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.

A Tour of Versioned Go (vgo)

Go & Versioning, Part 2

Russ Cox

February 20, 2018

research.swtch.com/vgo-tour

For me, design means building, tearing down, and building again, over and over. To write the new versioning proposal, I built an prototype, vgo, to work through many subtle details. This post shows what it's like to use vgo.

You can download and try vgo today by running `go get golang.org/x/vgo`. Vgo is a drop-in replacement for (and a forked copy of) the go command. You run vgo instead of go, and then it uses the standard compiler and libraries you already have installed in `$GOROOT` (Go 1.10beta1 or later).

The details of vgo's semantics and command lines are likely to change as we learn more about what works and what does not. However, we intend to avoid backwards-incompatible changes to the `go.mod` file format, so that a `go.mod` added to a project today will keep working in the future. As we refine the proposal, we'll update vgo accordingly.

Examples

This section demonstrates what it's like to use vgo. Please follow along and experiment with variations as you do.

Start by installing vgo:

```
$ go get -u golang.org/x/vgo
```

You are certain to run into interesting bugs, since vgo is at best only lightly tested right now. To file issues, please use the main Go issue tracker and add the prefix "x/vgo:" to the title. Thanks.

Hello, world

Let's write an interesting "hello, world" program. Create a directory outside your GOPATH/src tree and change into it:

```
$ cd $HOME
$ mkdir hello
$ cd hello
```

Then create a file `hello.go`:

```
package main // import "github.com/you/hello"

import (
    "fmt"
    "rsc.io/quote"
)

func main() {
    fmt.Println(quote.Hello())
}
```

Or download it:

```
$ curl -sS https://swtch.com/hello.go >hello.go
```

Create an empty go.mod file to mark the root of this module, and then build and run your new program:

```
$ echo >go.mod
$ vgo build
vgo: resolving import "rsc.io/quote"
vgo: finding rsc.io/quote (latest)
vgo: adding rsc.io/quote v1.5.2
vgo: finding rsc.io/quote v1.5.2
vgo: finding rsc.io/sampler v1.3.0
vgo: finding golang.org/x/text v0.0.0-20170915032832-14c0d48ead0c
vgo: downloading rsc.io/quote v1.5.2
vgo: downloading rsc.io/sampler v1.3.0
vgo: downloading golang.org/x/text v0.0.0-20170915032832-14c0d48ead0c
$ ./hello
Hello, world.
$
```

Notice that there is no explicit vgo get required. Plain vgo build will, upon encountering an unknown import, look up the module that contains it and add the latest version of that module as a requirement to the current module.

A side effect of running any vgo command is to update go.mod if necessary. In this case, the vgo build wrote a new go.mod:

```
$ cat go.mod
module github.com/you/hello

require rsc.io/quote v1.5.2
$
```

Because the go.mod was written, the next vgo build will not resolve the import again or print nearly as much:

```
$ vgo build
$ ./hello
Hello, world.
$
```

Even if rsc.io/quote v1.5.3 or v1.6.0 is released tomorrow, builds in this directory will keep using v1.5.2 until an explicit upgrade (see below).

The go.mod file lists a minimal set of requirements, omitting those implied by the ones already listed. In this case, rsc.io/quote v1.5.2 requires the specific versions of rsc.io/sampler and golang.org/x/text that were reported, so it would be redundant to repeat those in the go.mod file.

It is still possible to find out the full set of modules required by a build, using vgo list -m:

```
$ vgo list -m
MODULE                VERSION
github.com/you/hello  -
golang.org/x/text     v0.0.0-20170915032832-14c0d48ead0c
rsc.io/quote          v1.5.2
rsc.io/sampler        v1.3.0
$
```

At this point you might wonder why our simple "hello world" program uses golang.org/x/text. It turns out that rsc.io/quote depends on rsc.io/sampler, which in turn uses golang.org/x/text for language matching.

```
$ LANG=fr ./hello
Bonjour le monde.
$
```

Upgrading

We've seen that when a new module must be added to a build to resolve a new import, vgo takes the latest one. Earlier, it needed rsc.io/quote and found that v1.5.2 was the latest. But except when resolving new imports, vgo uses only versions listed in go.mod files. In our example, rsc.io/quote depended indirectly on specific versions of golang.org/x/text and rsc.io/sampler. It turns out that both of those packages have newer releases, as we can see by adding -u (check for updated packages) to the vgo list command:

```
$ vgo list -m -u
MODULE                VERSION                                 LATEST
github.com/you/hello  -                                       -
golang.org/x/text     v0.0.0-20170915032832-14c0d48ead0c  v0.3.0 (2017-12-14 08:08)
rsc.io/quote          v1.5.2 (2018-02-14 10:44)               -
rsc.io/sampler        v1.3.0 (2018-02-13 14:05)           v1.99.99 (2018-02-13 17:20)
$
```

Both of those packages have newer releases, so we might want to upgrade them in our hello program.

Let's upgrade golang.org/x/text first:

```
$ vgo get golang.org/x/text
vgo: finding golang.org/x/text v0.3.0
vgo: downloading golang.org/x/text v0.3.0
$ cat go.mod
module github.com/you/hello

require (
    golang.org/x/text v0.3.0
    rsc.io/quote v1.5.2
)
$
```

The vgo get command looks up the latest version of the given modules and adds that version as a requirement for the current module, by updating go.mod. Now future builds will use the newer text module:

```
$ vgo list -m
MODULE                VERSION
github.com/you/hello  -
golang.org/x/text     v0.3.0
rsc.io/quote          v1.5.2
rsc.io/sampler        v1.3.0
$
```

Of course, after an upgrade, it's a good idea to test that everything still works. Our dependencies `rsc.io/quote` and `rsc.io/sampler` have not been tested with the newer text module. We can run their tests in the configuration we've created:

```
$ vgo test all
?       github.com/you/hello    [no test files]
?       golang.org/x/text/internal/gen  [no test files]
ok      golang.org/x/text/internal/tag  0.020s
?       golang.org/x/text/internal/testtext [no test files]
ok      golang.org/x/text/internal/ucd  0.020s
ok      golang.org/x/text/language  0.068s
ok      golang.org/x/text/unicode/cldr  0.063s
ok      rsc.io/quote    0.015s
ok      rsc.io/sampler  0.016s
$
```

In the original go command, the package pattern `all` meant all packages found in GOPATH. That's almost always too many to be useful. In vgo, we've narrowed the meaning of `all` to be "all packages in the current module, and the packages they import, recursively." Version 1.5.2 of the `rsc.io/quote` module contains a buggy package:

```
$ vgo test rsc.io/quote/...
ok      rsc.io/quote    (cached)
--- FAIL: Test (0.00s)
    buggy_test.go:10: buggy!
FAIL
FAIL    rsc.io/quote/buggy  0.014s
(exit status 1)
$
```

Until something in our module imports buggy, however, it's irrelevant to us, so it's not included in `all`. In any event, the upgraded x/text seems to work. At this point we'd probably commit `go.mod`.

Another option is to upgrade all modules needed by the build, using `vgo get -u`:

```
$ vgo get -u
vgo: finding golang.org/x/text latest
vgo: finding rsc.io/quote latest
vgo: finding rsc.io/sampler latest
vgo: finding rsc.io/sampler v1.99.99
vgo: finding golang.org/x/text latest
vgo: downloading rsc.io/sampler v1.99.99
$ cat go.mod
module github.com/you/hello

require (
    golang.org/x/text v0.3.0
    rsc.io/quote v1.5.2
    rsc.io/sampler v1.99.99
)
$
```

Here, `vgo get -u` has kept the upgraded text module and also upgraded `rsc.io/sampler` to its latest version, v1.99.99.

Let's run our tests:

```
$ vgo test all
?       github.com/you/hello    [no test files]
?       golang.org/x/text/internal/gen  [no test files]
ok      golang.org/x/text/internal/tag  (cached)
?       golang.org/x/text/internal/testtext [no test files]
ok      golang.org/x/text/internal/ucd  (cached)
ok      golang.org/x/text/language  0.070s
ok      golang.org/x/text/unicode/cldr  (cached)
--- FAIL: TestHello (0.00s)
    quote_test.go:19: Hello() = "99 bottles of beer on the wall, 99 bottles of beer, ...", want "...
FAIL
FAIL    rsc.io/quote    0.014s
--- FAIL: TestHello (0.00s)
    hello_test.go:31: Hello([en-US fr]) = "99 bottles of beer on the wall, 99 bottles of beer, .....
    hello_test.go:31: Hello([fr en-US]) = "99 bottles of beer on the wall, 99 bottles of beer, .....
FAIL
FAIL    rsc.io/sampler  0.014s
(exit status 1)
$
```

It appears that something is wrong with `rsc.io/sampler` v1.99.99. Sure enough:

```
$ vgo build
$ ./hello
99 bottles of beer on the wall, 99 bottles of beer, ...
$
```

The `vgo get -u` behavior of taking the latest of every dependency is exactly what `go get` does when packages being downloaded aren't in GOPATH. On a system with nothing in GOPATH:

```
$ go get -d rsc.io/hello
$ go build -o badhello rsc.io/hello
$ ./badhello
99 bottles of beer on the wall, 99 bottles of beer, ...
$
```

The important difference is that `vgo` *does not behave this way by default*. Also you can undo it by downgrading.

