program will not compile. 但如果我们传递了 w 的地址，该程序将不会编译。

```
fmt.Fprintf(&w, "hello, world\n") // 编译时错误。
```

一个例外就是任何值，甚至接口指针都能被赋予一个空接口类型（interface{}）的变量。即便如此，若该值是一个指针接口，就几乎可以断定它是个错误；其结果仍会产生混淆。

我应当为值或指针定义方法吗?

```
func (s *MyStruct) pointerMethod() { } // 为指针定义的方法
func (s MyStruct)  valueMethod()   { } // 为值定义的方法
```

对于不习惯指针的程序员，这两个例子之间的差别会造成混乱，但这种情况其实是非常简单的。当为一个类型定义了一个方法，则接收者（上面例子中的 s）的表现正好就是该方法的实参。将接收者定义为值还是指针都是一样的问题，就像一个函数的实参应该是值还是指针一样。有几点需要考虑的地方。

首先，也是最重要的一点，方法需要修改接收者吗？如果是，则接收者*必须*是一个指针。（切片和映射的行为类似于引用，所以它们的状况有一点微妙，但比如说要改变方法中切片的长度，则接受者仍然必须是指针。）在上面的例子中，如果 pointerMethod 修改了 s 的字段，那么调用者将观察到那些改变，但 valueMethod 是由调用者实参的副本调用的（这是传值的规定），因此对它的更改对于调用者来说是不可见的。

顺便一提，指针接收者在Java中的情况和Go是相同的，尽管在Java中指针隐藏在幕后；而Go的值接受者则不相同。

其次是效率问题的考虑。若接受者很大，比如说一个大型的 struct，使用指针接收器将更廉价。

接着是一致性问题。若某些类型的方法必须拥有指针接收者，则其余的也应该这样，因此不管该类型被如何使用，方法集都始终如一。更多详情见方法集一节。

对于诸如基本类型、切片以及小型 struct 这样的类型，值接收者是非常廉价的，因此除非该方法的语义需要一个指针，一个有效而清楚的值接收者。

new与make之间有什么不同?

简单来说：new 分配内存，make 初始化切片、映射和信道类型。

更多信息请访问高效Go编程的相关章节。

int 在64位机器上的大小是多少？

int 和 uint 的大小取决于具体实现，但在给定平台上它们彼此之间相同。64位Go编译器（gc和gccgo）使用32位来表示 int。依赖于值具体大小的代码应当使用确定大小的类型，比如说 int64。另一方面，浮点数标量和复数的大小总是固定的：float32、 complex64 等等，因为程序员在使用浮点数时应当知道精度。浮点数常量的默认大小为 float64。

目前，所有的实现都用32位int，基本上是很武断的决定。然而，我们希望在未来的Go发行版中， int 会在64位架构上增加到64位。

我如何知道变量分配在堆上还是栈上？

从正确性立场上看，你无须了解它。Go中存在的每一个变量都只需引用它就行。由实现选择的存储位置与该语言的语义无关。

存储位置对于编写有效的程序来说并无影响。如果可能，Go编译器会在该函数的栈帧中，分配该函数的局部变量。然而，如果编译器不能证明在该函数返回后，该变量不再被引用，那么编译器必须在垃圾回收的堆上分配该变量，以避免悬空指针错误。此外，若局部变量非常大，它可能会更合理地将变量存储在堆而非栈中。

在当前编译器中，若一个变量的地址已被占用，该变量对于堆上的分配来说就是个候选的。然而，当这样的变量不会在该函数返回后继续存在并驻留在栈上时，一个基本的*逃逸分析*就能识别这一情况。

为什么我的Go进程会使用那么多虚拟内存？

Go的内存分配器在虚拟内存中预留了一大块区域作为分配的地方。这块虚拟内存局部于具体的Go进程，而这种预留并不会剥夺内存中的其它进程。

To find the amount of actual memory allocated to a Go process, use the Unix top command and consult the RES (Linux) or RSIZE (Mac OS X) columns.

