GO命令教程

极客学院出版

前言

作者介绍

郝林，10年软件开发从业经验。搞过银行、电信软件和互联网社交等产品。对Go语言和Docker都情有独钟。现在正在从事互联网软件基础组件的构建以及Go语言的技术推广和社区运营等工作。他也是图灵原创技术图书《Go并发编程实战》和在线免费教程《Go语言第一课》的作者。

本教程的由来

这份Go命令教程原先是我著的图书《Go并发编程实战》中的一部分。这部分内容与并发编程的关系不大，故被砍掉。但是它是有价值的，也算是我对Go语言官方提供的标准命令的一个学习笔记。所以，我觉得应该把它做成免费资源分享给大家。经出版社的认可，我将这份教程放在这里供广大Go语言爱好者阅读。

相关源

本教程在Github上的地址在这里。如果你喜欢本教程，请不吝Star。如果你想贡献一份力量，欢迎提交Pull Request。

针对Go 1.3和1.4版本的教程已被放入分支go1.3。而主分支此后一段时间内会致力于针对Go 1.5进行更新。

本教程中会提及一个名为 `goc2p` 的项目。该项目实际上是《Go并发编程实战》随书附带的示例项目。这本书中讲到的所有源码都在 `goc2p` 项目中。我已在《Go并发编程实战》出版之时将 `goc2p` 项目放出。

关于协议

版本信息

书中演示代码基于以下版本：

系统	版本信息
Go	1.3+

目录

1

标准命令详解

Go语言的1.5版本在标准命令方面有了重大变更。这倒不是说它们的用法有多大的变化，而是说它们的底层支持已经大变样了。让我们先来对比一下 `$GOROOT/pkg/tool/<平台相关目录>` 中的内容。以下简称此目录为Go工具目录。

插播：平台相关目录即以_命名的目录，用于存放因特定平台的不同而不同的代码包归档文件或可执行文件。其中，代表特定平台的操作系统代号，而则代表特定平台的计算架构代号。使用 `go env` 命令便可查看它们在你的计算机中的实际值。

1.4版本的Go工具目录的内容如下：

```
5a      5l      6g      8c      addr2line dist      objdump   tour
5c      6a      6l      8g      cgo       fix       pack      vet
5g      6c      8a      8l      cover     nm        pprof     yacc
```

下面是Go 1.5版本的：

```
addr2line asm      compile  dist     fix      nm       pack     tour     vet
api       cgo      cover    doc      link     objdump  pprof    trace    yacc
```

可以看到，1.5版本的目录内容精简了不少。这是因为Go 1.5的编译器、链接器都已经完全用Go语言重写了。而在这之前，它们都是用C语言写的，因此不得不为每类平台编写不同的程序并生成不同的文件。例如，8g、6g和5g分别是gc编译器在x86（32bit）、x86-64（64bit）和ARM计算架构的计算机上的实现程序。相比之下，用Go语言实现的好处就是，编译器和链接器都将是跨平台的了。简要来说，Go 1.5版本的目录中的文件compile即是统一后的编译器，而文件link则是统一后的链接器。

本教程并不会讲解Go语言的编译器、链接器以及其它工具是怎样被编写出来的，并只会关注于怎样用好包含它们在内的Go语言自带的命令和工具。

为了让讲解更具关联性，也为了让读者能够更容易的理解这些命令和工具，本教程并不会按照这些命令的字典顺序描述它们，而会按照我们在实际开发过程中通常的使用顺序以及它们的重要程度来逐一进行说明。现在，我们就先从 `go build` 命令开始。

go build

`go build` 命令用于编译我们指定的源码文件或代码包以及它们的依赖包。

例如，如果我们在执行 `go build` 命令时不后跟任何代码包，那么命令将试图编译当前目录所对应的代码包。例如，我们想编译goc2p项目的代码包 `logging` 。其中一个方法是进入 `logging` 目录并直接执行该命令：

```
hc@ubt:~/golang/goc2p/src/logging$ go build
```

因为在代码包 `logging` 中只有库源码文件和测试源码文件，所以在执行 `go build` 命令之后不会在当前目录和goc2p项目的pkg目录中产生任何文件。

插播：Go语言的源码文件有三大类，即：命令源码文件、库源码文件和测试源码文件。他们的功用各不相同，而写法也各有各的特点。命令源码文件总是作为可执行的程序的入口。库源码文件一般用于集中放置各种待被使用的程序实体（全局常量、全局变量、接口、结构体、函数等等）。而测试源码文件主要用于对前两种源码文件中的程序实体的功能和性能进行测试。另外，后者也可以用于展现前两者中程序的使用方法。

另外一种编译 `logging` 包的方法是：

```
hc@ubt:~/golang/goc2p/src$ go build logging
```

在这里，我们把代码包 `logging` 的导入路径作为参数传递给 `go build` 命令。另一个例子：如果我们要编译代码包 `cnet/ctcp` ，只需要在任意目录下执行命令 `go build cnet/ctcp` 即可。

插播：之所以这样的编译方法可以正常执行，是因为我们已经在环境变量 `GOPATH` 中加入了goc2p项目的根目录（即 `~/golang/goc2p/` ）。这时，goc2p项目的根目录就成为了一个工作区目录。只有这样，Go语言才能正确识别我们提供的goc2p项目中某个代码包的导入路径。而代码包的导入路径是指，相对于Go语言自身的源码目录（即 `$GOROOT/src` ）或我们在环境变量 `GOPATH` 中指定的某个目录的 `src` 子目录下的子路径。例如，这里的代码包 `logging` 的绝对路径是 `~/golang/goc2p/src/logging` 。而不论goc2p项目的根文件夹被放在哪儿， `logging` 包的导入路径都是 `logging` 。显而易见，我们在称呼一个代码包的时候总是以其导入路径作为其称谓。

言归正传，除了上面的简单用法，我们还可以同时编译多个Go源码文件：

```
hc@ubt:~/golang/goc2p/src$ go build logging/base.go logging/console_logger.go logging/log_manager.go logging/tag.go
```

但是，使用这种方法会有一个限制。作为参数的多个Go源码文件必须在同一个目录中。也就是说，如果我们想用这种方法既编译 `logging` 包又编译 `basic` 包是不可能的。不过别担心，在需要的时候，那些被编译目标依赖的代码包会被 `go build` 命令自动的编译。例如，如果有一个导入路径为 `app` 的代码包，同时依赖了 `logging` 包和 `basic` 包。那么在执行 `go build app` 的时候，该命令就会自动的在编译 `app` 包之前去检查 `logging` 包和 `basic` 包

的编译状态。如果发现它们的编译结果文件不是最新的，那么该命令就会先去的编译这两个代码包，然后再编译 app 包。

注意， go build 命令在编译只包含库源码文件的代码包（或者同时编译多个代码包）时，只会做检查性的编译，而不会输出任何结果文件。

另外， go build 命令既不能编译包含多个命令源码文件的代码包，也不能同时编译多个命令源码文件。因为，如果把多个命令源码文件作为一个整体看待，那么每个文件中的main函数就属于重名函数，在编译时会抛出重复定义错误。假如，在goc2p项目的代码包 cmd （此代码包仅用于示例目的，并不会永久存在于该项目中）中包含有两个命令源码文件showds.go和initpkg_demo.go，那么我们在使用 go build 命令同时编译它们时就会失败。示例如下：

```
hc@ubt:~/golang/goc2p/src/cmd$ go build showds.go initpkg_demo.go
 # command-line-arguments

./initpkg_demo.go:19: main redeclared in this block
        previous declaration at ./showds.go:56
```

请注意上面示例中的 command-line-arguments 。在这个位置上应该显示的是作为编译目标的源码文件所属的代码包的导入路径。但是，这里显示的并不是它们所属的代码包的导入路径 cmd 。这是因为，命令程序在分析参数的时候如果发现第一个参数是Go源码文件而不是代码包，则会在内部生成一个虚拟代码包。这个虚拟代码包的导入路径和名称都会是 command-line-arguments 。在其他基于编译流程的命令程序中也有与之一致的操作，比如 go install 命令和 go run 命令。

另一方面，如果我们编译的多个属于 main 包的源码文件中没有 main 函数的声明，那么就会使编译器立即报出“未定义 main 函数声明”的错误并中止编译。换句话说，在我们同时编译多个 main 包的源码文件时，要保证其中有且仅有一个 main 函数声明，否则编译是无法成功的。

现在我们使用 go build 命令编译单一命令源码文件。我们在执行命令时加入一个标记 -v 。这个标记的意义在于可以使命令把执行过程中构建的包名打印出来。我们会在稍后对这个标记进行详细说明。现在我们先来看一个示例：

```
hc@ubt:~/golang/goc2p/src/basic/pkginit$ ls
initpkg_demo.go
hc@ubt:~/golang/goc2p/src/basic/pkginit$ go build -v initpkg_demo.go
command-line-arguments
hc@ubt:~/golang/goc2p/src/basic/pkginit$ ls
initpkg_demo  initpkg_demo.go
```

我们在执行命令 go build -v initpkg_demo.go 之后被打印出的 command-line-arguments ”`就是命令程序为命令源码文件initpkg_demo.go生成的虚拟代码包的包名。顺带说一句，

命令 go build 会把编译命令源码文件后生成的结果文件存放到执行该命令时所在的目录下。这个所说的结果文件就是与命令源码文件对应的可执行文件。它的名称会与命令源码文件的主文件名相同。

顺便说一下，如果我们有多个声明为属于 main 包的源码文件，且其中只有一个文件声明了 main 函数的话，那么是可以使用 go build 命令同时编译它们的。在这种情况下，不包含 main 函数声明的那几个源码文件会被视为库源码文件（理所当然）。如此编译之后的结果文件的名称将会与我们指定的编译目标中最左边的那个源码文件的主文件名相同。

其实，除了让Go语言编译器自行决定可执行文件的名称，我们还可以自定义它。示例如下：

```
hc@ubt:~/golang/goc2p/src/basic/pkginit$ go build -o initpkg initpkg_demo.go
hc@ubt:~/golang/goc2p/src/basic/pkginit$ ls
initpkg    initpkg_demo.go
```

使用 -o 标记可以指定输出文件（在这个示例中指的是可执行文件）的名称。它是最常用的一个 go build 命令标记。但需要注意的是，当使用标记 -o 的时候，不能同时对多个代码包进行编译。

标记 -i 会使 go build 命令安装那些编译目标依赖的且还未被安装的代码包。这里的安装意味着产生与代码包对应的归档文件，并将其放置到当前工作区目录的 pkg 子目录的相应子目录中。在默认情况下，这些代码包是不会被安装的。

除此之外，还有一些标记不但受到 go build 命令的支持，而且对于后面会提到的 go install 、 go run 、 go test 等命令同样是有效的。下表列出了其中比较常用的标记。

表0-1 go build 命令的常用标记说明

标记名称	标记描述
-a	强行对所有涉及到的代码包（包含标准库中的代码包）进行重新构建，即使它们已经是最新的了。
-n	打印编译期间所用到的其它命令，但是并不真正执行它们。
-p n	指定编译过程中执行各任务的并行数量（确切地说应该是并发数量）。在默认情况下，该数量等于CPU的逻辑核数。但是在 darwin/arm 平台（即iPhone和iPad所用的平台）下，该数量默认是 1 。
-race	开启竞态条件的检测。不过此标记目前仅在 linux/amd64 、 freebsd/amd64 、 darwin/amd64 和 windows/amd64 平台下受到支持。
-v	打印出那些被编译的代码包的名字。
-work	打印出编译时生成的临时工作目录的路径，并在编译结束时保留它。在默认情况下，编译结束时会删除该目录。
-x	打印编译期间所用到的其它命令。注意它与 -n 标记的区别。

我们在这里忽略了一些并不常用的或作用于编译器或连接器的标记。在本小节的最后将会对这些标记进行简单的说明。如果读者有兴趣，也可以查看Go语言的官方文档以获取相关信息。

下面我们就用其中几个标记来查看一下在构建代码包 logging 时创建的临时工作目录的路径：

```
hc@ubt:~/golang/goc2p/src$ go build -v -work logging
WORK=/tmp/go-build888760008
logging
```

上面命令的结果输出的第一行是为了编译 logging 包，Go创建的一个临时工作目录，这个目录被创建到了Linux的临时目录下。输出的第二行是对标记 -v 的响应。这意味着此次命令执行时仅编译了 logging 包。关于临时工作目录的用途和内容，我们会在讲解 go run 命令和 go test 命令的时候详细说明。

现在我们再来看看如果强制重新编译会涉及到哪些代码包：

```
hc@ubt:~/golang/goc2p/src$ go build -a -v -work logging
WORK=/tmp/go-build929017331
runtime
errors
sync/atomic
math
unicode/utf8
unicode
sync
io
syscall
strings
time
strconv
reflect
os
fmt
log
logging
```

怎么会多编译了这么多代码包呢？可以确定的是，代码包 logging 中的代码直接依赖了标准库中的 runtime 包、 strings 包、 fmt 包和 log 包。那么其他的代码包为什么也会被重新编译呢？

从代码包编译的角度来说，如果代码包A依赖代码包B，则称代码包B是代码包A的依赖代码包（以下简称依赖包），代码包A是代码包B的触发代码包（以下简称触发包）。

go build 命令在执行时，编译程序会先查找目标代码包的所有依赖包，以及这些依赖包的依赖包，直至找到最深层的依赖包为止。在此过程中，如果发现有循环依赖的情况，编译程序就会输出错误信息并立即退出。此过程完成之后，所有的依赖关系也就形成了一棵含有重复元素的依赖树。对于依赖树中的一个节点（代码包）来说，它的直接分支节点（前者的依赖包），是按照代码包导入路径的字典序从左到右排列的。最左边的分支节点会最先被编译。编译程序会依此设定每个代码包的编译优先级。

执行 go build 命令的计算机如果拥有多个逻辑CPU核心，那么编译代码包的顺序可能会存在一些不确定性。但是，它一定会满足这样的约束条件： 依赖代码包 -> 当前代码包 -> 触发代码包 。

标记 -p n 可以限制编译过程中任务执行的并发数量， n 默认为当前计算机的CPU逻辑核数。如果在执行 go build 命令时加入标记 -p 1 ，那么就可以保证代码包编译顺序严格按照预先设定好的优先级进行。现在我们再来编译 logging 包：

```
hc@ubt:~/golang/goc2p/src$ go build -a -v -work -p 1 logging
WORK=/tmp/go-build114039681
runtime
errors
sync/atomic
sync
io
math
syscall
time
os
unicode/utf8
strconv
reflect
fmt
log
unicode
strings
logging
```

我们可以认为，以上示例中所显示的代码包的顺序，就是 logging 包直接或间接依赖的代码包按照优先级从高到低排列后的排序。

另外，如果在命令中加入标记 -n ，那么编译程序只会输出所用到的命令而不会真正运行。在这种情况下，编译过程不会使用并发模式。

在本节的最后，我们对一些并不太常用的标记进行简要的说明：

- -asmflags

此标记可以后跟另外一些标记，如 -D 、 -I 、 -S 等。这些后跟的标记用于控制Go语言编译器编译汇编语言文件时的行为。

- -buildmode

此标记用于指定编译模式，使用方式如 -buildmode=default （这等同于默认情况下的设置）。此标记支持的编译模式目前有6种。借此，我们可以控制编译器在编译完成后生成静态链接库（即.a文件，也就是我们之前说的归档文件）、动态链接库（即.so文件）或/和可执行文件（在Windows下是.exe文件）。

- -compiler

此标记用于指定当前使用的编译器的名称。其值可以为 gc 或 gccgo 。其中，gc编译器即为Go语言自带的编辑器，而gccgo编译器则为GCC提供的Go语言编译器。而GCC则是GNU项目出品的编译器套件。GNU是一个众所周知的自由软件项目。在开源软件界不应该有人不知道它。好吧，如果你确实不知道它，赶紧去google吧。

- -gccgoflags

此标记用于指定需要传递给gccgo编译器或链接器的标记的列表。

- -gcflags

此标记用于指定需要传递给 go tool compile 命令的标记的列表。

- -installsuffix

为了使当前的输出目录与默认的编译输出目录分离，可以使用这个标记。此标记的值会作为结果文件的父目录名称的后缀。其实，如果使用了 -race 标记，这个标记会被自动追加且其值会为 race 。如果我们同时使用了 -race 标记和 -installsuffix ，那么在 -installsuffix 标记的值的后面会再被追加 _race ，并以此来作为实际使用的后缀。

- -ldflags

此标记用于指定需要传递给 go tool link 命令的标记的列表。

- -linkshared

此标记用于与 -buildmode=shared 一同使用。后者会使作为编译目标的非 main 代码包都被合并到一个动态链接库文件中，而前者则会在此之上进行链接操作。

- -pkgdir

使用此标记可以指定一个目录。编译器会只从该目录中加载代码包的归档文件，并会把编译可能会生成的代码包归档文件放置在该目录下。

- -tags

此标记用于指定在实际编译期间需要受理的编译标签（也可被称为编译约束）的列表。这些编译标签一般会作为源码文件开始处的注释的一部分，例如，在 $GOROOT/src/os/file_posix.go 开始处的注释为：

```
// Copyright 2009 The Go Authors. All rights reserved.
// Use of this source code is governed by a BSD-style
// license that can be found in the LICENSE file.

// +build darwin dragonfly freebsd linux nacl netbsd openbsd solaris windows
```

最后一行注释即包含了与编译标签有关的内容。大家可以查看代码包 go/build 的文档已获得更多的关于编译标签的信息。

- -toolexec

此标记可以让我们去自定义在编译期间使用一些Go语言自带工具（如 vet 、 asm 等）的方式。

go install

命令 go install 用于编译并安装指定的代码包及它们的依赖包。当指定的代码包的依赖包还没有被编译和安装时，该命令会先去处理依赖包。与 go build 命令一样，传给 go install 命令的代码包参数，应该以导入路径的形式提供。并且， go build 命令的绝大多数标记也都可以用于 go install 命令。实际上， go install 命令只比 go build 命令多做了一件事，即：安装编译后的结果文件到指定目录。

在对 go install 命令进行详细说明之前，让我们先回顾一下goc2p的目录结构。为了节省篇幅，我们在这里隐藏了代码包中的源码文件。如下：

```
$HOME/golang/goc2p:
    bin/
    pkg/
    src/
        cnet/
        logging/
        helper/
            ds/
        pkgtool/
```

我们看到，goc2p项目中有三个子目录，分别是bin目录、pkg目录和src目录。现在只有src目录中包含了一些目录，而其他两个目录都是空的。

现在，我们来看看安装代码包的规则。

安装代码包

如果 go install 命令后跟的代码包中仅包含库源码文件，那么 go install 命令会把编译后的结果文件保存在源码文件所在工作区的pkg目录下。对于仅包含库源码文件的代码包来说，这个结果文件就是对应的代码包归档文件。相比之下，我们在使用 go build 命令对仅包含库源码文件的代码包进行编译时，是不会在当前工作区的src目录和pkg目录下产生任何结果文件的。结果文件会出于编译的目的被生成在临时目录中，但并不会对当前工作区目录产生任何影响。

如果我们在执行 go install 命令时不后跟任何代码包参数，那么命令将试图编译当前目录所对应的代码包。比如，我们现在要安装代码包 pkgtool ：

```
hc@ubt:~/golang/goc2p/src/pkgtool$ go install -v -work
WORK=D:\cygwin\tmp\go-build758586887
pkgtool
```

我们刚刚说过，执行 go install 命令后会对指定代码包先编译再安装。其中，编译代码包使用了与 go build 命令相同的程序。所以，执行 go install 命令后也会首先建立一个名称以go-build为前缀的临时目录。如果我们想强行重新安装指定代码包及其依赖包，那么就需要加入标记 -a：

```
hc@ubt:~/golang/goc2p/src/pkgtool$ go install -a -v -work
WORK=/tmp/go-build014992994
runtime
errors
sync/atomic
unicode
unicode/utf8
sort
sync
io
syscall
bytes
strings
time
bufio
os
path/filepath
pkgtool
```

可以看到，代码包 pkgtool 仅仅依赖了标准库中的代码包。

现在我们再来查看一下goc2p项目目录：

```
$HOME/golang/goc2p:
    bin/
     pkg/
        linux_386/
            pkgtool.a
    src/
```

现在pkg目录中多了一个子目录。读过本书第二章的读者应该已经知道，linux_386被叫做平台相关目录。它的名字可以由 ${GOOS}_${GOARCH} 来得到。其中， ${GOOS} 和 ${GOARCH} 分别是环境变量GOOS和GOARCH的值。上述示例在计算架构为386（ ）且操作系统为Linux的计算机上运行。所以，这里的平台相关目录即为linux_386。我们还看到，在goc2p项目中的平台相关目录下存在一个文件，名称是pkgtool.a。这就是代码包 pkgtool 的归档文件，文件名称是由代码包名称与“.a”后缀组合而来的。

实际上，代码包的归档文件并不都会被保存在pkg目录的平台相关目录下，还可能被保存在这个平台相关目录的子目录下。 下面我们来安装 cnet/ctcp 包：

```
hc@ubt:~/golang/goc2p/src/pkgtool$ go install -a -v -work ../cnet/ctcp
WORK=/tmp/go-build083178213
runtime
errors
sync/atomic
unicode
unicode/utf8
math
sort
sync
io
syscall
bytes
strings
bufio
time
strconv
math/rand
os
reflect
fmt
log
runtime/cgo
logging
net
cnet/ctcp
```

请注意，我们是在代码包 pkgtool 对应的目录下安装 cnet/ctcp 包的。我们使用了一个目录相对路径。

实际上，这种提供代码包位置的方式被叫做本地代码包路径方式，也是被所有Go命令接受的一种方式，这包括之前已经介绍过的 go build 命令。但是需要注意的是，本地代码包路径只能以目录相对路径的形式呈现，而不能使用目录绝对路径。请看下面的示例：

```
hc@ubt:~/golang/goc2p/src/cnet/ctcp$ go install -v -work ~/golang/goc2p/src
   /cnet/ctcp
can't load package: package /home/hc/golang/goc2p/src/cnet/ctcp:
   import "/home/hc/golang/goc2p/src/cnet/ctcp": cannot import absolute path
```

从上述示例中的命令提示信息我们可以看到，以目录绝对路径的形式提供代码包位置是不会被Go命令认可的。

这是由于Go认为本地代码包路径的表示只能以“./”或“../”开始，再或者直接为“.”或“..”。而代码包的代码导入路径又不允许以“/”开始。所以，这种用绝对路径表示代码包位置的方式也就不被支持了。

上述规则适用于所有Go命令。读者可以自己尝试一下，比如在执行 go build 命令时分别以代码包导入路径、目录相对路径和目录绝对路径的形式提供代码包位置，并查看执行结果。

我们已经通过上面的示例强行的重新安装了 cnet/ctcp 包及其依赖包。现在我们再来看一下goc2p的项目目录：

```
$HOME/golang/goc2p:
   bin/
   pkg/
      linux_386/
         /cnet
            ctcp.a
         logging.a
         pkgtool.a
   src/
```

我们发现在pkg目录的平台相关目录下多了一个名为cnet的目录，而在这个目录下的就是名为ctcp.a的代码包归档文件。由此我们可知，代码包归档文件的存放路径的相对路径（相对于当前项目的pkg目录的平台相关目录）即为代码包导入路径除去最后一个元素后的路径。而代码包归档文件的名称即为代码包导入路径中的最后一个元素再加“.a”后缀。再举一个例子，如果代码包导入路径为x/y/z，则它的归档文件存放路径的相对路径即为x/y/，而这个归档文件的名称即为z.a。

现在来看代码包 pkgtool 的归档文件的存放路径。因为它的导入路径中只有一个元素，所以其归档文件就被直接存放到了goc2p项目的pkg目录的平台相关目录下。

此外，我们还发现pkg目录的平台相关目录下还有一个名为logging.a的文件。很显然，我们并没有显式的安装代码包 logging 。这是怎么回事呢？这是因为 go install 命令在安装指定的代码包之前，会先去安装指定代码包的依赖包。当依赖包被正确安装后，指定的代码包的安装才会开始。由于代码包 cnet/ctcp 依赖于代码包 loggin g ，所以当代码包 logging 被成功安装之后，代码包 cnet/ctcp 才会被安装。

还有一个问题：上述的安装过程涉及到了那么多代码包，那为什么goc2p项目的pkg目录中只包含该项目中代码包的归档文件呢？实际上， go install 命令会把标准库中的代码包的归档文件存放到Go根目录的pkg目录中，而把指定代码包依赖的第三方项目的代码包的归档文件存放到那个项目的pkg目录下。这样就实现了Go语言标准库代码包的归档文件与用户代码包的归档文件，以及处在不同工作区的用户代码包的归档文件之间的彻底分离。

安装命令源码文件

除了安装代码包之外， go install 命令还可以安装命令源码文件。为了看到安装命令源码文件是goc2p项目目录的变化，我们先把该目录还原到原始状态，即清除bin子目录和pkg子目录下的所有目录和文件。然后，我们来安装代码包 helper/ds 下的命令源码文件showds.go，如下：

```
hc@ubt:~/golang/goc2p/src$ go install helper/ds/showds.go
go install: no install location for directory
   /home/hc/golang/goc2p/src/helper/ds outside GOPATH
```

这次我们没能成功安装。该Go命令认为目录/home/hc/golang/goc2p/src/helper/ds不在环境GOPATH中。我们可以通过Linux的 echo 命令来查看一下环境变量GOPATH的值：

```
hc@ubt:~/golang/goc2p/src$ echo $GOPATH
/home/hc/golang/lib:/home/hc/golang/goc2p
```

环境变量GOPATH的值中确实包含了goc2p项目的根目录。这到底是怎么回事呢?