Downgrading

To downgrade a package, use `vgo list -t` to show the available tagged versions:

```
$ vgo list -t rsc.io/sampler
rsc.io/sampler
    v1.0.0
    v1.2.0
    v1.2.1
    v1.3.0
    v1.3.1
    v1.99.99
$
```

Then use vgo get to ask for a specific version, like maybe v1.3.1:

```
$ cat go.mod
module github.com/you/hello

require (
    golang.org/x/text v0.3.0
    rsc.io/quote v1.5.2
    rsc.io/sampler v1.99.99
)
$ vgo get rsc.io/sampler@v1.3.1
vgo: finding rsc.io/sampler v1.3.1
vgo: downloading rsc.io/sampler v1.3.1
$ vgo list -m
MODULE                VERSION
github.com/you/hello  -
golang.org/x/text     v0.3.0
rsc.io/quote          v1.5.2
rsc.io/sampler        v1.3.1
$ cat go.mod
module github.com/you/hello

require (
    golang.org/x/text v0.3.0
    rsc.io/quote v1.5.2
    rsc.io/sampler v1.3.1
)
$ vgo test all
?       github.com/you/hello    [no test files]
?       golang.org/x/text/internal/gen  [no test files]
ok      golang.org/x/text/internal/tag  (cached)
?       golang.org/x/text/internal/testtext [no test files]
ok      golang.org/x/text/internal/ucd  (cached)
ok      golang.org/x/text/language  (cached)
ok      golang.org/x/text/unicode/cldr  (cached)
ok      rsc.io/quote    0.016s
ok      rsc.io/sampler  0.015s
$
```

Downgrading one package may require downgrading others. For example:

```
$ vgo get rsc.io/sampler@v1.2.0
vgo: finding rsc.io/sampler v1.2.0
vgo: finding rsc.io/quote v1.5.1
vgo: finding rsc.io/quote v1.5.0
vgo: finding rsc.io/quote v1.4.0
vgo: finding rsc.io/sampler v1.0.0
vgo: downloading rsc.io/sampler v1.2.0
$ vgo list -m
MODULE                VERSION
github.com/you/hello  -
golang.org/x/text     v0.3.0
rsc.io/quote          v1.4.0
rsc.io/sampler        v1.2.0
$ cat go.mod
module github.com/you/hello
```

```
require (
    golang.org/x/text v0.3.0
    rsc.io/quote v1.4.0
    rsc.io/sampler v1.2.0
)
$
```

In this case, rsc.io/quote v1.5.0 was the first to require rsc.io/sampler v1.3.0; earlier versions only needed v1.0.0 (or later). The downgrade selected rsc.io/quote v1.4.0, the last version compatible with v1.2.0.

It is also possible to remove a dependency entirely, an extreme form of downgrade, by specifying none as the version.

```
$ vgo get rsc.io/sampler@none
vgo: downloading rsc.io/quote v1.4.0
vgo: finding rsc.io/quote v1.3.0
$ vgo list -m
MODULE                VERSION
github.com/you/hello  -
golang.org/x/text     v0.3.0
rsc.io/quote          v1.3.0
$ cat go.mod
module github.com/you/hello

require (
    golang.org/x/text v0.3.0
    rsc.io/quote v1.3.0
)
$ vgo test all
vgo: downloading rsc.io/quote v1.3.0
?       github.com/you/hello    [no test files]
ok      rsc.io/quote    0.014s
$
```

Let's go back to the state where everything is the latest version, including rsc.io/sampler v1.99.99:

```
$ vgo get -u
vgo: finding golang.org/x/text latest
vgo: finding rsc.io/quote latest
vgo: finding rsc.io/sampler latest
vgo: finding golang.org/x/text latest
$ vgo list -m
MODULE                VERSION
github.com/you/hello  -
golang.org/x/text     v0.3.0
rsc.io/quote          v1.5.2
rsc.io/sampler        v1.99.99
$
```

Excluding

Having identified that v1.99.99 isn't okay to use in our hello world program, we may want to record that fact, to avoid future problems. We can do that by adding an exclude directive to go.mod:

```
exclude rsc.io/sampler v1.99.99
```

Future operations behave as if that module does not exist:

```
$ echo 'exclude rsc.io/sampler v1.99.99' >>go.mod
$ vgo list -t rsc.io/sampler
rsc.io/sampler
    v1.0.0
    v1.2.0
    v1.2.1
    v1.3.0
    v1.3.1
    v1.99.99 # excluded
$ vgo get -u
vgo: finding golang.org/x/text latest
vgo: finding rsc.io/quote latest
vgo: finding rsc.io/sampler latest
vgo: finding rsc.io/sampler latest
vgo: finding golang.org/x/text latest
$ vgo list -m
MODULE                VERSION
github.com/you/hello  -
golang.org/x/text     v0.3.0
rsc.io/quote          v1.5.2
rsc.io/sampler        v1.3.1
$ cat go.mod
module github.com/you/hello

require (
    golang.org/x/text v0.3.0
    rsc.io/quote v1.5.2
    rsc.io/sampler v1.3.1
)

exclude "rsc.io/sampler" v1.99.99
$ vgo test all
?       github.com/you/hello    [no test files]
?       golang.org/x/text/internal/gen  [no test files]
ok      golang.org/x/text/internal/tag  (cached)
?       golang.org/x/text/internal/testtext [no test files]
ok      golang.org/x/text/internal/ucd  (cached)
ok      golang.org/x/text/language  (cached)
ok      golang.org/x/text/unicode/cldr  (cached)
ok      rsc.io/quote    (cached)
ok      rsc.io/sampler  (cached)
$
```

Exclusions only apply to builds of the current module. If the current module were required by a larger build, the exclusions would not apply.= For example, an exclusion in `rsc.io/quote`'s `go.mod` will not apply to our "hello, world" build. This policy balances giving the authors of the current module almost arbitrary control over their own build, without also subjecting them to almost arbitrary control exerted by the modules they depend on.

At this point, the right next step is to contact the author of `rsc.io/sampler` and report the problem in v1.99.99, so it can be fixed in v1.99.100. Unfortunately, the author has a blog post that depends on not fixing the bug.

Replacing

If you do identify a problem in a dependency, you need a way to replace it with a fixed copy temporarily. Suppose we want to change something about the behavior of `rsc.io/quote`. Perhaps we want to work around the problem in `rsc.io/sampler`, or perhaps we want to do something else. The first step is to check out the `quote` module, using an ordinary `git` command:

```
$ git clone https://github.com/rsc/quote ../quote
Cloning into '../quote'...
```

Then edit `../quote/quote.go` to change something about `func Hello`. For example, I'm going to change its return value from `sampler.Hello()` to `sampler.Glass()`, a more interesting greeting.

```
$ cd ../quote
$ <edit quote.go>
$
```

Having changed the fork, we can make our build use it in place of the real one by adding a replacement directive to `go.mod`:

```
replace rsc.io/quote v1.5.2 => ../quote
```

Then we can build our program using it:

```
$ cd ../hello
$ echo 'replace rsc.io/quote v1.5.2 => ../quote' >>go.mod
$ vgo list -m
MODULE                 VERSION
github.com/you/hello   -
golang.org/x/text      v0.3.0
rsc.io/quote           v1.5.2
 => ../quote
rsc.io/sampler         v1.3.1
$ vgo build
$ ./hello
I can eat glass and it doesn't hurt me.
$
```

You can also name a different module as a replacement. For example, you can fork `github.com/rsc/quote` and then push your change to your fork.

```
$ cd ../quote
$ git commit -a -m 'my fork'
[master 6151719] my fork
 1 file changed, 1 insertion(+), 1 deletion(-)
$ git tag v0.0.0-myfork
$ git push https://github.com/you/quote v0.0.0-myfork
To https://github.com/you/quote
 * [new tag]         v0.0.0-myfork -> v0.0.0-myfork
$
```

Then you can use that as the replacement:

```
$ cd ../hello
$ echo 'replace rsc.io/quote v1.5.2 => github.com/you/quote v0.0.0-myfork' >>go.mod
$ vgo list -m
vgo: finding github.com/you/quote v0.0.0-myfork
MODULE                    VERSION
github.com/you/hello      -
golang.org/x/text         v0.3.0
rsc.io/quote              v1.5.2
 => github.com/you/quote  v0.0.0-myfork
rsc.io/sampler            v1.3.1
$ vgo build
vgo: downloading github.com/you/quote v0.0.0-myfork
$ LANG=fr ./hello
Je peux manger du verre, ça ne me fait pas mal.
$
```

Backwards Compatibility

Even if you want to use vgo for your project, you probably don't want to require all your users to have vgo. Instead, you can create a vendor directory that allows go command users to produce nearly the same builds (building inside GOPATH, of course):

```
$ vgo vendor
$ mkdir -p $GOPATH/src/github.com/you
$ cp -a . $GOPATH/src/github.com/you/hello
$ go build -o vhello github.com/you/hello
$ LANG=es ./vhello
Puedo comer vidrio, no me hace daño.
$
```

I said the builds are "nearly the same," because the import paths seen by the toolchain and recorded in the final binary are different. The vendored builds see vendor directories:

```
$ go tool nm hello | grep sampler.hello
 1170908 B rsc.io/sampler.hello
$ go tool nm vhello | grep sampler.hello
 11718e8 B github.com/you/hello/vendor/rsc.io/sampler.hello
$
```

Except for this difference, the builds should produce the same binaries. In order to provide for a graceful transition, vgo-based builds ignore vendor directories entirely, as will module-aware go command builds.

What's Next?

Please try vgo. Start tagging versions in your repositories. Create and check in go.mod files. File issues at golang.org/issue, and please include "x/vgo:" at the start of the title. More posts tomorrow. Thanks, and have fun!

