Use internal packages to reduce your public API surface

by Dave Cheney

In the beginning, before the `go` tool, before Go 1.0, the Go distribution stored the standard library in a subdirectory called `pkg/` and the commands which built upon it in `cmd/`. This wasn't so much a deliberate taxonomy but a by product of the original `make` based build system. In September 2014, the Go distribution dropped the `pkg/` subdirectory, but then this tribal knowledge had set root in large Go projects and continues to this day.

I tend to view empty directories inside a Go project with suspicion. Often they are a hint that the module's author may be trying to create a taxonomy of packages rather than ensuring each package's name, and thus its enclosing directory, uniquely describes its purpose. While the symmetry with `cmd/` for `package main` commands is appealing, a directory that exists only to hold other packages is a potential design smell.

More importantly, the boilerplate of an empty `pkg/` directory distracts from the more useful idiom of an `internal/` directory. `internal/` is a special directory name recognised by the `go` tool which will prevent one package from being imported by another unless both share a common ancestor. Packages within an `internal/` directory are therefore said to be *internal packages*.

To create an internal package, place it within a directory named `internal/`. When the `go` command sees an import of a package with `internal/` in the import path, it verifies that the importing package is within the tree rooted at the *parent* of the `internal/` directory.

For example, a package `/a/b/c/internal/d/e/f` can only be imported by code in the directory tree rooted at `/a/b/c`. It cannot be imported by code in `/a/b/g` or in any other repository.

If your project contains multiple packages you may find you have some exported symbols which are intended to be used by other packages in your project, but are not intended to be part of your project's public API. Although Go has limited visibility modifiers–public, exported, symbols and private, non exported, symbols–internal packages provide a useful mechanism for controlling visibility to parts of your project which would otherwise be considered part of its public versioned API.

You can, of course, promote internal packages later if you want to commit to supporting that API; just move them up a directory level or two. The key is this process is *opt-in*. As the author, internal packages give you control over which symbols in your project's public API without being forced to glob concepts together into unwieldy mega packages to avoid exporting them.