Go += Package Versioning

Go & Versioning, Part 1

Russ Cox

February 20, 2018

research.swtch.com/vgo-intro

We need to add package versioning to Go.

More precisely, we need to add the concept of package versions to the working vocabulary of both Go developers and our tools, so that they can all be precise when talking to each other about exactly which program should be built, run, or analyzed. The `go` command needs to be able to tell developers exactly which versions of which packages are in a particular build, and vice versa.

Versioning will let us enable reproducible builds, so that if I tell you to try the latest version of my program, I know you're going to get not just the latest version of my code but the exact same versions of all the packages my code depends on, so that you and I will build completely equivalent binaries.

Versioning will also let us ensure that a program builds exactly the same way tomorrow as it does today. Even when there are newer versions of my dependencies, the `go` command shouldn't start using them until asked.

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. If we get the model and the defaults right, we should be able to add versioning in such a way that programmers *still* mostly don't pay attention to versioning, and everything just works better and is easier to understand. Existing workflows should change as little as possible. Releasing new versions should be very easy. In general, version management work must fade to the background, not be a day-to-day concern.

In short, we need to add package versioning, but we need to do it without breaking `go get`. This post sketches a proposal for doing exactly that, along with a prototype demonstration that you can try today and that hopefully will be the basis for eventual `go` command integration. I intend this post to be the start of a productive discussion about what works and what doesn't. Based on that discussion, I will make adjustments to both the proposal and the prototype, and then I will submit an official Go proposal, for integration into Go 1.11 as an opt-in feature.

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. That said, this proposal is still also in its early stages. If details are not right yet, we will take the time to fix them before the work lands in the main Go distribution.

Background

Before we look at the proposal, let's look at how we got where we are today. This is maybe a little long, but the history has important lessons for the present and helps to understand why the proposal changes what it does. If you are impatient, feel free to skip ahead to the proposal, or read the accompanying example blog post.

Makefiles, `goinstall`, and `go get`

In November 2009, the initial release of Go was a compiler, linker, and some libraries. You had to run `6g` and `6l` to compile and link your programs, and we

included sample makefiles. There was a minimal wrapper `gobuild` that could build a single package and write an appropriate makefile, in most cases. There was no established way to share code with other people. We knew more was needed, but we released what we had, planning to do the rest with the community.

In February 2010, we proposed `goinstall`, a new, zero-configuration command for downloading packages from source control repositories like Bitbucket and GitHub. `Goinstall` introduced the import path conventions Go developers take for granted today. Because no code at the time followed those conventions, `goinstall` at first only worked with packages that imported nothing beyond the standard library. But developers quickly migrated from their own varied naming schemes to the uniform convention we know today, and the set of published Go packages grew into a coherent ecosystem.

`Goinstall` also eliminated makefiles and, with them, the complexity of user-defined build variations. While it is occasionally inconvenient to package authors not to be able to generate code during each build, that simplification has been incredibly important to package *users*: a user never has to worry about first installing the same set of tools as the package author used before building a package. The simplification has also been crucial to tooling. A makefile is an imperative, step-by-step recipe for compiling a package; reverse-engineering how to apply a different tool, like `go vet` or code completion, to the same package, can be quite difficult. Even getting build dependencies right, so that packages are rebuilt when necessary and only when necessary, is much harder with arbitrary makefiles. Although some people objected at the time that flexibility was being taken away, it is clear in retrospect that the benefits far outweighed the inconvenience.

In December 2011, as part of preparation for Go 1, we introduced the `go` command, which replaced `goinstall` with `go get`.

On the whole, `go get` has been transformative, enabling Go developers to share source code and build on each other's work, and enabling tooling by isolating details of the build system inside the `go` command. But `go get` is missing any concept of versioning. It was clear in the very first discussions of `goinstall` that we needed to do something about versioning. Unfortunately, it was not clear, at least to us on the Go team, exactly what to do. When `go get` needs a package, it always fetches the latest copy, delegating the download and update operations to version control systems like Git or Mercurial. This ignorance of package versioning has led to at least two significant shortcomings.

