

Defining Go Modules

Go & Versioning, Part 6

Russ Cox

February 22, 2018

research.swtch.com/vgo-module

As introduced in the overview post, 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. In particular:

- 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.
- 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: packages developed using Bazaar or Fossil, for example, are effectively unavailable to users who cannot or choose not to install these tools. The version control tools have also been a source of exciting security problems. It would be good to move them outside the security perimeter.
- We want to allow multiple modules to be developed in a single source code repository but versioned independently. While most developers will likely keep working with one module per repo, larger projects might benefit from having multiple modules in a single repo. For example, we'd like to keep `golang.org/x/text` a single repository but be able to version experimental new packages separately from established packages.
- We want to make it easy for individuals and companies to put caching proxies in front of `go get` downloads, whether for availability (use a local copy to ensure the download works tomorrow) or security (vet packages before they can be used inside a company).
- 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.
- 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.

This post presents the parts of the `vgo` design that address these issues. Everything here is preliminary: we will change the design if we find that it is not right.

Versioned Releases

Abstraction boundaries let projects scale. Originally, all Go packages could be imported by all other Go packages. We introduced the internal directory convention in Go 1.4 to eliminate the problem that developers who chose to structure a program as multiple packages had to worry about other users importing and depending on details of helper packages never meant for public use.

The Go community has a similar visibility problem now with repository commits. Today, it's very common for users to identify package versions by commit identifiers (usually Git hashes), with the result that developers who structure work as a sequence of commits need to worry, at least in the back of their mind, about users pinning to any of those commits, which again were never meant for public use. We need to change the expectations in the Go open source community, to establish a norm that authors tag releases and users prefer those.

I don't think this point, that users should be choosing from versions issued by authors instead of picking out individual commits from the Git history, is particularly controversial. The difficult part is shifting the norm. We need to make it easy for authors to tag commits and easy for users to use those tags.

The most common way authors share code today is on code hosting sites, especially GitHub. For code on GitHub, all authors will need to do is tag a commit and push the tag. We also plan to provide a tool, maybe called `go release`, to compare different versions of a module for API compatibility at the type level, to catch inadvertent breaking changes that are visible in the type system, and also to help authors decide between issuing should be a minor release (because it adds new API or changes many lines of code) or only a patch release.

For users, `vgo` itself operates entirely in terms of tagged versions. However, we know that at least during the transition from old practices to new, and perhaps indefinitely as a way to bootstrap new projects, an escape hatch will be necessary, to allow specifying a commit. This is possible in `vgo`, but it has been designed so as to make users prefer explicitly tagged versions.

Specifically, `vgo` understands the special pseudo-version `v0.0.0-yyyymmddhhmmss-commit` as referring to the given commit identifier, which is typically a shortened Git hash and which must have a commit time matching the (UTC) timestamp. This form is a valid semantic version string for a prerelease of `v0.0.0`. For example, this pair of `Gopkg.toml` stanzas:

```
[[projects]]
  name = "google.golang.org/appengine"
  packages = [
    "internal",
    "internal/base",
    "internal/datastore",
    "internal/log",
    "internal/remote_api",
    "internal/urlfetch",
    "urlfetch"
  ]
  revision = "150dc57a1b433e64154302bdc40b6bb8aefa313a"
  version = "v1.0.0"

[[projects]]
  branch = "master"
  name = "github.com/google/go-github"
  packages = ["github"]
  revision = "922ceac0585d40f97d283d921f872fc50480e06e"
```

correspond to these `go.mod` lines:

```
require (
    "google.golang.org/appengine" v1.0.0
    "github.com/google/go-github" v0.0.0-20180116225909-922ceac0585d
)
```

The pseudo-version form is chosen so that the standard semver precedence rules compare two pseudo-versions by commit time, because the timestamp encoding makes string comparison match time comparison. The form also ensures that `vgo` will always prefer a tagged semantic version over an untagged pseudo-version, because even if `v0.0.1` is very old, it has a greater semver precedence than any `v0.0.0` prerelease. (Note also that this matches the choice made by `dep` when adding a new dependency to a project.) And of course pseudo-version strings are unwieldy: they stand out in `go.mod` files and `vgo list -m` output. All these inconveniences help encourage authors and users to prefer explicitly tagged versions, a bit like the extra step of having to write `import "unsafe"` encourages developers to prefer writing safe code.