Semantic Import Versioning

Go & Versioning, Part 3

Russ Cox

February 21, 2018

research.swtch.com/vgo-import

How do you deploy an incompatible change to an existing package? This is the fundamental challenge, the fundamental decision, in any package management system. The answer decides the complexity of the resulting system It decides how easy or difficult package management will be to use. (It also decides how easy or difficult package management will be to implement, but the user experience is more important.)

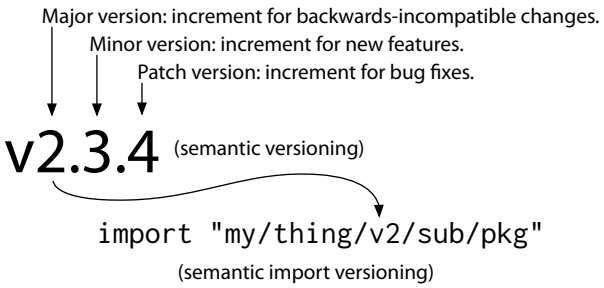
To answer this question, this post first presents the *import compatibility rule* for Go:

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

We've argued for this principle from the start of Go, but we haven't given it a name or such a direct statement.

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.

Developers today expect to use semantic versions to describe packages, so we adopt them into the model. Specifically, a module `my/thing` is imported as `my/thing` for v0, the incompatibility period when breaking changes are expected and not protected against, and then also during v1, the first stable major version. But when it's time to add v2, instead of redefining the meaning of the now-stable `my/thing`, we give it a new name: `my/thing/v2`.



I call this convention *semantic import versioning*, the result of following the import compatibility rule while using semantic versioning.

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. In this post I hope to give you a sense of why I changed my mind.

A Dependency Story

To make the discussion concrete, consider the following story. The story is hypothetical, of course, but it's motivated by a real problem. When dep was released, the team at Google that wrote the OAuth2 package asked me how they should go about introducing some incompatible improvements they've wanted to do for a long time. The more I thought about it, the more I realized that this was not

as easy as it sounded, at least not without semantic import versioning.

Prologue

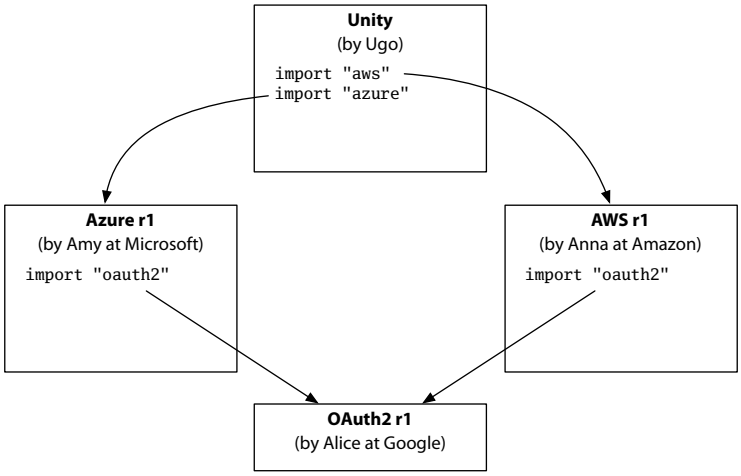
From the perspective of a package management tool, there are Authors of code and Users of code. Alice, Anna, and Amy are Authors of different code packages. Alice works at Google and wrote the OAuth2 package. Amy works at Microsoft and wrote the Azure client libraries. Anna works at Amazon and wrote the AWS client libraries. Ugo is the User of all these packages. He's working on the ultimate cloud app, Unity, which uses all of those packages and others.

As the Authors, Alice, Anna, and Amy need to be able to write and release new versions of their packages. Each version of a package specifies a required version for each of its dependencies.

As the User, Ugo needs to be able to build Unity with these other packages; he needs control over exactly which versions are used in a particular build; and he needs to be able to update to new versions when he chooses.

There's more that our friends might expect from a package management tool, especially around discovery, testing, portability, and helpful diagnostics, of course, but those are not relevant to the story.

As our story opens, Ugo's Unity build dependencies look like:



Chapter 1

Everyone is writing software independently.

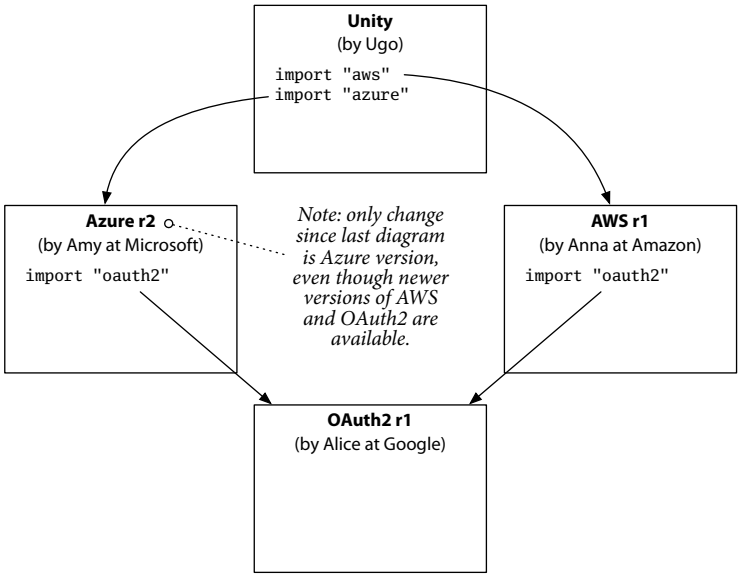
At Google, Alice has been busy designing a new, simpler, easier-to-use API for the OAuth2 package. It can still do everything that the old package can do, but with half the API surface. She releases it as OAuth2 r2. (The 'r' here stands for revision. For now, the revision numbers don't indicate anything other than sequencing: in particular, they're not semantic versions.)

At Microsoft, Amy is on a well-deserved long vacation, and her team decides not to make any changes related to OAuth2 r2 until she returns. The Azure package will keep using OAuth2 r1 for now.

At Amazon, Anna finds that using OAuth2 r2 will let her delete a lot of ugly code from the implementation of AWS r1, so she changes AWS to use OAuth2 r2. She fixes a few bugs along the way and issues the result as AWS r2.

Ugo gets a bug report about behavior on Azure and tracks it down to a bug in the Azure client libraries. Amy already released a fix for that bug in Azure r2 before leaving for vacation. Ugo adds a test case to Unity, confirms that it fails, and asks the package management tool to update to Azure r2.

After the update, Ugo's build looks like:



He confirms that the new test passes and that all his old tests still pass. He locks in the Azure update and ships an updated Unity.

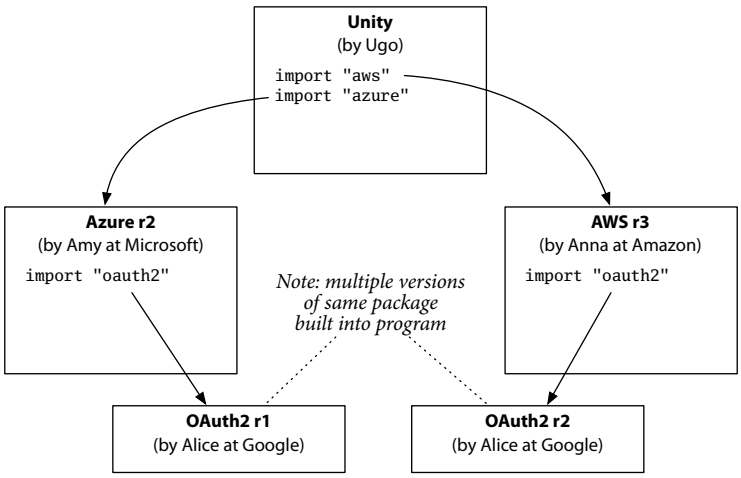
Chapter 2

To much fanfare, Amazon launches their new cloud offering, Amazon Zeta Functions. In preparation for the launch, Anna added Zeta support to the AWS package, which she now releases as AWS r3.

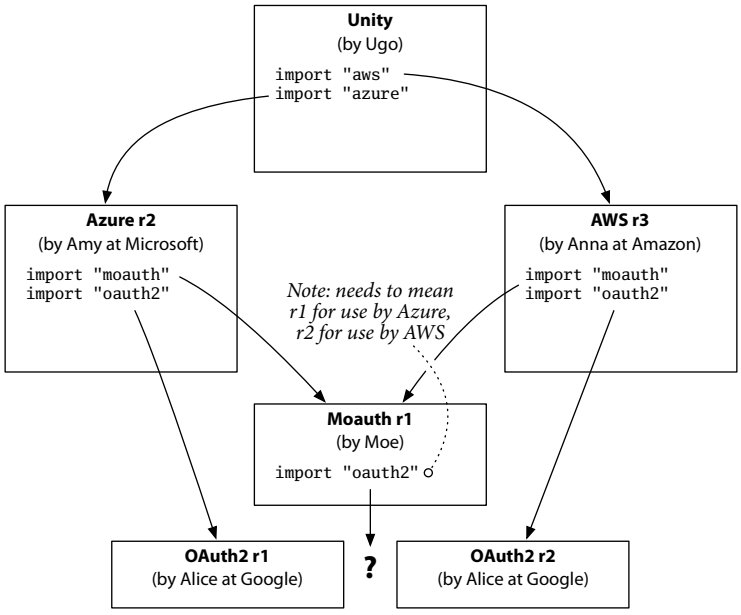
When Ugo hears about Amazon Zeta, he writes some test programs and is so excited about how well they work that he skips lunch to update Unity. Today's update does not go as well as the last one. Ugo wants to build Unity with Zeta support using Azure r2 and AWS r3, the latest version of each. But Azure r2 needs OAuth2 r1 (not r2), while AWS r3 needs OAuth2 r2 (not r1). Classic diamond dependency, right? Ugo doesn't care what it is. He just wants to build Unity.