Versioning and API Stability

The first significant shortcoming of `go get` is that, without a concept of versioning, it cannot convey to users any expectations about what kinds of changes to expect in a given update.

In November 2013, Go 1.2 added a FAQ entry about package versioning that gave this basic advice (unchanged as of Go 1.10):

> *Packages intended for public use should try to maintain backwards compatibility as they evolve. The Go 1 compatibility guidelines are a good reference here: don't remove exported names, encourage tagged composite literals, and so on. If different functionality is required, add a new name instead of changing an old one. If a complete break is required, create a new package with a new import path.*

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 differ-

ent commits (perhaps on different branches) of a single Git repository. Following semantic versioning, authors are expected to introduce a new major version when making a breaking change, so that later versions of a `v1` import path can be expected to be drop-in replacements for earlier ones, while a `v2` import path may be a completely different API.

In August 2015, Dave Cheney filed a proposal to adopt semantic versioning. That prompted an interesting discussion over the next few months, in which everyone seemed to agree that tagging code with semantic versions seemed like a fine idea, but no one knew the next step: what should tools do with these versions?

Any discussion of semantic versioning inevitably includes counterarguments citing Hyrum's law, which states:

> With a sufficient number of users of an API, it does not matter what you promise in the contract. All observable behaviors of your system will be depended on by somebody.

While Hyrum's law is empirically true, semantic versioning is still a useful way to frame expectations about the relationships between releases. Updating from 1.2.3 to 1.2.4 should not break your code, while updating from 1.2.3 to 2.0.0 may. If your code stops working after an update to 1.2.4, the author is likely to welcome a bug report and issue a fix in 1.2.5. If your code stops working (or even compiling) after an update to 2.0.0, that change has a much greater chance of being intentional and a correspondingly lesser chance of being fixed to your code's liking in 2.0.1.

Instead of concluding from Hyrum's law that semantic versioning is impossible, I conclude that builds should be careful to use exactly the same versions of each dependency that the author did, unless forced to do otherwise. That is, builds should default to being as reproducible as possible.

Vendoring and Reproducible Builds

The second significant shortcoming of `go get` is that, without a concept of versioning, it cannot ensure or even express the idea of a reproducible build. There is no way to be sure that your users are compiling the same versions of your code's dependencies that you did. In November 2013, the Go 1.2 FAQ also added this basic advice:

> 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.) Store the copy under a new import path that identifies it as a local copy. For example, you might copy `"original.com/pkg"` to `"you.com/external/original.com/pkg"`. Keith Rarick's `goven` is one tool to help automate this process.

`Goven`, which Keith Rarick had started in March 2012, copied a dependency into your repository and also updated all the import paths within it to reflect the new location. Modifying the source code of the dependency in this way was necessary to make it build but was also unfortunate. The modifications made it harder to compare against and incorporate newer copies and required updates to other copied code using that dependency.

In September 2013, Keith announced `godep`, "a new tool for freezing package dependencies." The main advance in `godep` was to add what we now understand as Go vendoring—that is, to copy dependencies into the project *without* modifying the source files—without direct toolchain support, by setting up GOPATH in a certain way.

In October 2014, Keith proposed adding support for the concept of "external packages" to the Go toolchain, so that tools could better understand projects using that convention. By then, there were multiple efforts similar to `godep`. Matt Farina wrote a blog post, "Glide in the Sea of Go Package Managers," comparing `godep` with the newer arrivals, most notably `glide`.

In April 2015, Dave Cheney introduced `gb`, a "project-based build tool ... that permits repeatable builds via source vendoring," again without import rewriting. (Another motivation for `gb` was to avoid the requirement that code be stored in specific directories in GOPATH, which is not a good match for many developer workflows.)

That spring, Jason Buberel surveyed the Go package management landscape to understand what could be done to unify these multiple efforts and avoid duplication and wasted work. His survey made it clear to us on the Go team that the `go` command needed direct support for vendoring without import rewriting. At the same time, Daniel Theophanes started a specification for a file format to describe the exact provenance and version of code in a vendor directory. In June 2015, we accepted Keith's proposal as the Go 1.5 vendor experiment, optional in Go 1.5 and enabled by default in Go 1.6. We encouraged all vendoring tool authors to work with Daniel to adopt a single metadata file format.

