Go, the unwritten parts

Articles mostly about Go and what I am currently working on. Conventions, best practices, little known practical tips.

by **JBD**

Home
Archive
About
GitHub

Subscribe to the **feed**.

Style guideline for Go packages

Sat, Jan 14, 2017

Go is about naming and organization as much as everything else in the language. Well-organized Go code is easy to discover, use and read. Well-organized code is as critical as well designed APIs. The location, name, and the structure of your packages are the first elements your users see and interact with.

This document's goal is to guide you with common good practices not to set rules. You will always need to use your own judgement to pick the most elegant solution for your specific case.

Packages

All Go code is organized into packages. A package in Go is simply a directory/folder with one or more `.go` files inside of it. Go packages provide isolation and organization of code similar to how directories/folders organize files on a computer.

All Go code lives in a package and a package is the entry point to access Go code. Understanding and establishing good practices around packages is important to write effective Go code.

Package Organization

Let’s begin with suggestions how you should organize Go code and explain conventions about locating Go packages.

Use multiple files

A package is a directory with one or more Go files. Feel free to separate your code into as many files as logically make sense for optimal readability.

For example, an HTTP package might have been separated into different files according to the HTTP aspect the file handles. In the following example, an HTTP package is broken down into a few files: header types and code, cookie types and code, the actual HTTP implementation, and documentation of the package.

```
- doc.go       // package documentation
- headers.go   // HTTP headers types and code
- cookies.go   // HTTP cookies types and code
- http.go      // HTTP client implementation, request and response types, etc.
```

Keep types close

As a rule of thumb, keep types closer to where they are used. This makes it easy for any maintainer (not just the original author) to find a type. A good place for a Header struct type might be in `headers.go`.

```
$ cat headers.go
package http

// Header represents an HTTP header.
type Header struct {...}
```

Even though, the Go language doesn’t restrict where you define types, it is often a good practice to keep the core types grouped at the top of a file.

Organize by responsibility

A common practise from other languages is to organize types together in a package called models or types. In Go, we organize code by their functional responsibilities.

```
package models // DON'T DO IT!!!

// User represents a user in the system.
type User struct {...}
```

Rather than creating a models package and declare all entity types there, a User type should live in a service-layer package.

```
package mngtservice

// User represents a user in the system.
type User struct {...}

func UsersByQuery(ctx context.Context, q *Query) ([]*User, *Iterator, error)

func UserIDByEmail(ctx context.Context, email string) (int64, error)
```

Optimize for godoc

It is a great exercise to use godoc in the early phases of your package's API design to see how your concepts will be rendered on doc. Sometimes, the visualization also has an impact on the design. Godoc is the way your users will consume a package, so it is ok to tweak things to make them more accessible. Run `godoc -http=<hostport>` to start a godoc server locally.

Provide examples to fill the gaps

In some cases, you may not be able to provide all related types from a single package. It might be noisy to do so, or you might want to publish concrete implementations of a common interface from a separate package, or those types could be owned by a third-party package. Give examples to help the user to discover and understand how they are used together.

```
$ godoc cloud.google.com/go/datastore
func NewClient(ctx context.Context, projectID string, opts ...option.ClientOption) (*Cl
...
```

NewClient works with option.ClientOptions but it is neither the datastore package nor the option package that export all the option types.

```
$ godoc google.golang.org/extraoption
func WithCustomValue(v string) option.ClientOption
...
```

If your API requires many non-standard packages to be imported, it is often useful to add a **Go example** to give your users some working code.

Examples are a good way to increase visibility of a less discoverable package. For example, an example for datastore.NewClient might reference the extraoption package.

Don't export from main

An identifier may be **exported** to permit access to it from another package.

Main packages are not importable, so exporting identifiers from main packages is unnecessary. Don't export identifiers from a main package if you are building the package to a binary.

Exceptions to this rule might be the main packages built into a .so, or a .a or Go plugin. In such cases, Go code might be used from other languages via **cgo's export functionality** and exporting identifiers are required.

Package Naming

A package name and import path are both significant identifiers of your package and represent everything your package contains. Naming your packages canonically not just improves your code quality but also your users'.

Lowercase only

Package names should be lowercase. Don't use snake_case or camelCase in package names. The Go blog has a **comprehensive guide** about naming packages with a good variety of examples.

Short, but representative names

Package names should be short, but should be unique and representative. Users of the package should be able to grasp its purpose from just the package's name.

Avoid overly broad package names like "common" and "util".

```
import "pkgs.org/common" // DON'T!!!
```

Avoid duplicate names in cases where user may need to import the same package.

If you cannot avoid a bad name, it is very likely that there is a problem with your overall structure and code organization.

Clean import paths

Avoid exposing your custom repository structure to your users. Align well with the GOPATH conventions. Avoid having src/, pkg/ sections in your import paths.

```
github.com/user/repo/src/httputil   // DON'T DO IT, AVOID SRC!!

github.com/user/repo/gosrc/httputil // DON'T DO IT, AVOID GOSRC!!
```

No plurals

In go, package names are not plural. This is surprising to programmers who came from other languages and are retaining an old habit of pluralizing names. Don't name a package httputils, but httputil!

```
package httputils  // DON'T DO IT, USE SINGULAR FORM!!
```

Renames should follow the same rules

If you are importing more than one packages with the same name, you can locally rename the package names. The renames should follow the same rules mentioned on this article. There is no rule which package you should rename. If you are renaming the standard package library, it is nice to add a go prefix to make the name self document that it is "Go standard library's" package, e.g. `gourl`, `goioutil`.

```
import (
    gourl "net/url"

    "myother.com/url"
)
```

Enforce vanity URLs

`go get` supports getting packages by a URL that is different than the URL of the package's repo. These URLs are called vanity URLs and require you to serve a page with specific meta tags the Go tools recognize. You can serve a package with a custom domain and path using vanity URLs.

For example,

```
$ go get cloud.google.com/go/datastore
```

checks out the source code from `https://code.googlesource.com/gocloud` behind the scenes and puts it in your workspace under $GOPATH/src/cloud.google.com/go/datastore.

Given code.googlesource.com/gocloud is already serving this package, would it be possible to go get the package from that URL? The answer is no, if you enforce the vanity URL.

To do that, add an import statement to the package. The go tool will reject any import of this package from any other path and will display a friendly error to the user. If you don't enforce your vanity URLs, there will be two copies of your package that cannot work together due to the different namespace.

```
package datastore // import "cloud.google.com/go/datastore"
```

Package Documentation

Always document the package. Package documentation is a top-level comment immediately preceding the package clause. For non-main packages, godoc always starts with “Package {pkgname}” and follows with a description. For main packages, documentation should explain the binary.

```
// Package ioutil implements some I/O utility functions.
package ioutil

// Command gops lists all the processes running on your system.
package main

// Sample helloworld demonstrates how to use x.
package main
```

Use doc.go

Sometimes, package docs can get very lengthy, especially when they provide details of usage and guidelines. Move the package godoc to a `doc.go` file. (See an example of a **doc.go**.)

If you have any suggestions or comments, please ping **@rakyll.**