layout: post
title: 002-如何编写Go代码
category: golang架构师之路
tags: golang

keywords: golang, go package, go install,架构师

前言

该文档将演示简单的Go package的开发并介绍Go工具(包含`fetch,build,install`等命令)，以及GOPATH,GOROOT,GOBIN各种Go相关的环境变量配置。

Go工具要求以特定的方式组织代码，请详细阅读该文档。你会了解到Go使用安装和最快运行Go程序的方法。

Go代码组织结构详解

概述

- Go开发者通常将所有Go代码保存在单个`工作区`中
- 工作区包含多个代码版本仓库，例如git库
- 每个代码仓库包含一个或多个包
- 每个包由单个目录中的一个或多个Go源文件组成
- 包路径确定其导入路径($GOPATH/src/**${pacakge_path}**,${package_path}重要)

注意Go的工作区组织方式不同于那些每个项目都有单独的工作区并且每个工作区与版本控制库紧密相关的其他编程环境。比如Java一个项目一个工作区(工作空间)也是一个版本库(java工程的最佳实践)。而Go工作区则是多个版本控制库，每个版本控制库对应一个项目(单个工作区多个项目代码库)

```
java
project->workspace->git
1-1-1    // 对应关系

go
workspace->git->project
```

```
1-n-n // 对应关系
```

工作区

工作区(workspace)是一个目录层次结构，其根目录有三个目录构成:

- `src` 包含Go源文件
- `pkg` 包含包对象
- `bin` 包含可执行命令

go工具编译构建 `src` 源软件包并将生成的二进制文件安装到 `pkg` 和 `bin` 目录。

`src` 子目录通常包含多个代码版本控制仓库，用于追踪一个或多个源码包的开发。

为了了解实践中的工作区是什么样子，演示个例子:

```
bin/
    hello                           # command executable
    outyet                          # command executable
pkg/
    linux_amd64/
        github.com/golang/example/
            stringutil.a            # package object
src/
    github.com/golang/example/
        .git/                       # Git repository metadata
    hello/
        hello.go               # command source
    outyet/
        main.go                # command source
        main_test.go           # test source
    stringutil/
        reverse.go             # package source
        reverse_test.go        # test source
    golang.org/x/image/
        .git/                       # Git repository metadata
    bmp/
        reader.go              # package source
        writer.go              # package source
    ... (many more repositories and packages omitted) ...
```

以上展示了包含两个代码仓库(example和image)的工作区(注意: `一个` 工作区内)。example仓库包含两个命令(hello和outyet)和一个类库(stringutil)。image仓库包含bmp包和几个其他的。

典型的工作区包含着由许多包和命令的源码仓库组成。**大多数Go开发者将Go源码和依赖保存在单个工作区中。**

命令和库由不同类型的源码包构建。

GOPATH默认环境变量

GOPATH环境变量特指工作区的路径。

默认情况下在系统"家目录"(`$HOME`)下的`go`目录(`$HOME/go`)。
* Unix `$HOME/go`
* Plan9 `$HOME/go`
* Windows `%USERPROFILE%\go` (通常是`c:\Users\$UserName\go`)

如果想要在非默认工作区的路径工作，那么你需要设置`GOPATH`环境变量指向那个目录(另一种常见的设置比如是设置`GOPATH=$HOME`)。**注意GOPATH一定不能是Go SDK的安装目录。**

小贴士:

命令`go env GOPATH`打印当前的`GOPATH`。

如果没有设置其他位置的环境变量它将打印默认位置`$HOME/go`。

```
$ go env GOPATH
/Users/fqc/github/golang_sidepro
```

为方便起见(编译后的程序可以在任意目录下直接启动运行，后面会演示)，将工作区的`bin`子目录添加到系统环境变量`PATH`中:

```
$ export PATH=$PATH:$(go env GOPATH)/bin
```

注意:不是`go env` $GOPATH

另外为了简洁起见，本文档的其他脚本使用`$GOPATH`替代`$(go env GOPATH)`。如果你没有设置`GOPATH`却想让脚本正常运行，可以通过`$HOME/go`替代`$(go env GOPATH)`命令或者运行

```
export GOPATH=$(go env GOPATH)
```