作者通过查看Go命令的源码文件（$GOROOT/src/go/*.go）找到了其根本原因。在上一小节我们提到过，在环境变量GOPATH中包含多个工作区目录路径时，我们需要在编译命令源码文件前先对环境变量GOBIN进行设置。实际上，这个环境变量所指的目录路径就是命令程序生成的结果文件的存放目录。 go install 命令会把相应的可执行文件放置到这个目录中。

由于命令 go build 在编译库源码文件后不会产生任何结果文件，所以自然也不用会在意结果文件的存放目录。在该命令编译单一的命令源码文件时，在结果文件存放目录无效的情况下会将结果文件（也就是可执行文件）存放到执行该命令时所在的目录下。因此，即使环境变量GOBIN的值无效，我们在执行 go build 命令时也不会见到这个错误提示信息。

然而， go install 命令中一个很重要的步骤就是将结果文件（归档文件或者可执行文件）存放到相应的目录中。所以，命令 go install 在安装命令源码文件时，如果环境变量GOBIN的值无效，则它会在最后检查结果文件存放目录的时候发现这一问题，并打印与上述示例所示内容类似的错误提示信息，最后直接退出。

这个错误提示信息在我们安装多个库源码文件时也有可能遇到。示例如下：

```
hc@ubt:~/golang/goc2p/src/pkgtool$ go install envir.go fpath.go ipath.go pnode.go util.go
go install: no install location for directory /home/hc/golang
    /goc2p/src/pkgtool outside GOPATH
```

而且，在我们为环境变量GOBIN设置了正确的值之后，这个错误提示信息仍然会出现。这是因为，只有在安装命令源码文件的时候，命令程序才会将环境变量GOBIN的值作为结果文件的存放目录。而在安装库源码文件时，在命令程序内部的代表结果文件存放目录路径的变量不会被赋值。最后，命令程序会发现它依然是个无效的空值。所以，命令程序会同样返回一个关于“无安装位置”的错误。这就引出一个结论，我们只能使用安装代码包的方式来安装库源码文件，而不能在 go install 命令罗列并安装它们。另外， go install 命令目前无法接受标记 -o 以自定义结果文件的存放位置。这也从侧面说明了 go install 命令当前还不支持针对库源码文件的安装操作。

单从上述问题来讲，Go工具在执行错误识别及其提示信息的细分方面还没有做到最好。

go get

```
hc@ubt:~$ go get github.com/hyper-carrot/go_lib
```

命令 go get 可以根据要求和实际情况从互联网上下载或更新指定的代码包及其依赖包，并对它们进行编译和安装。在上面这个示例中，我们从著名的代码托管站点Github上下载了一个项目（或称代码包），并安装到了环境变量GOPATH中包含的第一个工作区中。在本机中，这个代码包的导入路径就是github.com/hyper-carrot/go_lib。

一般情况下，为了分离自己与第三方的代码，我们会设置两个及以上的工作区。我们现在新建一个目录路径为~/golang/lib的工作区，并把这个工作区路径作为环境变量GOPATH值中的第一个目录路径。注意，环境变量GOPATH中包含的路径不能与环境变量GOROOT的值重复。如此一来，如果我们再使用 go get 命令下载和安装代码包，那么这些代码包就都会被安装在这个新的工作区中了。我们暂且把这个工作区叫做Lib工作区。假如我们在Lib工作区建立和设置完毕之后运行了上面示例中的命令，那么这个代码包就应该会被保存在Lib工作的src目录下，并且已经被安装妥当，如下所示：

```
$HOME/golang/lib:
    bin/
    pkg/
        linux_386/
            github.com/
            hyper-carrot/
        go_lib.a
    src/
        github.com/
        hyper-carrot/
        go_lib/
    ...
```

从另一个角度来说，如果我们想把一个项目上传到Github站点（或其他代码托管站点）上，并被其他人使用的话，那么我们就应该把这个项目当做一个代码包来看待。我们在之前已经提到过原因， go get 命令会将项目下的所有子目录和源码文件存放到工作区src目录下，而src目录下的所有子目录都会是某个代码包导入路径的一部分或者全部。也就是说，我们应该直接在项目目录下存放子代码包和源码文件，并且直接存放在项目目录下的源码文件所声明的包名应该与该项目名相同（除非它是命令源码文件）。这样做可以让其他人使用 go get 命令从Github站点上下载你的项目之后直接就能使用它。

实际上，像goc2p项目这样直接以项目根目录的路径作为工作区路径的做法是不被推荐的。之所以这样做主要是想让读者更容易的理解Go语言的工程结构和工作区概念，也可以让读者看到另一种项目结构。

** 远程导入路径分析**

实际上， `go get` 命令所做的动作也被叫做代码包远程导入，而传递给该命令的作为代码包导入路径的参数又被叫做代码包远程导入路径。

实际上， `go get` 命令不仅可以从像Github这样著名的代码托管站点上下载代码包，还可以从任何命令支持的代码版本控制系统（英文为Version Control System，简称为VCS）检出代码包。任何代码托管站点都是通过某个或某些代码版本控制系统来提供代码上传下载服务的。所以，更严格的讲， `go get` 命令所做的是从代码版本控制系统的远程仓库中检出/更新代码包并对其进行编译和安装。

该命令所支持的VCS的信息如下表：

表0-2 `go get` 命令支持的VCS

名称	主命令	说明
Mercurial	hg	Mercurial是一种轻量级分布式版本控制系统，采用Python语言实现，易于学习和使用，扩展性强。
Git	git	Git最开始是Linux Torvalds为了帮助管理 Linux 内核开发而开发的一个开源的分布式版本控制软件。但现在已被广泛使用。它是被用来进行有效、高速的各种规模项目的版本管理。
Subversion	svn	Subversion是一个版本控制系统，也是第一个将分支概念和功能纳入到版本控制模型的系统。但相对于Git和Mercurial而言，它只算是传统版本控制系统的一员。
Bazaar	bzr	Bazaar是一个开源的分布式版本控制系统。但相比而言，用它来作为VCS的项目并不多。

`go get` 命令在检出代码包之前必须要知道代码包远程导入路径所对应的版本控制系统和远程仓库的URL。

如果该代码包在本地工作区中已经存在，则会直接通过分析其路径来确定这几项信息。 `go get` 命令支持的几个版本控制系统都有一个共同点，那就是会在检出的项目目录中存放一个元数据目录，名称为“.”前缀加其主命令名。例如，Git会在检出的项目目录中加入名为“.git”的子目录。所以，这样就很容易判定代码包所用的版本控制系统。另外，又由于代码包已经存在，我们只需通过代码版本控制系统的更新命令来更新代码包，因此也就不需要知道其远程仓库的URL了。对于已存在于本地工作区的代码包，除非要求更新代码包，否则 `go get` 命令不会进行重复下载。如果想要求更新代码包，可以在执行 `go get` 命令时加入 `-u` 标记。这一标记会稍后介绍。

如果本地工作区中不存在该代码包，那么就只能通过对代码包远程导入路径进行分析来获取相关信息了。首先， `go get` 命令会对代码包远程导入路径进行静态分析。为了使分析过程更加方便快捷， `go get` 命令程序中已经预置了著名代码托管站点的信息。如下表：

表0-3 预置的代码托管站点的信息

名称	主域名	支持的VCS	代码包远程导入路径示例

Bitbucket	bitbucket.org	Git, Mercurial	bitbucket.org/user/project bitbucket.org/user/project/sub/directory
GitHub	github.com	Git	github.com/user/project github.com/user/project/sub/directory
Google Code	code.google.com	Git, Mercurial, Subversion	code.google.com/p/project code.google.com/p/project/sub/directory code.google.com/p/project.subrepository code.google.com/p/project.subrepository/sub/directory
Launchpad	launchpad.net	Bazaar	launchpad.net/project launchpad.net/project/series launchpad.net/project/series/sub/directory launchpad.net/~user/project/branch launchpad.net/~user/project/branch/sub/directory

一般情况下，代码包远程导入路径中的第一个元素就是代码托管站点的主域名。在静态分析的时候，`go get`命令会将代码包远程导入路径与预置的代码托管站点的主域名进行匹配。如果匹配成功，则在对代码包远程导入路径的初步检查后返回正常的返回值或错误信息。如果匹配不成功，则会再对代码包远程导入路径进行动态分析。至于动态分析的过程，我们就不在这里赘述了。

如果对代码包远程导入路径的静态分析或/和动态分析成功并获取到对应的版本控制系统和远程仓库URL，那么`go get`命令就会进行代码包检出或更新的操作。随后，`go get`命令会在必要时以同样的方式检出或更新这个代码包的所有依赖包。

命令特有标记

命令`go get`可以接受所有可用于`go build`命令和`go install`命令的标记。这是因为`go get`命令的内部步骤中完全包含了编译和安装这两个动作。另外，`go get`命令还有一些特有的标记，如下表所示：

表0-4 `go get`命令的特有标记说明

标记名称	标记描述
-d	让命令只执行下载动作，而不执行安装动作。
-fix	让命令在下载代码包后先执行修正动作，而后再进行编译和安装。
-u	让命令利用网络来更新已有代码包及其依赖包。默认情况下，该命令只会从网络上下载本地不存在的代码包，而不会更新已有的代码包。

为了更好的理解这几个特有标记，我们先清除Lib工作区的src目录和pkg目录中的所有子目录和文件。现在我们使用带有`-d`标记的`go get`命令来下载同样的代码包：

```
hc@ubt:~$ go get -d github.com/hyper-carrot/go_lib
```

现在，让我们再来看一下Lib工作区的目录结构：

```
$HOME/golang/lib:
   bin/
   pkg/
   src/
      github.com/
      hyper-carrot/
      go_lib/
   ...
```

我们可以看到， go get 命令只将代码包下载到了Lib工作区（环境变量GOPATH中的第一个目录）的src目录，而没有进行后续的编译和安装动作。

我们知道，绝大多数计算机编程语言在进行升级和演进过程中，不可能保证100%的向后兼容（Backward Compatibility）。在计算机世界中，向后兼容是指在一个程序或者代码库在更新到较新的版本后，用旧的版本程序创建的软件和系统仍能被正常操作或使用，或在旧版本的代码库的基础上编写的程序仍能正常编译运行的能力。Go语言的开发者们已想到了这点，并提供了官方的代码升级工具—— fix 。 fix 工具可以修复因Go语言规范变更而造成的语法级别的错误。关于fix工具，我们将放在本节的稍后位置予以说明。

假设我们本机安装的Go语言版本是1.3，但我们的程序需要用到一个很早之前用Go语言的0.9版本开发的代码包。那么我们在使用 go get 命令的时候可以加入 -fix 标记。这个标记的作用是在检出代码包之后，先对该代码包中不符合Go语言1.3版本的语言规范的语法进行修正，然后再下载它的依赖包，最后再对它们进行编译和安装。

标记 -u 的意图和执行的动作都比较简单。我们在执行 go get 命令时加入 -u 标记就意味着，如果在本地工作区中已存在相关的代码包，那么就是用对应的代码版本控制系统的更新命令更新它，并进行编译和安装。这相当于强行更新指定的代码包及其依赖包。我们来看如下示例：

```
hc@ubt:~$ go get -v github.com/hyper-carrot/go_lib
```

因为我们在之前已经检出并安装了代码包 go_lib ，所以我们执行上面这条命令后什么也没发生。还记得加入标记 -v 标记意味着会打印出被构建的代码包的名字吗？现在我们使用标记 -u 来强行更新代码包：

```
hc@ubt:~$ go get -v -u  github.com/hyper-carrot/go_lib
github.com/hyper-carrot/go_lib (download)
github.com/hyper-carrot/go_lib/logging
github.com/hyper-carrot/go_lib
```

其中，带“(download)”后缀意味着命令从远程仓库检出或更新了代码包。从打印出的信息可以看到，`go get`命令先更新了参数指定的已存在于本地工作区的代码包，而后编译了它的唯一依赖包，最后编译了该代码包。我们还可以加上一个`-x`标记，以打印出用到的命令。读者可以自己试用一下它。

智能的下载

命令`go get`还有一个很值得称道的功能。在使用它检出或更新代码包之后，它会寻找与本地已安装Go语言的版本号相对应的标签（tag）或分支（branch）。比如，本机安装Go语言的版本是1.x，那么`go get`命令会在该代码包的远程仓库中寻找名为“go1”的标签或者分支。如果找到指定的标签或者分支，则将本地代码包的版本切换到此标签或者分支。如果没有找到指定的标签或者分支，则将本地代码包的版本切换到主干的最新版本。

前面我们说在执行`go get`命令时也可以加入`-x`标记，这样可以看到`go get`命令执行过程中所使用的所有命令。不知道读者是否已经自己尝试了。下面我们还是以代码包`github.com/hyper-carrot/go_lib`为例，并且通过之前示例中的命令的执行此代码包已经被检出到本地。这时我们再次更新这个代码包：

```
hc@ubt:~$ go get -v -u -x github.com/hyper-carrot/go_lib
github.com/hyper-carrot/go_lib (download)
cd /home/hc/golang/lib/src/github.com/hyper-carrot/go_lib
git fetch
cd /home/hc/golang/lib/src/github.com/hyper-carrot/go_lib
git show-ref
cd /home/hc/golang/lib/src/github.com/hyper-carrot/go_lib
git checkout origin/master
WORK=/tmp/go-build034263530
```

在上述示例中，`go get`命令通过`git fetch`命令将所有远程分支更新到本地，而后有用`git show-ref`命令列出本地和远程仓库中记录的代码包的所有分支和标签。最后，当确定没有名为“go1”的标签或者分支后，`go get`命令使用`git checkout origin/master`命令将代码包的版本切换到主干的最新版本。下面，我们在本地增加一个名为“go1”的标签，看看`go get`命令的执行过程又会发生什么改变：

```
hc@ubt:~$ cd ~/golang/lib/src/github.com/hyper-carrot/go_lib
hc@ubt:~/golang/lib/src/github.com/hyper-carrot/go_lib$ git tag go1
hc@ubt:~$ go get -v -u -x github.com/hyper-carrot/go_lib
github.com/hyper-carrot/go_lib (download)
cd /home/hc/golang/lib/src/github.com/hyper-carrot/go_lib
git fetch
cd /home/hc/golang/lib/src/github.com/hyper-carrot/go_lib
git show-ref
cd /home/hc/golang/lib/src/github.com/hyper-carrot/go_lib
git show-ref tags/go1 origin/go1
cd /home/hc/golang/lib/src/github.com/hyper-carrot/go_lib
git checkout tags/go1
WORK=/tmp/go-build636338114
```

将这两个示例进行对比，我们会很容易发现它们之间的区别。第二个示例的命令执行过程中使用 `git show-ref` 查看所有分支和标签，当发现有匹配的信息又通过 `git show-ref tags/go1 origin/go1` 命令进行精确查找，在确认无误后将本地代码包的版本切换到标签“go1”之上。

命令 `go get` 的这一功能是非常有用的。我们的代码在直接或间接依赖某些同时针对多个Go语言版本开发的代码包时，可以自动的检出其正确的版本。也可以说， `go get` 命令内置了一定的代码包多版本依赖管理的功能。

go clean

执行 go clean 命令会删除掉执行其它命令时产生的一些文件和目录，包括：

1. 在使用 go build 命令时在当前代码包下生成的与包名同名或者与Go源码文件同名的可执行文件。在Windows下，则是与包名同名或者Go源码文件同名且带有“.exe”后缀的文件。
2. 在执行 go test 命令并加入 -c 标记时在当前代码包下生成的以包名加“.test”后缀为名的文件。在Windows下，则是以包名加“.test.exe”后缀为名的文件。我们会在后面专门介绍 go test 命令。
3. 如果执行 go clean 命令时带有标记 -i ，则会同时删除安装（执行 go install 命令）当前代码包时所产生的结果文件。如果当前代码包中只包含库源码文件，则结果文件指的就是在工作区的pkg目录的相应目录下的归档文件。如果当前代码包中只包含一个命令源码文件，则结果文件指的就是在工作区的bin目录下的可执行文件。
4. 还有一些目录和文件是在编译Go或C源码文件时留在相应目录中的。包括：“_obj”和“_test”目录，名称为“_testmain.go”、“test.out”、“build.out”或“a.out”的文件，名称以“.5”、“.6”、“.8”、“.a”、“.o”或“.so”为后缀的文件。这些目录和文件是在执行 go build 命令时生成在临时目录中的。如果你忘记了这个临时目录是怎么回事儿，可以再回顾一下前面关于 go build 命令的介绍。临时目录的名称以 go-build 为前缀。
5. 如果执行 go clean 命令时带有标记 -r ，则还包括当前代码包的所有依赖包的上述目录和文件。

我们再以goc2p项目的 logging 为例。为了能够反复体现每个标记的作用，我们会使用标记 -n 。使用标记 -n 会让命令在执行过程中打印用到的系统命令，但不会真正执行它们。如果想既打印命令又执行命令则需使用标记 -x 。现在我们来试用一下 go clean 命令：

```
hc@ubt:~/golang/goc2p/src$ go clean -x logging
cd /home/hc/golang/goc2p/src/logging
rm -f logging logging.exe logging.test logging.test.exe
```

现在，我们加上标记 -i ：

```
hc@ubt:~/golang/goc2p/src$ go clean -x -i logging
cd /home/hc/golang/goc2p/src/logging
rm -f logging logging.exe logging.test logging.test.exe
rm -f /home/hc/golang/goc2p/pkg/linux_386/logging.a
```

如果再加上标记 -r 又会打印出哪些命令呢？请读者自己试一试吧。

godoc

命令 godoc 是一个很强大的工具，用于展示指定代码包的文档。我们可以通过运行 go get code.google.com/p/go.tools/cmd/godoc 安装它。

该命令有两种模式可供选择。如果在执行命令时不加入 -http 标记，则该命令就以命令行模式运行。在打印纯文本格式的文档到标准输出后，命令执行就结束了。比如，我们用命令行模式查看代码包fmt的文档：

```
hc@ubt:~$ godoc fmt
```

由于篇幅原因，我们在本小节中略去了文档查询结果。读者可以自己运行一下上述命令。在该命令被执行之后，我们就可以看到编排整齐有序的文档内容了。这包括代码包 fmt 的综述和所有可导出成员的声明、文档以及例子。

有时候我们只是想查看某一个函数或者结构体类型的文档，那么我们可以将这个函数或者结构体的名称加入命令的最后面，像这样：

```
hc@ubt:~$ godoc fmt Printf
```

或者：

```
hc@ubt:~$ godoc os File
```

如果我们想同时查看一个代码包中的几个函数的文档，则仅需将函数或者结构体名称追加到命令后面。比如我们要查看代码包 fmt 中函数 Printf 和函数 Println 的文档：

```
hc@ubt:~$ godoc fmt Printf Println
```

如果我们不但想在文档中查看可导出成员的声明，还想看到它们的源码，那么我们可以在执行 godoc 命令的时候加入标记 -src ，比如这样：

```
hc@ubt:~$ godoc -src fmt Printf
```

Go语言为程序使用示例代码设立了专有的规则。我们在这里暂不讨论这个规则的细节。只需要知道正因为有了这个专有规则，使得 godoc 命令可以根据这些规则提取相应的示例代码并把它们加入到对应的文档中。如果我们想在查看代码包 net 中的结构体 Listener 的文档的同时查看关于它的示例代码，那么我们只需要在执行命令时加入标记 -ex 。使用方法如下：

```
hc@ubt:~$ godoc -ex net Listener
```

在实际的Go语言环境中，我们可能会遇到一个命令源码文件所产生的可执行文件与代码包重名的情况。比如本节介绍的命令 go 和官方代码包 go 。现在我们要明确的告诉 godoc 命令要查看可执行文件go的文档，我们需要在名称前加入“cmd/”前缀：

```
hc@ubt:~$ godoc cmd/go
```

另外，如果我们想查看HTML格式的文档，就需要加入标记 -html 。当然，这样在命令行模式下的查看效果是很差的。但是，如果仔细查看的话，可以在其中找到一些相应源码的链接地址。

一般情况下， godoc 命令会去Go语言根目录和环境变量GOPATH的值（一个或多个工作区）指向的工作区目录中查找代码包。不过，我们还可以通过加入标记 -goroot 来制定一个Go语言根目录。这个被指定的Go语言根目录仅被用于当次命令的执行。示例如下：

```
hc@ubt:~$ godoc -goroot="/usr/local/go" fmt
```

现在让我们来看看另外一种模式。如果我们在执行命令时加上 -http 标记则会启用另一模式。这种模式被叫做Web服务器模式，它以Web页面的形式提供Go语言文档。

我们使用如下命令启动这个文档Web服务器：

```
hc@ubt:~/golang/goc2p$ godoc -http=:6060
```

标记 -http 的值 :6060 表示启动的Web服务器使用本机的6060端口。之后，我们就可以通过在网络浏览器的地址栏中输入http://localhost:6060来查看以网页方式展现的Go文档了。

本机的Go文档Web服务首页

图片 1.1 本机的Go文档Web服务首页

图0-1 本机的Go文档Web服务首页

这与Go语言官方站点的Web服务页面如出一辙。这使得我们在不方便访问Go语言官方站点的情况下也可以查看Go语言文档。并且，更便利的是，通过本机的Go文档Web服务，我们还可以查看所有本机工作区下的代码的文档。比如，goc2p项目中的代码包 pkgtool 的页面如下图：

goc2p项目中的pkgtool包的Go文档页面

图片 1.2 goc2p项目中的pkgtool包的Go文档页面

图0-2 goc2p项目中的pkgtool包的Go文档页面

现在，我们在本机开启Go文档Web服务器，端口为9090。命令如下：

```
hc@ubt:~$ godoc -http=:9090 -index
```

注意，要使用 -index 标记开启搜索索引，这个索引会在服务器启动时创建并维护。否则无论在Web页面还是命令行终端中提交查询都会返回错误“Search index disabled: no results available”。

索引中提供了标示符和全文本搜索信息（通过正则表达式为可搜索性提供支持）。全文本搜索结果显示条目的最大数量可以通过标记 -maxresults 提供。标记 -maxresults 默认值是10000。如果不想提供如此多的结果条目，可以设置小一些的值。甚至，如果不想提供全文本搜索结果，可以将标记 -maxresults 的值设置为0，这样服务器就只会创建标识符索引，而根本不会创建全文本搜索索引了。标识符索引即为对程序实体（变量、常量、函数、结构体和接口）名称的索引。

正因为在使用了 -index 标记的情况下文档服务器会在启动时创建索引，所以在文档服务器启动之后还不能立即提供搜索服务，需要稍等片刻。在索引为被创建完毕之前，我们的搜索操作都会得到提示信息“Indexing in progress: result may be inaccurate”。

如果我们在本机用 godoc 命令启动了Go文档Web服务器，且IP地址为192.168.1.4、端口为9090，那么我们就可以在另一个命令行终端甚至另一台能够与本机联通的计算机中通过如下命令进行查询了。查询命令如下：

```
hc@ubt:~$ godoc -q -server="192.168.1.4:9090" Listener
```

命令的最后为要查询的内容，可以是任何你想搜索的字符串，而不仅限于代码包、函数或者结构体的名称。

标记 -q 开启了远程查询的功能。而标记 -server="192.168.1.4:9090" 则指明了远程文档服务器的IP地址和端口号。实际上，如果不指明远程查询服务器的地址，那么该命令会自行将地址“:6060”和“golang.org”作为远程查询服务器的地址。这两个地址即是默认的本机文档Web站点地址和官方的文档Web站点地址。所以执行如下命令我们也可以查询到标准库的信息：

```
hc@ubt:~$ godoc -q=true fmt
```

命令 godoc 还有很多可用的标记，但在通常情况下并不常用。读者如果有兴趣，可以在命令行环境下执行 godoc 进行查看。

至于怎样才能写出优秀的代码包文档，我在《Go并发编程实战》的5.2节中做了详细说明。

go run

在本书第二章中，我们介绍过Go源码文件的分类。Go源码文件包括：命令源码文件、库源码文件和测试源码文件。其中，命令源码文件总应该属于 main 代码包，且在其中有无参数声明、无结果声明的main函数。单个命令源码文件可以被单独编译，也可以被单独安装（需要设置环境变量GOBIN）。当然，命令源码文件也可以被单独运行。我们想要运行命令源码文件就需要使用命令 go run 。

go run 命令可以编译并运行命令源码文件。由于它其中包含了编译动作，因此它也可以接受所有可用于 go build 命令的标记。除了标记之外， go run 命令只接受Go源码文件作为参数，而不接受代码包。与 go build 命令和 go install 命令一样， go run 命令也不允许多个命令源码文件作为参数，即使它们在同一个代码包中也是如此。而原因也是一致的，多个命令源码文件都有相同的main函数声明。

如果命令源码文件可以接受参数，那么在使用 go run 命令运行它的时候就可以把它的参数放在它的文件名后面，像这样：

```
hc@ubt:~/golang/goc2p/src/helper/ds$ go run showds.go -p ~/golang/goc2p
```

在上面的示例中，我们使用 go run 命令运行命令源码文件showds.go。这个命令源码文件可以接受一个名称为“p”的参数。我们用“-p”这种形式表示“p”是一个参数名而不是参数值。它与源码文件名之间需要用空格隔开。参数值会放在参数名的后面，两者成对出现。它们之间也要用空格隔开。如果有第二个参数，那么第二个参数的参数名与第一个参数的参数值之间也要有一个空格。以此类推。

go run 命令只能接受一个命令源码文件以及若干个库源码文件（需同属于 main包 ）作为文件参数，且不能接受测试源码文件。它在执行时会检查源码文件的类型。如果参数中有多个或者没有命令源码文件，那么 go run 命令就只会打印错误提示信息并退出，而不会继续执行。

在通过参数检查后， go run 命令会将编译参数中的命令源码文件，并把编译后的可执行文件存放到临时工作目录中。

编译和运行过程

为了更直观的体现出 go run 命令中的操作步骤，我们在执行命令时加入标记 -n ，用于打印相关命令而不实际执行。现在让我们来模拟运行goc2p项目中的代码包helper/ds的命令源码文件showds.go。示例如下：

```
hc@ubt:~/golang/goc2p/src/helper/ds$ go run -n showds.go

#
# command-line-arguments
#
```

```
mkdir -p $WORK/command-line-arguments/_obj/
mkdir -p $WORK/command-line-arguments/_obj/exe/
cd /home/hc/golang/goc2p/src/helper/ds
/usr/local/go/pkg/tool/linux_386/8g -o $WORK/command-line-arguments/_obj
   /_go_.8 -p command-line-arguments -complete -D _/home/freej/mybook/goc2p
         /src/helper/ds -I $WORK ./showds.go
/usr/local/go/pkg/tool/linux_386/pack grcP $WORK $WORK
   /command-line-arguments.a $WORK/command-line-arguments/_obj/_go_.8
cd .
/usr/local/go/pkg/tool/linux_386/8l -o $WORK/command-line-arguments
   /_obj/exe/showds -L $WORK $WORK/command-line-arguments.a
$WORK/command-line-arguments/_obj/exe/showds
```

在上面的示例中并没有显示针对命令源码文件showds.go的依赖包进行编译和运行的相关打印信息。这是因为该源码文件的所有依赖包已经在之前被编译过了。

现在，我们来逐行解释这些被打印出来的信息。

以前缀“#”开始的是注释信息。我们看到信息中有三行注释信息，并在中间行出现了内容“command-line-arguments”。我们在讲 go build 命令的时候说过，编译命令在分析参数的时候如果发现第一个参数是Go源码文件而不是代码包时，会在内部生成一个名为“command-line-arguments”的虚拟代码包。所以这里的注释信息就是要告诉我们下面的几行信息是关于虚拟代码包“command-line-arguments”的。

打印信息中的“$WORK”表示临时工作目录的绝对路径。为了存放对虚拟代码包“command-line-arguments”的编译结果，命令在临时工作目录中创建了名为command-line-arguments的子目录，并在其下又创建了_obj子目录和_obj/exe子目录。

然后，命令程序使用Go语言工具目录 8g 命令对命令源码文件showds.go进行了编译，并把结果文件存放到了$WORK/command-line-arguments/*obj目录下，名为_go*.8。我们在讲 go build 命令时提到过， 8g 命令是Go语言的官方编译器在x86（32bit）计算架构的计算机上所使用的编译程序。我们看到，编译结果文件的扩展名与底层编译命令名中的数字相对应。

编译成功后，命令程序使用 pack 命令将编译文件打包并直接存放到临时工作目录中。而后，它再用连接命令 8l 生成最终的可执行文件，并存于$WORK/command-line-arguments/_obj/exe/目录中。打印信息中的最后一行表示，命令运行了生成的可执行文件。

通过对这些打印出来的命令的解读，我们了解了临时工作目录的用途以和内容。

在上面的示例中，我们只是让 go run 命令打印出运行命令源码文件showds.go过程中需要执行的命令，而没有真正运行它。如果我们想真正运行命令源码文件showds.go并且想知道临时工作目录的位置，就需要去掉标记 -n 并且加上标记 -work 。当然，如果依然想看到过程中执行的命令，可以加上标记 -x 。如果读者已经看过之前

我们对 go build 命令的介绍，就应该知道标记 -x 与标记 -n 一样会打印出过程执行的命令，但不同的这些命令会被真正的执行。调整这些标记之后的命令就像这样：

```
hc@ubt:~/golang/goc2p/src/helper/ds$ go run -x -work showds.go
```

当命令真正执行后，临时工作目录中就会出现实实在在的内容了，像这样：

```
/tmp/go-build204903183:
    path/
        _obj/
            _go_.8
    path.a
    command-line-arguments/
        _obj/
            exe/
                showds
            _go_.8
    command-line-arguments.a
```

由于上述命令中包含了 -work 标记，所以我们可以从其输出中找到实际的工作目录（这里是/tmp/go-build204903183）。有意思的是，我们恰恰可以通过运行命令源码文件showds.go来查看这个临时工作目录的目录树：

```
hc@ubt:~/golang/goc2p/src/helper/ds$ go run showds.go -p /tmp/go-build204903183
```

读者可以自己试一试。

我们在前面介绍过，命令源码文件如果可以接受参数，则可以在执行 go run 命令运行这个命令源码文件时把参数名和参数值成对的追加在后面。实际上，如果在命令后追加参数，那么在最后执行生成的可执行文件的时候也会追加一致的参数。例如，如果这样执行命令：

```
hc@ubt:~/golang/goc2p/src/helper/ds$ go run -n showds.go -p ~/golang/goc2p
```

那么打印的最后一个命令就是：

```
$WORK/command-line-arguments/_obj/exe/showds -p /home/freej/golang/goc2p
```

可见， go run 命令会把追加到命令源码文件后面的参数原封不动的传给对应的可执行文件。

这就是一个命令源码文件从编译到运行的全过程。请记住， go run 命令包含了两个动作：编译命令源码文件和运行对应的可执行文件。

go test

`go test` 命令用于对Go语言编写的程序进行测试。这种测试是以代码包为单位的。当然，这还需要测试源码文件的帮助。关于怎样编写并写好Go程序测试代码，我们会在本章的第二节加以详述。在这里，我们只讨论怎样使用命令启动测试。

`go test` 命令会自动测试每一个指定的代码包。当然，前提是指定的代码包中存在测试源码文件。关于测试源码文件方面的知识，我们已经在第二章的第二节中介绍过。测试源码文件是名称以“_test.go”为后缀的、内含若干测试函数的源码文件。测试函数一般是以“Test”为名称前缀并有一个类型为“testing.T”的参数声明的函数.

