On Go modules

> Go and Cloud Native User Group Leipzig
> https://golangleipzig.space

2019-04-12, 19:00 at Basislager.co

> Martin Czygan

Repository with slides will be:

- github.com/golang-leipzig/gomodintro

History

- dependency management has been a pain point for years
- `$GOPATH` was (or is) unintuitive

Many project (at least 15) developed over time.

- https://github.com/golang/go/wiki/PackageManagementTools

Official proposal

> The official Go proposal is at https://golang.org/issue/24301, filed on March 20, 2018 and accepted on May 21, 2018.

The go tool focus

Focused on simplicity in a couple of ways:

- single code tree
- unified go command and simple fetching of libraries and executables with `go get`

Background

Go and versioning series from Russ Cox:

- https://research.swtch.com/vgo

67 pages. XXX: Link to concatenated version.

A few highlights (part 1)

> Although we must add versioning, we also must not remove the best parts of the current go command: its simplicity, speed, and understandability. Today, many programmers mostly don't pay attention to versioning, and everything mostly works fine.

https://research.swtch.com/vgo-intro

A few highlights (2)

> This proposal keeps the best parts of go get, adds reproducible builds, adopts semantic versioning, eliminates vendoring, deprecates GOPATH in favor of a project-based workflow, and provides for a smooth migration from dep and its predecessors.

A few highlights (3)

> In March 2014, Gustavo Niemeyer created gopkg.in, advertising “stable APIs for the Go language.” The domain is a version-aware GitHub redirector, allowing import paths like gopkg.in/yaml.v1 and gopkg.in/yaml.v2 to refer to different commits (perhaps on different branches) of a single Git repository.

A few highlights (4)

> If you're using an externally supplied package and worry that it might change in unexpected ways, the simplest solution is to copy it to your local repository. (This is the approach Google takes internally.)

2014-2017

Proliferation of tools.

- godep
- glide
- gb
- and many more

Also: GOVENDOREXPERIMENT in Go 1.5.

- 2017, GopherCon talk, The New Era of Go Package Management

A few highlights (5)

- **import compatibility rule** hinted at by the Go FAQ and gopkg.in; that is, establish the expectation that newer versions of a package with a given import path should be backwards-compatible with older versions
- **minimal version selection**, to choose which package versions are used in a given build
- introduce the concept of a **Go module**, a group of packages versioned as a single unit and that declare the minimum requirements that must be satisfied by their dependencies
- define how to **retrofit** all this into the existing go command, so that basic workflows do not change significantly from today

A few highlights (6) (part 2)

> For me, design means building, tearing down, and building again, over and over.

https://research.swtch.com/vgo-tour

Example of vgo usage.

> I personally did not try it out.

A few highlights (7) (part 3)

> A year ago, I believed that putting **versions in import paths** like this was **ugly**, undesirable, and probably avoidable. But over the past year, I've come to understand just how much **clarity and simplicity** they bring to the system.

https://research.swtch.com/vgo-import

A few highlights (8)

The *import compatibility rule*:

> If an old package and a new package have the same import path, the new package **must be backwards compatible** with the old package.

The import compatibility rule dramatically simplifies the experience of using incompatible versions of a package. When each different version has a different import path, there is no ambiguity about the intended semantics of a given import statement. This makes it easier for both developers and tools to understand Go programs.

A few highlights (9)

> The problem here is that the **semver spec** is really not much more than a way to choose and compare version strings. It says nothing else.

> The most valuable part of semver is the encouragement to make backwards-compatible changes when possible.

A few highlighs (10)

Ok, we use a new name, but which?

> But choosing a new name at every backwards-incompatible change is difficult and unhelpful to users.

> We can use semantic versioning and follow the import compatibility rule by including the major version in the import path. Instead of needing to invent a cute but unrelated new name like Pocoauth, Alice can call her new API OAuth2 2.0.0, with the new import path "oauth2/v2".