要得出分配给某个Go进程的实际内存数额，请使用Unix的 top 命令并查阅 RES（Linux）或 RSIZE（Mac OS X）一列。

并发

什么操作是原子性的？什么是互斥性的？

我们现在还没有完整地定义它，不过一些关于原子性的细节还是可以从 Go内存模型规范中找到的。

至于互斥性，sync包实现了它们，但我们希望Go的编程风格会鼓励人们尝试更高级的技巧。特别是，考虑结构化你的程序，以便一次只用一个Go程来负责一块特定的数据。

不要通过共享内存来进行通信，而应该通过通信来共享内存。

关于这个概念的详细讨论，请参阅 通过通信共享内存 的代码漫步及其 相关文章。

为什么我的多Go程程序不能使用多个CPU?

你必须设置GOMAXPROCS外壳环境变量或使用运行时包中同名的函数 `function` 以允许运行时支持利用不止一个的操作系统线程。

执行并行计算的程序应该能从增加 GOMAXPROCS 中获益。不过，你必须意识到 并发不是并行。

为什么使用 GOMAXPROCS > 1 有时会使我的程序变慢?

这取决于你程序的性质。在本质上连续的问题并不能通过添加更多Go程来提高速度。只有当问题在本质上并行的时候，并发才能编程并行处理。

在实际应用中，比起进行运算，在信道上花费更多时间通信的程序，会在使用多操作系统线程时出现性能下降。这是因为在线程间发送数据涉及到切换上下文，这需要很大的代价。比如说，在Go语言规范中 素数筛的例子并没有明显的并行性，尽管它启动了一些Go程，但增加 GOMAXPROCS 更有可能会减慢速度，而非提高速度。

Go的Go程调度并不如所需要的那么好。在将来，它应当能识别这类情况，并优化它对操作系统线程的使用。现在，GOMAXPROCS 应当根据每个应用来进行设置。

关于此话题的更多详情见 并发不是并行。

函数与方法

为什么T与*T拥有不同的方法集?

根据Go规范中的定义

> 其它任意已命名类型 T 的方法集由所有带接收者类型 T 的方法组成。与指针类型 *T 相应的方法集为所有带接收者 *T 或 T 的方法的集（就是说，它也包含 T 的方法集）。

如果一个接口值包含一个指针 *T，一个方法调用可通过解引用该指针来获得一个值，但如果一个接口值包含一个值 T，就没有可用的方式让一个方法调用获得一个指针。

即便在编译器可以获得传入方法的值的地址的情况下，若该方法修改了该值，则更改会在调用者中丢失。一个常见的例子是，代码：

```
var buf bytes.Buffer
io.Copy(buf, os.Stdin)
```

会将标准输入复制到 buf 的**副本**中，而不是复制到 buf 自身。这几乎是从不期望的行为。

闭包作为Go程在运行时会发生什么?

当闭包与并发一起使用时，可能会产生一些混乱。考虑以下程序：

```
func main() {
    done := make(chan bool)

    values := []string{"a", "b", "c"}
    for _, v := range values {
        go func() {
            fmt.Println(v)
            done <- true
        }()
    }

    // 在退出前等待所有Go程完成
    for _ = range values {
        <-done
    }
}
```

有人可能会错误地希望看到 a，b，c 作为输出。而你可能会看到 c，c，c。这是因为每一次循环迭代中都使用了变量 v 的相同实例，因此每一个闭包都共享了单一的变量。当该闭包运行时，它将在 fmt.Println 执行后打印出 v 的值，但 v 可能已经在Go程启动后被修改了。要在这类问题发生前发现它们，请运行 go vet。

要将 v 的当前值在每一个闭包启动后绑定至它们，就必须在每一次迭代中，通过修改内部循环来创建新的变量。其中一种方式就是将变量作为实参传至该闭包中：

```
    for _, v := range values {
        go func(u string) {
            fmt.Println(u)
            done <- true
        }(v)
    }
```

在这个例子中，v 的值作为一个实参传入了该匿名函数。然后这个值就可作为变量 u 在该函数中访问了。

甚至只需简单地创建新的变量，使用声明的风格看起来可能有点怪，但这在Go中能很好地工作：

```
    for _, v := range values {
        v := v // 创建新的“v”。
        go func() {
            fmt.Println(v)
            done <- true
        }()
    }
```

流程控制

Go有没有 ?: 操作符?