现在，我们来测试goc2p项目中的几个代码包。在使用 `go test` 命令时指定代码包的方式与其他命令无异——使用代码包导入路径。如果需要测试多个代码包，则需要在它们的导入路径之间加入空格以示分隔。示例如下：

```
hc@ubt:~$ go test basic cnet/ctcp pkgtool
ok      basic   0.010s
ok      cnet/ctcp       2.018s
ok      pkgtool 0.009s
```

`go test` 命令在执行完所有的代码包中的测试文件之后，会以代码包为单位打印出测试概要信息。在上面的示例中，对应三个代码包的三行信息的第一列都是“ok”。这说明它们都通过了测试。每行的第三列显示运行相应测试所用的时间，以秒为单位。我们还可以在代码包目录下运行不加任何参数的运行 `go test` 命令。其作用和结果与上面的示例是一样的。

另外，我们还可以指定测试源码文件来进行测试。这样的话， `go test` 命令只会执行指定文件中的测试，像这样：

```
hc@ubt:~/golang/goc2p/src/pkgtool$ go test envir_test.go
# command-line-arguments
./envir_test.go:20: undefined: GetGoroot
./envir_test.go:34: undefined: GetAllGopath
./envir_test.go:74: undefined: GetSrcDirs
./envir_test.go:76: undefined: GetAllGopath
./envir_test.go:83: undefined: GetGoroot
FAIL    command-line-arguments [build failed]
```

我们看到，与指定源码文件进行编译或运行一样，命令程序会为指定的源码文件生成一个虚拟代码包——“command-line-arguments”。但是，测试并没有通过。但其原因并不是测试失败，而是编译失败。对于运行这次测试的命令程序来说，测试源码文件envir_test.go是属于代码包“command-line-arguments”的。并且，这个测试源码文件中使用了库源码文件envir.go中的函数。可以，它却没有显示导入这个库源码文件所属的代码

包，这当然会引起编译错误。如果想解决这个问题，我们还需要在执行命令时加入这个测试源码文件所测试的那个源码文件。示例如下：

```
hc@ubt:~/golang/goc2p/src/pkgtool$ go test envir_test.go envir.go
ok      command-line-arguments  0.008s
```

现在，我们故意使代码包 pkgtool 中的某个测试失败。现在我们再来运行测试：

```
hc@ubt:~$ go test basic cnet/ctcp pkgtool
ok      basic   0.010s
ok      cnet/ctcp       2.015s
--- FAIL: TestGetSrcDirs (0.00 seconds)
        envir_test.go:85: Error: The src dir '/usr/local/go/src/pkg' is incorrect.
FAIL
FAIL    pkgtool 0.009s
```

我们通过以上示例中的概要信息获知，测试源码文件中envir_test.go的测试函数 TestGetSrcDirs 中的测试失败了。在包含测试失败的测试源码文件名的那一行信息中，紧跟测试源码文件名的用冒号分隔的数字是错误信息所处的行号，在行号后面用冒号分隔的是错误信息。这个错误信息的内容是用户自行编写的。另外，概要信息的最后一行以“FAIL”为前缀。这表明针对代码包pkgtool的测试未通过。未通过的原因在前面的信息中已有描述。

一般情况下，我们会把测试源码文件与被测试的源码文件放在同一个代码包中。并且，这些源码文件中声明的包名也都是相同的。除此之外我们还有一种选择，那就是测试源码文件中声明的包名可以是所属包名再加“_test”后缀。我们把这种测试源码文件叫做包外测试源码文件。不过，包外测试源码文件存在一个弊端，那就是在它们的测试函数中无法测试被测源码文件中的包级私有的程序实体，比如包级私有的变量、函数和结构体类型。这是因为这两者的所属代码包是不相同的。所以，我们一般很少会编写包外测试源码文件。

关于标记

go test 命令的标记处理部分是庞大且繁杂的，以至于使Go语言的开发者们不得不把这一部分的逻辑从 go test 命令程序主体中分离出来并建立单独的源码文件。因为 go test 命令中包含了编译动作，所以它可以接受可用于 go build 命令的所有标记。另外，它还有很多特有的标记。这些标记的用于控制命令本身的动作，有的用于控制和设置测试的过程和环境，还有的用于生成更详细的测试结果和统计信息。

可用于 go test 命令的两个比较常用的标记是 -i 和标记 -c 。这两个就是用于控制 go test 命令本身的动作的标记。详见下表。

表0-6 go test 命令的标记说明

标记名称	标记描述
-c	生成用于运行测试的可执行文件，但不执行它。
-i	安装/重新安装运行测试所需的依赖包但不编译和运行测试代码。

上述这两个标记可以搭配使用。搭配使用的目的就是让 `go test` 命令既安装依赖包又编译测试代码，但不运行测试。也就是说，让命令程序跑一遍运行测试之前的所有流程。这可以测试一下测试过程。需要注意的是，在加入 `-c` 标记后，命令程序在编译测试代码并生成用于运行测试的一系列文件之后会把临时工作目录及其下的所有内容一并删除。如果想在命令执行结束后再去查看这些内容的话，我们还需要加入 `-work` 标记。

除此之外， `go test` 命令还有很多功效各异的标记。但是由于这些标记的复杂性，我们需要结合测试源码文件进行详细的讲解。所以我们把这些内容放在了本章的第二节中。

go list

`go list` 命令的作用是列出指定的代码包的信息。与其他命令相同，我们需要以代码包导入路径的方式给定代码包。被给定的代码包可以有多个。这些代码包对应的目录中必须直接保存有Go语言源码文件，其子目录中的文件不算在内。否则，代码包将被看做是不完整的。现在我们来试用一下：

```
hc@ubt:~$ go list cnet/ctcp pkgtool
cnet/ctcp
pkgtool
```

我们看到，在不加任何标记的情况下，命令的结果信息中只包含了我们指定的代码包的导入路径。我们刚刚提到，作为参数的代码包必须是完整的代码包。例如：

```
hc@ubt:~$ go list cnet pkgtool
can't load package: package cnet: no Go source files in /home/hc/golang/goc2p
    /src/cnet
pkgtool
```

这时，`go list` 命令报告了一个错误——代码包 `cnet` 对应的目录下没有Go源码文件。但是命令还是把代码包pkgtool的导入路径打印出来了。然而，当我们在执行 `go list` 命令并加入标记 `-e` 时，即使参数中包含有不完整的代码包，命令也不会提示错误。示例如下：

```
hc@ubt:~$ go list -e cnet pkgtool
cnet
pkgtool
```

标记 `-e` 的作用是以容错模式加载和分析指定的代码包。在这种情况下，命令程序如果在加载或分析的过程中遇到错误只会在内部记录一下，而不会直接把错误信息打印出来。我们为了看到错误信息可以使用 `-json` 标记。这个标记的作用是把代码包的结构体实例用JSON的样式打印出来。

这里解释一下，JSON的全称是Javascript Object Notation。它一种轻量级的承载数据的格式。JSON的优势在于语法简单、短小精悍，且非常易于处理。JSON还是一种纯文本格式，独立于编程语言。正因为如此，得到了绝大多数编程语言和浏览器的支持，应用非常广泛。Go语言当然也不例外，在它的标准库中有专门用于处理和转换JSON格式的数据的代码包 `encoding/json` 。关于JSON格式的具体内容，读者可以去它的官方网站查看说明。

在了解了这些基本概念之后，我们来试用一下 `-json` 标记。示例如下：

```
hc@ubt:~$ go list -e -json cnet
{
        "Dir": "/home/hc/golang/goc2p/src/cnet",
```

```
    "ImportPath": "cnet",
    "Stale": true,
    "Root": "/home/hc/golang/goc2p",
    "Incomplete": true,
    "Error": {
        "ImportStack": [
            "cnet"
        ],
        "Pos": "",
        "Err": "no Go source files in /home/hc/golang/goc2p/src/cnet"
    }
}
```

在上述JSON格式的代码包信息中，对于结构体中的字段的显示是不完整的。因为命令程序认为我们指定 cnet 就是不完整的。在名为 Error 的字段中，我们可以看到具体说明。 Error 字段的内容其实也是一个结构体。在JSON格式下，这种嵌套的结构体被完美的展现了出来。 Error 字段所指代的结构体实例的 Err 字段说明了 cnet 不完整的原因。这与我们在没有使用 -e 标记的情况下所打印出来的错误提示信息是一致的。我们再来看 Incomplete 字段。它的值为 true 。这同样说明 cnet 是一个不完整的代码包。

实际上，在从这个代码包结构体实例到JSON格式文本的转换过程中，所有的值为其类型的空值的字段都已经被忽略了。

现在我们使用带 -json 标记的 go list 命令列出代码包 cnet/ctcp 的信息：

```
hc@ubt:~$ go list -json cnet/ctcp
{
    "Dir": "/home/freej/mybook/goc2p/src/cnet/ctcp",
    "ImportPath": "cnet/ctcp",
    "Name": "ctcp",
    "Target": "/home/freej/mybook/goc2p/pkg/linux_386/cnet/ctcp.a",
    "Stale": true,
    "Root": "/home/freej/mybook/goc2p",
    "GoFiles": [
        "base.go",
        "tcp.go"
    ],
    "Imports": [
        "bufio",
        "errors",
        "logging",
        "net",
        "sync",
        "time"
    ],
```

```
    "Deps": [
        "bufio",
        "bytes",
        "errors",
        "fmt",
        "io",
        "log",
        "logging",
        "math",
        "math/rand",
        "net",
        "os",
        "reflect",
        "runtime",
        "runtime/cgo",
        "sort",
        "strconv",
        "strings",
        "sync",
        "sync/atomic",
        "syscall",
        "time",
        "unicode",
        "unicode/utf8",
        "unsafe"
    ],
    "TestGoFiles": [
        "tcp_test.go"
    ],
    "TestImports": [
        "bytes",
        "fmt",
        "net",
        "strings",
        "sync",
        "testing",
        "time"
    ]
}
```

由于 cnet/ctcp 包是一个完整有效的代码包，所以我们不使用 -e 标记也是没有问题的。在上面打印的 cnet/ctcp 包的信息中没有 Incomplete 字段。这是因为完整的代码包中的Incomplete字段的其类型的空值 false 。它已经在转换过程中被忽略掉了。另外，在 cnet/ctcp 包的信息中我们看到了很多其它的字段。现在我就来看看在Go命令程序中的代码包结构体都有哪些公开的字段。如下表。

表0-7 代码包结构体中的基本字段

字段名称	字段类型	字段描述
Dir	字符串（string）	代码包对应的目录。
ImportPath	字符串（string）	代码包的导入路径。
Name	字符串（string）	代码包的名称。
Doc	字符串（string）	代码包的文档字符串。
Target	字符串（string）	代码包的安装路径。
Goroot	布尔（bool）	代码包是否在Go安装目录下。
Standard	布尔（bool）	代码包是否属于标准库的一部分。
Stale	布尔（bool）	代码包能否被```go install```命令安装。
Root	字符串（string）	代码包所属的工作区或Go安装目录的路径。

表0-8 代码包结构体中与源码文件有关的字段

字段名称	字段类型	字段描述
GoFiles	字符串切片（[]string）	Go源码文件的数组。不包含导入了代码包“C”的源码文件和测试源码文件。
CgoFiles	字符串切片（[]string）	导入了代码包“C”的源码文件的数组。
IgnoredGoFiles	字符串切片（[]string）	需要被编译器忽略的源码文件的数组。
CFiles	字符串切片（[]string）	名称中有“.c”后缀的文件的数组。
HFiles	字符串切片（[]string）	名称中有“.h”后缀的文件的数组。
SFiles	字符串切片（[]string）	名称中有“.s”后缀的文件的数组。
SysoFiles	字符串切片（[]string）	名称中有“.syso”后缀的文件的数组。这些文件需要被加入到归档文件中。
SwigFiles	字符串切片（[]string）	名称中有“.swig”后缀的文件的数组。
SwigCXXFiles	字符串切片（[]string）	名称中有“.swigcxx”后缀的文件的数组。

表0-9 代码包结构体中与Cgo指令有关的字段

字段名称	字段类型	字段描述
CgoCFLAGS	字符串切片（[]string）	需要传递给C编译器的标记的数组。针对于Cgo。
CgoLDFLAGS	字符串切片（[]string）	需要传递给链接器的标记的数组。针对于Cgo。
CgoPkgConfig	字符串切片（[]string）	pkg-config的名称的数组。针对于Cgo。

表0-10 代码包结构体中与依赖信息有关的字段

字段名称	字段类型	字段描述
Imports	字符串切片（[]string）	代码包中的源码文件显示导入的依赖包的导入路径的数组。
Deps	字符串切片（[]string）	所有的依赖包（包括间接依赖）的导入路径的数组。

表0-11 代码包结构体中与错误信息有关的字段

字段名称	字段类型	字段描述
Incomplete	布尔（bool）	代码包是否是完整的，也即在载入或分析代码包及其依赖包时是否有错误发生。
Error	*PackageError类型	载入或分析代码包时发生的错误。
Error	*PackageError类型的数组（[]*PackageError）	载入或分析代码包的依赖包时发生的错误。

表0-12 代码包结构体中与测试源码文件有关的字段

字段名称	字段类型	字段描述
TestGoFiles	字符串切片（[]string）	代码包中的测试源码文件的数组。
TestImports	字符串切片（[]string）	代码包中的测试源码文件显示导入的依赖包的导入路径的数组。
XTestGoFiles	字符串切片（[]string）	代码包中的外部测试源码文件的数组。
XTestImports	字符串切片（[]string）	代码包中的外部测试源码文件显示导入的依赖包的导入路径的数组。

代码包结构体中定义的字段很多，但有些时候我们只需要查看其中的一些字段。那要怎么做呢？标记 -f 可以满足这个需求。比如这样：

```
hc@ubt:~$ go list -f {{.ImportPath}} cnet/ctcp
cnet/ctcp
```

实际上， -f 标记的默认值就是 {{.ImportPath}} 。这也正是我们在使用不加任何标记的 go list 命令时依然能看到指定代码包的导入路径的原因了。

标记 -f 的值需要满足标准库的代码包````text/template 中定义的语法。比如， {{.S}} 代表根结构体的 S 字段的值。在 go list 命令的场景下，这个根结构体就是指定的代码包所对应的结构体。如果 S 字段的值也是一个结构体的话，那么 {{.S.F}} 就代表根结构体的 S 字段的值中的 F 字段的值。如果我们要查看 cnet/ctcp 包中的命令源码文件和库源码文件的列表，可以这样使用 -f```标记：

```
hc@ubt:~$ go list -f {{.GoFiles}} cnet/ctcp
[base.go tcp.go]
```

如果我们想查看不完整的代码包 cnet 的错误提示信息，还可以这样：

```
hc@ubt:~$ go list -e -f {{.Error.Err}} cnet
no Go source files in D:\Kanbox\gitrepo\goc2p\src\cnet
```

我们还可以利用代码包 text/template 中定义的强大语法让 go list 命令输出定制化更高的代码包信息。比如：

```
hc@ubt:~$ go list -e -f 'The package {{.ImportPath}} is {{if .Incomplete}}
   incomplete!{{else}}complete.{{end}}' cnet
The package cnet is incomplete!

hc@ubt:~$ go list -f 'The imports of package {{.ImportPath}}
   is [{{join .Imports ", "}}].' cnet/ctcp
The imports of package cnet/ctcp is [bufio, errors, logging, net, sync, time].
```

其中， join 是命令程序在 text/template 包原有语法之上自定义的语法，在底层使用标准库代码包 strings 中的 Join 函数。关于更多的语法规则，请读者查看代码包 text/template 的相关文档。

另外， -tags 标记也可以被 go list 接受。它与我们在讲 go build 命令时提到的 -tags 标记是一致的。读者可以查看代码包```go/build``的文档以了解细节。

go list 命令很有用。它可以为我们提供指定代码包的更深层次的信息。这些信息往往是我们无法从源码文件中直观看到的。

go fix与go tool fix

命令 go fix 会把指定代码包的所有Go语言源码文件中的旧版本代码修正为新版本的代码。这里所说的版本即Go语言的版本。代码包的所有Go语言源码文件不包括其子代码包（如果有的话）中的文件。修正操作包括把对旧程序调用的代码更换为对新程序调用的代码、把旧的语法更换为新的语法，等等。

这个工具其实非常有用。在编程语言的升级和演进的过程中，难免会对过时的和不够优秀的语法及标准库进行改进。这样的改进对于编程语言的向后兼容性是个挑战。我们在前面提到过向后兼容这个词。简单来说，向后兼容性就是指新版本的编程语言程序能够正确识别和解析用该编程语言的旧版本编写的程序和软件，以及在新版本的编程语言的运行时环境中能够运行用该编程语言的旧版本编写的程序和软件。对于Go语言来说，语法的改变和标准库的变更都会使得用旧版本编写的程序无法在新版本环境中编译通过。这就等于破坏了Go语言的向后兼容性。对于一个编程语言、程序库或基础软件来说，向后兼容性是非常重要的。但有时候为了让软件更加优秀，软件的开发者或维护者不得不在向后兼容性上做出一些妥协。这是一个在多方利益之间进行权衡的结果。本小节所讲述的工具正是Go语言的创造者们为了不让这种妥协给语言使用者带来困扰和额外的工作量而编写的自动化修正工具。这也充分体现了Go语言的软件工程哲学。下面让我们来详细了解它们的使用方法和内部机理。

命令 go fix 其实是命令 go tool fix 的简单封装。这甚至比 go fmt 命令对 gofmt 命令的封装更简单。像其它的Go命令一样， go fix 命令会先对作为参数的代码包导入路径进行验证，以确保它是正确有效的。像在本小节开始处描述的那样， go fix 命令会把有效代码包中的所有Go语言源码文件作为多个参数传递给 go tool fix 命令。实际上， go fix 命令本身不接受任何标记，它会把加入的所有标记都原样传递给 go tool fix 命令。 go tool fix 命令可接受的标记如下表。

表0-15 go tool fix 命令的标记说明

<

table class="table table-bordered table-striped table-condensed"> 标记名称 标记描述 -diff 不将修正后的内容写入文件，而只打印修正前后的内容的对比信息到标准输出。 -r 只对目标源码文件做有限的修正操作。该标记的值即为允许的修正操作的名称。多个名称之间用英文半角逗号分隔。 -force 使用此标记后，即使源码文件中的代码已经与Go语言的最新版本相匹配了，也会强行执行指定的修正操作。该标记的值就是需要强行执行的修正操作的名称，多个名称之间用英文半角逗号分隔。

<

table>

在默认情况下， go tool fix 命令程序会在目标源码文件上执行所有的修正操作。多个修正操作的执行会按照每个修正操作中标示的操作建立日期以从早到晚的顺序进行。我们可以通过执行 go tool fix -? 来查看 go tool fix 命令的使用说明以及当前支持的修正操作。

与本书对应的Go语言版本的 go tool fix 命令目前只支持两个修正操作。一个是与标准库代码包 go/printer 中的结构体类型 Config 的初始化代码相关的修正操作，另一个是与标准库代码包``net 中的结构体类型 IPAddr 、 UDPAddr 和 TCPAddr```的初始化代码相关的修正操作。从修正操作的数量来看，自第一个正式版发布以来，Go语言的向后兼容性还是很好的。从Go语言官网上的说明也可以获知，在Go语言的第二个大版本（ Go 2.x ）出现之前，它会一直良好的向后兼容性。

值得一提的是，上述的修正操作都是依靠Go语言的标准库代码包 go 及其子包中提供的功能来完成的。实际上， go tool fix 命令程序在执行修正操作之前，需要先将目标源码文件中的内容解析为一个抽象语法树实例。这一功能其实就是由代码包 go/parser 提供的。而在这个抽象语法树实例中的各个元素的结构体类型的定义以及检测、访问和修改它们的方法则由代码包 go/ast 提供。有兴趣的读者可以阅读这些代码包中的代码。这对于深入理解Go语言对代码的静态处理过程是非常有好处的。

回到正题。与 gofmt 命令相同， go tool fix 命令也有交互模式。我们同样可以通过执行不带任何参数的命令来进入到这个模式。但是与 gofmt 命令不同的是，我们在 go tool fix 命令的交互模式中输入的代码必须是完整的，即必须要符合Go语言源码文件的代码组织形式。当我们输入了不完整的代码片段时，命令程序将显示错误提示信息并退出。示例如下：

```
hc@ubt:~$ go tool fix -r='netipv6zone'
a := &net.TCPAddr{ip4, 8080}
standard input:1:1: expected 'package', found 'IDENT' a
```

相对于上面的示例，我们必须要这样输入源码才能获得正常的结果：

```
hc@ubt:~$ go tool fix -r='netipv6zone'
package main

import (
"fmt"
"net"
)

func main() {
addr := net.TCPAddr{"127.0.0.1", 8080}
fmt.Printf("TCP Addr: %s\n", addr)
}
standard input: fixed netipv6zone
package main
```

```
import (
	"fmt"
	"net"
)

func main() {
	addr := net.TCPAddr{IP: "127.0.0.1", Port: 8080}
	fmt.Printf("TCP Addr: %s\n", addr)
}
```

上述示例的输出结果中有这样一行提示信息：“standard input: fixed netipv6zone”。其中，“standard input”表明源码是从标准输入而不是源码文件中获取的，而“fixed netipv6zone”则表示名为netipv6zone的修正操作发现输入的源码中有需要修正的地方，并且已经修正完毕。另外，我们还可以看到，输出结果中的代码已经经过了格式化。

go vet与go tool vet

命令 go vet 是一个用于检查Go语言源码中静态错误的简单工具。与大多数Go命令一样， go vet 命令可以接受 -n 标记和 -x 标记。 -n 标记用于只打印流程中执行的命令而不真正执行它们。 -n 标记也用于打印流程中执行的命令，但不会取消这些命令的执行。示例如下：

```
hc@ubt:~$ go vet -n pkgtool
/usr/local/go/pkg/tool/linux_386/vet golang/goc2p/src/pkgtool/envir.go golang/goc2p/src/pkgtool/envir_test.go golang/goc
```

go vet 命令的参数既可以是代码包的导入路径，也可以是Go语言源码文件的绝对路径或相对路径。但是，这两种参数不能混用。也就是说， go vet 命令的参数要么是一个或多个代码包导入路径，要么是一个或多个Go语言源码文件的路径。

go vet 命令是 go tool vet 命令的简单封装。它会首先载入和分析指定的代码包，并把指定代码包中的所有Go语言源码文件和以“.s”结尾的文件的相对路径作为参数传递给 go tool vet 命令。其中，以“.s”结尾的文件是汇编语言的源码文件。如果 go vet 命令的参数是Go语言源码文件的路径，则会直接将这些参数传递给 go tool vet 命令。

如果我们直接使用 go tool vet 命令，则其参数可以传递任意目录的路径，或者任何Go语言源码文件和汇编语言源码文件的路径。路径可以是绝对的也可以是相对的。

实际上， vet 属于Go语言自带的特殊工具，也是比较底层的命令之一。Go语言自带的特殊工具的存放路径是$GOROOT/pkg/tool/$GOOS_$GOARCH/，我们暂且称之为Go工具目录。我们再来复习一下，环境变量GOROOT的值即Go语言的安装目录，环境变量GOOS的值代表程序构建环境的目标操作系统的标识，而环境变量$GOARCH的值则为程序构建环境的目标计算架构。另外，名为$GOOS_$GOARCH的目录被叫做平台相关目录。Go语言允许我们通过执行 go tool 命令来运行这些特殊工具。在Linux 32bit的环境下，我们的Go语言安装目录是/usr/local/go/。因此， go tool vet 命令指向的就是被存放在/usr/local/go/pkg/tool/linux_386目录下的名为 vet 的工具。

go tool vet 命令的作用是检查Go语言源代码并且报告可疑的代码编写问题。比如，在调用 Printf 函数时没有传入格式化字符串，以及某些不标准的方法签名，等等。该命令使用试探性的手法检查错误，因此并不能保证报告的问题确实需要解决。但是，它确实能够找到一些编译器没有捕捉到的错误。

go tool vet 命令程序在被执行后会首先解析标记并检查标记值。 go tool vet 命令支持的所有标记如下表。

表0-16 go tool vet 命令的标记说明

标记名称	标记描述

-all	进行全部检查。如果有其他检查标记被设置，则命令程序会将此值变为false。默认值为true。
-asmdecl	对汇编语言的源码文件进行检查。默认值为false。
-assign	检查赋值语句。默认值为false。
-atomic	检查代码中对代码包sync/atomic的使用是否正确。默认值为false。
-buildtags	检查编译标签的有效性。默认值为false。
-composites	检查复合结构实例的初始化代码。默认值为false。
-compositeWhiteList	是否使用复合结构检查的白名单。仅供测试使用。默认值为true。
-methods	检查那些拥有标准命名的方法的签名。默认值为false。
-printf	检查代码中对打印函数的使用是否正确。默认值为false。
-printfuncs	需要检查的代码中使用的打印函数的名称的列表，多个函数名称之间用英文半角逗号分隔。默认值为空字符串。
-rangeloops	检查代码中对在```range```语句块中迭代赋值的变量的使用是否正确。默认值为false。
-structtags	检查结构体类型的字段的标签的格式是否标准。默认值为false。
-unreachable	查找并报告不可到达的代码。默认值为false。

在阅读上面表格中的内容之后，读者可能对这些标签的具体作用及其对命令程序检查步骤的具体影响还很模糊。不过没关系，我们下面就会对它们进行逐一的说明。

-all标记

如果标记 `-all` 有效（标记值不为 `false` ），那么命令程序会对目标文件进行所有已知的检查。实际上，标记 `-all` 的默认值就是 `true` 。也就是说，在执行 `go tool vet` 命令且不加任何标记的情况下，命令程序会对目标文件进行全面的检查。但是，只要有一个另外的标记（ `-compositeWhiteList` 和 `-printfuncs` 这两个标记除外）有效，命令程序就会把标记 `-all` 设置为false，并只会进行与有效的标记对应的检查。

-assign标记

如果标记 `-assign` 有效（标记值不为 `false` ），则命令程序会对目标文件中的赋值语句进行自赋值操作检查。什么叫做自赋值呢？简单来说，就是将一个值或者实例赋值给它本身。像这样：

```
var s1 string = "S1"
s1 = s1 // 自赋值
```