Incorporating the concept of vendoring into the Go toolchain allowed program analysis tools like `go vet` to better understand projects using vendoring, and today there are a dozen or so Go package managers or vendoring tools that manage vendor directories. On the other hand, because these tools all use different metadata file formats, they do not interoperate and cannot easily share information about dependency requirements.

More fundamentally, vendoring is an incomplete solution to the package versioning problem. It only provides reproducible builds. It does nothing to help understand package versions and decide which version of a package to use. Package managers like `glide` and `dep` add the concept of versioning onto Go builds implicitly, without direct toolchain support, by setting up the vendor directory a certain way. As a result, the many tools in the Go ecosystem cannot be made properly aware of versions. It's clear that Go needs direct toolchain support for package versions.

An Official Package Management Experiment

At GopherCon 2016, a group of interested gophers got together on Hack Day (now Community Day) for a wide-ranging discussion of Go package management. One outcome was the formation of a committee and an advisory group for package management work, with a goal of creating a new tool for Go package management. The vision was for that tool to unify and replace the existing ones, but it would still be implemented outside the direct toolchain, using vendor directories. The committee—Andrew Gerrand, Ed Muller, Jessie Frazelle, and Sam Boyer, organized by Peter Bourgon—drafted a spec and then, led by Sam, implemented it as `dep`. For background, see Sam's February 2016 post "So you want to write a package manager," his December 2016 post "The Saga of Go Dependency Management," and his July 2017 GopherCon talk, "The New Era of Go Package Management."

`Dep` serves many purposes: it is an important improvement over current practice that's usable today, it is an important step toward a solution, and it is also an experiment—we call it an "official experiment"—that helps us learn more about what does and does not work well for Go developers. But `dep` is not a direct prototype of the eventual `go` command integration of package versioning. It is a powerful, almost arbitrarily flexible way to explore the design space, serving a role like makefiles did when we were grappling with how to build Go programs.

But once we understand the design space better and can narrow it down to the few key features that must be supported, it will help the Go ecosystem to remove the other features, to reduce expressiveness, to adopt enforced conventions that make Go code bases more uniform and easier to understand and make tooling easier to build.

This post is the beginning of the next step after dep: the first draft of a prototype of the final go command integration, the package management equivalent of goinstall. The prototype is a standalone command we call vgo. It is a drop-in replacement for the go command, but it adds support for package versioning. This is a new experiment, and we will see what we can learn from it. Like when we introduced goinstall, some code and projects already work with vgo today, and other projects will need changes to be made compatible. We will be taking away some control and expressiveness, just as we took away makefiles, in service of simplifying the system and eliminating complexity for users. Generally, we are looking for early adopters to help us experiment with vgo, so that we can learn as much as possible from users.

Starting to experiment with vgo does not mean ending support for dep. We will keep dep available until the path to full go command integration is decided, implemented, and generally available. We will also work to make the eventual transition from dep to the go command integration, in whatever form it takes, as smooth as possible. Projects that have not yet converted to dep can still reap real benefits from doing so. (Note that both godep and glide have ended active development and encourage migrating to dep.) Other projects may wish to move directly to vgo, if it serves their needs already.

Proposal

The proposal for adding versioning to the go command has four steps. First, adopt the *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. Second, use a simple, new algorithm, known as *minimal version selection*, to choose which package versions are used in a given build. Third, 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. Fourth, define how to retrofit all this into the existing go command, so that basic workflows do not change significantly from today. The rest of this section introduces each of these steps. Other blog posts this week will go into more detail.

The Import Compatibility Rule

Nearly all pain in package management systems is caused by trying to tame incompatibility. For example, most systems allow package B to declare that it requires package D 6 or later, and then allow package C to declare that it requires D 2, 3, or 4, but not 5 or later. If you are writing package A, and you want to use both B and C, then you are out of luck: there is no one single version of D that can be chosen to build both B and C into A. There is nothing you can do about it: these systems say that what B and C did was acceptable—they effectively encourage it—so you are just stuck.