Worse, it doesn't appear to be anyone's fault. Alice wrote a better OAuth2 package. Amy fixed some Azure bugs and went on vacation. Anna decided AWS should use the new OAuth2 (an internal implementation detail) and later added Zeta support. Ugo wants Unity to use the latest Azure and AWS packages. It's very hard to say any of them did something wrong. If these people aren't wrong, then maybe the package manager is. We've been assuming that there can be only one version of OAuth2 in Ugo's Unity build. Maybe that's the problem: maybe the package manager should allow different versions to be included in a single build. This example would seem to indicate that it must.

Ugo is still stuck, so he searches StackOverflow and finds out about the package manager's `-fmultiverse` flag, which allows multiple versions, so that his program builds as:

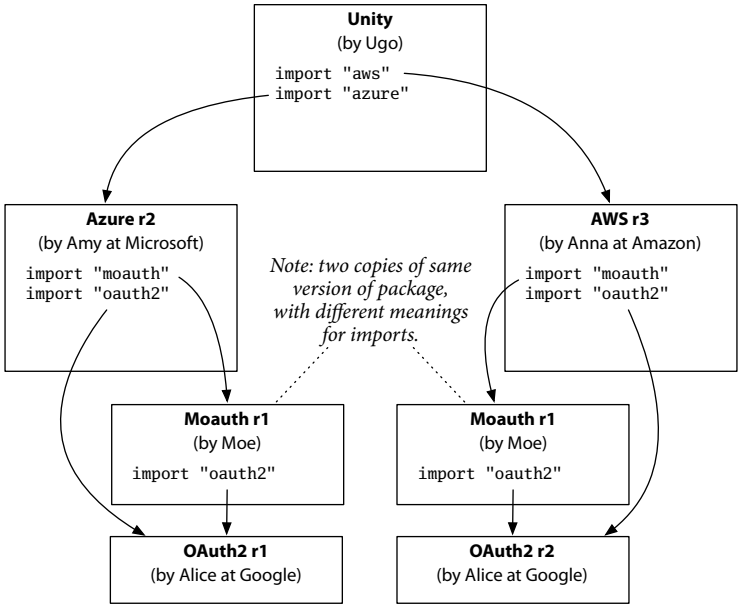


Ugo tries this. It doesn't work. Digging further into the problem, Ugo discovers that both Azure and AWS are using a popular OAuth2 middleware library called Moauth that simplifies part of the OAuth2 processing. Moauth is not a complete API replacement: users still import OAuth2 directly, but they use Moauth to simplify some of the API calls. The details that Moauth helps with didn't change from OAuth2 r1 to r2, so Moauth r1 (the only version that exists) is compatible with either. Both Azure r2 and AWS r3 use Moauth r1. That works fine in programs using only Azure or only AWS, but Ugo's Unity build actually looks like:



Unity needs both copies of OAuth2, but then which one does Moauth import?

In order to make the build work, it would seem that we need two identical copies of Moauth: one that imports OAuth2 r1, for use by Azure, and a second that imports OAuth2 r2, for use by AWS. A quick StackOverflow check shows that the package manager has a flag for that: `-fclone`. Using this flag, Ugo's program builds as:



This actually works and passes its tests, although Ugo now wonders if there are more problems lurking. He heads home for a late dinner.

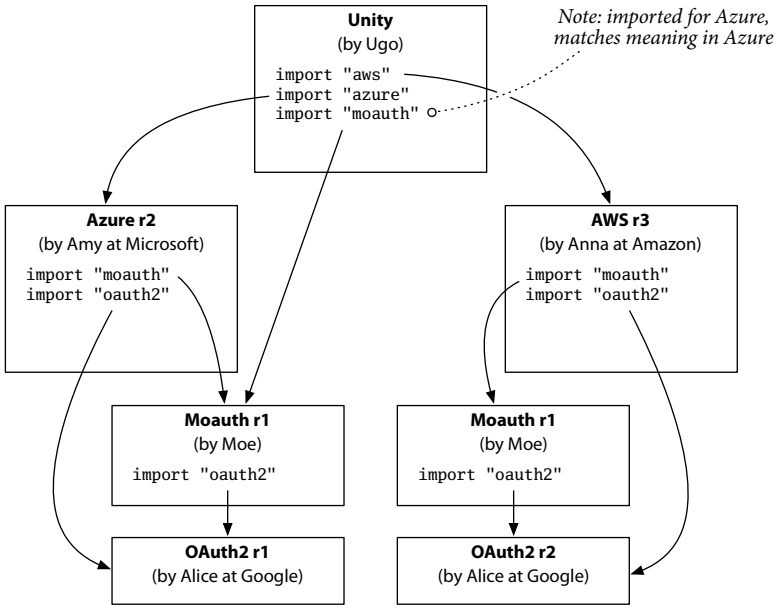
Chapter 3

Back at Microsoft, Amy has returned from vacation. She decides that Azure can keep using OAuth2 r1 for a while longer, but she realizes that it would help users to let them pass Moauth tokens directly into the Azure API. She adds this to the Azure package in a backwards-compatible way and releases Azure r3. Over at Amazon, Anna likes the Azure package's new Moauth-based API and adds a similar API to the AWS package, releasing AWS r4.

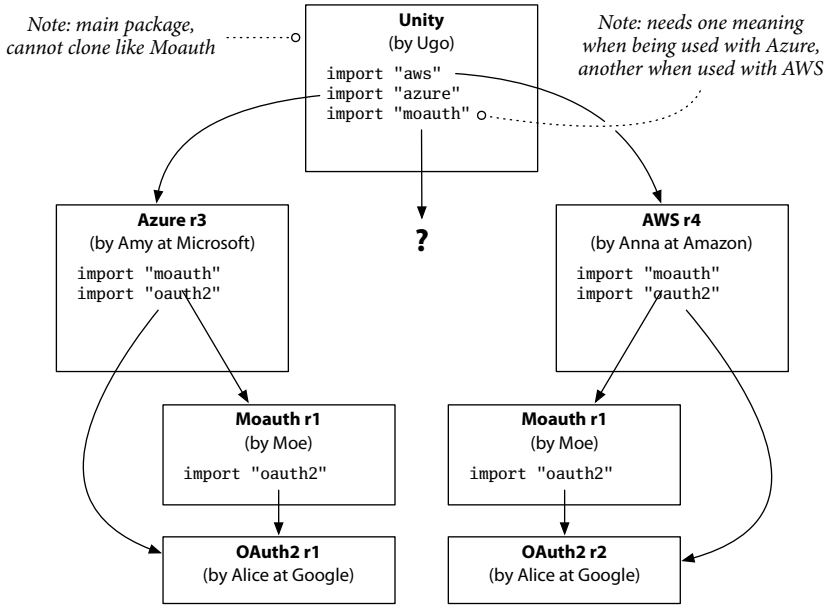
Ugo sees these changes and decides to update to the latest version of both Azure and AWS in order to use the Moauth-based APIs. This time he blocks off an afternoon. First he tentatively updates the Azure and AWS packages without modifying Unity at all. His program builds!

Excited, Ugo changes Unity to use the Moauth-based Azure API, and that builds too. When he changes Unity to also use the Moauth-based AWS API, though, the build fails. Perplexed, he reverts his Azure changes, leaving only the AWS changes, and the build succeeds. He puts the Azure changes back, and the build fails again. Ugo returns to StackOverflow.

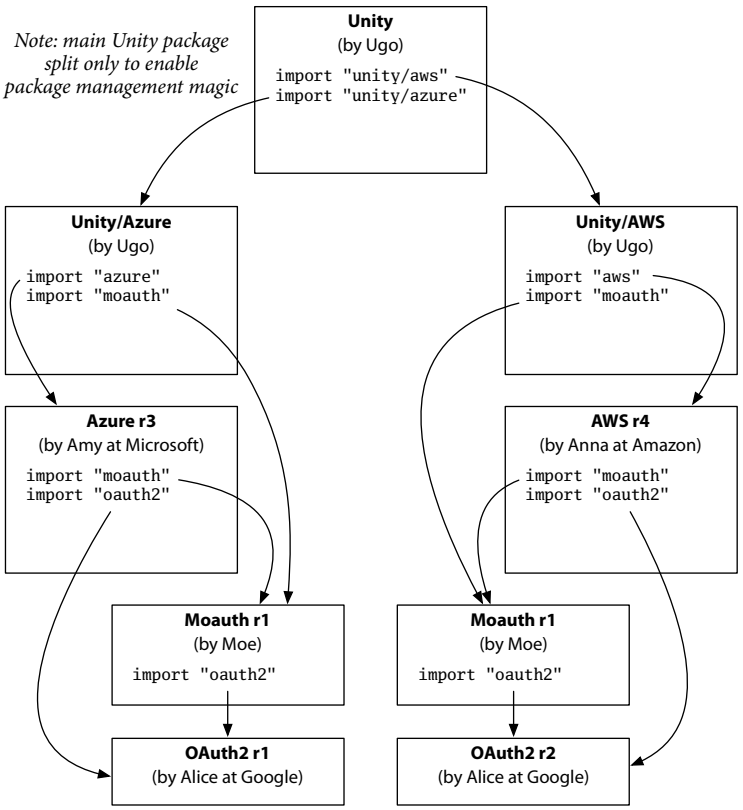
Ugo learns that when using just one Moauth-based API (in this case, Azure) with `-fmultiverse -fclone`, Unity implicitly builds as:



but when he is using both Moauth-based APIs, the single `import "moauth"` in Unity is ambiguous. Since Unity is the main package, it cannot be cloned (in contrast to Moauth itself):



A comment on StackOverflow suggests moving the Moauth import into two different packages and having Unity import them instead. Ugo tries this and, incredibly, it works:



Ugo makes it home on time. He's not terribly happy with his package manager, but he's now a big fan of StackOverflow.