上述的命令的含义实质还是将`$(go env GOPATH)`赋值给`GOPATH`变量，以后使用`$GOPATH`替

代 `$(go env GOPATH)` 。

学习更多的GOPATH环境变量，可以通过 `go help gopath` 命令。

```
☁  golang_sidepro [master] ⚡ go help gopath
                                    [master↑2|+66…
The Go path is used to resolve import statements.
It is implemented by and documented in the go/build package.

The GOPATH environment variable lists places to look for Go code.
On Unix, the value is a colon-separated string.
On Windows, the value is a semicolon-separated string.
On Plan 9, the value is a list.

If the environment variable is unset, GOPATH defaults
to a subdirectory named "go" in the user's home directory
($HOME/go on Unix, %USERPROFILE%\go on Windows),
unless that directory holds a Go distribution.
Run "go env GOPATH" to see the current GOPATH.

......

being checked out for the first time by 'go get': those are always
placed in the main GOPATH, never in a vendor subtree.

See https://golang.org/s/go15vendor for details.
```

注意上述都是在默认情况下的设置。

```
☁  golang架构师之路 [master] ⚡ cd $GOPATH
                                  [master↑1|+3…
☁  golang_sidepro [master] ⚡ ll
                                  [master↑2|+66…
total 48
-rw-r--r--   1 fqc  staff    19K Jun  1  2016 LICENSE
-rw-r--r--   1 fqc  staff    56B Jun  1  2016 README.md
drwxr-xr-x   9 fqc  staff   306B Apr 16 22:39 bin
drwxr-xr-x   3 fqc  staff   102B Apr 14 15:20 pkg
drwxr-xr-x  20 fqc  staff   680B Apr 14 14:58 src
```

自定义GOPATH环境变量

`GOPATH` 可以是你操作系统的任意目录(除了Go SDK目录)。在Unix示例中我们将它设置到 `$HOME/work` 。注意 `GOPATH` 一定不能和GO SDK的安装目录。另一个常用设置

是 `GOPATH=$HOME`

- Unix系统
  - Bash
  - Zsh
- Windows系统

Bash

编辑 `.bash_profile`，添加如下一行:

```sh
export GOPATH=$HOME/work
```

保存并退出编辑器。然后刷新 `~/.bash_profile` 。

```sh
source ~/.bash_profile
```

Zsh

编辑 `~/.zshrc` 文件，添加如下一行:

```sh
export GOPATH=$HOME/work
```

保存并退出编辑器。然后刷新 `~/.zshrc`

注意:由于zsh的特殊原因，可能需要在 `~/.zshrc` 中添加一行 `source ~/.bash_profile` 中使得配置永久生效，不用每次重启后需要刷新配置文件。

Windows

工作区你可以选择自己喜欢的，但是我们将采用 `C:work` 作为示例演示。注意 `GOPATH` 不能和你的 `GO SDK` 安装路径一样
1. 创建 `C:\work` 目录
2. 选择 `开始` -> `控制面板` -> `系统和安全` -> `系统` -> `高级系统设置` -> `环境变量`
3. 创建用户变量
4. 变量名一栏输入 `GOPATH`
5. 变量值一栏输入 `C:\work`
6. 点击完成

总结:

记住`GOPATH`并不是Go的安装路径，而是工作区WorkSpace路径。
`GOPATH`分两种情况

- 默认不设置`$HOME/go`
- 自定义

GOROOT

查看go安装的目录

```
$ go env GOROOT
/usr/local/go
```

查看go命令所在
`sh`
`➜ ~ which go`
`/usr/local/go/bin/go`

导入路径 Import path

`import path`是唯一标识包的字符串。包的导入路径对应其在工作区或远程代码库的位置(如下所述)。

- 标准库的代码包
  使用较短的导入路径即可，例如`"fmt"`或`"net/http"`。
- 自定义的包
  必须选择不太可能与标准库或未来添加的外部库相冲突的一个基本路径。

注意在可以成功构建代码之前不要发布代码到远程仓库。**平时注意组织好代码就好像某天你将发布它一样是个非常好的习惯**(说到这里就需要特别注意`code review`对代码质量的重要性)。实际上你可以选择任意的路径，只要它在针对工作区、标准库或更庞大的Go生态圈是唯一的。

我们将采用`github.com/user`作为`项目根路径`或`项目基本路径`。在工作区存放源码的文件夹下创建该目录：

```
mkdir -p $GOPATH/src/github.com/user
```

第一个go程序

为了编译和运行一个简单的程序，
- 首先我们需要选择`包路径`(我们这里采用`github.com/user/hello`)

在工作区创建相应的目录:

```sh
mkdir $GOPATH/src/github.com/user/hello
```

- 然后，在刚创建的目录里新建文件`hello.go`

  包含如下代码:

  ```
  package main

  import "fmt"

  func main(){
      fmt.Print("hello world.\n")
  }
  ```