或者

```
s1, s2 := "S1", "S2"
s2, s1 = s2, s1 // 自赋值
```

检查程序会同时遍历等号两边的变量或者值。在抽象语法树的语境中，它们都被叫做表达式节点。检查程序会检查等号两边对应的表达式是否相同。判断的依据是这两个表达式节点的字符串形式是否相同。在当前的场景下，这种相同意味着它们的变量名是相同的。如前面的示例。

有两种情况是可以忽略自赋值检查的。一种情况是短变量声明语句。根据Go语言的语法规则，当我们在函数中要在声明局部变量的同时对其赋值，就可以使用 := 形式的变量赋值语句。这也就意味着 := 左边的变量名称在当前的上下文环境中应该还未曾出现过（否则不能通过编译）。因此，在这种赋值语句中不可能出现自赋值的情况，忽略对它的检查也是合理的。另一种情况是等号左右两边的表达式个数不相等的变量赋值语句。如果在等号的右边是对某个函数或方法的调用，就会造成这种情况。比如：

```
file, err := os.Open(wp)
```

很显然，这个赋值语句肯定不是自赋值语句。因此，不需要对此种情况进行检查。如果等号右边并不是对函数或方法调用的表达式，并且等号两边的表达式数量也不相等，那么势必会在编译时引发错误，也不必检查。

-atomic标记

如果标记 -atomic 有效（标记值不为 false ），则命令程序会对目标文件中的使用代码包 sync/atomic 进行原子赋值的语句进行检查。原子赋值语句像这样：

```
var i32 int32
i32 = 0
newi32 := atomic.AddInt32(&i32, 3)
fmt.Printf("i32: %d, newi32: %d.\n", i32, newi32)
```

函数 AddInt32 会原子性的将变量 i32 的值加 3 ，并返回这个新值。因此上面示例的打印结果是：

```
i32: 3, newi32: 3
```

在代码包 sync/atomic 中，与 AddInt32 类似的函数还有 AddInt64 、 AddUint32 、 AddUint64 和 AddUintptr 。检查程序会对上述这些函数的使用方式进行检查。检查的关注点在破坏原子性的使用方式上。比如：

```
i32 = 1
i32 = atomic.AddInt32(&i32, 3)
_, i32 = 5, atomic.AddInt32(&i32, 3)
i32, _ = atomic.AddInt32(&i32, 1), 5
```

上面示例中的后三行赋值语句都属于原子赋值语句，但它们都破坏了原子赋值的原子性。以第二行的赋值语句为例，等号左边的 atomic.AddInt32(&i32, 3) 的作用是原子性的将变量 i32 的值增加 3 。但该语句又将函数的结果值赋值给变量 i32 ，这个二次赋值属于对变量 i32 的重复赋值，也使原本拥有原子性的赋值操作被拆分为了两个步骤的非原子操作。如果在对变量 i32 的第一次原子赋值和第二次非原子的重复赋值之间又有另一个程序对变量 i32 进行了原子赋值，那么当前程序中的这个第二次赋值就破坏了那两次原子赋值本应有的顺序性。因为，在另一

个程序对变量 i32 进行原子赋值后，当前程序中的第二次赋值又将变量 i32 的值设置回了之前的值。这显然是不对的。所以，上面示例中的第二行代码应该改为：

```
atomic.AddInt32(&i32, 3)
```

并且，对第三行和第四行的代码也应该有类似的修改。检查程序如果在目标文件中查找到像上面示例的第二、三、四行那样的语句，就会打印出相应的错误信息。

另外，上面所说的导致原子性被破坏的重复赋值语句还有一些类似的形式。比如：

```
i32p := &i32
*i32p = atomic.AddUint64(i32p, 1)
```

这与之前的示例中的代码的含义几乎是一样。另外还有：

```
var counter struct{ N uint32 }
counter.N = atomic.AddUint64(&counter.N, 1)
```

和

```
ns := []uint32{10, 20}
ns[0] = atomic.AddUint32(&ns[0], 1)
nps := []*uint32{&ns[0], &ns[1]}
*nps[0] = atomic.AddUint32(nps[0], 1)
```

在最近的这两个示例中，虽然破坏原子性的重复赋值操作因结构体类型或者数组类型的介入显得并不那么直观了，但依然会被检查程序发现并及时打印错误信息。

顺便提一句，对于原子赋值语句和普通赋值语句，检查程序都会忽略掉对等号两边的表达式的个数不相等的赋值语句的检查。

-buildtags标记

前文已提到，如果标记 -buildtags 有效（标记值不为 false ），那么命令程序会对目标文件中的编译标签（如果有的话）的格式进行检查。什么叫做条件编译？在实际场景中，有些源码文件中包含了平台相关的代码。我们希望只在某些特定平台下才编译它们。这种有选择的编译方法就被叫做条件编译。在Go语言中，条件编译的配置就是通过编译标签来完成的。编译器需要依据源码文件中编译标签的内容来决定是否编译当前文件。编译标签可必须出现在任何源码文件（比如扩展名为“.go”，“.h”，“.c”，“.s”等的源码文件）的头部的单行注释中，并且在其后面需要有空行。

至于编译标签的具体写法，我们就不在此赘述了。读者可以参看Go语言官方的相关文档。我们在这里只简单罗列一下 -buildtags 有效时命令程序对编译标签的检查内容：

1. 若编译标签前导符“+build”后没有紧随空格，则打印格式错误信息。
2. 若编译标签所在行与第一个多行注释或代码行之间没有空行，则打印错误信息。
3. 若在某个单一参数的前面有两个英文叹号“!!”，则打印错误信息。
4. 若单个参数包含字母、数字、“_”和“.”以外的字符，则打印错误信息。
5. 若出现在文件头部单行注释中的编译标签前导符“+build”未紧随在单行注释前导符“//”之后，则打印错误信息。

如果一个在文件头部的单行注释中的编译标签通过了上述的这些检查，则说明它的格式是正确无误的。由于只有在文件头部的单行注释中编译标签才会被编译器认可，所以检查程序只会查找和检查源码文件中的第一个多行注释或代码行之前的内容。

-composites标记和-compositeWhiteList标记

如果标记 `-composites` 有效（标记值不为 `false` ），则命令程序会对目标文件中的复合字面量进行检查。请看如下示例：

```
type counter struct {
    name   string
    number int
}
...
c := counter{name: "c1", number: 0}
```

在上面的示例中，代码 `counter{name: "c1", number: 0}` 是对结构体类型 `counter` 的初始化。如果复合字面量中涉及到的类型不在当前代码包内部且未在所属文件中被导入，那么检查程序不但会打印错误信息还会将退出代码设置为1，并且取消后续的检查。退出代码为1意味着检查程序已经报告了一个或多个问题。这个问题比仅仅引起错误信息报告的问题更加严重。

在通过上述检查的前提下，如果复合字面量中包含了对结构体类型的字段的赋值但却没有指明字段名，像这样：

```
var v = flag.Flag{
    "Name",
    "Usage",
    nil, // Value
    "DefValue",
}
```

那么检查程序也会打印错误信息，以提示在复合字面量中包含有未指明的字段赋值。

这有一个例外，那就是当标记 -compositeWhiteList 有效（标记值不为 false ）的时候。只要类型在白名单中，即使其初始化语句中含有未指明的字段赋值也不会被提示。这是出于什么考虑呢？先来看下面的示例：

```
type sliceType []string
...
st1 := sliceType{"1", "2", "3"}
```

上面示例中的 sliceType{"1", "2", "3"} 也属于复合字面量。但是它初始化的类型实际上是一个切片值，只不过这个切片值被别名化并被包装为了另一个类型而已。在这种情况下，复合字面量中的赋值不需要指明字段，事实上这样的类型也不包含任何字段。白名单中所包含的类型都是这种情况。它们是在标准库中的包装了切片值的类型。它们不需要被检查，因为这种情况是合理的。

在默认情况下，标记 -compositeWhiteList 是有效的。也就是说，检查程序不会对它们的初始化代码进行检查，除非我们在执行 go tool vet 命令时显示的将 -compositeWhiteList 标记的值设置为false。

-methods标记

如果标记 -methods 有效（标记值不为 false ），则命令程序会对目标文件中的方法定义进行规范性的进行检查。这里所说的规范性是狭义的。

在检查程序内部存有一个规范化方法字典。这个字典的键用来表示方法的名称，而字典的元素则用来描述方法应有的参数和结果的类型。在该字典中列出的都是Go语言标准库中使用最广泛的接口类型的方法。这些方法的名字都非常通用。它们中的大多数都是它们所属接口类型的唯一方法。我们在第4章中提到过，Go语言中的接口类型实现方式是非侵入式的。只要结构体类型实现了某一个接口类型中的所有方法，就可以说这个结构体类型是该接口类型的一个实现。这种判断方式被称为动态接口检查。它只在运行时进行。如果我们想让一个结构体类型成为某一个接口类型的实现，但又写错了要实现的接口类型中的方法的签名，那么也不会引发编译器报错。这里所说的方法签名包括方法的参数声明列表和结果声明列表。虽然动态接口检查失败时并不会报错，但是它却会间接的引发其它错误。而这些被间接引发的错误只会在运行时发生。示例如下：

```
type MySeeker struct {
    // 忽略字段定义
}

func (self *MySeeker) Seek(whence int, offset int64) (ret int64, err error) {
    // 想实现接口类型io.Seeker中的唯一方法，但是却把参数的顺序写颠倒了。
    // 忽略实现代码
}

func NewMySeeker io.Seeker {
    return &MySeeker{/* 忽略字段初始化 */} // 这里会引发一个运行时错误。
                                    //由于MySeeker的Seek方法的签名写错了，所以MySeeker不是io.Seeker的实现。
}
```

这种运行时错误看起来会比较诡异，并且错误排查也会相对困难，所以应该尽量避免。`-methods` 标记所对应的检查就是为了达到这个目的。检查程序在发现目标文件中某个方法的名字被包含在规范化方法字典中但其签名与对应的描述不对应的时候，就会打印错误信息并设置退出代码为1。

我在这里附上在规范化方法字典中列出的方法的信息：

表0-17 规范化方法字典中列出的方法

方法名称	参数类型	结果类型	所属接口	唯一方法
Format	"fmt.State", "rune"	<无>	fmt.Formatter	是
GobDecode	"[]byte"	"error"	gob.GobDecoder	是
GobEncode	<无>	"[]byte", "error"	gob.GobEncoder	是
MarshalJSON	<无>	"[]byte", "error"	json.Marshaler	是
Peek	"int"	"[]byte", "error"	image.reader	否
ReadByte	"int"	"[]byte", "error"	io.ByteReader	是
ReadFrom	"io.Reader"	"int64", "error"	io.ReaderFrom	是
ReadRune	<无>	"rune", "int", "error"	io.RuneReader	是
Scan	"fmt.ScanState", "rune"	"error"	fmt.Scanner	是
Seek	"int64", "int"	"int64", "error"	io.Seeker	是
UnmarshalJSON	"[]byte"	"error"	json.Unmarshaler	是
UnreadByte	<无>	"error"	io.ByteScanner	否
UnreadRune	<无>	"error"	io.RuneScanner	否
WriteByte	"byte"	"error"	io.ByteWriter	是
WriteTo	"io.Writer"	"int64", "error"	io.WriterTo	是

-printf标记和-printfuncs标记

标记 `-printf` 旨在目标文件中检查各种打印函数使用的正确性。而标记 `-printfuncs` 及其值则用于明确指出需要检查的打印函数。`-printfuncs` 标记的默认值为空字符串。也就是说，若不明确指出检查目标则检查所有打印函数。可被检查的打印函数如下表：

表0-18 格式化字符串中动词的格式要求

函数全小写名称	支持格式化	可自定义输出	自带换行
error	否	否	是
fatal	否	否	是
fprint	否	是	否
fprintln	否	是	是
panic	否	否	否
panicln	否	否	是

print	否	否	否
println	否	否	是
sprint	否	否	否
sprintln	否	否	是
errorf	是	否	否
fatalf	是	否	否
fprintf	是	是	否
panicf	是	否	否
printf	是	否	否
sprintf	是	是	否

以字符串格式化功能来区分，打印函数可以分为可打印格式化字符串的打印函数（以下简称格式化打印函数）和非格式化打印函数。对于格式化打印函数来说，其第一个参数必是格式化表达式，也可被称为模板字符串。而其余参数应该为需要被填入模板字符串的变量。像这样：

```
fmt.Printf("Hello, %s!\n", "Harry")
// 会输出：Hello, Harry!
```

而非格式化打印函数的参数则是一个或多个要打印的内容。比如：

```
fmt.Println("Hello,", "Harry!")
// 会输出：Hello, Harry!
```

以指定输出目的地功能区分，打印函数可以被分为可自定义输出目的地的的打印函数（以下简称自定义输出打印函数）和标准输出打印函数。对于自定义输出打印函数来说，其第一个函数必是其打印的输出目的地。比如：

```
fmt.Fprintf(os.Stdout, "Hello, %s!\n", "Harry")
// 会在标准输出设备上输出：Hello, Harry!
```

上面示例中的函数 `fmt.Fprintf` 既能够让我们自定义打印的输出目的地，又能够格式化字符串。此类打印函数的第一个参数的类型应为 `io.Writer` 接口类型。只要某个类型实现了该接口类型中的所有方法，就可以作为函数 `Fprintf` 的第一个参数。例如，我们还可以使用代码包 `bytes` 中的结构体 `Buffer` 来接收打印函数打印的内容。像这样：

```
var buff bytes.Buffer
fmt.Fprintf(&buff, "Hello, %s!\n", "Harry")
fmt.Print("Buffer content:", buff.String())
// 会在标准输出设备上输出：Buffer content: Hello, Harry!
```

而标准输出打印函数则只能将打印内容到标准输出设备上。就像函数 `fmt.Printf` 和 `fmt.Println` 所做的那样。

检查程序会首先关注打印函数的参数数量。如果参数数量不足，则可以认为在当前调用打印函数的语句中并不会出现用法错误。所以，检查程序会忽略对它的检查。检查程序中对打印函数的最小参数是这样定义的：对于可以

自定义输出的打印函数来说，最小参数数量为2，其它打印函数的最小参数数量为1。如果打印函数的实际参数数量小于对应的最小参数数量，就会被判定为参数数量不足。

对于格式化打印函数，检查程序会进行如下检查：

1. 如果格式化字符串无法被转换为基本字面量（标识符以及用于表示int类型值、float类型值、char类型值、string类型值的字面量等），则检查程序会忽略剩余的检查。如果 `-v` 标记有效，则会在忽略检查前打印错误信息。另外，格式化打印函数的格式化字符串必须是字符串类型的。因此，如果对应位置上的参数的类型不是字符串类型，那么检查程序会立即打印错误信息，并设置退出代码为1。实际上，这个问题已经可以引起一个编译错误了。

2. 如果格式化字符串中不包含动词（verbs），而格式化字符串后又有多余的参数，则检查程序会立即打印错误信息，并设置退出代码为1，且忽略后续检查。我现在举个例子。我们拿之前的一个示例作为基础，即：

   fmt.Printf("Hello, %s!\n", "Harry")

在这个示例中，格式化字符串中的“%s”就是我们所说的动词，“%”就是动词的前导符。它相当于一个需要被填的空。一般情况下，在格式化字符串中被填的空的数量应该与后续参数的数量相同。但是可以出现在格式化字符串中没有动词并且在格式化字符串之后没有额外参数的情况。在这种情况下，该格式化打印函数就相当于一个非格式化打印函数。例如，下面这个语句会导致此步检查不通过：

```
fmt.Printf("Hello!\n", "Harry")
```

1. 检查程序还会检查动词的格式。这部分检查会非常严格。检查程序对于格式化字符串中动词的格式要求如表0-19。表中对每个动词只进行了简要的说明。读者可以查看标准库代码包 `fmt` 的文档以了解关于它们的详细信息。命令程序会按照表5-19中的要求对格式化及其后续参数进行检查。如上表所示，这部分检查分为两步骤。第一个步骤是检查格式化字符串中的动词上是否附加了不合法的标记，第二个步骤是检查格式化字符串中的动词与后续对应的参数的类型是否匹配。只要检查出问题，检查程序就会打印出错误信息并且设置退出代码为1。

2. 如果格式化字符串中的动词不被支持，则检查程序同样会打印错误信息后，并设置退出代码为1。

表0-19 格式化字符串中动词的格式要求

动词	合法的附加标记	允许的参数类型	简要说明
b	“ ”，“-”，“+”，“.”和“0”	int或float	用于二进制表示法。
c	“-”	rune或int	用于单个字符的Unicode表示法。
d	“ ”，“-”，“+”，“.”和“0”	int	用于十进制表示法。

e	“ ”，“-”，“+”，“.”和“0”	float	用于科学记数法。
E	“ ”，“-”，“+”，“.”和“0”	float	用于科学记数法。
f	“ ”，“-”，“+”，“.”和“0”	float	用于控制浮点数精度。
F	“ ”，“-”，“+”，“.”和“0”	float	用于控制浮点数精度。
g	“ ”，“-”，“+”，“.”和“0”	float	用于压缩浮点数输出。
G	“ ”，“-”，“+”，“.”和“0”	float	用于动态选择浮点数输出格式。
o	“ ”，“-”，“+”，“.”，“0”和“#”	int	用于八进制表示法。
p	“-”和“#”	pointer	用于表示指针地址。
q	“ ”，“-”，“+”，“.”，“0”和“#”	rune，int或string	用于生成带双引号的字符串形式的内容。
s	“ ”，“-”，“+”，“.”和“0”	rune，int或string	用于生成字符串形式的内容。
t	“-”	bool	用于生成与布尔类型对应的字符串值。（“true”或“false”）
T	“-”	任何类型	用于用Go语法表示任何值的类型。
U	“-”和“#”	rune或int	用于针对Unicode的表示法。
v	“ ”，“-”，“+”，“.”，“0”和“#”	任何类型	以默认格式格式化任何值。
x	“ ”，“-”，“+”，“.”，“0”和“#”	rune，int或string	以十六进制、全小写的形式格式化每个字节。
X	“ ”，“-”，“+”，“.”，“0”和“#”	rune，int或string	以十六进制、全大写的形式格式化每个字节。

对于非格式化打印函数，检查程序会进行如下检查：

1. 如果打印函数不是可以自定义输出的打印函数，那么其第一个参数就不能是标准输出 `os.Stdout` 或者标准错误输出 `os.Stderr` 。否则，检查程序将打印错误信息并设置退出代码为1。这主要是为了防止程序编写人员的笔误。比如，他们可能会把函数 `fmt.Println` 当作函数 `fmt.Printf` 来用。

2. 如果打印函数是不自带换行的，比如 `fmt.Printf` 和 `fmt.Print` ，则它必须只少有一个参数。否则，检查程序将打印错误信息并设置退出代码为1。像这样的调用打印函数的语句是没有任何意义的。并且，如果这个打印函数还是一个格式化打印函数，那么这还会引起一个编译错误。需要注意的是，函数名称为 `Error` 的方法不会在被检查之列。比如，标准库代码包 `testing` 中的结构体类型 `T` 和 `B` 的方法 `Error` 。这是因为它们可能实现了接口类型 `Error` 。这个接口类型中唯一的方法 `Error` 无需任何参数。

3. 如果第一个参数的值为字符串类型的字面量且带有格式化字符串中才应该有的动词的前导符“%”，则检查程序会打印错误信息并设置退出代码为1。因为非格式化打印函数中不应该出现格式化字符串。

4. 如果打印函数是自带换行的，那么在打印内容的末尾就不应该有换行符“\n”。否则，检查程序会打印错误信息并设置退出代码为1。换句话说，检查程序认为程序中如果出现这样的代码：

   fmt.Println("Hello!\n")

常常是由于程序编写人员的笔误。实际上，事实确实如此。如果我们确实想连续输入多个换行，应该这样写：

```
fmt.Println("Hello!")
fmt.Println()
```

至此，我们详细介绍了 `go tool vet` 命令中的检查程序对打印函数的所有步骤和内容。打印函数的功能非常简单，但是 `go tool vet` 命令对它的检查却很细致。从中我们可以领会到一些关于打印函数的最佳实践。

-rangeloops标记

如果标记 `-rangeloop` 有效（标记值不为 `false` ），那么命令程序会对使用 `range` 进行迭代的 `for` 代码块进行检查。我们之前提到过，使用 `for` 语句需要注意两点：

1. 不要在 `go` 代码块中处理在迭代过程中被赋予值的迭代变量。比如：

   mySlice := []string{"A", "B", "C"} for index, value := range mySlice { go func() { fmt.Printf("Index: %d, Value: %s\n", index, value) }() }

在Go语言的并发编程模型中，并没有线程的概念，但却有一个特有的概念——Goroutine。Goroutine也可被称为Go例程或简称为Go程。关于Goroutine的详细介绍在第6章和第7章。我们现在只需要知道它是一个可以被并发执行的代码块。

1. 不要在 `defer` 语句的延迟函数中处理在迭代过程中被赋予值的迭代变量。比如：

   myDict := make(map[string]int) myDict["A"] = 1 myDict["B"] = 2 myDict["C"] = 3 for key, value := range myDict { defer func() { fmt.Printf("Key: %s, Value: %d\n", key, value) }() }

其实，上述两点所关注的问题是相同的，那就是不要在可能被延迟处理的代码块中直接使用迭代变量。 `go` 代码块和 `defer` 代码块都有这样的特质。这是因为等到go函数（跟在 `go` 关键字之后的那个函数）或延迟函数真正被执行的时候，这些迭代变量的值可能已经不是我们想要的值了。

另一方面，当检查程序发现在带有 `range` 子句的 `for` 代码块中迭代出的数据并没有赋值给标识符所代表的变量时，则会忽略对这一代码块的检查。比如像这样的代码：

```
func nonIdentRange(slc []string) {
    l := len(slc)
    temp := make([]string, l)
    l--
    for _, temp[l] = range slc {
        // 忽略了使用切片值temp的代码。
        if l > 0 {
            l--
        }
    }
}
```

就不会受到检查程序的关注。另外，当被迭代的对象的大小为 0 时， for 代码块也不会被检查。

据此，我们知道如果在可能被延迟处理的代码块中直接使用迭代中的临时变量，那么就可能会造成与编程人员意图不相符的结果。如果由此问题使程序的最终结果出现偏差甚至使程序报错的话，那么看起来就会非常诡异。这种隐晦的错误在排查时也是非常困难的。这种不正确的代码编写方式应该彻底被避免。这也是检查程序对迭代代码块进行检查的最终目的。如果检查程序发现了上述的不正确的代码编写方式，就会打印出错误信息以提醒编程人员。

-structtags标记

如果标记``-structtags 有效（标记值不为 false``），那么命令程序会对结构体类型的字段的标签进行检查。我们先来看下面的代码：

```
type Person struct {
    XMLName    xml.Name    `xml:"person"`
    Id         int         `xml:"id,attr"`
    FirstName  string      `xml:"name>first"`
    LastName   string      `xml:"name>last"`
    Age        int         `xml:"age"`
    Height     float32     `xml:"height,omitempty"`
    Married    bool
    Address
    Comment    string      `xml:",comment"`
}
```

在上面的例子中，在结构体类型的字段声明后面的那些字符串形式的内容就是结构体类型的字段的标签。对于Go语言本身来说，结构体类型的字段标签就是注释，它们是可选的，且会被Go语言的运行时系统忽略。但是，这些标签可以通过标准库代码包 reflect 中的程序访问到。因此，不同的代码包中的程序可能会赋予这些结构体类型的字段标签以不同的含义。比如上面例子中的结构体类型的字段标签就对代码包 encoding/xml 中的程序非常有用处。

严格来讲，结构体类型的字段的标签应该满足如下要求：

1. 标签应该包含键和值，且它们之间要用英文冒号分隔。
2. 标签的键应该不包含空格、引号或冒号。
3. 标签的值应该被英文双引号包含。
4. 如果标签内容符合了第3条，那么标签的全部内容应该被反引号“`”包含。否则它需要被双引号包含。
5. 标签可以包含多个键值对，其它们之间要用空格“ ”分隔。例如： `key:"value" _gofix:"_magic"`

检查程序首先会对结构体类型的字段标签的内容做去引号处理，也就是把最外面的双引号或者反引号去除。如果去除失败，则检查程序会打印错误信息并设置退出代码为1，同时忽略后续检查。如果去引号处理成功，检查程序则会根据前面的规则对标签的内容进行检查。如果检查出问题，检查程序同样会打印出错误信息并设置退出代码为1。

-unreachable标记

如果标记``-unreachable 有效（标记值不为 false```），那么命令程序会在函数或方法定义中查找死代码。死代码就是永远不会被访问到的代码。例如：

```
func deadCode1() int {
    print(1)
    return 2
    println() // 这里存在死代码
}
```

在上面示例中，函数 deadCode1 中的最后一行调用打印函数的语句就是死代码。检查程序如果在函数或方法中找到死代码，则会打印错误信息以提醒编码人员。我们把这段代码放到命令源码文件deadcode_demo.go中，并在main函数中调用它。现在，如果我们编译这个命令源码文件会马上看到一个编译错误：“missing return at end of function”。显然，这个错误侧面的提醒了我们，在这个函数中存在死代码。实际上，我们在修正这个问题之前它根本就不可能被运行，所以也就不存在任何隐患。但是，如果在这个函数不需要结果的情况下又会如何呢？我们稍微改造一下上面这个函数：

```
func deadCode1() {
    print(1)
    return
    println() // 这里存在死代码
}
```

好了，我们现在把函数 deadcode1 的声明中的结果声明和函数中 return 语句后的数字都去掉了。不幸的是，当我们再次编译文件时没有看到任何报错。但是，这里确实存在死代码。在这种情况下，编译器并不能帮助我们找到问题，而 go tool vet 命令却可以。

```
hc@ubt:~$ go tool vet deadcode_demo.go
deadcode_demo.go:10: unreachable code
```

`go tool vet` 命令中的检查程序对于死代码的判定有几个依据，如下：

1. 在这里，我们把 `return` 语句、 `goto` 语句、 `break` 语句、 `continue` 语句和 `panic` 函数调用语句都叫做流程中断语句。如果在当前函数、方法或流程控制代码块的分支中的流程中断语句的后面还存在其他语句或代码块，比如：

   `func deadCode2() { print(1) panic(2) println() // 这里存在死代码 }`

   或

   `func deadCode3() { L: { print(1) goto L } println() // 这里存在死代码 }`

   或

   `func deadCode4() { print(1) return { // 这里存在死代码 } }`

则后面的语句或代码块就会被判定为死代码。但检查程序仅会在错误提示信息中包含第一行死代码的位置。

1. 如果带有 `else` 的 `if` 代码块中的每一个分支的最后一条语句均为流程中断语句，则在此流程控制代码块后的代码都被判定为死代码。比如：

   `func deadCode5(x int) { print(1) if x == 1 { panic(2) } else { return } println() // 这里存在死代码 }`

注意，只要其中一个分支不包含流程中断语句，就不能判定后面的代码为死代码。像这样：

```
func deadCode5(x int) {
   print(1)
   if x == 1 {
      panic(2)
   } else if x == 2 {
      return
   }
   println() // 这里并不是死代码
}
```

1. 如果在一个没有显式中断条件或中断语句的 `for` 代码块后面还存在其它语句，则这些语句将会被判定为死代码。比如：

   `func deadCode6() { for { for { break } } println() // 这里存在死代码 }`

或

```
func deadCode7() {
    for {
        for {
        }
        break // 这里存在死代码
    }
    println()
}
```

而我们对这两个函数稍加改造后，就会消除 `go tool vet` 命令发出的死代码告警。如下：

```
func deadCode6() {
    x := 1
    for x == 1 {
        for {
            break
        }
    }
    println() // 这里存在死代码
}
```

以及

```
func deadCode7() {
    x := 1
    for {
        for x == 1 {
        }
        break // 这里存在死代码
    }
    println()
}
```