The `go.mod` File

A module version is defined by a tree of source files. The `go.mod` file describes the module and also indicates the root directory. When `vgo` is run in a directory, it looks in the current directory and then successive parents to find the `go.mod` marking the root.

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:

```
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
```

The leading verb can be factored out of adjacent lines, leading to a block, like in Go imports:

```
require (
    "new/thing" v2.3.4
    "old/thing" v1.2.3
)
```

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. For example, the approach used in `Gopkg.toml` above leads to three lines for each requirement, making them harder to skim, sort, and diff. Instead I designed a minimal format reminiscent of the top of a Go program, but hopefully not close enough to be confusing. I adapted an existing comment-friendly parser.

The eventual `go` command integration may change the file format, perhaps even adopting a more standard framing, but for compatibility we will keep the ability to read today's `go.mod` files, just as `vgo` can also read requirement information from `GLOCKFILE`, `Godeps/Godeps.json`, `Gopkg.lock`, `dependencies.tsv`, `glide.lock`, `vendor.conf`, `vendor.yml`, `vendor/manifest`, and `vendor/vendor.json` files.

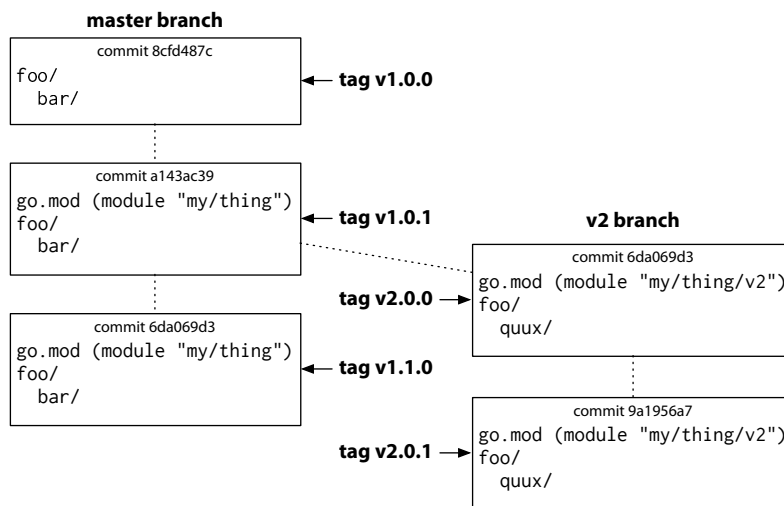
From Repository to Modules

Developers work in version control systems, and clearly vgo must make that as easy as possible. It is not reasonable to expect developers to prepare module archives themselves, for example. Instead, vgo makes it easy to export modules directly from any version control repository following some basic, unobtrusive conventions.

To start, it suffices to create a repository and tag a commit, using a semver-formatted tag like `v0.1.0`. The leading `v` is required, and having three numbers is also required. Although vgo itself accepts shorthands like `v0.1` on the command line, the canonical form `v0.1.0` must be used in repository tags, to avoid ambiguity. Only the tag is required. In order to use commits made without use of vgo, a `go.mod` file is not strictly required at this point. Creating new tagged commits creates new module versions. Easy.

When developers reach v2, semantic import versioning means that a `/v2/` is added to the import path at the end of the module root prefix: `my/thing/v2/sub/pkg`. There are good reasons for this convention, as described in the earlier post, but it is still a departure from existing tools. Realizing this, vgo will not use any v2 or later tag in a source code repository without first checking that it has a `go.mod` with a module path declaration ending in that major version (for example, module `"my/thing/v2"`). Vgo uses that declaration as evidence that the author is using semantic import versioning to name packages within that module. This is especially important for multi-package modules, since the import paths within the module must contain the `/v2/` element to avoid referring back to the v1 module.