A Retelling with Semantic Versioning

Let's wave a magic wand and retell the story with semantic versions, assuming that the package manager uses them instead of the original story's 'r' numbers.

Here's what changes:

- OAuth2 r1 becomes OAuth2 1.0.0.
- Moauth r1 becomes Moauth 1.0.0.
- Azure r1 becomes Azure 1.0.0.
- AWS r1 becomes AWS 1.0.0.
- OAuth2 r2 becomes OAuth2 2.0.0 (partly incompatible API).
- Azure r2 becomes Azure 1.0.1 (bug fix).
- AWS r2 becomes AWS 1.0.1 (bug fix, internal use of OAuth2 2.0.0).
- AWS r3 becomes AWS 1.1.0 (feature update: add Zeta).
- Azure r3 becomes Azure 1.1.0 (feature update: add Moauth APIs).
- AWS r4 becomes AWS 1.2.0 (feature update: add Moauth APIs).

Nothing else about the story changes. Ugo still runs into the same build problems, and he still has to turn to StackOverflow to learn about build flags and refactoring techniques just to keep Unity building. According to semver, though, Ugo should have had no trouble at all with any of his updates: not one of

the packages that Unity imports changed its major version during the story. Only OAuth2 did, deep in Unity's dependency tree. Unity itself does not import OAuth2. What went wrong?

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. In particular, it says nothing about how to handle incompatible changes after incrementing the major version number.

The most valuable part of semver is the encouragement to make backwards-compatible changes when possible. The FAQ correctly notes:

> "Incompatible changes should not be introduced lightly to software that has a lot of dependent code. The cost that must be incurred to upgrade can be significant. Having to bump major versions to release incompatible changes means you'll think through the impact of your changes and evaluate the cost/benefit ratio involved."

I certainly agree that "incompatible changes should not be introduced lightly." Where I think semver falls short is the idea that "having to bump major versions" is a step that will make you "think through the impact of your changes and evaluate the cost/benefit ratio involved." Quite the opposite: it's far too easy to read semver as implying that as long as you increment the major version when you make an incompatible change, everything else will work out. The example shows that this is not the case.

From Alice's point of view, the OAuth2 API needed backwards-incompatible changes, and when she made them, semver seemed to promise it would be fine to release an incompatible OAuth2 package, provided she gave it version 2.0.0. But that semver-approved change triggered the cascade of problems that befell Ugo and Unity.

Semantic versions are an important way for authors to convey expectations to users, but that's all they are. By itself, it can't be expected to solve these larger build problems. Instead, let's look at an approach that does solve the build problems. Afterward, we can consider how to fit semver into that approach.

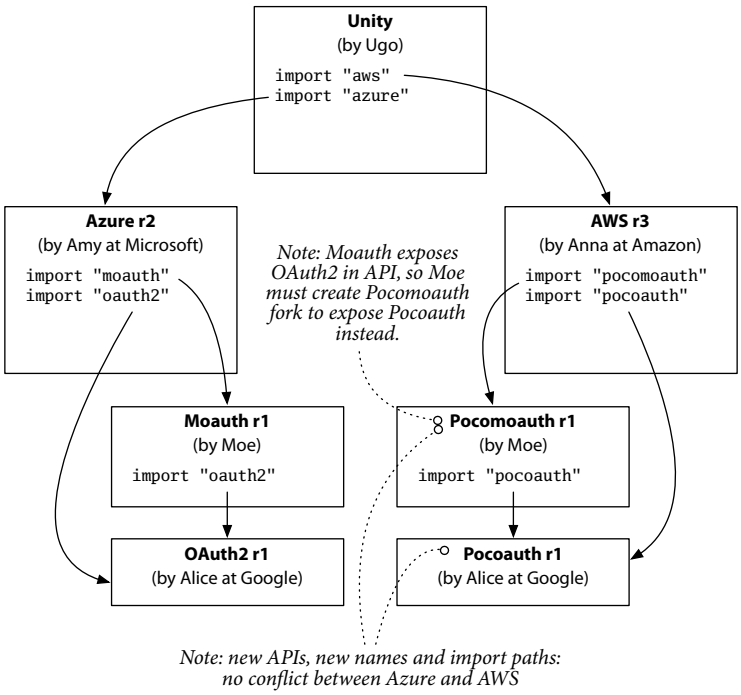
A Retelling with Import Versioning

Once again, let's retell the story, this time using the import compatibility rule:

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

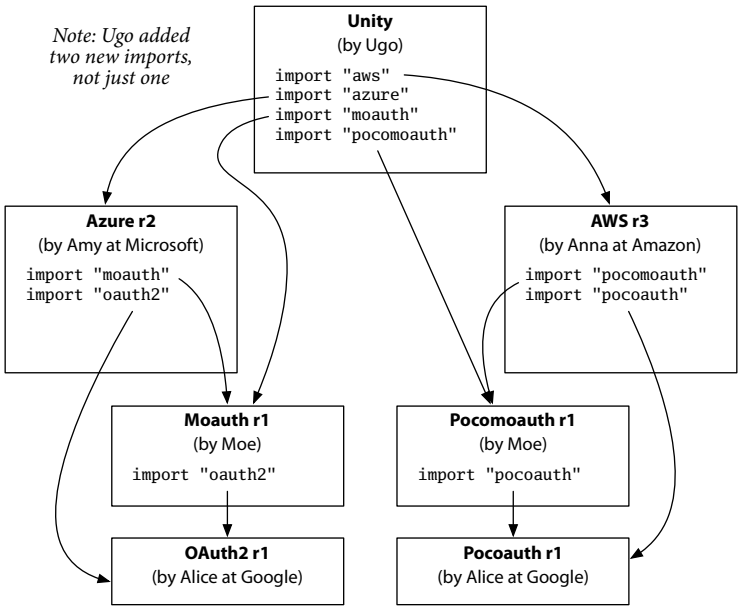
Now the plot changes are more significant. The story starts out the same way, but in Chapter 1, when Alice decides to create a new, partly incompatible OAuth2 API, she cannot use `"oauth2"` as its import path. Instead, she names the new version Pocoauth and gives it the import path `"pocoauth"`. Presented with two different OAuth2 packages, Moe (the author of Moauth) must write a second package, Moauth for Pocoauth, which he names Pocomoauth and gives the import path `"pocomoauth"`. When Anna updates the AWS package to the new OAuth2 API, she also changes the import paths in that code from `"oauth2"` to `"pocoauth"` and from `"moauth"` to `"pocomoauth"`. Then the story proceeds as before, with the release of AWS r2 and AWS r3.

In Chapter 2, when Ugo eagerly adopts Amazon Zeta, everything just works. The imports in all the packages code exactly match what needs to be built. He doesn't have to look up special flags on StackOverflow, and he's only five minutes late to lunch.



In Chapter 3, Amy adds Moauth-based APIs to Azure while Anna adds equivalent Pocomoauth-based APIs to AWS.

When Ugo decides to update both Azure and AWS, again there's no problem. His updated program builds without any special refactoring:



At the end of this version of the story, Ugo doesn't even think about his package manager. It just works; he barely notices that it's there.

In contrast to the semantic versioning translation of the story, the use of import versioning here changed two critical details. First, when Alice introduced her backwards-incompatible OAuth2 API, she had to release it as a new package (Pocoauth). Second, because Moe's wrapper package Moauth exposed the OAuth2 package's type definitions in its own API, Alice's release of a new package forced Moe's release of a new package (Pocomoauth). Ugo's final Unity build

went well because Alice's and Moe's package splits created exactly the structure needed to keep clients like Unity building and running. Instead of Ugo and users like him needing incomplete package manager complexity like `-fmulti-verse -fclone` aided by extraneous refactorings, the import compatibility rule pushes a small amount of additional work onto package authors, and all users benefit.

There is certainly a cost to needing to introduce a new name for each backwards-incompatible API change, but as the semver FAQ says, that cost should encourage authors to more clearly consider the impact of such changes and whether they are truly necessary. And in the case of Import Versioning, the cost pays for significant benefits to users.

An advantage of Import Versioning here is that package names and import paths are well-understood concepts for Go developers. If you tell an author that making a backwards-incompatible change requires creating a different package with a different import path, then—without any special knowledge of versioning—the author can reason through the implications on client packages: clients are going to need to change their own imports one at a time; Moauth is not going to work with the new package; and so on.

Able to predict the effects on users more clearly, authors might well make different, better decisions about their changes. Alice might look for way to introduce the new, cleaner API into the original OAuth2 package alongside the existing APIs, to avoid a package split. Moe might look more carefully at whether he can use interfaces to make Moauth support both OAuth2 and Pocoauth, avoiding a new Pocomoauth package. Amy might decide it's worth updating to Pocoauth and Pocomoauth instead of exposing the fact that the Azure APIs use outdated OAuth2 and Moauth packages. Anna might have tried to make the AWS APIs allow either Moauth or Pocomoauth, to make it easier for Azure users to switch.