Instead of designing a system that inevitably leads to large programs not building, this proposal requires that package authors follow 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 rule is a restatement of the suggestion from the Go FAQ, quoted earlier.

The quoted FAQ text ended by saying, "If a complete break is required, create a new package with a new import path." Developers today expect to use semantic versioning to express such a break, so we integrate semantic versioning into our proposal. Specifically, major version 2 and later can be used by including the version in the path, as in:

```
import "github.com/go-yaml/yaml/v2"
```

Creating v2.0.0, which in semantic versioning denotes a major break, therefore creates a new package with a new import path, as required by import compatibility. Because each major version has a different import path, a given Go executable might contain one of each major version. This is expected and desirable. It keeps programs building and allows parts of a very large program to update from v1 to v2 independently.

Expecting authors to follow the import compatibility rule lets us avoid trying to tame incompatibility, making the overall system exponentially simpler and the package ecosystem less fragmented. In practice, of course, despite the best efforts of authors, updates within the same major version do occasionally break users. Therefore, it's important to use an upgrade mechanism that doesn't upgrade too quickly. That brings us to the next step.

Minimal Version Selection

Nearly all package managers today, including `dep` and `cargo`, use the newest allowed version of packages involved in the build. I believe this is the wrong default, for two important reasons. First, the meaning of "newest allowed version" can change due to external events, namely new versions being published. Maybe tonight someone will introduce a new version of some dependency, and then tomorrow the same sequence of commands you ran today would produce a different result. Second, to override this default, developers spend their time telling the package manager "no, don't use X," and then the package manager spends its time searching for a way not to use X.

This proposal takes a different approach, which I call *minimal version selection*. It defaults to using the *oldest* allowed version of every package involved in the build. This decision does not change from today to tomorrow, because no older version will be published. Even better, to override this default, developers spend their time telling the package manager, "no, use at least Y," and then the package manager can trivially decide which version to use. I call this minimal version selection because the versions chosen are minimal and also because the system as a whole is perhaps also minimal, avoiding nearly all the complexity of existing systems.

Minimal version selection allows modules to specify only minimum requirements for their dependency modules. It gives well-defined, unique answers for both upgrade and downgrade operations, and those operations are efficient to implement. It also allows the author of the overall module being built to specify dependency versions to exclude, or to specify that a specific dependency version be replaced by a forked copy, either in the local file system or published as its own module. These exclusions and replacements do not apply when the module is being built as a dependency of some other module. This gives users full control over how their own programs build, but not over how other people's programs build.

Minimal version selection delivers reproducible builds by default, without a lock file.

Import compatibility is key to minimal version selection's simplicity. Instead of users saying "no, that's too new," they can only say "no, that's too old." In that case, the solution is clear: use a (minimally) newer version. And newer versions are agreed to be acceptable replacements for older ones.

Defining Go Modules

A Go *module* is a collection of packages sharing a common import path prefix, known as the module path. The module is the unit of versioning, and module versions are written as semantic version strings. When developing using Git, developers will define a new semantic version of a module by adding a tag to the module's Git repository. Although semantic versions are strongly preferred, referring to specific commits will be supported as well.

A module defines, in a new file called `go.mod`, the minimum version requirements of other modules it depends on. For example, here is a simple `go.mod` file:

```
// My hello, world.

module "rsc.io/hello"

require (
    "golang.org/x/text" v0.0.0-20180208041248-4e4a3210bb54
    "rsc.io/quote" v1.5.2
)
```

This file defines a module, identified by path `rsc.io/hello`, which itself depends on two other modules: `golang.org/x/text` and `rsc.io/quote`. A build of a module by itself will always use the specific versions of required dependencies listed in the `go.mod` file. As part of a larger build, it will only use a newer version if something else in the build requires it.

Authors will be expected to tag releases with semantic versions, and `vgo` encourages using tagged versions, not arbitrary commits. The `rsc.io/quote` module, served from `github.com/rsc/quote`, has tagged versions, including v1.5.2. The `golang.org/x/text` module, however, does not yet provide tagged versions. To name untagged commits, the pseudo-version v0.0.0-*yyyymmddhhmmss-commit* identifies a specific commit made on the given date. In semantic versioning, this string corresponds to a v0.0.0 prerelease, with prerelease identifier *yyyymmddhhmmss-commit*. Semantic versioning precedence rules order such prereleases before v0.0.0 or any later version, and they order prereleases by string comparison. Placing the date first in the pseudo-version syntax ensures that string comparison matches date comparison.