We expect that most developers will prefer to follow the usual “major branch” convention, in which different major versions live in different branches. In this case, the root directory in a v2 branch would have a `go.mod` indicating v2, like this:

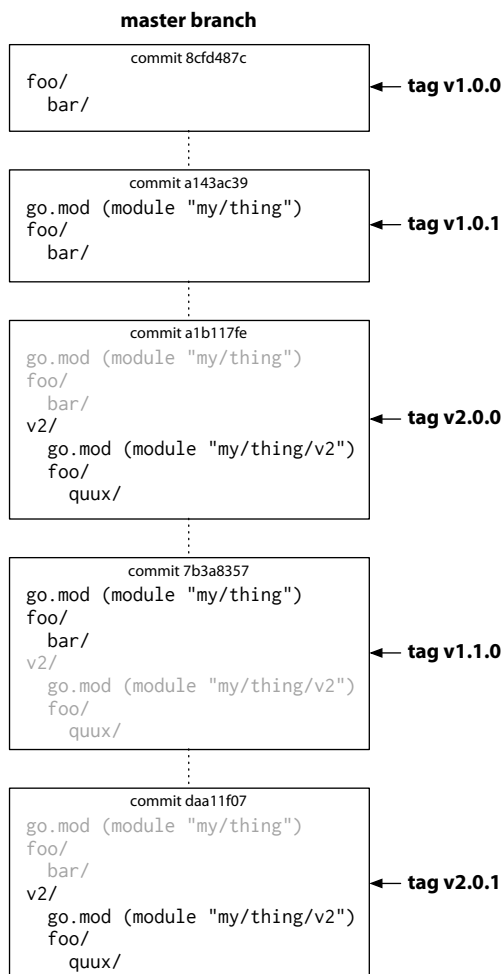


Go module using major branches

This is roughly how most developers already work. In the picture, the `v1.0.0` tag points to a commit that predates vgo. It has no `go.mod` file at all, and that works fine. In the commit tagged `v1.0.1`, the author has added a `go.mod` file that says module `"my/thing"`. After that commit, however, the author forks a new v2 development branch. In addition to whatever code changes prompted v2 (including the replacement of `bar` with `quux`), the `go.mod` in that new branch is updated to say module `"my/thing/v2"`. The branches can then proceed indepen-

dently. In truth, vgo really has no idea about branches. It just resolves the tag to a commit and then looks at the go.mod file in the commit. Again, the go.mod file is required for v2 and later so that vgo can use the module line as a sign that the code has been written with semantic import versioning in mind, so the imports in foo say my/thing/v2/foo/quux, not my/thing/foo/quux.

As an alternative, vgo also supports a “major subdirectory” convention, in which major versions above v1 are developed in subdirectories:



Go module using major subdirectories

In this case, v2.0.0 is created not by forking the whole tree into a separate branch but by copying it into a subdirectory. Again the go.mod must be updated to say "my/thing/v2". Afterward, v1.x.x tags pointing at commits address the files in the root directory, excluding v2/, while v2.x.x tags pointing at commits address the files in the v2/ subdirectory only. The go.mod file lets vgo distinguish the two cases. It would also be meaningful to have a v1.x.x and a v2.x.x tag pointing at the same commit: they would address different subtrees of the commit.

We expect that developers may feel strongly about choosing one convention or the other. Instead of taking sides, vgo supports both. Note that for major versions above v2, the major subdirectory approach may provide a more graceful transition for users of go get. On the other hand, users of dep or vendoring tools should be able to consume repositories using either convention. Certainly we will make sure dep can.

Multiple-Module Repositories

Developers may also find it useful to maintain a collection of modules in a single source code repository. We want vgo to support this possibility. In general, there is already wide variation in how different developers, teams, projects, and companies apply source control, and we do not believe it is productive to impose a single mapping like “one repository equals one module” onto all developers. Having some flexibility here should also help vgo adapt as best practices around source control continue to change.

In the major subdirectory convention, `v2/` contains the module `"my/thing/v2"`. A natural extension is to allow subdirectories not named for major versions. For example, we could add a `blue/` subdirectory that contains the module `"my/thing/blue"`, confirmed by a `blue/go.mod` file with that module path. In this case, the source control commit tags addressing that module would take the form `blue/v1.x.x`. Similarly, the tag `blue/v2.x.x` would address the `blue/v2/` subdirectory. The existence of the `blue/go.mod` file excludes the `blue/` tree from the outer `my/thing` module.

In the Go project, we intend to explore using this convention to allow repositories like `golang.org/x/text` to define multiple, independent modules. This lets us retain the convenience of coarse-grained source control but still promote different subtrees to v1 at different times.