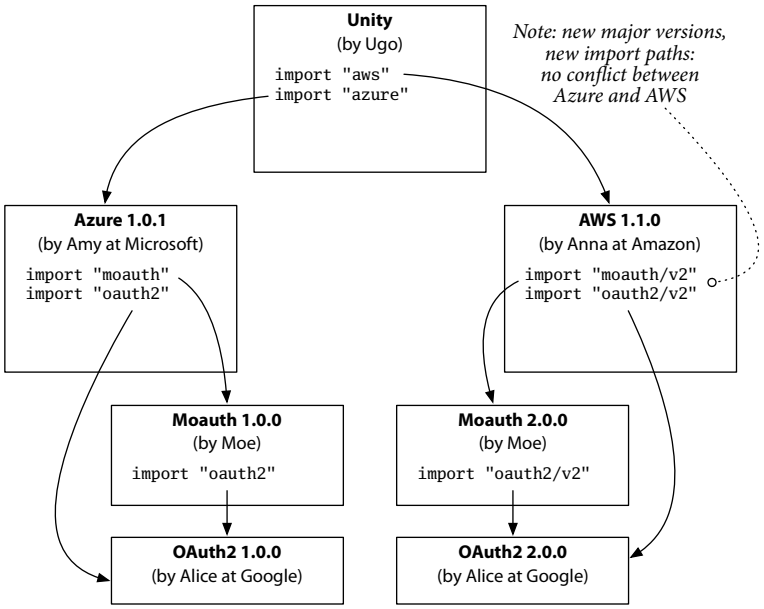
In contrast, the implications of a semver "major version bump" are far less clear and do not exert the same kind of design pressure on authors. To be clear, this approach creates a bit more work for authors, but that work is justified by delivering significant benefits to users. In general, this balance makes sense, because packages aim to have many more users than authors, and hopefully all packages at least have as many users as they do authors.

Semantic Import Versioning

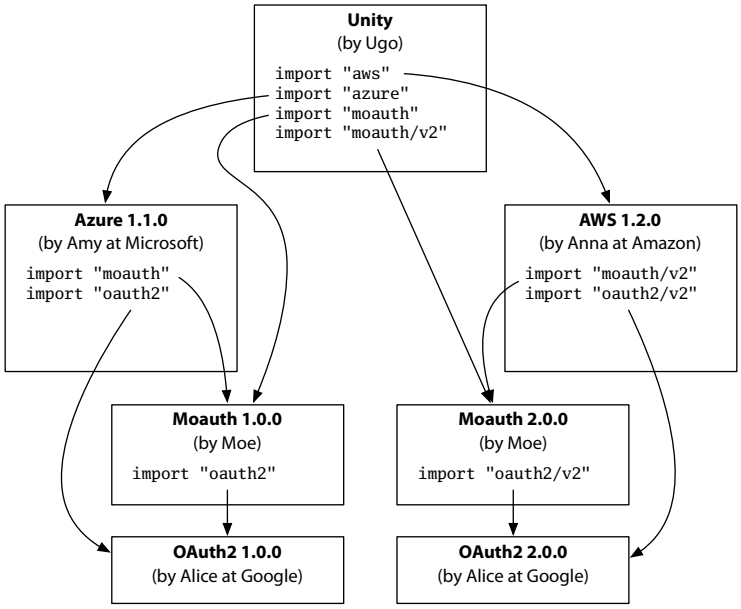
The previous section showed how import versioning leads to simple, predictable builds during updates. But choosing a new name at every backwards-incompatible change is difficult and unhelpful to users. Given the choice between OAuth2 and Pocoauth, which should Amy use? Without further investigation, there's no way to know. In contrast, semantic versioning makes this easy: OAuth2 2.0.0 is clearly the intended replacement for OAuth2 1.0.0.

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"`. The same for Moe: Moauth 2.0.0 (imported as `"moauth/v2"`) can be the helper package for OAuth2 2.0.0, just as Moauth 1.0.0 was the helper package for OAuth2 1.0.0.

When Ugo adds Zeta support in Chapter 2, his build looks like:



Because `"moauth"` and `"moauth/v2"` are simply different packages, it is perfectly clear to Ugo what he needs to do to use `"moauth"` with Azure and `"moauth/v2"` with AWS: import both.



For compatibility with existing Go usage and as a small encouragement not to make backwards-incompatible API changes, I am assuming here that major version 1 is omitted from import paths: `import "moauth"`, not `"moauth/v1"`. Similarly, major version 0, which explicitly disavows compatibility, is also omitted from import paths. The idea here is that by using a v0 dependency, users are explicitly acknowledging the possibility of breakage and taking on the responsibility to deal with it when they choose to update. (Of course, it's then important that updates don't happen automatically. We'll see in the next post how minimal version selection helps with that.)

Functional Names & Immutable Meanings

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. The old name had one meaning. By using a different meaning for the new name and eliminating the old one, we ensured the compiler would complain loudly about every piece of code that needed to be examined and updated, instead of silently compiling incorrect code. And if people had their own programs using the function, they'd get a compile-time failure instead of a long debugging session. In today's world of distributed version control, that last problem is magnified, making the name change even more important. A merge of concurrently-written code expecting the old semantics should not silently get the new semantics instead.

Of course, deleting an old function works only when all the uses can be found, or when users understand that they are responsible for keeping up with changes, as was the case in a research system like Plan 9. For exported APIs, it's usually much better to leave the old name and old behavior intact and only add a new name with new behavior. Rich Hickey made the point in his "Speculation" 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. The functional approach brings benefits in clarity and predictability in small-scale programming, and the benefits are even larger when applied, as in the import compatibility rule, to whole APIs: dependency hell is really just mutability hell writ large. That's just one small observation in the talk; the whole thing is worth watching.

In the early days of "`go get`", when people asked about making backwards-incompatible changes, our response—based on intuition derived from years of experience with these kinds of software changes—was to give the import versioning rule, but without a clear explanation why this approach was better than not putting the major version in the import paths. 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.*

One motivation for this blog post is to show, using a clear, believable example, why following the rule is so important.

Avoiding Singleton Problems

One common objection to the semantic import versioning approach is that package authors today expect that there is only ever one copy of their package in a given build. Allowing multiple packages at different major versions may cause problems due to unintended duplications of singletons. An example would be registering an HTTP handler. If `my/thing` registers an HTTP handler for `/debug/my/thing`, then having two copies of the package will result in duplicate registrations, which causes a panic at registration time. Another problem would be if there were two HTTP stacks in the program. Clearly only one HTTP stack can listen on port 80; we wouldn't want half the program registering handlers that will not be used. Go developers are already running into problems like this due to vendoring inside vendored packages.

Moving to vgo and semantic import versioning clarifies and simplifies the current situation though. Instead of the uncontrolled duplication caused by vendoring inside vendoring, authors will have a guarantee that there is only one instance of each major version of their packages. 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. Perhaps that means exporting debug information for v2 on `/debug/my/thing/v2`. Or perhaps it means coordinating. Maybe v2 can take charge of registering the handler but also provide a hook for v1 to supply information to display on the page. This would mean `my/thing` importing `my/thing/v2` or vice versa; with different import paths, that's easy to do and easy to understand. In contrast, if both v1 and v2 are `my/thing` it's hard to comprehend what it means for one to import its own import path and get the other.

Automatic API Updates

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. (See my 2016 article, "Codebase Refactoring (with help from Go)," for more on that problem.)

In addition to keeping programs building, semantic import versioning has a significant benefit to gradual code repair, which I touched on in the previous section: one major version of a package can import and be written in terms of another. It is trivial for the v2 API to be written as a wrapper of the v1 implementation, or vice versa. This lets them share the code and, with appropriate design choices and perhaps use of type aliases, might even allow clients using v1 and v2 to interoperate. It may also help resolve a key technical problem in defining automatic API updates.

Before Go 1, we relied heavily on `go fix`, which users ran after updating to a new Go release and finding their programs no longer compiled. Updating code that doesn't compile makes it impossible to use most of our program analysis tools, which require that their inputs are valid programs. Also, we've wondered how to allow authors of packages outside the Go standard library to supply "fixes" specific to their own API updates. The ability to name and work with multiple incompatible versions of a package in a single program suggests a possible solution: if a v1 API function can be implemented as a wrapper around the v2 API, the wrapper implementation can double as the fix specification. For example, suppose v1 of an API has functions `EnableFoo` and `DisableFoo` and v2 replaces the pair with a single `SetFoo(enabled bool)`. After v2 is released, v1 can be implemented as a wrapper around v2:

```
package p // v1

import v2 "p/v2"

func EnableFoo() {
    //go:fix
    v2.SetFoo(true)
}

func DisableFoo() {
    //go:fix
    v2.SetFoo(false)
}
```

The special `//go:fix` comments would indicate to `go fix` that the wrapper body that follows should be inlined into the call site. Then running `go fix` would rewrite calls to v1 `EnableFoo` to v2 `SetFoo(true)`. The rewrite is easily specified and type-checked, since it is plain Go code. Even better, the rewrite is clearly safe: v1 `EnableFoo` is *already* calling v2 `SetFoo(true)`, so rewriting the call site plainly does not change the meaning of the program.

It is plausible that `go fix` might use symbolic execution to fix even the reverse API change, from a v1 with `SetFoo` to a v2 with `EnableFoo` and `DisableFoo`. The v1 `SetFoo` implementation could read:

```
package q // v1

import v2 "q/v2"

func SetFoo(enabled bool) {
    if enabled {
        //go:fix
        v2.EnableFoo()
    } else {
        //go:fix
        v2.DisableFoo()
    }
}
```

and then `go fix` would update `SetFoo(true)` to `EnableFoo()` and `SetFoo(false)` to `DisableFoo()`. This kind of fix would even apply to API updates within a single major version. For example, v1 could be deprecating (but keeping) `SetFoo` and introducing `EnableFoo` and `DisableFoo`. The same kind of fix would help callers move away from the deprecated API.