在Go中没有三元操作符的形式。你可以使用下面的方法来识相相同的结果：

```
if expr {
    n = trueVal
} else {
    n = falseVal
}
```

包与测试

我如何创建多文件包?

把所有源文件都放进与它们自己的包相同的目录中去。源文件可随意从不同的文件中引用项，而无需提前声明或头文件。

除分割成多个文件外，包可以像个单文件包一样编译并测试。

我如何编写单元测试?

在相同的目录中创建一个以 `_test.go` 结尾的新文件作为你的包源文件。在该文件中，加入 `import "testing"` 并编写以下形式的函数：

```
func TestFoo(t *testing.T) {
    ...
}
```

在该目录中运行 `go test`。该脚本会查找 `Test` 函数，构建一个测试二进制文件并运行它。

更多详情见如何使用Go编程文档、 `testing` 包以及 `go test`子命令。

我最喜欢的测试助手函数在哪?

Go的标准 `testing` 包使编写单元测试更加容易，但是它缺乏在其它语言的测试框架提供的

特性，比如断言函数。在本文档前面的一小节中解释了为什么Go没有断言，同样的论点也适用于在测试中对 assert 的使用。适当的错误处理意味着可以在一项测试失败后让其它测试继续运行，因此调试错误的人能够得到一幅关于错误的完整的画面。比起 isPrime 对于2给出错误答案的报告后就不再运行更多测试来说，isPrime 对于2、3、5和7（或2、4、8和16）给出错误答案的报告更加有用。触发测试失败的程序员可以无需熟悉失败的代码。

一个相关的问题是，测试框架往往会发展成他们自己的带条件测试、流程控制以及打印机制的迷你语言，但Go已经拥有了所有的那些能力，为什么要重新创造它们呢？我们更愿意在Go中写测试，因为它只需学习少量的语言，而这种方式会使测试直截了当且易于理解。

如果需要编写额外的代码量，良好的错误似乎是重复的和压倒一切的，如果通过表格控制，在数据结构中定义的输入输出列表上进行迭代（Go对于数据结构字面有着极好的支持），则该测试可能会工作得更好。编写良好的测试及良好的错误信息的工作会分担很多测试情况。在Go的标准库中充满了说明性的例子，比如说在 fmt 包中的格式化测试。

实现

该编译器使用什么编译器技术构建？

gccgo 拥有一个耦合到标准GCC后端的，带递归下降解析器的C++前端。 Gc 是用C编写的，它使用 yacc/bison 作为解析器。尽管它是个新的程序，但它也适用于Plan 9的C编译器套件（ http://plan9.bell-labs.com/sys/doc/compiler.html）并使用了Plan 9加载程序的一种变体来生成ELF/Mach-O/PE二进制文件。

我们考虑过用Go自身编写官方的Go编译器 gc，但由于自举的困难和开源分布的特殊性，我们决定不这样做——你需要一个Go编译器来设置一个Go的环境。gccgo 出现得稍晚一些，它让考虑使用Go来编写编译器成为了可能，这很有可能发生。（Go是实现编译器的不错的语言；原生的词法分析器和解析器在 go 包中已经可用，类型检查器正在开发。）

我们也考虑过为 gc 使用LLVM，但我们认为它过于庞大且慢得难以满足我们的性能目标。

运行时如何支持实现？

还是因为自举的问题，运行时代码大部分以C编写（以及一丁点汇编），尽管现在Go已经能实现它的大部分功能。gccgo 的运行时使用 glibc 支持。gc 使用了一个定制的库以保证它的封装在控制之下；它使用了Plan 9 C编译器的一个为Go程支持分段栈的版本进行编译。gccgo 编译器只在Linux上实现了分段栈，由gold连接器最近的修改所支持。