现在可以使用go工具构建并安装该程序:

```
go install github.com/user/hello
```

注意：**可以在系统的任意位置运行上述命令**。go工具会根据配置的环境变量`GOPATH`下寻找`github.com/user/hello`包。

错误示例:

```
➜  ~  go install $GOPATH/src/github.com/user/hello
can't load package: package /Users/fqc/work/src/github.com/user/hello: impor
t "/Users/fqc/work/src/github.com/user/hello": cannot import absolute path
➜  ~  go install $GOPATH/src/github.com/user/hello/hello.go
go install: no install location for .go files listed on command line (GOBIN
not set)
```

如果**已经在包路径下**，也可以忽略包路径直接执行`go install`:

```
$ cd $GOPATH/src/github.com/user/hello
```

```
$ go install
```

上述命令构建`hello` command，生成可执行二进制文件。然后将二进制文件安装到工作区的bin目录为hello(或在Windows下，hello.exe)。在我们的示例中，将为`$GOPATH/bin/hello`，也就是`$HOME/work/bin/hello`。

go工具只会在发生错误时打印输出，因此如果这些命令不产生输入，则说明它们已成功执行。

你可以通过输入下面的全路径运行程序:

```
$ $GOPATH/bin/hello
Hello, wolrd.
```

或者你可以更加聪明便捷的方式，将`$GOPATH/bin`将入到`$PATH`中，以后只需要输入二进制名称即可:

```
$ hello
Hello, world.
```

如果你正在使用源码控制系统，现在是个初始化仓库的绝佳时机，添加文件并提交第一次修改.另外，这个步骤是可选的:你不是非要使用代码版本控制来写Go代码(但是最好用上)。

```
$ cd $GOPATH/src/github.com/user/hello
$ git init
Initialized empty Git repository in /home/user/work/src/github.com/user/hell
o/.git/
$ git add hello.go
$ git commit -m "initial commit"
[master (root-commit) 0b4507d] initial commit
 1 file changed, 1 insertion(+)
  create mode 100644 hello.go
```

最后就是关联远程仓库推送代码啦，读者可以自行完成。

第一个类库

我们来写一个类库并在`hello`项目中使用。

再次重申强调，**第一步是选择包路径**(我们将使用`github.com/user/stringutil`)并创建包目录:

`sh`

```
$ mkdir $GOPATH/src/github.com/user/stringutil
```

第二步，在`stringutil`中创建`reverse.go`文件，包含以下内容:

```
package stringutil
func Reverse(s string) string{
    r := []rune(s)
    for i, j := 0, len(r)-1; i < len(r)/2; i, j = i+1,j-1{
       r[i], r[j] = r[j], r[i]
    }
    return string(r)
}
```

现在使用`go build`构建包:

```
$ go build github.com/user/stringutil
```

或者，你就位于包目录下，直接运行:

```
$ go build
```

`go build`并不产生输出文件，针对于非main函数的类库，而针对有主函数的包会在当前目录编译输出二进制命令文件。需要的话，必须使用`go install`，该命令将包对象放到工作区的`pkg`目录中，将二进制命令的文件放到`bin`目录，这俩目录会通过`go install`执行时自动生成。

在确保`stringutil`包构建后，使用它来修改原先的`hello.go`(位于$GOPATH/src/github.com/user/hello):

```
package main
import (
    "fmt"
    "github.com/user/stringutil"
)
func main(){
    fmt.Printf(stringutil.Reverse("!oG ,olleH"))
}
```

每当Go工具安装包或二进制文件时，它都会安装它的所有依赖。所以当你安装(install)`hello`项目时:

```
$ go install github.com/user/hello
```

stringutil也会自动地被安装好。

运行新版本的程序，你将会看到一个新的且翻转的信息:

```
$ hello
Hello, Go!
```

经过以上操作步骤之后，你的工作区看起来应该向如下:

```
sh
bin/
hello # command executable
pkg/
linux_amd64/ # this will reflect your OS and architecture
github.com/user/
stringutil.a # package object
src/
github.com/user/
hello/
hello.go # command source
stringutil/
reverse.go # package source
```

注意 `go install` 将 `stringutil.a` 对象保存到它的源码目录镜像 `pkg/linux_amd64` 中。这是为了以后Go工具的调用可以 `找到包对象避免不必要的重复编译` 。linux_amd64部分是帮助交叉编译，并将反映操作系统和体系结构。

Go命令可执行文件是 `静态链接` 的；包对象不需要出现在运行的Go程序中。

注:除了main包之外，其他的包都会生成 `*.a文件` 即包文件

包名

Go源文件第一个语句必须是 `package name` ，其中 `name` 是包的导入的默认名称。(所有的包文件必须使用相同的 `name` 。)