我们只是加了一个显式的中断条件就能够使之通过死代码检查。但是，请注意！这两个函数中在被改造后仍然都包含死循环代码！这说明检查程序并不对中断条件的逻辑进行检查。

1. 如果 `select` 代码块的所有 `case` 中的最后一条语句均为流程中断语句（ `break` 语句除外），那么在 `select` 代码块后面的语句都会被判定为死代码。比如：

   ```
   func deadCode8(c chan int) { print(1) select { case <-c: print(2) panic(3) } println() // 这里存在死代码 }
   ```

或

```
func deadCode9(c chan int) {
L:
```

```
    print(1)
    select {
    case <-c:
        print(2)
        panic(3)
    case c <- 1:
        print(4)
        goto L
    }
    println() // 这里存在死代码
}
```

另外，在空的 select 语句块之后的代码也会被认为是死代码。比如：

```
func deadCode10() {
    print(1)
    select {}
    println() // 这里存在死代码
}
```

或

```
func deadCode11(c chan int) {
    print(1)
    select {
    case <-c:
        print(2)
        panic(3)
    default:
        select {}
    }
    println() // 这里存在死代码
}
```

上面这两个示例中的语句 select {} 都会引发一个运行时错误：“fatal error: all goroutines are asleep - dead lock!”。这就是死锁！关于这个错误的详细说明在第7章。

1. 如果 switch 代码块的所有 case 和 default case 中的最后一条语句均为流程中断语句（除了 break 语句），那么在 switch 代码块后面的语句都会被判定为死代码。比如：

   func deadCode14(x int) { print(1) switch x { case 1: print(2) panic(3) default: return } println(4) // 这里存在死代码 }

我们知道，关键字 fallthrough 可以使流程从 switch 代码块中的一个 case 转移到下一个 case 或 default case 。死代码也可能由此产生。例如：

```
func deadCode15(x int) {
    print(1)
    switch x {
    case 1:
        print(2)
        fallthrough
    default:
        return
    }
    println(3) // 这里存在死代码
}
```

在上面的示例中，第一个case总会把流程转移到第二个case，而第二个case中的最后一条语句为return语句，所以流程永远不会转移到语句 println(3) 上。因此， println(3) 语句会被判定为死代码。如果我们把 fallthrough 语句去掉，那么就可以消除这个死代码判定。实际上，只要某一个 case 或者 default case 中的最后一条语句是break语句，就不会有死代码的存在。当然，这个 break 语句本身不能是死代码。另外，与 select 代码块不同的是，空的 switch 代码块并不会使它后面的代码成为死代码。

综上所述，死代码的判定虽然看似比较复杂，但其实还是有原则可循的。我们应该在编码过程中就避免编写可能会造成死代码的代码。如果我们实在不确定死代码是否存在，也可以使用 go tool vet 命令来检查。不过，需要提醒读者的是，不存在死代码并不意味着不存在造成死循环的代码。当然，造成死循环的代码也并不一定就是错误的代码。但我们仍然需要对此保持警觉。

-asmdecl标记

如果标记``-asmdecl 有效（标记值不为 false```），那么命令程序会对汇编语言的源码文件进行检查。对汇编语言源码文件及相应编写规则的解读已经超出了本书的范围，所以我们并不在这里对此项检查进行描述。如果读者有兴趣的话，可以查看此项检查的程序的源码文件asmdecl.go。它在Go语言安装目录的子目录src/cmd/vet下。

至此，我们对 go vet 命令和 go tool vet 命令进行了全面详细的介绍。之所以花费如此大的篇幅来介绍这两个命令，不仅仅是为了介绍此命令的使用方法，更是因为此命令程序的检查工作涉及到了很多我们在编写Go语言代码时需要避免的“坑”。由此我们也可以知晓应该怎样正确的编写Go语言代码。同时，我们也应该在开发Go语言程序的过程中经常使用 go tool vet 命来检查代码。

go tool pprof

我们可以使用 go tool pprof 命令来交互式的访问概要文件的内容。命令将会分析指定的概要文件，并会根据我们的要求为我们提供高可读性的输出信息。

在Go语言中，我们可以通过标准库的代码包 runtime 和 runtime/pprof 中的程序来生成三种包含实时性数据的概要文件，分别是CPU概要文件、内存概要文件和程序阻塞概要文件。下面我们先来分别介绍用于生成这三种概要文件的API的用法。

CPU概要文件

在介绍CPU概要文件的生成方法之前，我们先来简单了解一下CPU主频。CPU的主频，即CPU内核工作的时钟频率（CPU Clock Speed）。CPU的主频的基本单位是赫兹（Hz），但更多的是以兆赫兹（MHz）或吉赫兹（GHz）为单位。时钟频率的倒数即为时钟周期。时钟周期的基本单位为秒（s），但更多的是以毫秒（ms）、微妙（us）或纳秒（ns）为单位。在一个时钟周期内，CPU执行一条运算指令。也就是说，在1000 Hz的CPU主频下，每1毫秒可以执行一条CPU运算指令。在1 MHz的CPU主频下，每1微妙可以执行一条CPU运算指令。而在1 GHz的CPU主频下，每1纳秒可以执行一条CPU运算指令。

在默认情况下，Go语言的运行时系统会以100 Hz的的频率对CPU使用情况进行取样。也就是说每秒取样100次，即每10毫秒会取样一次。为什么使用这个频率呢？因为100 Hz既足够产生有用的数据，又不至于让系统产生停顿。并且100这个数上也很容易做换算，比如把总取样计数换算为每秒的取样数。实际上，这里所说的对CPU使用情况的取样就是对当前的Goroutine的堆栈上的程序计数器的取样。由此，我们就可以从样本记录中分析出哪些代码是计算时间最长或者说最耗CPU资源的部分了。我们可以通过以下代码启动对CPU使用情况的记录。

```
func startCPUProfile() {
    if *cpuProfile != "" {
        f, err := os.Create(*cpuProfile)
        if err != nil {
            fmt.Fprintf(os.Stderr, "Can not create cpu profile output file: %s",
                err)
            return
        }
        if err := pprof.StartCPUProfile(f); err != nil {
            fmt.Fprintf(os.Stderr, "Can not start cpu profile: %s", err)
            f.Close()
            return
        }
    }
}
```

在函数 startCPUProfile 中，我们首先创建了一个用于存放CPU使用情况记录的文件。这个文件就是CPU概要文件，其绝对路径由 *cpuProfile 的值表示。然后，我们把这个文件的实例作为参数传入到函数```pprof.StartCPUProfile``中。如果此函数没有返回错误，就说明记录操作已经开始。需要注意的是，只有CPU概要文件的绝对路径有效时此函数才会开启记录操作。

如果我们想要在某一时刻停止CPU使用情况记录操作，就需要调用下面这个函数：

```
func stopCPUProfile() {
    if *cpuProfile != "" {
        pprof.StopCPUProfile() // 把记录的概要信息写到已指定的文件
    }
}
```

在这个函数中，并没有代码用于CPU概要文件写入操作。实际上，在启动CPU使用情况记录操作之后，运行时系统就会以每秒100次的频率将取样数据写入到CPU概要文件中。 pprof.StopCPUProfile 函数通过把CPU使用情况取样的频率设置为0来停止取样操作。并且，只有当所有CPU使用情况记录都被写入到CPU概要文件之后， pprof.StopCPUProfile 函数才会退出。从而保证了CPU概要文件的完整性。

内存概要文件

内存概要文件用于保存在用户程序执行期间的内存使用情况。这里所说的内存使用情况，其实就是程序运行过程中堆内存的分配情况。Go语言运行时系统会对用户程序运行期间的所有的堆内存分配进行记录。不论在取样的那一时刻、堆内存已用字节数是否有增长，只要有字节被分配且数量足够，分析器就会对其进行取样。开启内存使用情况记录的方式如下：

```
func startMemProfile() {
    if *memProfile != "" && *memProfileRate > 0 {
        runtime.MemProfileRate = *memProfileRate
    }
}
```

我们可以看到，开启内存使用情况记录的方式非常简单。在函数 startMemProfile 中，只有在 *memProfile 和 *memProfileRate 的值有效时才会进行后续操作。 *memProfile 的含义是内存概要文件的绝对路径。 *memProfileRate 的含义是分析器的取样间隔，单位是字节。当我们将这个值赋给int类型的变量 runtime.MemProfileRate 时，就意味着分析器将会在每分配指定的字节数量后对内存使用情况进行取样。实际上，即使我们不给 runtime.MemProfileRate 变量赋值，内存使用情况的取样操作也会照样进行。此取样操作会从用户程序开始时启动，且一直持续进行到用户程序结束。 runtime.MemProfileRate 变量的默认值是 512 * 1024 ，即512K个字节。只有当我们显式的将 0 赋给 runtime.MemProfileRate 变量之后，才会取消取样操作。

在默认情况下，内存使用情况的取样数据只会被保存在运行时内存中，而保存到文件的操作只能由我们自己来完成。请看如下代码：

```
func stopMemProfile() {
    if *memProfile != "" {
        f, err := os.Create(*memProfile)
        if err != nil {
            fmt.Fprintf(os.Stderr, "Can not create mem profile output file: %s", err)
            return
        }
        if err = pprof.WriteHeapProfile(f); err != nil {
            fmt.Fprintf(os.Stderr, "Can not write %s: %s", *memProfile, err)
        }
        f.Close()
    }
}
```

从函数名称上看， stopMemProfile 函数的功能是停止对内存使用情况的取样操作。但是，它只做了将取样数据保存到内存概要文件的操作。在 stopMemProfile 函数中，我们调用了函数 pprof.WriteHeapProfile ，并把代表内存概要文件的文件实例作为了参数。如果 pprof.WriteHeapProfile 函数没有返回错误，就说明数据已被写入到了内存概要文件中。

需要注意的是，对内存使用情况进行取样的程序会假定取样间隔在用户程序的运行期间内都是一成不变的，并且等于 runtime.MemProfileRate 变量的当前值。因此，我们应该在我们的程序中只改变内存取样间隔一次，且应尽早改变。比如，在命令源码文件的main函数的开始处就改变它。

程序阻塞概要文件

程序阻塞概要文件用于保存用户程序中的Goroutine阻塞事件的记录。我们来看开启这项操作的方法：

```
func startBlockProfile() {
    if *blockProfile != "" && *blockProfileRate > 0 {
        runtime.SetBlockProfileRate(*blockProfileRate)
    }
}
```

与开启内存使用情况记录的方式类似，在函数 startBlockProfile 中，当 *blockProfile 和 *blockProfileRate 的值有效时，我们会设置对Goroutine阻塞事件的取样间隔。 *blockProfile 的含义为程序阻塞概要文件的绝对路径。 *blockProfileRate 的含义是分析器的取样间隔，单位是次。函数 runtime.SetBlockProfileRate 的唯一参数是int类型的。它的含义是分析器会在每发生几次Goroutine阻塞事件时对这些事件进行取样。如果我们不显式的使用 runtime.SetBlockProfileRate 函数设置取样间隔，那么取样间隔就为1。也就是说，在默认情况下，每发生一次Goroutine阻塞事件，分析器就会取样一次。与内存使用情况记录一样，运行时系统对Goroutine阻塞事件的取样操作也会贯穿于用户程序的整个运行期。但是，如果我们通过 runtime.SetBlockProfileRate 函数将这个取样间隔设置为 0 或者负数，那么这个取样操作就会被取消。

我们在程序结束之前可以将被保存在运行时内存中的Goroutine阻塞事件记录存放到指定的文件中。代码如下：

```
func stopBlockProfile() {
    if *blockProfile != "" && *blockProfileRate >= 0 {
        f, err := os.Create(*blockProfile)
        if err != nil {
            fmt.Fprintf(os.Stderr, "Can not create block profile output file: %s", err)
            return
        }
        if err = pprof.Lookup("block").WriteTo(f, 0); err != nil {
            fmt.Fprintf(os.Stderr, "Can not write %s: %s", *blockProfile, err)
        }
        f.Close()
    }
}
```

在创建程序阻塞概要文件之后， `stopBlockProfile` 函数会先通过函数 `pprof.Lookup` 将保存在运行时内存中的内存使用情况记录取出，并在记录的实例上调用 `WriteTo` 方法将记录写入到文件中。

更多的概要文件

我们可以通过 `pprof.Lookup` 函数取出更多种类的取样记录。如下表：

表0-20 可从pprof.Lookup函数中取出的记录

名称	说明	取样频率
goroutine	活跃Goroutine的信息的记录。	仅在获取时取样一次。
threadcreate	系统线程创建情况的记录。	仅在获取时取样一次。
heap	堆内存分配情况的记录。	默认每分配512K字节时取样一次。
block	Goroutine阻塞事件的记录。	默认每发生一次阻塞事件时取样一次。

在上表中，前两种记录均为一次取样的记录，具有即时性。而后两种记录均为多次取样的记录，具有实时性。实际上，后两种记录“heap”和“block”正是我们前面讲到的内存使用情况记录和程序阻塞情况记录。

我们知道，在用户程序运行期间各种状态是在不断变化的。尤其对于后两种记录来说，随着取样次数的增多，记录项的数量也会不断增长。而对于前两种记录“goroutine”和“threadcreate”来说，如果有新的活跃Goroutine产生或新的系统线程被创建，其记录项数量也会增大。所以，Go语言的运行时系统在从内存中获取记录时都会先预估一个记录项数量。如果在从预估记录项数量到获取记录之间的时间里又有新记录项产生，那么运行时系统会试图重新获取全部记录项。另外，运行时系统使用切片来装载所有记录项的。如果当前使用的切片装不下所有记录项，运行时系统会根据当前记录项总数创建一个更大的切片，并再次试图装载所有记录项。直到这个切片足以装载所有的记录项为止。但是，如果记录项增长过快的话，运行时系统将不得不不断的进行尝试。这可能导致过多的时间消耗。对于前两种记录“goroutine”和“threadcreate”来说，运行时系统创建的切片的大小为当前记录项总数再加10。对于前两种记录“heap”和“block”来说，运行时系统创建的切片的大小为当前记录项总数再加50。虽然上述情况发生的概率可能并不会太高，但是如果我们在对某些高并发的用户程序获取上述记录

的时候耗费的时间过长，可以先排查一下这类原因。实际上，我们在前面介绍的这几 种记录操作更适合用于对高并发的用户程序进行状态检测。

我们可以通过下面这个函数分别将四种记录输出到文件。

```
func SaveProfile(workDir string, profileName string, ptype ProfileType, debug int) {
    absWorkDir := getAbsFilePath(workDir)
    if profileName == "" {
        profileName = string(ptype)
    }
    profilePath := filepath.Join(absWorkDir, profileName)
    f, err := os.Create(profilePath)
    if err != nil {
        fmt.Fprintf(os.Stderr, "Can not create profile output file: %s", err)
        return
    }
    if err = pprof.Lookup(string(ptype)).WriteTo(f, debug); err != nil {
        fmt.Fprintf(os.Stderr, "Can not write %s: %s", profilePath, err)
    }
    f.Close()
}
```

函数 SaveProfile 有四个参数。第一个参数是概要文件的存放目录。第二个参数是概要文件的名称。第三个参数是概要文件的类型。其中，类型 ProfileType 只是为string类型起的一个别名而已。这样是为了对它的值进行限制。它的值必须为“goroutine”、“threadcreate”、“heap”或“block”中的一个。我们现在来重点说一下第四个参数。参数 debug 控制着概要文件中信息的详细程度。这个参数也就是传给结构体 pprof.Profile 的指针方法 WriteTo 的第二个参数。而 pprof.Profile 结构体的实例的指针由函数 pprof.Lookup 产生。下面我们看看参数debug的值与写入概要文件的记录项内容的关系。

表0-21 参数debug对概要文件内容的影响

记录\debug	小于等于0	等于1	大于等于2
goroutine	为每个记录项提供调用栈中各项的以十六进制表示的内存地址。	在左边提供的信息的基础上，为每个记录项的调用栈中各项提供与内存地址对应的带代码包导入路径的函数名和源码文件路径及源码所在行号。	以高可读的方式提供各活跃Goroutine的状态信息和调用栈信息。
threadcreate	同上。	同上。	同左。

heap	同上。	在左边提供的信息的基础上，为每个记录项的调用栈中各项提供与内存地址对应的带代码包导入路径的函数名和源码文件路径及源码所在行，并提供内存状态信息。	同左。
block	同上。	在左边提供的信息的基础上，为每个记录项的调用栈中各项提供与内存地址对应的带代码包导入路径的函数名和源码文件路径及源码所在行号。	同左。

从上表可知，当 debug 的值小于等于 0 时，运行时系统仅会将每个记录项中的基本信息写入到概要文件中。记录项的基本信息只包括其调用栈中各项的以十六进制表示的内存地址。 debug 的值越大，我们能从概要文件中获取的信息越多。但是， go tool pprof 命令会无视那些除基本信息以外的附加信息。实际上，运行时系统在向概要文件中写入附加信息时会在最左边加入“#”，以表示当前行为注释行。也正因为有了这个前缀， go tool pprof 命令才会略过对这些附加信息的解析。这其中有一个例外，那就是当 debug 大于等于 2 时，Goroutine记录并不是在基本信息的基础上附加信息，而是完全以高可读的方式写入各活跃Goroutine的状态信息和调用栈信息。并且，在所有行的最左边都没有前缀“#”。显然，这个概要文件是无法被 go tool pprof 命令解析的。但是它对于我们来说会更加直观和有用。

至此，我们已经介绍了使用标准库代码包 runtime 和 runtime/pprof 中的程序生成概要文件的全部方法。在上面示例中的所有代码都被保存在goc2p项目的代码包 basic/prof 中。代码包 basic/prof 中的这些程序非常易于使用。不过由于Go语言目前没有类似停机钩子（Shutdown Hook）的API（应用程序接口），所以代码包 basic/prof 中的程序目前只能以侵入式的方式被使用。

pprof工具

我们在上一小节中提到过，任何以 go tool 开头的Go命令内部指向的特殊工具都被保存在目录$GOROOT/pkg/tool/$GOOS_$GOARCH/中。我们把这个目录叫做Go工具目录。与其他特殊工具不同的是，pprof工具并不是用Go语言编写的，而是由Perl语言编写的。（Perl是一种通用的、动态的解释型编程语言）与Go语言不同，Perl语言可以直接读取源码并运行。正因为如此， pprof 工具的源码文件被直接保存在了Go工具目录下。而对于其它Go工具，存在此目录的都是经过编译而生成的可执行文件。我们可以直接用任意一种文本查看工具打开在Go工具目录下的pprof工具的源码文件pprof。实际上，这个源码文件拷贝自Google公司发起的开源项目gperftools。此项目中包含了很多有用的工具。这些工具可以帮助开发者创建更健壮的应用程序。pprof就是其中的一个非常有用的工具。

因为 pprof 工具是用Perl语言编写的，所以执行 go tool pprof 命令的前提条件是需要在当前环境下安装Perl语言，推荐的版本号是5.x。关于Perl语言的安装方法就不在这里叙述了，读者可以自己找到方法并自行安装它。在安装完Perl语言之后，我们可以在命令行终端中进入到Go工具目录，并执行命令 perl pprof 。它与我们在任意目录下执行 go tool pprof 命令的效果是一样的。当然，如果想要让 go tool pprof 命令在任意目录下都可以被执行，我们需要先设置好与Go语言相关的环境变量。

我们在本小节已经讨论过， go tool pprof 命令会分析指定的概要文件并使得我们能够以交互式的方式访问其中的信息。但是光有概要文件还不够，我们还需要概要文件中信息的来源——命令源码文件的可执行文件。毕竟，概

要文件中的信息是对在运行期间的用户程序取样的结果。而可以运行的Go语言程序只能是编译命令源码文件后生成的可执行文件。因此，为了演示 go tool pprof 命令的用法，我们还创建或改造一个命令源码文件。在我们的goc2p项目的代码包中有一个名称为showpds.go的命令源码文件。这个命令源码文件用来解析指定的代码包的依赖关系，并将这些依赖关系打印到标准输出设备上。选用这个命令源码文件的原因是，我们可以通过改变指定的代码包来控制这个命令源码文件的运行时间的长短。不同的代码包可能会有不同数量的直接依赖包和间接依赖包。依赖包越多的代码包会使这个命令源码文件耗费更多的时间来解析它的依赖关系。命令源码文件运行的时间越长，我们得到的概要文件中的信息就越有意义。为了生成概要文件，我们需要稍微的改造一下这个命令源码文件。首先我们需要在这个文件中导入代码包 basic/prof 。然后，我们需要在它的main函数的开头加入一行代码 prof.Start() 。这行代码的含义是检查相关标记，并在标记有效时开启或设置对应的使用情况记录操作。最后，我们还需要在main函数的defer代码块中加入一行代码 prof.Stop() 。这行代码的含义是，获取已开启的记录的取样数据并将它们写入到指定的概要文件中。通过这三个步骤，我们就已经把生成运行时概要文件的功能附加到这个命令源码文件中了。为了开启这些功能，我还需要在通过执行 go run 命令来运行这个命令源码文件的时候，加入相应的标记。对代码包 basic/prof 中的程序有效的标记如下表。

表0-22 对代码包basic/prof的API有效的标记

标记名称	标记描述
-cpuprofile	指定CPU概要文件的保存路径。该路径可以是相对路径也可以是绝对路径，但其父路径必须已存在。
-blockprofile	指定程序阻塞概要文件的保存路径。该路径可以是相对路径也可以是绝对路径，但其父路径必须已存在。
-blockprofilerate	定义其值为n。此标记指定每发生n次Goroutine阻塞事件时，进行一次取样操作。
-memprofile	指定内存概要文件的保存路径。该路径可以是相对路径也可以是绝对路径，但其父路径必须已存在。
-memprofilerate	定义其值为n。此标记指定每分配n个字节的堆内存时，进行一次取样操作。

下面我们使用 go run 命令运行改造后的命令源码文件showpds.go。示例如下：

```
hc@ubt:~/golang/goc2p$ mkdir pprof
hc@ubt:~/golang/goc2p$ cd helper/pds
hc@ubt:~/golang/goc2p/helper/pds$ go run showpds.go -p="runtime" cpuprofile="../../../pprof/cpu.out" -blockprofile="../../
The package node of 'runtime': {/usr/local/go/src/pkg/ runtime [] [] false}
The dependency structure of package 'runtime':
runtime->unsafe
```

在上面的示例中，我们使用了所有的对代码包 basic/prof 的API有效的标记。另外，标记 -p 是对命令源码文件showpds.go有效的。其含义是指定要解析依赖关系的代码包的导入路径。

现在我们来查看一下goc2p项目目录下的pprof子目录：

```
hc@ubt:~/golang/goc2p/helper/pds$ ls ../../../pprof
block.out  cpu.out  mem.out
```

这个目录中的三个文件分别对应了三种包含实时性数据的概要文件。这也证明了我们对命令源码文件showpds.go的改造是有效的。

好了，一切准备工作就绪。现在，我们就来看看 go tool pprof 命令都能做什么。首先，我们来编译命令源码文件showpds.go。

```
hc@ubt:~/golang/goc2p/helper/pds$ go build showpds.go
hc@ubt:~/golang/goc2p/helper/pds$ ls
showpds showpds.go
```

然后，我们需要准备概要文件。标准库代码包 runtime 的依赖包极少，这使得可执行文件showpds在极短的时间内就会运行完毕。之前我们说过，程序运行的时间越长越好。所以我们需要找到一个直接和间接依赖包都很多的代码包。做过Web应用系统开发的同行们都知道，一个Web应用系统的后端程序可能会有很多的依赖，不论是代码库还是其他资源。根据我们的直觉，在Go语言的世界里也应该是在这样。在Go语言的标准库中，代码包 net/http 专门用来为Web应用系统开发提供各种API支持。我们就用这个代码包来生成所需的概要文件。

```
hc@ubt:~/golang/goc2p/helper/pds$ ./showpds -p="net/http" -cpuprofile="../../../pprof/cpu.out" -blockprofile="../../../pprof/
```

标准库代码包 net/http 的依赖包很多。也正因为如此，我忽略了所有输出的内容。读者可以自己试试上面的这个命令。我们一口气生成了所有能够生成的概要文件作为备用。这写概要文件被保存在了goc2p项目的pprof目录中。如果在上面的命令被执行前还没有pprof目录，命令会报错。所以读者需要先创建这个目录。

现在我们就以可执行文件showpds和pprof目录下的CPU概要文件cpu.out作为参数来执行 go tool pprof 命令。实际上，我们通过 go tool pprof 命令进入的就是pprof工具的交互模式的界面。

```
hc@ubt:~/golang/goc2p/helper/pds$ go tool pprof showpds ../../../pprof/cpu.out
Welcome to pprof!  For help, type 'help'.
(pprof)
```

我们可以在提示符“(pprof)”后面输入一些命令来查看概要文件。pprof工具在交互模式下支持的命令如下表。

表0-23 pprof工具在交互模式下支持的命令

名称	参数	标签	说明
gv	[focus]		将当前概要文件以图形化和层次化的形式显示出来。当没有任何参数时，在概要文件中的所有抽样都会被显示。如果指定了focus参数，则只显示调用栈中有名称与此参数相匹配的函数或方法的抽样。focus参数应该是一个正则表达式。
web	[focus]		与gv命令类似，web命令也会用图形化的方式来显示概要文件。但不同的是，web命令是在一个Web浏览器中显示它。如果你的Web浏览器已经启动，那么它的显示速度会非常快。如果想改变所使用的Web浏览器，可以在Linux下设置符号链接/etc/alternatives/gnome-www-browser或/etc/alternatives/x-www-browser，或在OS X下改变SVG文件的关联Finder。

list	[routine_regexp]		列出名称与参数“routine_regexp”代表的正则表达式相匹配的函数或方法的相关源代码。
weblist	[routine_regexp]		在Web浏览器中显示与list命令的输出相同的内容。它与list命令相比的优势是，在我们点击某行源码时还可以显示相应的汇编代码。
top[N]		[--cum]	top命令可以以本地取样计数为顺序列出函数或方法及相关信息。如果存在标记“--cum”则以累积取样计数为顺序。默认情况下top命令会列出前10项内容。但是如果在top命令后面紧跟一个数字，那么其列出的项数就会与这个数字相同。
disasm	[routine_regexp]		显示名称与参数“routine_regexp”相匹配的函数或方法的反汇编代码。并且，在显示的内容中还会标注有相应的取样计数。
callgrind	[filename]		利用callgrind工具生成统计文件。在这个文件中，说明了程序中函数的调用情况。如果未指定“filename”参数，则直接调用kcachegrind工具。kcachegrind可以以可视化的方式查看callgrind工具生成的统计文件。
help			显示帮助信息。
quit			退出go tool pprof命令。Ctrl-d也可以达到同样效果。

在上面表格中的绝大多数命令（除了help和quit）都可以在其所有参数和标签后追加一个或多个参数，以表示想要忽略显示的函数或方法的名称。我们需要在此类参数上加入减号“-”作为前缀，并且多个参数之间需要以空格分隔。当然，我们也可以用正则表达式替代函数或方法的名称。追加这些约束之后，任何调用栈中包含名称与这类参数相匹配的函数或方法的抽样都不会出现在命令的输出内容中。下面我们对这几个命令进行逐一说明。

gv命令

对于命令gv的用法，请看如下示例：

```
hc@ubt:~/golang/goc2p/helper/pds$ go tool pprof showpds ../../../pprof/cpu.out
Welcome to pprof!  For help, type 'help'.
(pprof) gv
Total: 101 samples
```