A few highlights (11)

Hello, Plan 9.

> Twenty years ago, Rob Pike and I were modifying the internals of a Plan 9 C library, and Rob taught me the rule of thumb that when you **change a function's behavior**, you also **change its name**.

A few highlights (12)

Enter: Rich Hickey.

> Rich Hickey made the point in his “Spec-ulation” talk in 2016 that this approach of only adding new names and behaviors, never removing old names or redefining their meanings, is exactly what **functional programming** encourages with respect to individual variables or data structures.

A few highlights (13)

- The singleton problem

> By including the major version into the import path, it should be clearer to authors that my/thing and my/thing/v2 are different and need to be able to **coexist**.

A few highlights (14)

Gradual code repair.

> One of the key reasons to allow both v1 and v2 of a package to **coexist** in a large program is to make it possible to upgrade the clients of that package one at a time and still have a buildable result. This is specific instance of the more general problem of **gradual code repair**.

A few highlights (15)

Automatic API updates via `go fix`.

> To be clear, this is not implemented today, but it seems promising, and this kind of tooling is made possible by giving different things different names. These examples demonstrate the power of having durable, immutable names attached to specific code behavior.

A few highlights (16)

More work for library authors.

> Semantic import versioning is more work for authors of packages. They can't just decide to issue v2, walk away from v1, and leave users [like Ugo] to deal with the fallout.

A few highlights (17)

> Let's commit to compatibility.

On minimum version selection (MVS) (part 4)

- https://research.swtch.com/vgo-mvs

> A versioned Go command must decide which module versions to use in each build. I call this list of modules and versions for use in a given build the **build list**.

> For stable development, today's build list must also be tomorrow's build list. But then developers must also be allowed to change the build list: to upgrade all modules, to upgrade one module, or to downgrade one module.

Greetings from New Jersey

> This post presents minimal version selection, a new, simple approach to the version selection problem. Minimal version selection is easy to understand and predict, which should make it easy to work with. It also produces high-fidelity builds, in which the dependencies a user builds are as close as possible to the ones the author developed against.

> **It is also efficient to implement, using nothing more complex than recursive graph traversals, so that a complete minimal version selection implementation in Go is only a few hundred lines of code.**

MVS and the import compatibility rule

> Minimal version selection assumes that each module declares its own dependency requirements: a list of minimum versions of other modules.

> Modules are assumed to follow the import compatibility rule—packages in any newer version should work as well as older ones—so a dependency requirement gives only a minimum version, never a maximum version or a list of incompatible later versions.

Current behaviour

> Go's current version selection algorithm is simplistic, providing two different version selection algorithms, neither of which is right.

- The first algorithm is the default behavior of go get: if you have a local version, use that one, or else download and use the latest version.
- The second algorithm is the behavior of go get -u: download and use the latest version of everything.

Algorithm

- recursive version
- graph traversal

Details at:

- https://research.swtch.com/vgo-mvs#algorithms

> An equivalent, more efficient construction is based on graph reachability.

In one sentence

> In contrast, minimal version selection prefers the minimum allowed version, which is the exact version requested by some go.mod in the project.

Helpers

Theory plus some helpers for the real world.

- exclusions
- replacements

No two files

> In contrast, most other systems, including Cargo and Dep, use the newest version available that meets requirements listed in a “manifest file.” The release of a new version changes their build decisions. To get reproducible builds, these systems add a second mechanism, the “lock file,” which lists the specific versions a build should use.

> I realized that both manifest and lock exist for the same purpose: to work around the “upgrade everything to latest” default behavior.

Reproducible builds (part 5)

- https://research.swtch.com/vgo-repro (Part 5)

> A reproducible build is one that, when repeated, produces the **same result**.

There is a movement around this, too, e.g. in Debian.

> A **verifiable build** is one that records enough information to be precise about exactly how to repeat it.
> A **verified build** is one that checks that it is using the expected source code.