Go语言的约定是 `包名为导入语句的最后一个元素` :例如导入为"crypto/rot13"包应该命名为 `rot13` 。

`可执行二进制命令`必须始终使用`package main`。

不要求将包名称链接到单个二进制文件中的所有包都是唯一的，只要求`导入路径`(完整文件名)是`唯一`的。

测试

Go有一个由`go test`测试命令和测试包组成的轻量级测试框架。

1. 创建一个名称以`_test.go`结尾的源文件
2. 创建一个方法签名为`func TestXXX(t *testing.T)`的方法(也可以使用性能测试Benchmark为前缀测试 参数类型`*testing.B`) `func BenchmarkXXX(b *testing.B)` **注意方法名必须是以Test开头或者Benchmark开头！**
3. 运行 go test 普通测试，性能benchmark测试需要加参数运行 `-bench="."` 测试框架将执行每个这样的函数;如果函数调用诸如`t.Error`或`t.Fail`的失败函数，则该测试被认为是失败的。

给stringutil添加一个测试。

1. 新建stringutil_test.go

```sh
$ mkdir $GOPATH/src/github.com/user/stringutil/stringutil_test.go
```

2. 编写测试代码

```
package stringutil
import "testing"

func TestReverse(t *testing.T) {
    cases := []struct {
        in, want string
    }{
        {"Hello, world", "dlrow ,olleH"},
        {"Hello, 世界", "界世 ,olleH"},
        {"", ""},
    }
    for _, c := range cases {
        got := Reverse(c.in)
        if got != c.want {
            t.Errorf("Reverse(%q) == %q, want %q", c.in, got, c.want)
        }
    }
```

```
}
```

然后运行 `go test` 命令执行该文件的测试

```
$ go test github.com/user/stringutil
ok      github.com/user/stringutil  0.007s
```

像之前一样，如果你已经在要编译运行的目录下，就可以忽略包路径:

```
$ go test
PASS
ok      github.com/user/stringutil  0.007s
```

运行 `go help test` 可以查看更多详细文档说明。

benchmark test

golang 测试单个文件

1，测试单个文件，一定要带上被测试的原文件

`Go test -v wechat_test.go wechat.go`

2，测试单个方法

`go test -v -test.run TestRefreshAccessToken`

远程包

包导入路径可以描述如何使用Git或Mercurial版本控制工具获取包的源代码。go工具使用这个属性功能自动的从远程仓库获取软件包。例如，本示例中描述的文档也保存在github.com/golang/example Github仓库中。如果你在包路径引入了该仓库路径， `go get` 将fetch,build and install it automatically(自动下载获取，构建，安装):

```
$ go get github.com/golang/example/hello
$ $GOPATH/bin/hello
```

如果指定的包不存在工作区中， `go get` 将会下载的内容存放于GOPATH设置的第一个工作区中。(如果程序包已经存在， `go get` 将会跳过远程获取，然后执行 `go install`)

在发出上面的`go get`命令后，工作区目录树看起来应该像下面:

```
bin/
    hello                           # command executable
pkg/
    linux_amd64/
        github.com/golang/example/
            stringutil.a            # package object
        github.com/user/
            stringutil.a            # package object
src/
    github.com/golang/example/
    .git/                       # Git repository metadata
        hello/
            hello.go                # command source
        stringutil/
            reverse.go              # package source
            reverse_test.go         # test source
    github.com/user/
        hello/
            hello.go                # command source
        stringutil/
            reverse.go              # package source
            reverse_test.go         # test source
```

在GitHub托管的hello命令依赖于统一仓库中的`stringutil`包。在`hello.go`文件中使用相同的导入路径，然后`go get`命令也可以定位并按照依赖程序包。

```
import "github.com/golang/example/stringutil"
```

这种规约是使开发者将自己的Go软件包分享出去的最简单的方式。 Go Wiki和godoc.org提供外部Go过程的列表。

接下来

- 订阅golang-announce邮件列表，当新的稳定版本的Go发布时可以收到通知
- 阅读实践Effective Go，掌握编写简洁清晰有效惯用的Go代码
- 学习《A Tour of Go》正确地掌握Go
- 访问官方文档，了解有关Go语言及其库和工具的一组深入的文章。

获取帮助

- 获得实时的帮助，可以通过Freenode IRC Server的#go-nuts询问gophers获得帮助。
- Go官方邮件列表的Go Nuts讨论组
- 使用Go issue tracker报告bug

参考

golang文档

SettingGOPATH
golang-rune

linux静态链接库与动态链接库的区别及动态库的创建
Effective Go