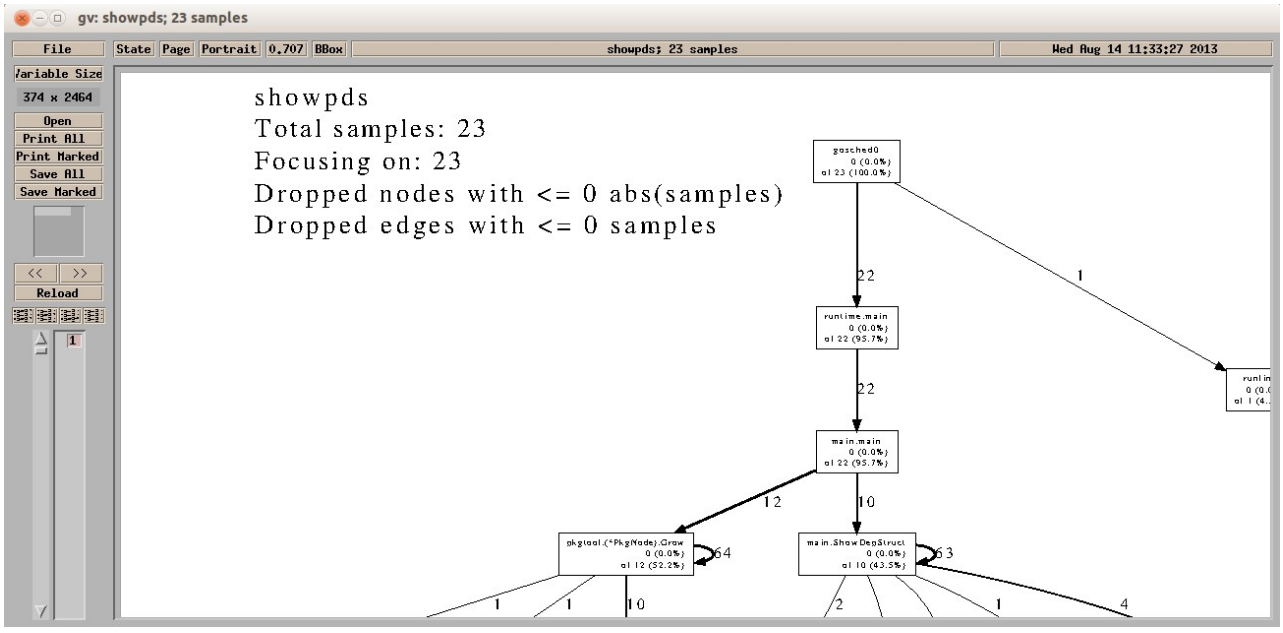
```
sh: 1: dot: not found
go tool pprof: signal: broken pipe
```

其中，“(pprof)”是pprof工具在交互模式下的提示符。

从输出信息中我们可以看到，gv命令并没有正确的被执行。原因是没有找到命令dot。经查，这个命令属于一个开源软件Graphviz。Graphviz的核心功能是图表的可视化。我们可以通过命令 `sudo apt-get install graphviz` 来安装这个软件。注意，上面这条命令仅可用于Debian的Linux发行版及其衍生版。如果是在Redhat的Linux发行版及其衍生版下，可以使用命令“yum install graphviz”来安装Graphviz。安装好Graphviz后，我们再来执行gv命令。

```
(pprof) gv
Total: 101 samples
gv -scale 0
(pprof) sh: 1: gv: not found
```

现在，输出信息有提示我们没有找到命令gv。gv是自由软件工程项目GNU（GNU's Not Unix）中的一款开源软件，用来以图形化的方式查看PDF文档。我们以同样的方式安装它。在Debian的Linux发行版及其衍生版下，执行命令 `sudo apt-get install gv` ，在Redhat的Linux发行版及其衍生版下，执行命令 `yum install gv` 。软件gv被安装好后，我们再次执行gv命令。在运行着图形界面软件的Linux操作系统下，会弹出这样一个窗口。如图5-3。



图片 1.3 pprof工具的gv命令的执行结果

图0-3 pprof工具的gv命令的执行结果

我们看到，在概要图的最上面显示了一些基本的信息。其中，“showpds”是我们生成概要文件时用到的那个可执行文件。它也是概要文件中内容的来源。“Total samples：”后面的数字23的含义是在本次程序执行期间分析器一共进行了23次取样。我们已经知道，CPU使用情况的取样操作会以每10毫秒一次的频率进行。因此，取样

23次就意味着程序运行所花费的CPU时间大约为 10毫秒 * 23 = 0.23秒 。由于我们并没有在gv命令后加入用于约束显示内容的参数focus，所以在“Focusing on: ”后面的数字也是23。也正是由于这个原因，后边两行信息中的数字均为0。读者可以自行试验一下在gv命令后加入focus参数的情形，例如： gv ShowDepStruct 。在下面的描述中，我们把函数和方法统称为函数。

现在，我们把视线放在主要的图形上。此图形由矩形和有向线段组成。在此图形的大多数矩形中都包含三行信息。第一行是函数的名字。第二行包含了该函数的本地取样计数（在括号左边的数字）及其在取样总数中所占的比例（在括号内的百分比）。第三行则包含了该函数的累积取样计数（括号左边的数字）及其在取样总数中所占的比例（在括号内的百分比）。

首先，读者需要搞清楚两个相似但不相同的概念，即：本地取样计数和累积取样计数。本地取样计数的含义是当前函数在取样中直接出现的次数。累积取样计数的含义是当前函数以及当前函数直接或间接调用的函数在取样中直接出现的次数。所以，存在这样一种场景：对于一个函数来说，它的本地取样计数是0。因为它没有在取样中直接出现过。但是，由于它直接或间接调用的函数频繁的直接出现在取样中，所以这个函数的累积取样计数却会很高。我们以上图中的函数mian.main为例。由于main.main函数在所有取样中都没有直接出现过，所以它的本地取样计数为0。但又由于它是命令源码文件中的入口函数，程序中其他的函数都直接或间接的被它调用。所以，它的累积取样计数是所有函数中最高的，达到了22。注意，不论是本地取样计数还是累积取样计数都没有把函数对自身的调用计算在内。函数对自身的调用又被称为递归调用。

最后需要说明的是，图形中的有向线段表示函数之间的调用关系。有向线段旁边的数字为线段起始位置的函数对线段末端位置的函数的调用计数。这里所说的调用计数其实是以函数的累积取样计数为依托的。更具体的讲，如果有一个从函数A到函数B的有向线段且旁边的数字为10，那么就说明在函数B的累加取样计数中有10次计数是由函数A对函数B的直接调用所引起的。也由于这个原因，函数A对函数B的调用计数必定小于等于函数B的累积取样计数。

至此，我们已经对概要图中的所有元素都进行了说明，相信读者已经能够读懂它了。那么，我们怎样通过概要图对程序进行分析呢?

我们可以把概要图中的这个主要图形看成是一张函数调用关系图。在一般情况下，处在非终端节点位置的函数的本地取样计数会非常小，至少会比该函数的累积取样计数小很多。因为它们都是通过对其它函数的调用来实现自身的功能的。进一步说，所有使用Go语言编写的代码的功能最后都需要依托操作系统所提供的API来实现。处在终端节点位置的函数一般都存在于平台相关的源码文件中，甚至有的函数本身就是操作系统的某个API在Go语言中的映射。它们的累积取样计数与本地取样计数是一致的。因此，这类函数的描述信息只有两行，即它的名称和它的累积取样计数。

现在我们已经明确了在概要图中出现的一个函数的本地取样计数、累积取样计数和调用计数的概念和含义以及它们之间的关系。这三个计数是我们分析程序性能的重要依据。

我们可以通过一个函数的累积取样次数计算出执行它所花费的时间。一个函数的累积取样计数越大就说明调用它所花费的CPU时间越多。具体来说，我们可以用CPU取样间隔（10毫秒）乘以函数的累积取样计数得出它所花费的实际时间。虽然这个实际时间只精确到了10毫秒的级别，但是这对于程序性能分析来说已经足够了。即使一个函数的累积取样计数很大，我们也不能判定这个函数本身就是有问题的。我们应该顺藤摸瓜，去寻找这个函数直接或间接调用的函数中最耗费CPU时间的那些函数。其实，这样的查找很容易，因为我们已经有了概要图。在其中的函数调用关系图中，累积取样计数越大的函数就拥有更大的节点（图中的矩形）面积。不过这也有例外，那就是程序的入口函数。广义来讲，在整个函数调用关系中处在初始位置附近且与之相连的有向线段在同一方向上至多只有一个的函数都可以被称作入口函数。无论它们的累积取样计数有多大，其所属的节点的面积都是在函数调用关系图中最小的。由于出现在取样和函数调用关系图中的所有函数几乎都源自入口函数的直接或间接的调用，所以入口函数的累积取样次数必定是它们中最大的。一般情况下，我们并不需要在意入口函数的计数数值，所以在函数调用关系图中也就不需要使用大面积的节点来强调它们。在图5-3中，函数 runtime.main 和 main.main 都可以被视为入口函数。另外，在函数调用关系图中，有向线段的粗细也反应了对应的调用计数的大小。

下面，作者总结了根据函数的相关计数来对其进行分析的三个过程：

1. 如果一个处在终端节点位置上的函数的累积取样计数和百分比都很大，就说明它自身花费了过多的CPU时间。这时，需要检查这个函数所实现的功能是否确实需要花费如此多的时间。如果花费的时间超出了我们的估算，则需要通过list命令找出函数体内最耗时的代码并进行进一步分析。如果我们发现这个函数所承担的职责过多，那么可以直接将这个函数拆分成多个拥有不同职责的更小的函数。

2. 如果一个处在非终端节点位置上的函数的累积取样计数和百分比都很大并且超出了我们的估算，那么我们应该首先查看其本地取样计数的大小。如果它的本地取样计数和百分比也很大，我们就需要通过list命令对这个函数体中的代码进行进一步分析。否则，我们就应该把关注点放在其下的分支节点所对应的函数上。如果当前节点下的所有直接分支节点的函数的累积取样计数都不大，但是直接分支节点的数量却非常多（十几甚至几十个），那么大致上可以断定当前节点的函数承担了过多的与流程控制相关的职责，我们需要对它进行拆分甚至重新设计。如果当前节点下的分支节点中包含累积取样计数和百分比很大的函数，那么我们就应该根据这个分支节点的类型（终端节点或非终端节点）来对其进行过程1或过程2的分析。

3. 单从调用计数的角度，我们并不能判断一个函数是否承担了过多的职责或者包含了过多的流程控制逻辑。但是，我们可以把调用计数作为定位问题的一种辅助手段。举个例子，如果根据过程1和过程2中的分析，我们怀疑在函数 B 及其调用的函数中可能存在性能问题，并且我们还发现函数 A 对函数 B 的调用计数也非常大，那么我们就应该想到函数 B 在取样中的频繁出现也许是由函数 A 对函数 B 的频繁调用引起的。在这种情况下，我们就应该先查看函数 A 中的代码，检查其中是否包含了过多的对函数 B 的不合理调用。如果存在不合理的调用，我们就应该对这部分代码进行重新设计。除此之外，我们还可以根据调用计数来判定一些小问题。比如，如果一个函数与调用它的所有函数都处于同一个代码包，那么我们就应该考虑把被调用的函数的访问权限设置为包内私有。如果对一个函数的调用都是来自于同一个函数，我们可以考虑在符合单一职

责原则的情况下把这两个函数合并。读者可能已经注意到，这与过程1中的一个建议是相互对立的。实际上，这也属于一种推迟优化策略。

在上述几个分析过程中的所有建议都不是绝对的。程序优化是一个复杂的过程，在很多时候都需要在多个指标或多个解决方案之间进行权衡和博弈。

在这几个分析过程的描述中，我们多次提到了list命令。现在我们就来对它进行说明。先来看一个示例：

```
(pprof) list ShowDepStruct
Total: 23 samples
ROUTINE ======================== main.ShowDepStruct in /home/hc/golang/goc2p
   /src/helper/pds/showpds.go
    0     20 Total samples (flat / cumulative)
    .      .  44:    }
    .      .  45:    fmt.Printf("The dependency structure of package '%s':\n",
   pkgImportPath)
    .      .  46:    ShowDepStruct(pn, "")
    .      .  47: }
    .      .  48:
---
    .      .  49: func ShowDepStruct(pnode *pkgtool.PkgNode, prefix string) {
    .      .  50:    var buf bytes.Buffer
    .      .  51:    buf.WriteString(prefix)
    .      .  52:    importPath := pnode.ImportPath()
    .      2  53:    buf.WriteString(importPath)
    .      1  54:    deps := pnode.Deps()
    .      .  55:    //fmt.Printf("P_NODE: '%s', DEP_LEN: %d\n", importPath,
   len(deps))
    .      .  56:    if len(deps) == 0 {
    .      5  57:        fmt.Printf("%s\n", buf.String())
    .      .  58:        return
    .      .  59:    }
    .      .  60:    buf.WriteString(ARROWS)
    .      .  61:    for _, v := range deps {
    .     12  62:        ShowDepStruct(v, buf.String())
    .      .  63:    }
    .      .  64: }
---
    .      .  65:
    .      .  66: func getPkgImportPath() string {
    .      .  67:    if len(pkgImportPathFlag) > 0 {
    .      .  68:        return pkgImportPathFlag
    .      .  69:    }
(pprof)
```

我们在pprof工具的交互界面中输入了命令 list ShowDepStruct 之后得到了很多输出信息。其中，ShowDepStruct为参数routine_regexp的值。输出信息的第一行告诉我们CPU概要文件中的取样一共有23个。这与我们之前讲解gv命令时看到的一样。输出信息的第二行显示，与我们提供的程序正则表达式（也就是参数routine_regexp）的值匹配的函数是 main.ShowDepStruct ，并且这个函数所在的源码文件的绝对路径是/home/hc/golang/goc2p/src/helper/pds/showpds.go。输出信息中的第三行告诉我们，在 main.ShowDepStruct 函数体中的代码的本地取样计数的总和是0，而累积取样计数的总和是20。在第三行最右边的括号中，flat代表本地取样计数，而cumulative代表累积取样计数。这是对该行最左边的那两个数字（也就是0和20）的含义的提示。从输出信息的第四行开始是对上述源码文件中的代码的截取，其中包含了 main.ShowDepStruct 函数的源码。list命令在这些代码的左边添加了对应的行号，这让我们查找代码更加容易。另外，在代码行号左边的对应位置上显示了每行代码的本地取样计数和累积取样计数。如果计数为0，则用英文句号“.”代替。这使得我们可以非常方便的找到存在计数值的代码行。

一般情况下，每行代码对应的本地取样计数和累积取样计数都应该与我们用gv命令生成的函数调用关系图中的计数相同。但是，如果一行代码中存在多个函数调用的话，那么在代码行号左边的计数值就会有偏差。比如，在上述示例中，第62行代码 ShowDepStruct(v, buf.String()) 的累积取样计数是12。但是从之前的函数调用关系图中我们得知，函数 main.ShowDepStruct 的累积取样计数是10。它们之间的偏差是2。实际上，在程序被执行的时候，第62行代码是由两个操作步骤组成的。第一个步骤是执行函数调用 buf.String() 并获得结果。第二个步骤是，调用函数 ShowDepStruct ，同时将变量 v``和执行第一个步骤所获得的结果作为参数传入。所以，这2个取样计数应该归咎于第62行代码中的函数调用子句 buf.String() 。也就是说，第62行代码的累积取样计数由两部分组成，即函数 main.ShowDepStruct 的累积取样计数和函数 bytes.(*Buffer).String 的累积取样计数。同理，示例中的第57行代码 fmt.Printf("%s\n", buf.String())```的累积取样计数也存在偏差。读者可以试着分析一下原因。

如果读者想验证上面所说的产生偏差的原因的话，可以将上面示例中的第62行代码和第57行代码分别拆成两行，然后再对命令源码文件showpds.go进行编译、运行（记得加入相关标记）并用pprof工具的list命令进行查看。不过，验证这个原因还有一个更简便的方式——使用pprof工具中的disasm命令。我们在下面的示例中执行disasm命令并后跟routine_regexp参数值ShowDepStruct。

```
bash
(pprof) disasm ShowDepStruct
Total: 23 samples
ROUTINE ====================== main.ShowDepStruct
    0     20 samples (flat, cumulative) 87.0% of total
---------------------- /home/hc/mybook/goc2p/src/helper/pds/showpds.go
    .     .    49: func ShowDepStruct(pnode *pkgtool.PkgNode, prefix string) {
<省略部分输出内容>
    .    10    62: ShowDepStruct(v, buf.String())
    .     .     80490ce: MOVL    main.&buf+3c(SP),AX
    .     .     80490d2: XORL    BX,BX
    .     .     80490d4: CMPL    BX,AX
```

```
    .     .    80490d6: JNE    main.ShowDepStruct+0x25f(SB)
    .     .    80490d8: LEAL    go.string.*+0x12d4(SB),BX
    .     .    80490de: MOVL    0(BX),CX
    .     .    80490e0: MOVL    4(BX),AX
    .     .    80490e3: MOVL    main.v+48(SP),BX
    .     .    80490e7: MOVL    BX,0(SP)
    .     .    80490ea: MOVL    CX,4(SP)
    .     .    80490ee: MOVL    AX,8(SP)
    .    10    80490f2: CALL    main.ShowDepStruct(SB)
    .     .    80490f7: MOVL    main.autotmp_0046+44(SP),DX
    .     .    80490fb: MOVL    main.autotmp_0048+70(SP),CX
    .     .   61: for _, v := range deps {
    .     .    80490ff: INCL    DX
    .     .    8049100: MOVL    main.autotmp_0047+2c(SP),BX
    .     .    8049104: CMPL    BX,DX
    .     .    8049106: JLT    main.ShowDepStruct+0x20b(SB)
    .     .   64: }
    .     .    8049108: ADDL    $80,SP
    .     .    804910e:    RET
    .     2   62: ShowDepStruct(v, buf.String())
    .     .    804910f: MOVL    8(AX),DI
    .     .    8049112: MOVL    4(AX),DX
    .     .    8049115: MOVL    c(AX),CX
    .     .    8049118: CMPL    CX,DX
    .     .    804911a: JCC    main.ShowDepStruct+0x273(SB)
    .     .    804911c: CALL    runtime.panicslice(SB)
    .     .    8049121:    UD2
    .     .    8049123: MOVL    DX,SI
    .     .    8049125: SUBL    CX,SI
    .     .    8049127: MOVL    DI,DX
    .     .    8049129: SUBL    CX,DX
    .     .    804912b: MOVL    0(AX),BP
    .     .    804912d: ADDL    CX,BP
    .     .    804912f: MOVL    BP,main.autotmp_0073+74(SP)
    .     .    8049133: MOVL    main.autotmp_0073+74(SP),BX
    .     .    8049137: MOVL    BX,0(SP)
    .     .    804913a: MOVL    SI,4(SP)
    .     .    804913e: MOVL    DX,8(SP)
    .     2    8049142: CALL    runtime.slicebytetostring(SB)
<省略部分输出内容>
(pprof)

(pprof)
```

由于篇幅原因，我们只显示了部分输出内容。disasm命令与list命令的输出内容有几分相似。实际上，disasm命令在输出函数 main.ShowDepStruct 的源码的同时还在每一行代码的下面列出了与这行代码对应的汇编指令。并且，命令还在每一行的最左边的对应位置上标注了该行汇编指令的本地取样计数和累积取样计数，同样以英文句号“.”代表计数为0的情况。另外，在汇编指令的左边且仅与汇编指令以一个冒号相隔的并不是像Go语言代码行中那样的行号，而是汇编指令对应的内存地址。

在上面这个示例中，我们只关注命令源码文件showpds.go中的第62行代码```ShowDepStruct(v, buf.String())``所对应的汇编指令。请读者着重查看在累积取样计数的列上有数字的行。像这样的行一共有四个。为了方便起见，我们把这四行摘抄如下：

```
.    10    62: ShowDepStruct(v, buf.String())
.    10     80490f2: CALL    main.ShowDepStruct(SB)
.     2    62: ShowDepStruct(v, buf.String())
.     2     8049142: CALL    runtime.slicebytetostring(SB)
```

其中的第一行和第三行说明了第62行代码的累积取样计数的组成，而第二行和第四行说明了存在这样的组成的原因。其中，汇编指令 CALL main.ShowDepStruct(SB) 的累积取样计数为10。也就是说，调用main.ShowDepStruct函数期间分析器进行了10次取样。而汇编指令 runtime.slicebytetostring(SB) 的累积取样计数为2，意味着在调用函数runtime.slicebytetostring期间分析器进行了2次取样。但是， runtime.slicebytetostring 函数又是做什么用的呢？实际上， runtime.slicebytetostring 函数正是被函数 bytes.(*Buffer).String 函数调用的。它实现的功能是把元素类型为byte的切片转换为字符串。综上所述，确实像我们之前说的那样，命令源码文件showpds.go中的第62行代码 ShowDepStruct(v, buf.String()) 的累积取样计数12由函数 main.ShowDepStruct 的累积取样计数10和函数 bytes.(*Buffer).String的 累积取样计数2组成。

至此，我们介绍了三个非常有用的命令，它们是gv命令、list命令和disasm命令。我们可以通过gv命令以图像化的方式查看程序中各个函数的本地取样计数、累积取样计数以及它们之间的调用关系和调用计数，并且可以很容易的通过节点面积的大小和有向线段的粗细找到计数值较大的节点。当我们依照之前所描述的分析过程找到可疑的高耗时的函数时，便可以使用list命令来查看函数内部各个代码行的本地取样计数和累积取样计数情况，并能够准确的找到使用了过多的CPU时间的代码。同时，我们还可以使用disasm命令来查看函数中每行代码所对应的汇编指令，并找到代码耗时的根源所在。因此，只要我们适时配合使用上述的这三条命令，就几乎可以在任何情况下理清程序性能问题的来龙去脉。可以说，它们是Go语言为我们提供的用于解决程序性能问题的瑞士军刀。

但是，有时候我们只是想了解哪些函数花费的CPU时间最多。在这种情况下，前面讲到的那几个命令所产生的数据就显得不那么直观了。不过不要担心，pprof工具为此提供了top命令。请看如下示例：

```
bash
(pprof) top
Total: 23 samples
      5  21.7%  21.7%        5  21.7% runtime.findfunc
      5  21.7%  43.5%        5  21.7% stkbucket
```

```
     3  13.0%  56.5%       3  13.0% os.(*File).write
     1   4.3%  60.9%       1   4.3% MHeap_AllocLocked
     1   4.3%  65.2%       1   4.3% getaddrbucket
     1   4.3%  69.6%       2   8.7% runtime.MHeap_Alloc
     1   4.3%  73.9%       1   4.3% runtime.SizeToClass
     1   4.3%  78.3%       1   4.3% runtime.aeshashbody
     1   4.3%  82.6%       1   4.3% runtime.atomicload64
     1   4.3%  87.0%       1   4.3% runtime.convT2E
(pprof)
```

在默认情况下，top命令会输出以本地取样计数为顺序的列表。我们可以把这个列表叫做本地取样计数排名列表。列表中的每一行都有六列。我们现在从左到右看，第一列和第二列的含义分别是：函数的本地取样计数和该本地取样计数在总取样计数中所占的比例。第四列和第五列的含义分别是：函数的累积取样计数和该累积取样计数在总取样计数中所占的比例。第五列的含义是左边几列数据所对应的函数的名称。读者应该对它们已经很熟悉了。这里需要重点说明的是第三列。第三列的含义是目前已打印出的函数的本地取样计数之和在总取样计数中所占的百分比。更具体的讲，第三行第三列上的百分比值就是列表前三行的三个本地取样计数的总和13除以总取样计数23而得出的。我们还可以通过将第二行上的百分比值43.5%与第三行第二列上的百分比值13.0%相加得到第三行第三列上的百分比值。第三列的百分比值可以使我们很直观的了解到最耗时的几个函数总共花费掉的CPU时间的比重。我们可以利用这一比重为性能优化任务制定更加多样化的目标。比如，我们的性能优化目标是把前四个函数的总耗时比重占比从60.9%降低到50%，等等。

从上面的示例我们可以看出，本地取样计数较大的函数都属于标准库的代码包或者Go语言内部。所以，我们无法或者不方便对这些函数进行优化。我们在之前提到过，在一般情况下，用户程序中的函数的本地取样计数都会非常低甚至是0。所以，如果我们编写的函数处在本地取样计数排名列表中的前几名的位置上话，就说明这个函数可能存在着性能问题。这时就需要我们通过list命令产生针对于这个函数的数据并仔细进行分析。举个例子，如果我们在函数中加入了一些并发控制代码（不论是同步并发控制还是异步的并发控制）使得这个函数本身的执行时间很长并在本地取样计数排名列表中处于前几名的位置，那么我们就应该仔细查看该函数中各行代码的取样计数以及它们的逻辑合理性。比如，用于同步并发控制的代码中是否存在产生死锁的可能性，或者用于异步并发控制的代码中是否存在协调失衡或者资源分配不均的地方。与编写合理和优秀的并发控制代码有关的内容在本书的第三部分。

在默认情况下，top命令输出的列表中只包含本地取样计数最大的前十个函数。如果我们想自定义这个列表的项数，那么需要在top命令后面紧跟一个项数值。比如：命令top5会输出行数为5的列表，命令top20会输出行数为20的列表，等等。

如果我们在top命令后加入标签 `--cum`，那么输出的列表就是以累积取样计数为顺序的。示例如下：

```
(pprof) top20 --cum
Total: 23 samples
     0   0.0%   0.0%      23 100.0% gosched0
     0   0.0%   0.0%      22  95.7% main.main
```

```
      0   0.0%   0.0%       22  95.7% runtime.main
      0   0.0%   0.0%       16  69.6% runtime.mallocgc
      0   0.0%   0.0%       12  52.2% pkgtool.(*PkgNode).Grow
      0   0.0%   0.0%       11  47.8% runtime.MProf_Malloc
      0   0.0%   0.0%       10  43.5% main.ShowDepStruct
      0   0.0%   0.0%       10  43.5% pkgtool.getImportsFromPackage
      0   0.0%   0.0%        8  34.8% cnew
      0   0.0%   0.0%        8  34.8% makeslice1
      0   0.0%   0.0%        8  34.8% runtime.cnewarray
      0   0.0%   0.0%        7  30.4% gostringsize
      0   0.0%   0.0%        7  30.4% runtime.slicebytetostring
      0   0.0%   0.0%        6  26.1% pkgtool.getImportsFromGoSource
      0   0.0%   0.0%        6  26.1% runtime.callers
      1   4.3%   4.3%        6  26.1% runtime.gentraceback
      0   0.0%   4.3%        6  26.1% runtime.makeslice
      5  21.7%  26.1%        5  21.7% runtime.findfunc
      5  21.7%  47.8%        5  21.7% stkbucket
      0   0.0%  47.8%        4  17.4% fmt.Fprintf
(pprof)
```

我们可以把这类列表叫做累积取样计数排名列表。在这个列表中，有命令源码文件showpds.go和代码包pkgtool中的函数上榜。它们都存在于项目goc2p中。在实际场景中，用户程序中的函数一般都处于函数调用关系图的上游。尤其是命令源码文件的入口函数 main.main 。所以，它们的累积取样计数一般都比较大，即使在累积取样计数排名列表中名列前茅也不足为奇。不过，如果一个函数的累积取样计数和百分比都很大，就应该引起我们的注意了。这在前面讲解gv命令的时候也有所提及。如果我们想在排名列表中过滤掉一些我们不关注的函数，还可以在命令的最后追加一个或多个我们想忽略的函数的名称或相应的正则表达式。像这样：

```
(pprof) top20 --cum -fmt\..* -os\..*
Ignoring samples in call stacks that match 'fmt\..*|os\..*'
Total: 23 samples
After ignoring 'fmt\..*|os\..*': 15 samples of 23 (65.2%)
      0   0.0%   0.0%       15  65.2% gosched0
      0   0.0%   0.0%       14  60.9% main.main
      0   0.0%   0.0%       14  60.9% runtime.main
      0   0.0%   0.0%       12  52.2% runtime.mallocgc
      0   0.0%   0.0%        8  34.8% pkgtool.(*PkgNode).Grow
      0   0.0%   0.0%        7  30.4% gostringsize
      0   0.0%   0.0%        7  30.4% pkgtool.getImportsFromPackage
      0   0.0%   0.0%        7  30.4% runtime.MProf_Malloc
      0   0.0%   0.0%        7  30.4% runtime.slicebytetostring
      0   0.0%   0.0%        6  26.1% main.ShowDepStruct
      0   0.0%   0.0%        6  26.1% pkgtool.getImportsFromGoSource
      0   0.0%   0.0%        5  21.7% cnew
      0   0.0%   0.0%        5  21.7% makeslice1
```

```
     0   0.0%   0.0%        5  21.7% runtime.cnewarray
     0   0.0%   0.0%        4  17.4% runtime.callers
     1   4.3%   4.3%        4  17.4% runtime.gentraceback
     0   0.0%   4.3%        3  13.0% MCentral_Grow
     0   0.0%   4.3%        3  13.0% runtime.MCache_Alloc
     0   0.0%   4.3%        3  13.0% runtime.MCentral_AllocList
     3  13.0%  17.4%        3  13.0% runtime.findfunc
(pprof)
```