Deprecated Versions

Authors also need to be able to deprecate a version, to indicate that it should not be used anymore. This is not yet implemented in the vgo prototype, but one way it could work would be to define that on code hosting sites, the existence of a tag `v1.0.0+deprecated` (ideally pointing at the same commit as `v1.0.0`) would indicate that the commit is deprecated. It is of course important not to remove the tag entirely, because that will break builds. Deprecated modules would be highlighted in some way in `vgo list -m -u` output (“show me my modules and information about updates”), so that users would know to update.

Also, because programs will have access to their own module lists and versions at runtime, a program could also be configured to check its own module versions against some chosen authority and self-report in some way when it is running deprecated versions. Again, the details here are not worked out, but it’s a good example of something that’s possible once developers and tools share a vocabulary for describing versions.

Publishing

Given a source control repository, developers need to be able to publish it in a form that vgo can consume. In the general case, we will provide a command that authors run to turn their source control repositories into file trees that can be served to vgo by any web server capable of serving static files. Similar to current `go get`, vgo expects a page with a `<meta>` tag to help translate from a module name to the tree of files for that module. For example, to look up `swtch.com/testmod`, the vgo command fetches the usual page:

```
$ curl -sSL 'https://swtch.com/testmod?go-get=1'
<!DOCTYPE html>
<meta name="go-import" content="swtch.com/testmod mod https://storage.googleapis.com/gomodels/rs...Nothing to
$
```

The `mod` server type indicates that modules are served in a file tree at that base URL. The relevant files at `storage.googleapis.com/gomodels/rsc` in this simple case are:

- .../swtch.com/testmod/@v/list
- .../swtch.com/testmod/@v/v1.0.0.info
- .../swtch.com/testmod/@v/v1.0.0.mod
- .../swtch.com/testmod/@v/v1.0.0.zip

The exact meaning of these URLs is discussed in the “Download Protocol” section later in the post.

Code Hosting Sites

A huge amount of development happens on code hosting sites, and we want that work to integrate into `vgo` as smoothly as possible. Instead of expecting developers to publish modules elsewhere, `vgo` will have support for reading the information it needs from those sites directly, using their HTTP-based APIs. In general, archive downloads can be significantly faster than the existing version control checkouts. For example, working on a laptop with a gigabit internet connection, it takes 10 seconds to download the CockroachDB source tree as a zip file from GitHub but almost four minutes to `git clone` it. Sites need only provide an archive of any form that can be fetched with a simple HTTP GET. Gerrit servers, for example, only support downloading gzipped tar files. `Vgo` translates downloaded archives into the standard form.

The initial prototype only includes support for GitHub and the Go project’s Gerrit server, but we will add support for Bitbucket and other major hosting sites too, before shipping anything in the main Go toolchain.

With the combination of the lightweight repository conventions, which mostly match what developers are already doing, and the support for known code hosting sites, we expect that most open source activity will be unaffected by the move to modules, other than simply adding a `go.mod` to each repository.

Companies taking advantage of old `go get`’s direct use of `git` and other source control tools will need to adjust. Perhaps it would make sense to write a proxy that serves the `vgo` expectations but using version control tools. Companies could then run one of those to produce an experience much like using the open source hosting sites.

Module Archives

The mapping from repositories to modules is a bit complex, because the way developers use source control varies. The end goal is to map all that complexity down into a common, single format for Go modules for use by proxies or other code consumers (for example, *godoc.org* or any code checking tools).

The standard format in the `vgo` prototype is zip archives in which all paths begin with the module path and version. For example, after running `vgo get` of `rsc.io/quote v1.5.2`, you can find the zip file in `vgo`’s download cache:

```
$ unzip -l $GOPATH/src/v/cache/rsc.io/quote/@v/v1.5.2.zip
 1479  00-00-1980  00:00   rsc.io/quote@v1.5.2/LICENSE
   131  00-00-1980  00:00   rsc.io/quote@v1.5.2/README.md
   240  00-00-1980  00:00   rsc.io/quote@v1.5.2/buggy/buggy_test.go
    55  00-00-1980  00:00   rsc.io/quote@v1.5.2/go.mod
   793  00-00-1980  00:00   rsc.io/quote@v1.5.2/quote.go
   917  00-00-1980  00:00   rsc.io/quote@v1.5.2/quote_test.go

$
```

I used zip because it is well-specified, widely supported, and cleanly extensible if needed, and it allows random access to individual files. (In contrast, tar files, the other obvious choice, are none of these things and don’t.)