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. We need only follow the rule that when you make a change to something, you also change its name.

Committing to Compatibility

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. But authors who do that are hurting their users. It seems to me a good thing if the system makes it harder to hurt users and instead naturally steers authors toward behaviors that don't hurt users.

More generally, Sam Boyer talked at GopherCon 2017 about how package managers moderate our social interactions, the collaboration of people building software. We get to decide. Do we want to work in a community built around a system that optimizes for compatibility, smooth transitions, and working well together? Or do we want to work in a community built around a system that optimizes for creating and describing incompatibility, that makes it acceptable for authors to break users' programs? Import versioning, and in particular handling semantic versioning by lifting the semantic major version into the import path, is how we can make sure we work in the first kind of community.

Let's commit to compatibility.

Minimal Version Selection

Go & Versioning, Part 4

Russ Cox

February 21, 2018

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.

The *version selection* problem therefore is to define the meaning of, and to give algorithms implementing, these four operations on build lists:

1. Construct the current build list.
2. Upgrade all modules to their latest versions.
3. Upgrade one module to a specific newer version.
4. Downgrade one module to a specific older version.

The last two operations specify one module to upgrade or downgrade, but doing so may require upgrading, downgrading, adding, or removing other modules, ideally as few as possible, to satisfy dependencies.

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.

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.

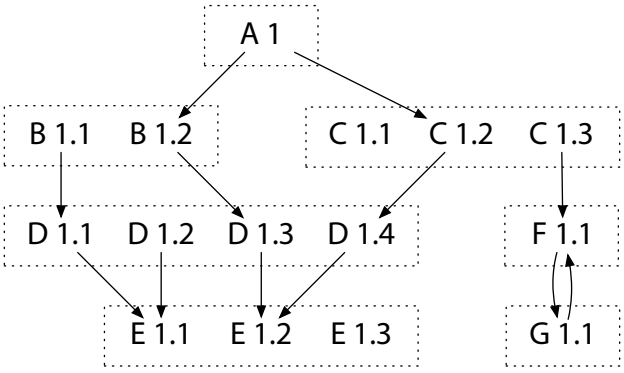
Then the definitions of the four operations are:

1. To construct the build list for a given target: start the list with the target itself, and then append each requirement's own build list. If a module appears in the list multiple times, keep only the newest version.
2. To upgrade all modules to their latest versions: construct the build list, but read each requirement as if it requested the latest module version.
3. To upgrade one module to a specific newer version: construct the non-upgraded build list and then append the new module's build list. If a module appears in the list multiple times, keep only the newest version.
4. To downgrade one module to a specific older version: rewind the required version of each top-level requirement until that requirement's build list no longer refers to newer versions of the downgraded module.

These operations are simple, efficient, and easy to implement.

Example

Before we examine minimal version selection in more detail, let's look at why a new approach is necessary. We'll use the following set of modules as a running example throughout the post:



The diagram shows the module requirement graph for seven modules (dotted boxes) with one or more versions. Following semantic versioning, all versions of a given module share a major version number. We are developing module A 1, and we will run commands to update its dependency requirements. The diagram shows both A 1's current requirements and the requirements declared by various versions of released modules B 1 through F 1.

Because the major version is part of the module's identifier, we must know that we are working on A 1 as opposed to A 2, but otherwise the exact version of A is unspecified—our work is unreleased. Similarly, different major versions are just different modules: for the purposes of these algorithms, B 1 is no more related to B 2 than to C 1. We could replace B 1 through F 1 in the diagram with A 2 through A 7 at a significant loss in clarity but without any change in how the algorithms handle the example. Because all the modules in the example do have major version 1, from now on we will omit the major version when possible, shortening A 1 to A. Our current development copy of A requires B 1.2 and C 1.2. B 1.2 in turn requires D 1.3. An earlier version, B 1.1, required D 1.1. And so on. Note that F 1.1 requires G 1.1, but G 1.1 also requires F 1.1. Declaring this kind of cycle can be important when singleton functionality moves from one module to another. Our algorithms must not assume the module requirement graph is acyclic.

Low-Fidelity Builds

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. This mode can use versions that are too old: if you have B 1.1 installed and run `go get` to download A, `go get` would not update to B 1.2, causing a failed or buggy build.

The second algorithm is the behavior of `go get -u`: download and use the latest version of everything. This mode fails by using versions that are too new: if you run `go get -u` to download A, it will correctly update to B 1.2, but it will also update to C 1.3 and E 1.3, which aren't what A asks for, may not have been tested, and may not work.

I call both these outcomes low-fidelity builds: viewed as attempts to reproduce the build that A's author used, these builds differ for no good reason. After we've seen the details of the minimal version selection algorithms, we'll look at why they produce high-fidelity builds instead.

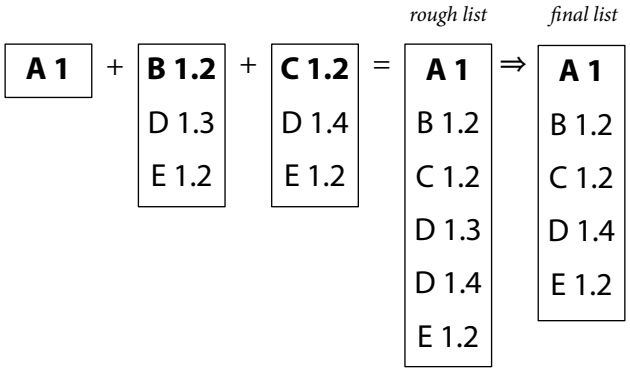
Algorithms

Now let's look at the algorithms in more detail.

Algorithm 1: Construct Build List

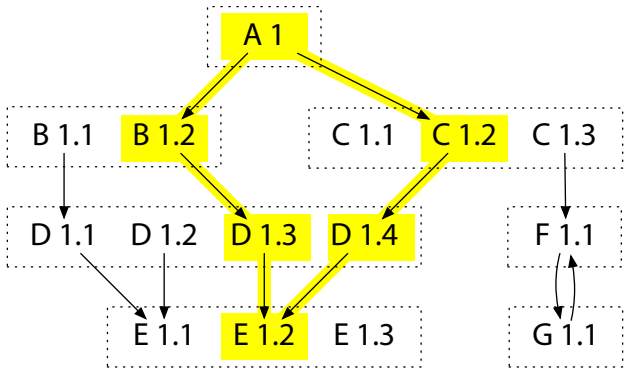
There are two useful (and equivalent) ways to define build list construction: as a recursive process and as a graph traversal.

The recursive definition of build list construction is as follows. Construct the rough build list for M by starting an empty list, adding M, and then appending the build list for each of M's requirements. Simplify the rough build list to produce the final build list, by keeping only the newest version of any listed module.



The recursive construction of build lists is useful mainly as a mental model. A literal implementation of that definition would be too inefficient, potentially requiring time exponential in the size of an acyclic module requirement graph and running forever on a cyclic graph.

An equivalent, more efficient construction is based on graph reachability. The rough build list for M is also just the list of all modules reachable in the requirement graph starting at M and following arrows. This can be computed by a trivial recursive traversal of the graph, taking care not to visit a node that has already been visited. For example, A's rough build list is the highlighted module versions found by starting at A and following the highlighted arrows:



(The simplification from rough build list to final build list remains the same.)

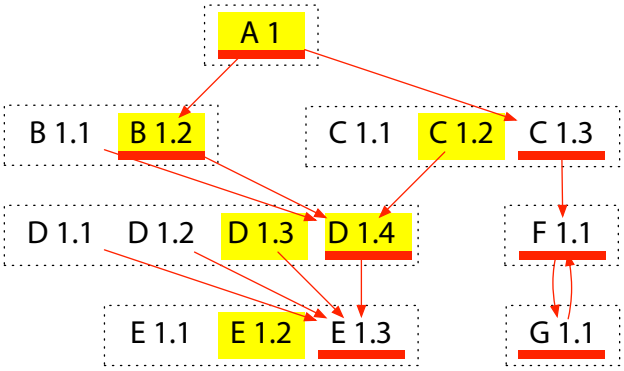
Note that this algorithm only visits each module in the rough build list once, and only those modules, so the execution time is proportional to the rough build list size $|B|$ plus the number of arrows that must be traversed (at most $|B|^2$). The algorithm completely ignores versions left off the rough build list: for example, it loads information about D 1.3, D 1.4, and E 1.2, but it does not load information about D 1.2, E 1.1 or E 1.3. In a dependency management setting, where loading information about each module version may mean a separate network round trip, avoiding unnecessary module versions is an important optimization.

Algorithm 2. Upgrade All Modules

Upgrading all modules is perhaps the most common modification made to build lists. It is what `go get -u` does today.

We compute an upgraded build list by upgrading the module requirement graph and then applying the previous algorithm. An upgraded module requirement graph is one in which every arrow pointing at any version of a module has been replaced by one pointing at the latest version of that module. (It is then also possible to discard all older versions from the graph, but the build list construction won't look at them anyway, so there's no need to clean up the graph.)