在上面的示例中，我们通过命令top20获取累积取样计数最大的20个函数的信息，同时过滤掉了来自代码包 `fmt` 和 `os` 中的函数。

我们要详细讲解的最后一个命令是callgrind。pprof工具可以将概要转化为强大的Valgrind工具集中的组件Callgrind支持的格式。Valgrind是可运行在Linux操作系统上的用来成分析程序性能及程序中的内存泄露错误的强力工具。而作为其中组件之一的Callgrind的功能是收集程序运行时的一些数据、函数调用关系等信息。由此可知，Callgrind工具的功能基本上与我们之前使用标准库代码包runtime的API对程序运行情况进行取样的操作是一致的。

我们可以通过callgrind命令将概要文件的内容转化为Callgrind工具可识别的格式并保存到文件中。示例如下：

```
(pprof) callgrind cpu.callgrind
Writing callgrind file to 'cpu.callgrind'.
(pprof)
```

文件cpu.callgrind是一个普通文本文件，所以我们可以使用任何文本查看器来查看其中的内容。但更方便的是，我们可以使用callgrind命令直接查看到图形化的数据。现在我们来尝试一下：

```
(pprof) callgrind
Writing callgrind file to '/tmp/pprof2641.0.callgrind'.
Starting 'kcachegrind /tmp/pprof2641.0.callgrind & '
(pprof) sh: 1: kcachegrind: not found
```

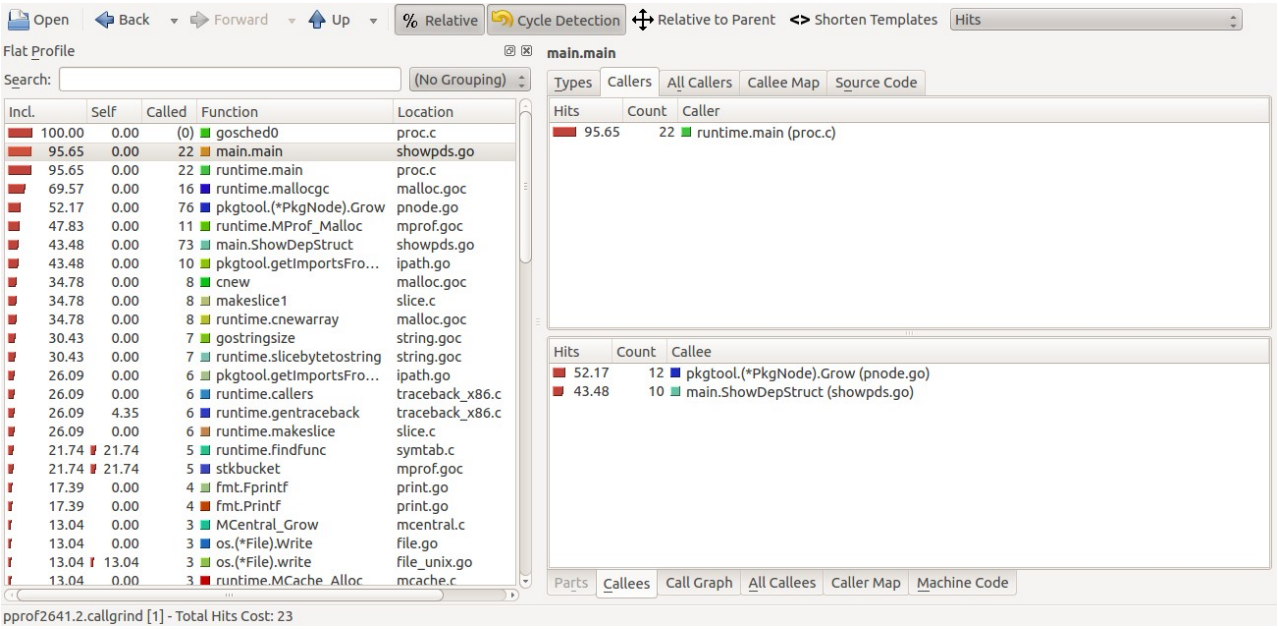
我们没有在callgrind命令后添加任何作为参数的统计文件路径。所以callgrind命令会自行使用kcachegrind工具以可视化的方式显示统计数据。然而，我们的系统中还没有安装kcachegrind工具。

在Debian的Linux发行版及其衍生版下，我们可以直接使用命令 `sudo apt-get install kcachegrind` 来安装kcachegrind工具。或者我们可以从其官方网站下载安装包来进行安装。

安装好kcachegrind工具之后，我们再来执行callgrind命令：

```
bash
(pprof) callgrind
Writing callgrind file to '/tmp/pprof2641.1.callgrind'.
Starting 'kcachegrind /tmp/pprof2641.1.callgrind & '
(pprof)
```

从命令输出的提示信息可以看出，实际上callgrind命令把统计文件保存到了Linux的临时文件夹/tmp中。然后使用kcachegrind工具进行查看。下图为在pprof工具交互模式下执行callgrind命令后弹出的kcachegrind工具界面。



图片 1.4 使用kcachegrind工具查看概要数据

图0-4 使用kcachegrind工具查看概要数据

从上图中我们可以看到，kcachegrind工具对数据的展现非常直观。总体上来说，界面被分为了左右两栏。在左栏中的是概要文件中记录的函数的信息列表。列表一共有五列，从左到右的含义 分别是函数的累积取样计数在总取样计数中的百分比、函数的本地取样计数在总取样计数中的百分比、函数被调用的总次数（包括递归调用）、函数的名称以及函数所在的源码文件的名称。而在界面的右栏，我们查看在左栏中选中的行的详细信息。kcachegrind工具的功能非常强大。不过由于对它的介绍超出了本书的范围，所以我们就此暂告一个段落。

我们刚刚提到过，不加任何参数callgrind命令的执行分为两个步骤——生成统计文件和使用kcachegrind工具查看文件内容。还记得我们在之前已经生成了一个名为统计文件cpu.callgrind吗？其实，我们可以使用命令 `kcachegrind cpu.callgrind` 直接对它进行查看。执行这个命令后所弹出的kcachegrind工具界面与我们之前看到的完全一致。

到现在为止，我们又介绍了两个可以更直观的统计和查看概要文件中数据的命令。top命令让我们可以在命令行终端中查看这些统计信息。而callgrind命令使我们通过kcachegrind工具查看概要文件的数据成为了可能。这两个命令都让我们可以宏观的、从不同维度的来查看和分析概要文件。它们都是非常给力的统计辅助工具。

除了上面详细讲述的那些命令之外，pprof工具在交互模式下还支持少许其它的命令。这在表5-23中也有所体现。这些命令有的只是主要命令的另一种形式（比如web命令和weblist命令），而有的只是为了提供辅助功能（比如help命令和quit命令）。

在本小节中，我们只使用 `go tool pprof` 命令对CPU概要文件进行了查看和分析。读者可以试着对内存概要文件和程序阻塞概要文件进行分析。

相对于普通的编程方式来讲，并发编程都是复杂的。所以，我们就更需要像pprof这样的工具为我们保驾护航。大家可以将本小节当作一篇该工具的文档，并在需要时随时查看。

go tool cgo

cgo也是一个Go语言自带的特殊工具。一般情况下，我们使用命令 go tool cgo 来运行它。这个工具可以使我们创建能够调用C语言代码的Go语言源码文件。这使得我们可以使用Go语言代码去封装一些C语言的代码库，并提供给Go语言代码或项目使用。

在执行 go tool cgo 命令的时候，我们需要加入作为目标的Go语言源码文件的路径。这个路径可以是绝对路径也可以是相对路径。但是，作者强烈建议在目标源码文件所属的代码包目录下执行 go tool cgo 命令并以目标源码文件的名字作为参数。因为， go tool cgo 命令会在当前目录（也就是我们执行 go tool cgo 命令的目录）中生成一个名为_obj的子目录。该目录下会包含一些Go源码文件和C源码文件。这个子目录及其包含的文件理应被保存在目标代码包目录之下。至于原因，我们稍后再做解释。

我们现在来看可以作为 go tool cgo 命令参数的Go语言源码文件。这个源码文件必须要包含一行只针对于代码包 C 的导入语句。其实，Go语言标准库中并不存在代码包 C 。代码包 C 是一个伪造的代码包。导入这个代码包是为了告诉cgo工具在这个源码文件中需要调用C代码，同时也是给予cgo所产生的代码一个专属的命名空间。除此之外，我们还需要在这个代码包导入语句之前加入一些注释，并且在注释行中写出我们真正需要使用的C语言接口文件的名称。像这样：

```
// #include <stdlib.h>
import "C"
```

在Go语言的规范中，把在代码包 C 导入语句之前的若干注释行叫做序文（preamble）。 在引入了C语言的标准代码库stdlib.h之后，我们就可以在后面的源码中调用这个库中的接口了。像这样：

```
func Random() int {
    return int(C.rand())
}

func Seed(i int) {
    C.srand(C.uint(i))
}
```

我们把上述的这些Go语言代码写入Go语言的库源码文件rand.go中，并将这个源码文件保存在goc2项目的代码包 basic/cgo/lib 的对应目录中。

在Go语言源码文件rand.go中对代码包 C 有四处引用，分别是三个函数调用语句 C.rand 、 C.srand 和 C.uint ，以及一个导入语句 import "C" 。其中，在Go语言函数 Random 中调用了C语言标准库代码中的函数 rand 并返回了它的结果。但是，C语言的 rand 函数返回的结果的类型是C语言中的int类型。在cgo工具的环境中，C语言中的int类型与C.int相对应。作为一个包装C语言接口的函数，我们必须将代码包 C 的使用限制在当前代码包

内。也就是说，我们必须对当前代码包之外的Go代码隐藏代码包 C 。这样做也是为了遵循代码隔离原则。我们在设计接口或者接口适配程序的时候经常会用到这种方法。因此， rand 函数的结果的类型必须是Go语言的。所以，我们在这里使用函数int对C.int类型的C语言接口的结果进行了转换。当然，为了更清楚的表达，我们也可以将函数 Random 中的代码 return int(C.rand()) 拆分成两行，像这样：

```
var r C.int = C.rand()
return int(r)
```

而Go语言函数 Seed 则恰好相反。C语言标准代码库中的函数 srand 接收一个参数，且这个参数的类型必须为C语言的uint类型，即 C.uint 。而Go语言函数Seed的参数为Go语言的int类型。为此，我们需要使用代码包 C 的函数 unit 对其进行转换。

实际上，标准C语言的数字类型都在cgo工具中有对应的名称，包括：C.char、C.schar（有符号字符类型）、C.uchar（无符号字符类型）、C.short、C.ushort（无符号短整数类型）、C.int、C.uint（无符号整数类型）、C.long、C.ulong（无符号长整数类型）、C.longlong（对应于C语言的类型long long，它是在C语言的C99标准中定义的新整数类型）、C.ulonglong（无符号的long long类型）、C.float和C.double。另外，C语言类型void*对应于Go语言的类型 unsafe.Pointer 。

如果想直接访问C语言中的struct、union或enum类型的话，就需要在名称前分别加入前缀struct_、union_或enum_。比如，我们需要在Go源码文件中访问C语言代码中的名为command的struct类型的话，就需要这样写：C.struct_command。那么，如果我们想在Go语言代码中访问C语言类型struct中的字段需要怎样做呢？解决方案是，同样以C语言类型struct的实例名以及选择符“.”作为前导，但需要在字段的名称前加入下划线“_”。例如，如果command1是名为command的C语言struct类型的实例名，并且这个类型中有一个名为name的字段，那么我们在Go语言代码中访问这个字段的方式就是command1._name。需要注意的是，我们不能在Go的struct类型中嵌入C语言类型的字段。这与我们在前面所说的代码隔离原则具有相同的意义。

在上面展示的库源码文件rand.go中有多处对C语言函数的访问。实际上，任何C语言的函数都可以 被Go语言代码调用。只要在源码文件中导入了代码包 C 。并且，我们还可以同时取回C语言函数的结果，以及一个作为错误提示信息的变量。这与我们在Go语言中同时获取多个函数结果的方法一样。同样的，我们可以使用下划线“_”直接丢弃取回的值。这在调用无结果的C语言函数时非常有用。请看下面的例子：

```
package cgo

/*
#cgo LDFLAGS: -lm
#include <math.h>
*/
import "C"

func Sqrt(p float32) (float32, error) {
```

```
    n, err := C.sqrt(C.double(p))
    return float32(n), err
}
```

上面这段代码被保存在了Go语言库源码文件math.go中，并与源码文件rand.go在同一个代码包目录。在Go语言函数 Sqrt 中的 C.sqrt 是一个在C语言标准代码库math.h中的函数。它会返回参数的平方根。但是在第一行代码中，我们接收由函数C.sqrt返回的两个值。其中，第一个值即为C语言函数 sqrt 的结果。而第二个值就是我们上面所说的那个作为错误提示信息的变量。实际上，这个变量的类型是Go语言的 error 接口类型。它包装了一个C语言的全局变量errno。这个全局变量被定义在了C语言代码库errno.h中。cgo工具在为我们生成C语言源码文件时会默认引入两个C语言标准代码库，其中一个就是errno.h。所以我们并不用在Go语言源码文件中使用指令符#include显式的引入这个代码库。cgo工具默认为我们引入的另一个是C语言标准代码库string.h。它包含了很多用于字符串处理和内存处理的函数。

在我们以“C.*”的形式调用C语言代码库时，有一点需要特别注意。在C语言中，如果一个函数的参数是一个具有固定尺寸的数组，那么实际上这个函数所需要的是指向这个数组的第一个元素的指针。C编译器能够正确识别和处理这个调用惯例。它可以自行获取到这个指针并传给函数。但是，这在我们使用cgo工具调用C语言代码库时是行不通的。在Go语言中，我们必须显式的将这个指向数组的第一个元素的指针传递给C语言的函数，像这样：``C.func1(&x[0])````。

另一个需要特别注意的地方是，在C语言中没有像Go语言中独立的字符串类型。C语言使用最后一个元素为‘\0’的字符数组来代表字符串。在Go语言的字符串和C语言的字符串之间进行转换的时候，我们就需要用到代码包 C 中的 C.C.CString 、 C.GoString 和 C.GoStringN 等函数。这些转换操作是通过对字符串数据的拷贝来完成的。Go语言内存管理器并不能感知此类内存分配操作。因为它们是由C语言代码引发的。所以，我们在使用与 C.CString 函数类似的会导致内存分配操作的函数之后，需要调用代码包C的free函数以手动的释放内存。这里有一个小技巧，即我们可以把对 C.free 函数的调用放到 defer 语句中或者放入在defer之后的匿名函数中。这样就可以保证在退出当前函数之前释放这些被分配的内存了。请看下面这个示例：

```
func Print(s string) {
        cs := C.CString(s)
        defer C.free(unsafe.Pointer(cs))
        C.myprint(cs)
}
```

上面这段代码被存放在goc2p项目的代码包 basic/cgo/lib 的库源码文件print.go中。其中的函数 C.myprint 是我们在该库源码文件的序文中自定义的。关于这种C语言函数定义方式，我们一会儿再解释。在这段代码中，我们首先把Go语言的字符串转换为了C语言的字符串。注意，变量 cs 的值实际上是指向字符串（在C语言中，字符串由字符数组代表）中第一个字符的指针。在cgo工具对应的上下文环境中， cs 变量的类型是 *C.Char 。然后，我们通过 defer 语句和 C.free 函数保证由C语言代码分配的内存得以释放。请注意子语句 unsafe.Pointer(cs) 。正因为 cs 变量在C语言中的类型是指针类型，且与之相对应的Go语言类型是 unsafe.Pointer 。所以，我

们需要先将其转换为Go语言可以识别的类型再作为参数传递给函数 C.free 。最后，我们将这个字符串打印到标准输出。

再次重申，我们在使用 C.CString 函数将Go语言的字符串转换为C语言字符串后，需要显式的调用 C.free 函数以释放用于数据拷贝的内存。而最佳实践是，将在 defer 语句中调用 C.free 函数。

在前面我们已经提到过，在导入代码包C的语句之上可以加入若干个为cgo工具而写的若干注释行（也被叫做序文）。并且，以#include和一个空格开始的注释行可以用来引入一个C语言的接口文件。我们也把序文中这种形式的字符串叫做指令符。指令符 #cgo 的用途是为编译器和连接器提供标记。这些标记在编译当前源码文件中涉及到代码包 C 的那部分代码时会被用到。

标记 CFLAGS 和 LDFLAGS``可以被放在指令符 #cgo```之后，并用于定制编译器gcc的行为。gcc（GNU Compiler Collection，GNU编译器套装），是一套由GNU开发的开源的编程语言编译器。它是GNU项目的关键部分，也是类Unix操作系统（也包括Linux操作系统）中的标准编译器。gcc（特别是其中的C语言编译器）也常被认为是跨平台编译器的事实标准。gcc原名为GNU C语言编译器（GNU C Compiler），因为它原本只能处理C语言。不过，gcc变得可以处理更多的语言。现在，gcc中包含了很多针对特定编程语言的编译器。我们在本节第一小节的末尾提及的gccgo就是这款套件中针对Go语言的编译器。标记CFLAGS可以指定用于gcc中的C编译器的选项。它尝尝用于指定头文件（.h文件）的路径。而标记LDFLAGS则可以指定gcc编译器会用到的一些优化参数，也可以用来告诉链接器需要用到的C语言代码库文件的位置。

为了清晰起见，我们可以把这些标记及其值拆分成多个注释行，并均以指令符 #cgo 作为前缀。另外，在指令符 #cgo 和标记之间，我们也可以加入一些可选的内容，即环境变量GOOS和GOARCH中的有效值。这样，我们就可以使这些标记只在某些操作系统和/或某些计算架构的环境下起作用了。示例如下：

```
// #cgo CFLAGS: -DPNG_DEBUG=1
// #cgo linux CFLAGS: -DLINUX=1
// #cgo LDFLAGS: -lpng
// #include <png.h>
import "C"
```

在上面的示例中，序文由四个注释行组成。第一行注释的含义是预定义一个名为 PNG_DEBUG 的宏并将它的值设置为1。而第二行注释的意思是，如果在Linux操作系统下，则预定义一个名为 LINUX 的宏并将它的值设置为1。第三行注释是与链接器有关的。它告诉链接器需要用到一个库名为png的代码库文件。最后，第四行注释引入了C语言的标准代码库png.h。

如果我们有一些在所有场景下都会用到的 CFLAGS 标记或 LDFLAGS 标记的值，那么就可以把它们分别作为环境变量 CGO_CFLAGS 和 CGO_LDFLAGS 的值。而对于需要针对某个导入了“C”的代码包的标记值就只能连同指令符 #cgo 一起放入Go语言源码文件的注释行中了。

相信读者对指令符 #cgo 和 #include 的用法已经有所了解了。

实际上，我们几乎可以在序文中加入任何C代码。像这样：

```
/*
#cgo LDFLAGS: -lsqlite3

#include <sqlite3.h>
#include <stdlib.h>

// These wrappers are necessary because SQLITE_TRANSIENT
// is a pointer constant, and cgo doesn't translate them correctly.
// The definition in sqlite3.h is:
//
// typedef void (*sqlite3_destructor_type)(void*);
// #define SQLITE_STATIC      ((sqlite3_destructor_type)0)
// #define SQLITE_TRANSIENT   ((sqlite3_destructor_type)-1)

static int my_bind_text(sqlite3_stmt *stmt, int n, char *p, int np) {
	return sqlite3_bind_text(stmt, n, p, np, SQLITE_TRANSIENT);
}
static int my_bind_blob(sqlite3_stmt *stmt, int n, void *p, int np) {
	return sqlite3_bind_blob(stmt, n, p, np, SQLITE_TRANSIENT);
}

*/
```

上面这段代码摘自开源项目gosqlite的Go语言源码文件sqlite.go。gosqlite项目是一个开源数据SQLite的Go语言版本的驱动代码库。实际上，它只是把C语言版本的驱动代码库进行了简单的封装。在Go语言的世界里，这样的封装随处可见，尤其是在Go语言发展早期。因为，这样可以非常方便的重用C语言版本的客户端程序，而大多数软件都早已拥有这类程序了。并且，封装C语言版本的代码库与从头开发一个Go语言版本的客户端程序相比，无论从开发效率还是运行效率上来讲都会是非常迅速的。现在让我们看看在上面的序文中都有些什么。很显然，在上面的序文中直接出现了两个C语言的函数 `my_bind_text` 和 `my_bind_blob` 。至于为什么要把C语言函数直接写到这里，在它们前面的注释中已经予以说明。大意翻译如下：这些包装函数是必要的，这是因为SQLITE_TRANSIENT是一个指针常量，而cgo并不能正确的翻译它们。看得出来，这是一种备选方案，只有在cgo不能帮我们完成工作时才会被选用。不管怎样，在序文中定义的这两个函数可以直接在当前的Go语言源码文件中被使用。具体的使用方式同样是通过“C.*”的形式来调用。比如源码文件sqlite.go中的代码：

```
rv := C.my_bind_text(s.stmt, C.int(i+1), cstr, C.int(len(str)))
```

和

```
rv := C.my_bind_blob(s.stmt, C.int(i+1), unsafe.Pointer(p), C.int(len(v)))
```

上述示例中涉及到的源码文件可以通过这个网址访问到。有兴趣的读者可以前往查看。

我们再来看看我们之前提到过的库源码文件print.go（位于goc2p项目的代码包 basic/cgo/lib 之中）的序文：

```
/*
#include <stdio.h>
#include <stdlib.h>

void myprint(char* s) {
	printf("%s", s);
}
*/
import "C"
```

我们在序文中定义一个名为 myprint 的函数。在这个函数中调用了C语言的函数 printf 。自定义函数 myprint 充当了类似于适配器的角色。之后，我们就可以在后续的代码中直接使用这个自定义的函数了：

```
C.myprint(cs)
```

关于在序文中嵌入C语言代码的方法我们就介绍到这里。

现在，让我们来使用 go tool cgo 命令并以rand.go作为参数生成_obj子目录和相关源码文件：

```
hc@ubt:~/golang/goc2p/src/basic/cgo/lib$ go tool cgo rand.go
hc@ubt:~/golang/goc2p/src/basic/cgo/lib$ ls
_obj  rand.go
hc@ubt:~/golang/goc2p/src/basic/cgo/lib$ ls _obj
_cgo_defun.c   _cgo_export.h  _cgo_gotypes.go  _cgo_.o      rand.cgo2.c
_cgo_export.c  _cgo_flags     _cgo_main.c     rand.cgo1.go
```

子目录_obj中一共包含了九个文件。

其中，cgo工具会把作为参数的Go语言源码文件rand.go转换为四个文件。其中包括两个Go语言源码文件rand.cgo1.go和_cgo_gotypes.go，以及两个C语言源码文件_cgo_defun.c和rand.cgo2.c。

文件rand.cgo1.go用于存放cgo工具对原始源码文件rand.go改写后的内容。改写具体细节包括去掉其中的代码包C导入语句，以及替换涉及到代码包C的语句，等等。最后，这些替换后的标识符所对应的Go语言的函数、类型或变量的定义，将会被写入到文件_cgo_gotypes.go中。

需要说明的是，替换涉及到代码包C的语句的具体做法是根据xxx的种类将标识符 C.xxx 替换为 _Cfunc_xxx 或者 _Ctype_xxx 。比如，作为参数的源码文件rand.go中存在如下语句：

```
C.srand(C.uint(i))
```

cgo工具会把它改写为：

```
_Cfunc_srand(_Ctype_uint(i))
```

其中，标识符 C.srand 被替换为 _Cfunc_srand ，而标识符 C.uint 被替换为了 _Ctype_uint 。并且，新的标识符 _Cfunc_srand 和 _Ctype_uint 的定义将会在文件_cgo_gotypes.go中被写明：

```
type _Ctype_uint uint32

type _Ctype_void [0]byte

func _Cfunc_srand(_Ctype_uint) _Ctype_void
```

其中，类型_Ctype_void可以表示空的参数列表或空的结果列表。

文件_cgo_defun.c中包含了相应的C语言函数的定义和实现。例如，C语言函数 _Cfunc_srand 的实现如下：

```
#pragma cgo_import_static _cgo_54716c7dc6a7_Cfunc_srand
void _cgo_54716c7dc6a7_Cfunc_srand(void*);

void
·_Cfunc_srand(struct{uint8 x[4];}p)
{
  runtime·cgocall(_cgo_54716c7dc6a7_Cfunc_srand, &p);
}
```

其中，十六进制数“54716c7dc6a7”是cgo工具由于作为参数的源码文件的内容计算得出的哈希值。这个十六进制数作为了函数名称 _cgo_54716c7dc6a7_Cfunc_srand 的一部分。这样做是为了生成一个唯一的名称以避免冲突。我们看到，在源码文件_cgo_defun.c中只包含了函数 _cgo_54716c7dc6a7_Cfunc_srand 的定义。而其实现被写入到了另一个C语言源码文件中。这个文件就是rand.cgo2.c。函数 _cgo_54716c7dc6a7_Cfunc_srand 对应的实现代码如下：

```
void
_cgo_f290d3e89fd1_Cfunc_srand(void *v)
{
  struct {
    unsigned int p0;
  } __attribute__((__packed__)) *a = v;
  srand(a->p0);
}
```

这个函数从指向函数 _Cfunc_puts 的参数帧中抽取参数，并调用系统C语言函数srand，最后将结果存储在帧中并返回。

下面我们对在子目录_obj中存放的其余几个文件进行简要说明：

- 文件_cgo_flags用于存放CFLAGS标记和LDFLAGS标记的值。

- 文件_cgo_main.c用于存放一些C语言函数的存根，也可以说是一些函数的空实现。函数的空实现即在函数体重没有任何代码（return语句除外）的实现。其中包括在源码文件_cgo_export.c出现的声明为外部函数的函数。另外，文件_cgo_main.c中还会有一个被用于动态链接处理的main函数。
- 在文件_cgo_export.h中存放了可以暴露给C语言代码的与Go语言类型相对应的C语言声明语句。
- 文件_cgo_export.c中则包含了与可以暴露给C语言代码的Go语言函数相对应的C语言函数定义和实现代码。
- 文件*cgo*.o是gcc编译器在编译C语言源码文件rand.cgo2.c、_cgo_export.c和_cgo_main.c之后生成的结果文件。

在上述的源码文件中，文件rand.cgo1.go和_cgo_gotypes.go将会在构建代码包时被Go官方Go语言编译器（6g、8g或5g）编译。文件_cgo_defun.c会在构建代码包时被Go官方的C语言的编译器（6c、8c或5c）编译。而文件rand.cgo2.c、_cgo_export.c和_cgo_main.c 则会被gcc编译器编译。

如果我们在执行 `go tool cgo` 命令时加入多个Go语言源码文件作为参数，那么在当前目录的_obj子目录下会出现与上述参数数量相同的x.cgo1.go文件和x.cgo2.c文件。其中，x为作为参数的Go语言源码文件主文件名。