Verifyable builds

How would you find out the version used to build a Go binary?

```
$ go get -u rsc.io/goversion
$ goversion /usr/bin | shuf -n 3
/usr/bin/containerd-shim go1.12.1
/usr/bin/yay go1.11.5
/usr/bin/gofmt go1.12.3
```

goversion tool and library versions

> Now that the vgo prototype understands module versions, it includes that information in the final binary too, and the new goversion -m flag prints it back out.

- https://research.swtch.com/vgo-repro#verifiable_builds

Example:

```
$ goversion -m goimports
```

An extra file

Was called `go.modverify`, now `go.sum`.

> The go.modverify file is a log of the hash of all versions ever encountered: lines are only added, never removed.

> The go.modverify feature helps detect **unexpected mismatches** between downloaded dependencies on different machines. It compares the hashes in go.modverify against hashes computed and saved at module download time.

Defining Go modules (part 6)

- https://research.swtch.com/vgo-module

> A Go module is a **collection of packages versioned as a unit**, along with a **go.mod** file listing other required modules.

> The move to modules is an opportunity for us to revisit and fix many details of how the go command manages source code. The current go get model will be about ten years old when we retire it in favor of modules. We need to make sure that the module design will serve us well for the next decade.

Use tags

> We want to encourage **more developers to tag releases of their packages**, instead of expecting that users will just pick a commit hash that looks good to them. Tagging explicit releases makes clear what is expected to be useful to others and what is still under development. At the same time, it must still be possible—although maybe not convenient—to request specific commits.

Abstracting away version control

> We want to move away from invoking version control tools such as bzr, fossil, git, hg, and svn to download source code. These fragment the ecosystem.

Community proxy

> We want to make it possible, at some future point, to introduce a shared proxy for use by the Go community, similar in spirit to those used by Rust, Node, and other languages. At the same time, the design must work well without assuming such a proxy or registry.

Move away from vendor

> We want to eliminate vendor directories. They were introduced for reproducibility and availability, but we now have better mechanisms. Reproducibility is handled by proper versioning, and availability is handled by caching proxies.

Tags and pseudo versions

Since not all code will be tagged, pseudo versions are possible.

The go.mod file

> A module version is defined by a tree of source files.

> The file format is line-oriented, with // comments only. Each line holds a single directive, which is a single verb (module, require, exclude, or replace, as defined by minimum version selection), followed by arguments

Example

```
module "my/thing"
require "other/thing" v1.0.2
require "new/thing" v2.3.4
exclude "old/thing" v1.2.3
replace "bad/thing" v1.4.5 => "good/thing" v1.4.5
```

Why not X?

> My goals for the file format were that it be (1) clear and simple, (2) easy for people to read, edit, manipulate, and diff, (3) easy for programs like vgo to read, modify, and write back, preserving comments and general structure, and (4) have room for limited future growth. I looked at JSON, TOML, XML, and YAML but none of them seemed to have those four properties all at once.

Tag requirements

> The leading v is required, and having three numbers is also required.

Versioned Go commands (part 7)

- https://research.swtch.com/vgo-cmd

> All commands (go build, go run, and so on) will download imported source code automatically, if the necessary version is not already present in the download cache on the local system.

Versioned Go commands

> The go list command will add access to module information.

> The all pattern is redefined to make sense in the world of modules.

> Developers can and will be encouraged to work in directories outside the GOPATH tree.

Automatic downloads

> The most significant implication of the isolation rule is that commands like go build, go install, and go test should download versioned dependencies as needed (that is, if not already downloaded and cached).

Demo

Resources

- Quick start
- Go tool docs: Module maintenance, Download modules to local cache, Edit go.mod from tools or scripts, ...
- Go modules design documents
- GopherCon SG 2018: Go with Versions
- Blog: Go Modules in 2019, Using Go Modules
- justforfunc: #42, #43