为什么我琐碎的程序其二进制文件却如此之大？

gc工具链（5l、6l 以及 8l）中的连接器做静态链接。因此所有的Go二进制文件都包括了Go运行时，连同运行时类型信息必须支持的动态类型检测、反射甚至恐慌时栈跟踪。

一个简单的C“hello, world”程序在Linux上使用gcc静态地编译并连接后大约有750KB，包括一个 printf 的实现。一个使用 fmt.Printf 的等价的Go程序大约有1.2MB，但它包含更多强大的运行时支持。

我能否停止关于我未使用变量/导入的抱怨？

未使用变量的存在可能预示着bug，而未使用的导入只会减慢编译速度。在你的代码树中积累太多的未使用导入可能会使事情变得非常慢。由于这些原因，Go都不允许它们出现。

在开发代码时，临时创建这些状况很常见，而在程序编译之前必须将它们编辑掉是很烦人的。

有些人要求加入一个编译器选项来关闭这些检查，或至少减少那些警告。然而，这样的选项还未被添加，因为编译器选项不能影响到语言的语义，而且Go编译器并不报告警告，只会报告错误来防止编译。

没有警告的理由有两个。其一，若它值得抱怨，也就值得在代码中修复它。（而如果它不值得修复，也就没必要提到。）其二，让编译器产生警告会鼓励实现就微弱的情况产生警告，这会使编译器变得嘈杂，从而掩盖那些*需要*被修复的真正的错误。

尽管解说这些情况是容易的，然而可以使用空白标识符来让未使用的东西在你的开发中存在一会儿。

```
import "unused"

// 此声明标记了从包中导入的被引用的项。
var _ = unused.Item  // TODO: 在提交前删除它!

func main() {
    debugData := debug.Profile()
    _ = debugData // 只在调试时使用
    ....
}
```

现在，大多数 Go 程序员会使用 goimports 工具，它能自动重写 Go 源文件使其拥有正确的导入，消除实践中的未使用导入问题。此程序很容易连接到大多数编辑器，使其在保存 Go 源文件时自动运行。

性能

为什么Go在基准测试X中表现很差?

Go的设计目标之一就是在可比较的程序上逼近C的性能，然而在一些基准测试中它的表现确实很差，包括几项test/bench/shootout中的测试。最慢的依赖库对于可比较性能的版本来说在Go中并不可用。例如 pidigits.go 依赖于一个多精度的数学包，而C版本的则使用GMP（它使用优化的汇编编写的）。依赖于正则表达式的基准测试（例如regex-dna.go）在本质上是将Go的原生regexp包与像PCRE那样成熟的，高度优化的正则表达式库相比较。

虽然基准测试游戏通过广泛的调优赢了，但大部分Go版本的基准测试还需要关注。如果你考量了C和Go的可比较程序（reverse-complement.go是其中一个例子），你就会发现这两种语言在这个套件上表明的原始性能非常接近。

不过，它还有提升的空间。编译器很好，但可以变得更好，一些库需要主要的性能工作，且垃圾回收器也还不够快。（即使这样，也要小心不要产生不必要的垃圾，否则会有巨大的影响。）

在任何情况下，Go都是非常有竞争力的。用该语言及工具开发的许多软件在性能上都有着明显的改善。请参阅博文Go程序性能分析 了解一个有益的例子。

对于C的改变

为什么它的语法和C如此不同?

除了声明语法外，它们之间的不同并不多，这主要源于两种需求。首先，语法应当感觉很轻量，没有太多强制性的关键字、重复或奥秘。其次，该语言被设计成易于分析的，无需符号表来解析。这会使构建诸如调试器、依赖分析器、自动文档提取器、IDE插件等工具变得更加容易。 C及其后代在这方面上是极其困难的。

为什么声明在后面?