通过上面的描述，我们基本了解了由cgo工具生成的文件的内容和用途。

与其它go命令一样，我们在执行 `go tool cgo` 命令的时候还可以加入一些标记。如下表。

表0-24 `go tool cgo` 命令可接受的标记

名称	默认值	说明
-cdefs	false	将改写后的源码内容以C定义模式打印到标准输出，而不生成相关的源码文件。
-godefs	false	将改写后的源码内容以Go定义模式打印到标准输出，而不生成相关的源码文件。
-objdir	""	gcc编译的目标文件所在的路径。若未自定义则为当前目录下的_obj子目录。
-dynimport	""	如果值不为空字符串，则打印为其值所代表的文件生成的动态导入数据到标准输出。
-dynlinker	false	记录在dynimport模式下的动态链接器信息。
-dynout	""	将-dynimport的输出（如果有的话）写入到其值所代表的文件中。
-gccgo	false	生成可供gccgo编译器使用的文件。
-gccgopkgpath	""	对应于gccgo编译器的-fgo-pkgpath选项。
-gccgoprefix	""	对应于gccgo编译器的-fgo-prefix选项。
-debug-define	false	打印相关的指令符#defines及其后续内容到标准输出。
-debug-gcc	false	打印gcc调用信息到标准输出。

-import_runtime_cgo	true	在生成的代码中加入语句“import runtime/cgo”。
-import_syscall	true	在生成的代码中加入语句“import syscall”。

在上表中，我们把标记分为了五类并在它们之间以空行分隔。

在第一类标记中，`-cdefs` 标记和 `-godefs` 标记都可以打印相应的代码到标准输出，并且使cgo工具不生成相应的源码文件。cgo工具在获取目标源码文件内容之后会改写其中的内容，包括去掉代码包 `C` 的导入语句，以及对代码包 `C` 的调用语句中属于代码包 `C` 的类型、函数和变量进行等价替换。如果我们加入了标记 `-cdefs` 或 `-godefs` ，那么cgo工具随后就会把改写后的目标源码打印到标准输出了。需要注意的是，我们不能同时使用这两个标记。使用这两个标记打印出来的源码内容几乎相同，而最大的区别也只是格式方面的。

第二类的三个标记都与动态链接库有关。在类Unix系统下，标记 `-dynimport` 的值可以是一个ELF（Executable and Linkable Format）格式或者Mach-O（Mach Object）格式的文件的路径。ELF即可执行链接文件格式。ELF格式的文件保存了足够的系统相关信息，以至于使它能够支持不同平台上的交叉编译和交叉链接，可移植性很强。同时，它在执行中支持动态链接共享库。我们在Linux操作系统下使用go命令生成的命令源码文件的可执行文件就是ELF格式的。而Mach-O是一种用于可执行文件、目标代码、动态链接库和内核转储的文件格式。在Windows下，这个标记的值应该是一个PE（Portable Execute）格式的文件的路径。在Windows操作系统下，使用go命令生成的命令源码文件的可执行文件就是PE格式的。

实质上，加入标记 `-dynimport` 的 `go tool cgo` 命令相当于一个被构建在cgo工具内部的独立的帮助命令。使用方法如 `go tool cgo -dynimport='cgo_demo.go'` 。这个命令会扫描这个标记的值所代表的可执行文件，并将其中记录的与已导入符号和已导入代码库相关的信息打印到标准输出。 `go build` 命令程序中有专门为cgo工具制定的规则。这使得它可以在编译直接或间接依赖了代码包C的命令源码文件时可以生成适当的可执行文件。在这个可执行文件中，直接包含了相关的已导入符号和已导入代码库的信息，以供之后使用。这样就无需使链接器复制gcc编译器的所有关于如何寻找已导入的符号以及使用它的位置的专业知识了。下面我们来试用一下 `go tool cgo -dynimport` 命令。

首先，我们创建一个命令源码文件cgo_demo.go，并把它存放在goc2p项目的代码包 `basic/cgo` 对应的目录下。命令源码文件cgo_demo.go的内容如下：

```
package main

import (
    cgolib "basic/cgo/lib"
    "fmt"
)

func main() {
    input := float32(2.33)
    output, err := cgolib.Sqrt(input)
```

```
    if err != nil {
        fmt.Errorf("Error: %s\n", err)
    }
    fmt.Printf("The square root of %f is %f.\n", input, output)
}
```

在这个命令源码文件中，我们调用了goc2p项目的代码包 basic/cgo/lib 中的函数 Sqrt 。这个函数是被保存在库源码文件math.go中的。而在文件math.go中，我们导入了代码包C。也就是说，命令源码文件cgo_demo.go间接的依赖了代码包C。现在，我们使用 go build 命令将这个命令源码文件编译成ELF格式的可执行文件。然后，我们就能够使用 go tool cgo -dynimport 命令查看其中的导入信息了。请看如下示例：

```
hc@ubt:~/golang/goc2p/basic/cgo$ go build cgo_demo.go
hc@ubt:~/golang/goc2p/basic/cgo$ go tool cgo -dynimport='cgo_demo'
#pragma cgo_import_dynamic pthread_attr_init pthread_attr_init#GLIBC_2.1
    "libpthread.so.0"
#pragma cgo_import_dynamic pthread_attr_destroy pthread_attr_destroy#GLIBC_2.0
    "libpthread.so.0"
#pragma cgo_import_dynamic stderr stderr#GLIBC_2.0 "libc.so.6"
#pragma cgo_import_dynamic sigprocmask sigprocmask#GLIBC_2.0 "libc.so.6"
#pragma cgo_import_dynamic free free#GLIBC_2.0 "libc.so.6"
#pragma cgo_import_dynamic fwrite fwrite#GLIBC_2.0 "libc.so.6"
#pragma cgo_import_dynamic malloc malloc#GLIBC_2.0 "libc.so.6"
#pragma cgo_import_dynamic strerror strerror#GLIBC_2.0 "libc.so.6"
#pragma cgo_import_dynamic srand srand#GLIBC_2.0 "libc.so.6"
#pragma cgo_import_dynamic setenv setenv#GLIBC_2.0 "libc.so.6"
#pragma cgo_import_dynamic __libc_start_main __libc_start_main#GLIBC_2.0
    "libc.so.6"
#pragma cgo_import_dynamic fprintf fprintf#GLIBC_2.0 "libc.so.6"
#pragma cgo_import_dynamic pthread_attr_getstacksize
     pthread_attr_getstacksize#GLIBC_2.1 "libpthread.so.0"
#pragma cgo_import_dynamic sigfillset sigfillset#GLIBC_2.0 "libc.so.6"
#pragma cgo_import_dynamic __errno_location __errno_location#GLIBC_2.0
    "libpthread.so.0"
#pragma cgo_import_dynamic sqrt sqrt#GLIBC_2.0 "libm.so.6"
#pragma cgo_import_dynamic rand rand#GLIBC_2.0 "libc.so.6"
#pragma cgo_import_dynamic pthread_create pthread_create#GLIBC_2.1
    "libpthread.so.0"
#pragma cgo_import_dynamic abort abort#GLIBC_2.0 "libc.so.6"
#pragma cgo_import_dynamic _ _ "libm.so.6"
#pragma cgo_import_dynamic _ _ "libpthread.so.0"
#pragma cgo_import_dynamic _ _ "libc.so.6"
```

从上面示例的输出信息中，我们可以看到可执行文件cgo_demo所涉及到的所有动态链接库文件以及相关的函数名和代码库版本等信息。

如果我们再加入一个标记 -dynlinker ，那么在命令的输出信息还会包含动态链接器的信息。示例如下：

```
hc@ubt:~/golang/goc2p/src/basic/cgo$ go tool cgo -dynimport='cgo_demo' -dynlinker
#pragma cgo_dynamic_linker "/lib/ld-linux.so.2"
<省略部分输出内容>
```

如果我们在命令 go tool cgo -dynimport 后加入标记 -dynout ，那么命令的输出信息将会写入到指定的文件中，而不是被打印到标准输出。比如命令 go tool cgo -dynimport='cgo_demo' -dynlinker -dynout='cgo_demo.di' 就会将可执行文件cgo_demo中的导入信息以及动态链接器信息转储到当前目录下的名为“cgo_demo.di”的文件中。

第四类标记包含了 -gccgo 、 -gccgopkgpath 和 -gccgoprefix 。它们都与编译器gccgo有关。标记 -gccgo 的作用是使cgo工具生成可供gccgo编译器使用的源码文件。这些源码文件会与默认情况下生成的源码文件在内容上有一些不同。实际上，到目前为止，cgo工具还不能很好的与gccgo编译器一同使用。但是，按照gccgo编译器的主要开发者Ian Lance Taylor的话来说，gccgo编译器并不需要cgo工具，也不应该使用gcc工具。不管怎样，这种情况将会在Go语言的1.3版本中得到改善。

第五类标记用于打印调试信息，包括标记 -debug-define 和 -debug-gcc 。gcc工具不但会生成新的Go语言源码文件以保存其对目标源码改写后的内容，还会生成若干个C语言源码文件。cgo工具为了编译这些C语言源码文件，就会用到gcc编译器。在加入 -debug-gcc 标记之后，gcc编译器的输出信息就会被打印到标准输出上。另外，gcc编译器在对C语言源码文件进行编译之后会产生一个结果文件。这个结果文件就是在*obj子目录下的名为_cgo*.o的文件。

第六类标记的默认值都为true。也就是说，在默认情况下cgo工具生成的_obj子目录下的Go语言源码文件_cgo_gotypes.go中会包含代码包导入语句 import _ "runtime/cgo" 和 import "syscall" 。代码包导入语句 import _ "runtime/cgo" 只是引发了代码包runtime/cgo中的初始化函数的执行而没有被分配到一个具体的命名空间上。在这些初始化函数中，包含了对一些C语言的全局变量和函数声明的初始化过程。需要注意的是，只要我们在执行 go tool cgo 命令的时候加入了标记 -gccgo ，即使标记 -import_runtime_cgo 有效，在Go语言源码文件_cgo_gotypes.go中也不会包含 import _ "runtime/cgo" 语句。

至此，我们在本小节讨论的都是Go语言代码如果通过cgo工具调用标准C语言编写的函数。其实，我们利用cgo工具还可以把Go语言编写的函数暴露给C语言代码。

Go语言可以使它的函数被C语言代码所用。这是通过使用一个特殊的注释“//export”来实现的。示例如下：

```
package cgo

/*
#include <stdio.h>
extern void CFunction1();
*/
```

```
import "C"

import "fmt"

//export GoFunction1
func GoFunction1() {
	fmt.Println("GoFunction1() is called.")
}

func CallCFunc() {
	C.CFunction1()
}
```

在这个示例中，我们使用注释行“//export GoFunction1”把Go语言函数 GoFunction1 暴露给了C语言代码。注意，注释行中在“//export ”之后的字符串必须与其下一行的那个函数的名字一致。我们也可以把字符串“//export”看成一种指令符，就像 #cgo 和 #include 。这里有一个限制，就是只要我们使用了指令符“//export”，在当前源码文件的序文中就不能包含任何C语言定义语句，只可以包含C语言声明语句。上面示例的序文中的 extern void CFunction1(); 就是一个很好的例子。序文中的这一行C语言声明语句会被拷贝到两个不同的cgo工具生成的C语言源码文件中。这也正是其序文中不能包含C语言定义语句的原因。那么C语言函数 CFunction1 的定义语句我们应该放在哪儿呢？答案是放到在同目录的其它Go语言源码文件的序文中，或者直接写到C语言源码文件中。

我们把上面示例中的内容保存到名为go_export.go的文件中，并放到goc2p项目的 basic/cgo/lib 代码包中。现在我们使用 go tool cgo 来处理这个源码文件。如下：

```
hc@ubt:~/golang/goc2p/basic/cgo/lib$ go tool cgo go_export.go
```

之后，我们会发现在_obj子目录下的C语言头文件_cgo_export.h中包含了这样一行代码：

```
extern void GoFunction1();
```

这说明C语言代码已经可以对函数 GoFunction1 进行调用了。现在我们使用 go build 命令构建goc2p项目的代码包 basic/cgo ，如下：

```
hc@ubt:~/golang/goc2p/basic/cgo/lib$ go build
# basic/cgo/lib
/tmp/go-build477634277/basic/cgo/lib/_obj/go_export.cgo2.o: In function `_cgo_cc103c85817e_Cfunc_CFunction1':
./go_export.go:34: undefined reference to `CFunction1'
collect2: ld return 1
```

构建并没有成功完成。根据错误提示信息我们获知，C语言函数 CFunction1 未被定义。这个问题的原因是我们并没有在Go语言源码文件go_export.go的序文中写入C语言函数 CFunction1 的实现，也即未对它进行定

义。我们之前说过，在这种情况下，对应函数的定义应该被放到其它Go语言源码文件的序文或者C语言源码文件中。现在，我们在当前目录下创建一个新的Go语言源码文件go_export_def.go。其内容如下：

```
package cgo

/*
#include <stdio.h>
void CFunction1() {
	printf("CFunction1() is called.\n");
	GoFunction1();
}
*/
import "C"
```

这个文件是专门用于存放C语言函数定义的。注意，由于C语言函数 printf 来自C语言标准代码库stdio.h，所以我们需要在序文中使用指令符#include将它引入。保存好源码文件go_export_def.go之后，我们重新使用 go tool cgo 命令处理这两个文件，如下：

```
hc@ubt:~/golang/goc2p/basic/cgo/lib$ go tool cgo go_export.go go_export_def.go
```

然后，我们再次执行 go build 命令构建代码包 basic/cgo/lib ：

```
hc@ubt:~/golang/goc2p/basic/cgo/lib$ go build
```

显然，这次的构建成功完成。当然单独构建代码包 basic/cgo/lib 并不是必须的。我们在这里是为了检查该代码包中的代码（包括Go语言代码和C语言代码）是否都能够被正确编译。

还记得goc2p项目的代码包basic/cgo中的命令源码文件cgo_demo.go。现在我们在它的main函数的最后加入一行新代码： cgo.CallCFunc() ，即调用在代码包``basic/cgo/lib```中的库源码文件go_export.go的函数。然后，我们运行这个命令源码文件：

```
hc@ubt:~/golang/goc2p/basic/cgo$ go run cgo_demo.go
The square root of 2.330000 is 1.526434.
ABC
CFunction1() is called.
GoFunction1() is called.
```

从输出的信息可以看出，我们定义的C语言函数 CFunction1 和Go语言函数 GoFunction1 都已被调用，并且调用顺序正如我们所愿。这个例子也说明，我们可以非常方便的使用cgo工具完成如下几件事：

1. Go语言代码调用标准C语言的代码。这也使得我们可以使用Go语言封装任何已存在的C语言代码库，并提供给其他Go语言代码使用。

2. 可以在Go语言源码文件的序文中自定义任何C语言代码并由Go语言代码使用。这使得我们可以更灵活的对C语言代码进行封装。同时，我们还可以利用这一特性在我们自定义的C语言代码中使用Go语言代码。

3. 通过指令符“//export”，可使C语言代码能够使用Go语言代码。这里所说的C语言代码是指我们在Go语言源码文件的序文中自定义的C语言代码。但是，`go tool cgo`命令会将序文中的C语言代码声明和定义分别写入到其生成的C语言头文件和C语言源码文件中。所以，从原则上讲，这已经具备了让外部C语言代码使用Go语言代码的能力。

综上所述，cgo工具不但可以使Go语言直接使用现存的非常丰富的C语言代码库，还可以使用Go语言代码扩展现有的C语言代码库。

至此，我们介绍了怎样独立的使用cgo工具。但实际上，我们可以直接使用标准go命令构建、安装和运行导入了代码包C的代码包和源码文件。标准go命令能够认出代码包C的导入语句并自动使用cgo工具进行处理。示例如下：

```
hc@ubt:~/golang/goc2p/src/basic/cgo$ rm -rf lib/_obj
hc@ubt:~/golang/goc2p/src/basic/cgo$ go run cgo_demo.go
The square root of 2.330000 is 1.526434.
ABC
CFunction1() is called.
GoFunction1() is called.
```

在上例中，我们首先删除了代码包`basic/cgo/lib`目录下的子目录_obj，以此来保证原始的测试环境。然后，我们直接运行了命令源码文件cgo_demo.go。在这个源码文件中，包含了对代码包`basic/cgo/lib`中函数的调用语句，而在这些函数中又包含了对代码包C的引用。从输出信息我们可以看出，命令源码文件cgo_demo.go的运行成功的完成了。这也验证了标准go命令在这方面的功能。不过，有时候我们还是很有必要单独使用`go tool cgo`命令，比如对相关的Go语言代码和C语言代码的功能进行验证或者需要通过标记定制化运行cgo工具的时候。另外，如果我们通过标准go命令构建或者安装直接或间接导入了代码C的命令源码文件，那么在生成的可执行文件中就会包含动态导入数据和动态链接器信息。我们可以使用`go tool cgo`命令查看可执行文件中的这些信息。

go env

命令 go env 用于打印Go语言的环境信息。其中的一些信息我们在之前已经多次提及，但是却没有进行详细的说明。在本小节，我们会对这些信息进行深入介绍。我们先来看一看 go env 命令情况下都会打印出哪些Go语言通用环境信息。

表0-25 go env 命令可打印出的Go语言通用环境信息

名称	说明
CGO_ENABLED	指明cgo工具是否可用的标识。
GOARCH	程序构建环境的目标计算架构。
GOBIN	存放可执行文件的目录的绝对路径。
GOCHAR	程序构建环境的目标计算架构的单字符标识。
GOEXE	可执行文件的后缀。
GOHOSTARCH	程序运行环境的目标计算架构。
GOOS	程序构建环境的目标操作系统。
GOHOSTOS	程序运行环境的目标操作系统。
GOPATH	工作区目录的绝对路径。
GORACE	用于数据竞争检测的相关选项。
GOROOT	Go语言的安装目录的绝对路径。
GOTOOLDIR	Go工具目录的绝对路径。

下面我们对这些环境信息进行逐一说明。

CGO_ENABLED

通过上一小节的介绍，相信读者对cgo工具已经很熟悉了。我们提到过，标准go命令可以自动的使用cgo工具对导入了代码包C的代码包和源码文件进行处理。这里所说的“自动”并不是绝对的。因为当环境变量CGO_ENABLED被设置为0时，标准go命令就不能处理导入了代码包C的代码包和源码文件了。请看下面的示例：

```
hc@ubt:~/golang/goc2p/src/basic/cgo$ export CGO_ENABLED=0
hc@ubt:~/golang/goc2p/src/basic/cgo$ go build -x
WORK=/tmp/go-build775234613
```

我们临时把环境变量CGO_ENABLED的值设置为0，然后执行 go build 命令并加入了标记 -x 。标记 -x 会让命令程序将运行期间所有实际执行的命令都打印到标准输出。但是，在执行命令之后没有任何命令被打印出来。这说明对代码包 basic/cgo 的构建操作并没有被执行。这是因为，构建这个代码包需要用到cgo工具，但cgo工具已经被禁用了。下面，我们再来运行调用了代码包 basic/cgo 中函数的命令源码文件cgo_demo.go。也就

是说，命令源码文件cgo_demo.go间接的导入了代码包 C 。还记得吗？这个命令源码文件被存放在goc2p项目的代码包 basic/cgo 中。示例如下：

```
hc@ubt:~/golang/goc2p/src/basic/cgo$ export CGO_ENABLED=0
hc@ubt:~/golang/goc2p/src/basic/cgo$ go run -work cgo_demo.go
WORK=/tmp/go-build856581210
# command-line-arguments
./cgo_demo.go:4: can't find import: "basic/cgo/lib"
```

在上面的示例中，我们在执行 go run 命令时加入了两个标记——-a 和 -work 。标记 -a 会使命令程序强行重新构建所有的代码包（包括涉及到的标准库），即使它们已经是最新的了。标记 -work 会使命令程序将临时工作目录的绝对路径打印到标准输出。命令程序输出的错误信息显示，命令程序没有找到代码包 basic/cgo 。其原因是由于代码包 basic/cgo 无法被构建。所以，命令程序在临时工作目录和工作区中都找不到代码包basic/cgo对应的归档文件cgo.a。如果我们使用命令 ll /tmp/go-build856581210 查看临时工作目录，也找不到名为basic的目录。

不过，如果我们在环境变量CGO_ENABLED的值为1的情况下生成代码包 basic/cgo 对应的归档文件cgo.a，那么无论我们之后怎样改变环境变量CGO_ENABLED的值也都可以正确的运行命令源码文件cgo_demo.go。即使我们在执行 go run 命令时加入标记 -a 也是如此。因为命令程序依然可以在工作区中找到之前在我们执行 go install 命令时生成的归档文件cgo.a。示例如下：

```
hc@ubt:~/golang/goc2p/src/basic/cgo$ export CGO_ENABLED=1
hc@ubt:~/golang/goc2p/src/basic/cgo$ go install ../basic/cgo
hc@ubt:~/golang/goc2p/src/basic/cgo$ export CGO_ENABLED=0
hc@ubt:~/golang/goc2p/src/basic/cgo$ go run -a -work cgo_demo.go
WORK=/tmp/go-build130612063
The square root of 2.330000 is 1.526434.
ABC
CFunction1() is called.
GoFunction1() is called.
```

由此可知，只要我们事先成功安装了引用了代码包C的代码包，即生成了对应的代码包归档文件，即使cgo工具在之后被禁用，也不会影响到其它Go语言代码对该代码包的使用。当然，命令程序首先会到临时工作目录中寻找需要的代码包归档文件。

关于cgo工具还有一点需要特别注意，即：当存在交叉编译的情况时，cgo工具一定是不可用的。在标准go命令的上下文环境中，交叉编译意味着程序构建环境的目标计算架构的标识与程序运行环境的目标计算架构的标识不同，或者程序构建环境的目标操作系统的标识与程序运行环境的目标操作系统的标识不同。在这里，我们可以粗略认为交叉编译就是在当前的计算架构和操作系统下编译和构建Go语言代码并生成针对于其他计算架构或/和操作系统的编译结果文件和可执行文件。

GOARCH

GOARCH的值的含义是程序构建环境的目标计算架构的标识，也就是程序在构建或安装时所对应的计算架构的名称。在默认情况下，它会与程序运行环境的目标计算架构一致。即它的值会与GOHOSTARCH的值是相同。但如果我们显式的设置了环境变量GOARCH，则它的值就会是这个环境变量的值。

GOBIN

GOBIN的值为存放可执行文件的目录的绝对路径。它的值来自环境变量GOBIN。在我们使用 `go tool install` 命令安装命令源码文件时生成的可执行文件会存放于这个目录中。

GOCHAR

GOCHAR的值是程序构建环境的目标计算架构的单字符标识。它的值会根据GOARCH的值来设置。当GOARCH的值为386时，GOCHAR的值就是8。当GOARCH的值为amd64时GOCHAR的值就是6。而GOCHAR的值5与GOARCH的值arm相对应。

GOCHAR主要有两个用途，列举如下：

1. Go语言官方的平台相关的工具的名称会以它的值为前缀。的名称会以GOCHAR的值为前缀。比如，在amd64计算架构下，用于编译Go语言代码的编译器的名称是6g，链接器的名称是6l。用于编译C语言代码的编译器的名称是6c。而用于编译汇编语言代码的编译器的名称为6a。
2. Go语言的官方编译器生成的结果文件会以GOCHAR的值作为扩展名。Go语言的官方编译器6g在对命令源码文件编译之后会把结果文件*go*.6存放到临时工作目录的相应位置中。

GOEXE

GOEXE的值会被作为可执行文件的后缀。它的值与GOOS的值存在一定关系，即只有GOOS的值为“windows”时GOEXE的值才会是“.exe”，否则其值就为空字符串“”。这与在各个操作系统下的可执行文件的默认后缀是一致的。

GOHOSTARCH

GOHOSTARCH的值的含义是程序运行环境的目标计算架构的标识，也就是程序在运行时所在的计算机系统的计算架构的名称。在通常情况下，它的值不需要被显式的设置。因为用来安装Go语言的二进制分发文件和MSI（Microsoft软件安装）软件包文件都是平台相关的。所以，对于不同计算架构的Go语言环境来说，它都会是一个常量。

GOHOSTOS

GOHOSTOS的值的含义是程序运行环境的目标操作系统的标识，也即程序在运行时所在的计算机系统的操作系统的名称。与GOHOSTARCH类似，它的值在不同的操作系统下是固定不变的，同样不需要显式的设置。

GOPATH

这个环境信息我们在之前已经提到过很多次。它的值指明了Go语言工作区目录的绝对路径。我们需要显式的设置环境变量GOPATH。如果有多个工作区，那么多个工作区的绝对路径之间需要用分隔符分隔。在windows操作系统下，这个分隔符为“;”。在其它操作系统下，这个分隔符为“:”。注意，GOPATH的值不能与GOROOT的值相同。

GORACE

GORACE的值包含了用于数据竞争检测的相关选项。数据竞争是在并发程序中最常见和最难调试的一类bug。数据竞争会发生在多个Goroutine争相访问相同的变量且至少有一个Goroutine中的程序在对这个变量进行写操作的情况下。

数据竞争检测需要被显式的开启。还记得标记 `-race` 吗？我们可以通过在执行一些标准go命令时加入这个标记来开启数据竞争检测。在这种情况下，GORACE的值就会被使用到了。支持 `-race` 标记的标准go命令包括： `go test` 命令、 `go run` 命令、 `go build` 命令和 `go install` 命令。

GORACE的值形如“option1=val1 option2=val2”，即：选项名称与选项值之间以等号“=”分隔，多个选项之间以空格“ ”分隔。数据竞争检测的选项包括log_path、exitcode、strip_path_prefix和history_size。为了设置GORACE的值，我们需要设置环境变量GORACE。或者，我们也可以在执行go命令时临时设置它，像这样：

```
hc@ubt:~/golang/goc2p/src/cnet/ctcp$ GORACE="log_path=/home/hc/golang/goc2p /race/report strip_path_prefix=hom
```

关于数据竞争检测的更多细节我们将会在本书的第三部分予以说明。

GOROOT

GOROOT会是我们在安装Go语言时第一个碰到Go语言环境变量。它的值指明了Go语言的安装目录的绝对路径。但是，只有在非默认情况下我们才需要显式的设置环境变量GOROOT。这里所说的默认情况是指：在Windows操作系统下我们把Go语言安装到c:\Go目录下，或者在其它操作系统下我们把Go语言安装到/usr/local/go目录下。另外，当我们不是通过二进制分发包来安装Go语言的时候，也不需要设置环境变量GOROOT的值。比如，在Windows操作系统下，我们可以使用MSI软件包文件来安装Go语言。

GOTOOLDIR

GOTOOLDIR的值指明了Go工具目录的绝对路径。根据GOROOT、GOHOSTOS和GOHOSTARCH来设置。其值为$GOROOT/pkg/tool/$GOOS_$GOARCH。关于这个目录，我们在之前也提到过多次。

除了上面介绍的这些通用的Go语言环境信息，还两个针对于非Plan 9操作系统的环境信息。它们是CC和GOGCCFLAGS。环境信息CC的值是操作系统默认的C语言编译器的命令名称。环境信息GOGCCFLAGS的值则是Go语言在使用操作系统的默认C语言编译器对C语言代码进行编译时加入的参数。

如果我们要有针对性的查看上述的一个或多个环境信息，可以在 `go env` 命令的后面加入它们的名字并执行之。在 `go env` 命令和环境信息名称之间需要用空格分隔，多个环境信息名称之间也需要用空格分隔。示例如下：

```
hc@ubt:~$ go env GOARCH GOCHAR
386
8
```

上例的 `go env` 命令的输出信息中，每一行对一个环境信息的值，且其顺序与我们输入的环境信息名称的顺序一致。比如，386为环境信息GOARCH，而8则是环境信息GOCHAR的值。

`go env` 命令能够让我们对当前的Go语言环境进行简要的了解。通过它，我们也可以对当前安装的Go语言的环境设置进行简单的检查。