Download Protocol

To download information about modules, as well as the modules themselves, the vgo prototype issues only simple HTTP GET requests. A key design goal was to make it possible to serve modules from static hosting sites, so the requests have no URL query parameters.

As we saw earlier, custom domains can specify that a module is hosted at a particular base URL. As implemented in vgo today (but, like all of vgo, subject to change), that module-hosting server must serve four request forms:

- GET *baseURL/module/@v/list* fetches a list of all known versions, one per line.
- GET *baseURL/module/@v/version.info* fetches JSON-formatted metadata about that version.
- GET *baseURL/module/@v/version.mod* fetches the go.mod file for that version.
- GET *baseURL/module/@v/version.zip* fetches the zip file for that version.

The JSON information served in the *version.info* form will likely evolve, but today it corresponds to this struct:

```
type RevInfo struct {
    Version string    // version string
    Name     string    // complete ID in underlying repository
    Short    string    // shortened ID, for use in pseudo-version
    Time     time.Time // commit time
}
```

The `vgo list -m -u` command shows the commit time of each available update by using the `Time` field.

A general module-hosting server may optionally respond to *version.info* requests for non-semver versions as well. A vgo command like

```
vgo get my/thing/v2@1459def
```

will fetch `1459def.info` and then derive a pseudo-version using the `Time` and `Short` fields.

There are two more optional request forms:

- GET *baseURL/module/@t/yyyymmddhhmmss* returns the `.info` JSON for the latest version at or before the given timestamp.
- GET *baseURL/module/@t/yyyymmddhhmmss/branch* does the same, but limiting the search to commits on a given branch.

These support the use of untagged commits in vgo. If vgo is adding a module and finds no tagged commits at all, it uses the first form to find the latest commit as of now. It does the same when looking for available updates, assuming there are still no tagged commits. The branch-limited form is used for the internal simulation of *gopkg.in*. These forms also support the command line syntaxes:

```
vgo get my/thing/v2@2018-02-01T15:34:45
vgo get my/thing/v2@2018-02-01T15:34:45@branch
```

These might be a mistake, but they're in the prototype today, so I'm mentioning them.

Proxy Servers

Both individuals and companies may prefer to download Go modules from proxy servers, whether for efficiency, availability, security, license compliance, or any other reason. Having a standard Go module format and a standard download protocol, as described in the last two sections, makes it trivial to introduce support for proxies. If the `$GOPROXY` environment variable is set, `vgo` fetches all modules from the server at the given base URL, not from their usual locations. For easy debugging, `$GOPROXY` can even be a `file:///` URL pointing at a local tree.

We intend to write a basic proxy server that serves from `vgo`'s own local cache, downloading new modules as needed. Sharing such a proxy among a set of computers would help reduce redundant downloads from the proxy's users but more importantly would ensure future availability, even if the original copies disappear. The proxy will also have an option not to allow downloads of new modules. In this mode, the proxy would limit the available modules to exactly those whitelisted by the proxy administrator. Both proxy modes are frequently requested features in corporate environments.

Perhaps some day it would make sense to establish a distributed collection of proxy servers used by default in `go get`, to ensure module availability and fast downloads for Go developers worldwide. But not yet. Today, we are focused on making sure that `go get` works as well as it can without assuming any kind of central proxy servers.

The End of Vendoring

Vendor directories serve two purposes. First, they specify by their contents the exact version of the dependencies to use during `go build`. Second, they ensure the availability of those dependencies, even if the original copies disappear. On the other hand, vendor directories are also difficult to manage and bloat the repositories in which they appear. With the `go.mod` file specifying the exact version of dependencies to use during `vgo build`, and with proxy servers for ensuring availability, vendor directories are now almost entirely redundant. They can, however, serve one final purpose: to enable a smooth transition to the new versioned world.

When building a module, `vgo` (and later `go`) will completely ignore vendored dependencies; those dependencies will also not be included in the module's zip file. To make it possible for authors to move to `vgo` and `go.mod` while still supporting users who haven't converted, the new `vgo vendor` command populates a module's vendor directory with the packages users need to reproduce the `vgo`-based build.

What's Next?

The details here may be revised, but today's `go.mod` files will be understood by any future tooling. Please start tagging your packages with release tags; add `go.mod` files if that makes sense for your project.

The next post in the series will cover changes to the `go` tool command line experience.