In addition to requirements, `go.mod` files can specify the exclusions and replacements mentioned in the previous section, but again those are only applied when building the module directly, not when building the module as part of a larger program. The examples illustrate all of these.

`Goinstall` and old `go get` invoke version control tools like `git` and `hg` directly to download code, leading to many problems, among them fragmentation: users without `bzr` cannot download code stored in Bazaar repositories, for example. In contrast, modules are always zip archives served over HTTP. Before, `go get` had special cases to choose the version control commands for popular code hosting sites. Now, `vgo` has special cases to use those hosting sites' APIs to fetch archives.

The uniform representation of modules as zip archives makes possible a trivial protocol for and implementation of a module-downloading proxy. Companies or individuals can run proxies for any number of reasons, including security and wanting to be able to work from cached copies in case the originals are removed. With proxies available to ensure availability and `go.mod` to define which code to use, `vendor` directories are no longer needed.

The go command

The go command must be updated to work with modules. One significant change is that ordinary build commands, like `go build`, `go install`, `go run`, and `go test`, will resolve new dependencies on demand, All it takes to use `golang.org/x/text` in a brand new module is to add an import to the Go source code and build the code.

The most significant change, though, is the end of GOPATH as a required place to work on Go code. Because the `go.mod` file includes the full module path and also defines the version of every dependency in use, a directory with a `go.mod` file marks the root of a directory tree that serves as a self-contained work space, separate from any other such directories. Now you just `git clone`, `cd`, and start writing. Anywhere. No GOPATH required.

What's Next?

I've also posted "A Tour of Versioned Go," showing what it's like to use `vgo`. See that post for how to download and experiment with `vgo` today. I'll post more throughout the week to add details that I skipped in this post. I encourage feedback in the comments on this post and the others, and I'll try to watch the Go subreddit and the golang-nuts mailing list too. On Friday I will post a FAQ as the final blog post in the series (at least for now). Next week I will submit a formal Go proposal.

Please try `vgo`. Start tagging versions in your repositories. Create and check in `go.mod` files. Note that if run in a repository that has an empty `go.mod` but that does have an existing `dep`, `glide`, `glock`, `godep`, `godeps`, `govend`, `govendor`, or `gvt` configuration file, `vgo` will use that to fill in the `go.mod` file.

I'm excited for Go to take the long-overdue step of adding versions to its working vocabulary. Some of the most common problems that developers run into when using Go are the lack of reproducible builds, `go get` ignoring release tags entirely, the inability of GOPATH to comprehend multiple versions of a package, and wanting or needing to work in source directories outside GOPATH. The design proposed here eliminates all these problems, and more.

Even so, I'm sure there are details that are wrong. I hope our users will help us get this design right by trying the new `vgo` prototype and engaging in productive discussions. I would like Go 1.11 to ship with preliminary support for Go modules, as a kind of technology preview, and then I'd like Go 1.12 to ship with official support. In some later release, we'll remove support for the old, unversioned `go get`. That's an aggressive schedule, though, and if getting the functionality right means waiting for later releases, we will.

I care very much about the transition from old `go get` and the myriad vendoring tools to the new module system. That process is just as important to me as getting the functionality right. If a successful transition means waiting for later releases, we will.

Thanks to Peter Bourgon, Jess Frazelle, Andrew Gerrand, and Ed Mueller, and Sam Boyer for their work on the package management committee and for many helpful discussions over the past year. Thanks also to Dave Cheney, Gustavo Niemeyer, Keith Rarick, and Daniel Theophanes for key contributions to the story of Go and package versioning. Thanks again to Sam Boyer for creating `dep`, and to him and the `dep` contributors for all their work on it. Thanks to everyone who has created or worked on the many earlier vendoring tools as well. Finally, thanks to everyone who will help us move this proposal forward, find and fix what's wrong, and add package versioning to Go as smoothly as possible.