如果你习惯于C，对你来说它们只是在后面而已。在C中，它的概念就像用表示它类型的表达式来声明变量。这是个好主意，不过类型和表达式的语法不要混合得太好，而结果会使人迷惑；考虑函数指针。 Go将表达式和类型语法大部分分离开，并简化了一些东西（对指针使用 * 前缀是检验该规则的例外）。在C中，声明

```
int* a, b;
```

会将 a 声明为指针，而 b 则不会；而在Go中

```
var a, b *int
```

会将二者都声明为指针。这样更清楚也更规则。另外，:= 短变量声明形式证明一个完整的变量声明应当以 := 呈现相同的顺序，因此

```
var a uint64 = 1
```

的效果等同于

```
a := uint64(1)
```

解析也通过拥有一个独特的，不只是表达式的类型语法而得到了简化，像 func 和 chan 这样的关键字让事情变得清晰。

更多详情请参阅Go的声明语法。

为什么没有指针运算?

为了安全。没有指针运算可创建一种语言，它不会派生出可以错误地成功访问的非法地址。编译器和硬件技术已经发展到了循环使用数组下标比循环使用指针运算更有效率的地步。此外，指针运算的缺失还可以简化垃圾收集器的实现。

为什么 ++ 与 -- 语句不是表达式？为什么只有后缀式而没有前缀式？

没有指针运算，前缀和后缀增量操作符的便利性就会减少。通过将它们从表达式层级中整体移除，表达式语法就会简化，而围绕 ++ 和 -- 求值顺序混乱的问题（考虑 `f(i++)` 和 `p[i] = q[++i]`）就能被很好地消除。这种简化是非常有意义的。至于后缀式与前缀式，二者都能很好地工作，但后缀式版本更加传统；前缀式则是STL所坚持的。具有讽刺意味的是，具有讽刺意味的是，它是为某种名字里包含后缀增量的语言写的。

为什么有大括号却没有分号？为什么我不能将开大括号放在下一行？

Go使用大括号为语句进行分组，这种语法对于使用C家族中任何语言工作的程序员来说是很熟悉的。然而，分号是为了解析器，而不是人们，而我们想要尽可能地消除它。为实现这个目标， Go从BCPL里借鉴了一个小诡计：用来分隔语句的分号还在正式的语法中，但词法分析器将所有行末都当做语句的结束，并自动插入分号，而无需前瞻。这种做法在实践中非常好，不过副作用就是强制的大括号风格。例如，函数的开大括号不能单独占据一行。

一些人争论词法分析器应当前瞻性地允许大括号占据下一行。而我们不这么认为。由于Go代码会自动地被`gofmt` 格式化， **一些**风格就必须被选择。那种风格可能不同于你在C或Java中使用的风格，但Go是一门新的语言，而且 `gofmt` 的风格比任何其它的风格都要好。更重要——还要重要的是，对于所有的Go程序来说，单一的、程序化的、强制性格式的优势，比任何独有的风格的劣势都更加好。还需要注意的是， Go的风格意味着Go的交互式实现可以使用标准的的语法，一次一行而无需特殊的规则。

为什么要有垃圾回收？代价会不会太高？

记账式系统编程的最大来源就是内存管理。我们觉得关键就在于消除程序员的开销，而垃圾回收技术的进步给了我们以足够低的开销和没有明显的延迟来实现它的信心。

另一点是并发和多线程编程的一大部分困难也源于内存管理；在线程之间传递的对象要保证它们被安全地释放是很麻烦的。自动垃圾回收使并发代码很容易编写。当然，在并发环境中实现垃圾回收本身也是一个挑战，但比起在每个程序中实现它来说，只需实现它一次就能帮助到每一个人。

最后，撇开并发不说，垃圾回收使接口更简单，因为它们无需指定内存该如何管理。

当前实现是并行的标记并清理试收集器，但将来的版本可能会使用不同的方法。