For example, here is the upgraded module requirement graph, with the original build list still marked in yellow and the upgraded build list now marked in red:



Although this tells us the upgraded build list, it does not yet tell us how to cause future builds to use that build list instead of the old build list (still marked in yellow). To upgrade the graph we changed the requirements for all modules, but an upgrade during development of module A must somehow be recorded only in A's requirement list (in A's `go.mod` file) in a way that causes Algorithm 1 to produce the build list we want, to pick the red modules instead of the yellow ones. To decide what to add to A's requirement list to cause that effect, we introduce a helper, Algorithm R.

Algorithm R. Compute a Minimal Requirement List

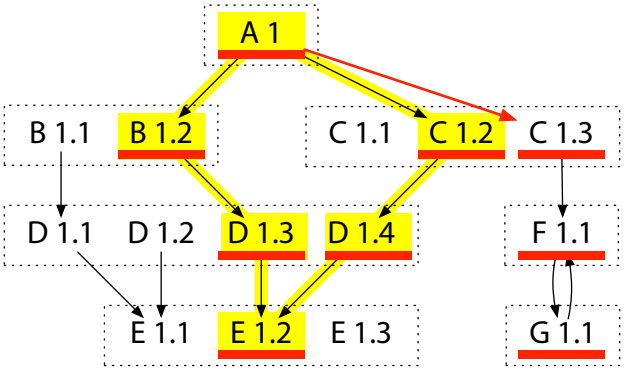
Given a build list compatible with the module requirement graph below the target, we want to compute a requirement list for the target that will induce that build list. It is always sufficient to list every module in the build list other than the target itself. For example, the upgrade we considered above could add C 1.3 (replacing C 1.2), D 1.4, E 1.3, F 1.1, and G 1.1 to A's requirement list. But in general not all of these additions are necessary, and we want to list as few additional modules as possible. For example, F 1.1 implies G 1.1 (and vice versa), so we need not list both. At first glance it seems natural to start by adding the module versions marked in red but not yellow (on the new list but missing from the old list). That heuristic would incorrectly drop D 1.4, which is implied by the old requirement C 1.2 but not by the new requirement C 1.3.

Instead, it is correct to visit the modules in reverse postorder—that is, to visit a module only after considering all modules that point into it—and only keep a module if it is not implied by modules already visited. For an acyclic graph, the result is a unique, minimal set of additions. For a cyclic graph, the reverse-postorder traversal must break cycles, and then the set of additions is unique and minimal for the modules not involved in cycles. As long as the result is correct and stable, we'll accept non-minimal answers in the case of cycles. In this example, the upgrade needs to add C 1.3 (replacing C 1.2), D 1.4, and E 1.3. It can drop F 1.1 (implied by C 1.3) and G 1.1 (also implied by C 1.3).

Algorithm 3. Upgrade One Module

Instead of upgrading all modules, cautious developers typically want to upgrade only one module, with as few other changes to the build list as possible. For example, we may want to upgrade to C 1.3, and we do not want that operation to make unnecessary changes like upgrading to E 1.3. Like in Algorithm 2, we can upgrade one module by upgrading the requirement graph, constructing a build list from it (Algorithm 1), and then reducing that list back to a set of requirements for the top-level module (Algorithm R). To upgrade the requirement graph, we add one new arrow from the top-level module to the upgraded module version.

For example, if we want to change A's build to upgrade to C 1.3, here is the upgraded requirement graph:



Like before, the new build list's modules are marked in red, and the old build list's are in yellow.

The upgrade's effect on the build list is the unique minimal way to make the upgrade, adding the new module version and any implied requirements but nothing else. Note that when constructing the upgraded graph, we must only add new arrows, not replace or remove old ones. For example, if the new arrow from A to C 1.3 replaced the old arrow from A to C 1.2, the upgraded build list would omit D 1.4. That is, the upgrade of C would downgrade D, an unexpected, unwanted, and non-minimal change. Once we've computed the build list for the upgrade, we can run Algorithm R (above) to decide how to update the requirements list. In this case we'd end up replacing C 1.2 with C 1.3 but then also adding a new requirement on D 1.4, to avoid the accidental downgrade of D. Note that this selective upgrade only updates other modules to C's minimum requirements: the upgrade of C does not simply fetch the latest of each of C's dependencies.

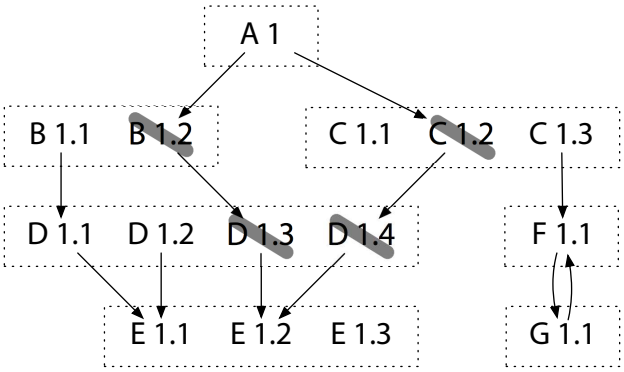
Algorithm 4. Downgrade One Module

We may also discover, perhaps after upgrading all modules, that the latest module version is buggy and must be avoided. In that situation, we need to be able to downgrade to an earlier version of the module. Downgrading one module may require downgrading other modules, but we want to downgrade as few other modules as possible. Like upgrades, downgrades must make their changes to the build list by modifying a target's requirements list. Unlike upgrades, downgrades must work by removing requirements, not adding them. This observation leads to a very simple downgrade algorithm that considers each of the target's requirements individually. If a requirement is incompatible with the proposed downgrade—that is, if the requirement's build list includes a now-disallowed module version—then try successively older versions until finding one that is compatible with the downgrade.

For example, starting with the original build graph, suppose we discover that

there is a problem with D 1.4, actually introduced in D 1.3, and so we decide to downgrade to D 1.2. Our target module A depends on B 1.2 and C 1.2. To downgrade from D 1.4 to D 1.2, we must find earlier versions of B and C that do not require (directly or indirectly) versions of D later than D 1.2.

Although we can consider each requirement separately, it is more efficient to consider the module requirement graph as a whole. In our example, the downgrade rule amounts to crossing out the unavailable versions of D and then following arrows backwards from unavailable modules to find and cross out other unavailable modules. At the end, the latest versions of A's requirements that remain can be recorded as the new requirements.



In this case, downgrading to D 1.2 implies downgrading to B 1.1 and C 1.1. To avoid an unnecessary downgrade to E 1.1, we must also add a new requirement on E 1.2. We can apply Algorithm R to find the minimal set of new requirements to write to `go.mod`.

Note that if we'd first upgraded to C 1.3, then the downgrade to D 1.2 would have continued to use C 1.3, which doesn't use any version of D at all. But downgrades are constrained to only downgrade packages, not also upgrade them; if an upgrade before downgrade is needed, the user must ask for it explicitly.

Theory

Minimal version selection is *very* simple. It achieves simplicity by eliminating all flexibility about what the answer must be: the build list is exactly the versions specified in the requirements. A real system needs more flexibility, for example the ability to exclude certain module versions or replace others. Before we add those, it is worth examining the theoretical basis for the current system's simplicity, so we understand which kinds of extensions preserve that simplicity and which do not.

If you are familiar with the way most other systems approach version selection, or if you remember my Version SAT post from a year ago, probably the most striking feature of Minimal version selection is that it does not solve general Boolean satisfiability, or SAT. As I explained in my earlier post, it takes very little for a version search to fall into solving SAT; version searches in these systems are inherently intricate, complex problems for which we know no general efficient solutions. If we want to avoid this fate, we need to know where the boundaries are, where not to step as we explore the design space. Conveniently, Schaefer's Dichotomy Theorem describes those boundaries precisely. It identifies six restricted classes of Boolean formulas for which satisfiability can be decided in polynomial time and then proves that for any class of formulas beyond those, satisfiability is NP-complete. To avoid NP-completeness, we need to limit the version selection problem to stay within one of Schaefer's restricted classes.

It turns out that minimal version selection lies in the intersection of three of the six tractable SAT subproblems: 2-SAT, Horn-SAT, and Dual-Horn-SAT. The

formula corresponding to a build in minimal version selection is the AND of a set of clauses, each of which is either a single positive literal (this version must be installed, such as during an upgrade), a single negative literal (this version is not available, such as during a downgrade), or the OR of one negative and one positive literal (an implication: if this version is installed, this other version must also be installed). The formula is a 2-CNF formula, because each clause has at most two variables. The formula is also a Horn formula, because each clause has at most one positive literal. The formula is also a dual-Horn formula, because each clause has at most one negative literal. That is, every satisfiability problem posed by minimal version selection can be solved by your choice of three different efficient algorithms. It is even simpler and more efficient to specialize further, as we did above, taking advantage of the very limited structure of these problems.

Although 2-SAT is the most well-known example of a SAT subproblem with an efficient solution, the fact that these problems are both Horn and dual-Horn formulas is more interesting. Every Horn formula has a unique satisfying assignment with the fewest variables set to true. This proves that there is a unique minimal answer for constructing a build list, as well for each upgrade. The unique minimal upgrade does not use a newer version of a given module unless absolutely necessary. Conversely, every dual-Horn formula also has a unique satisfying assignment with the fewest variables set to *false*. This proves that there is a unique minimal answer for each downgrade. The unique minimal downgrade does not use an older version of a given module unless absolutely necessary. If we want to extend minimal version selection, for example with the ability to exclude certain modules, we can only keep the uniqueness and mimimality properties by continuing to use constraints expressible as both Horn and dual-Horn formulas.

(Digression: The problem minimal version selection solves is NL-complete: it's in NL because it's a subset of 2-SAT, and it's NL-hard because st-connectivity can be trivially transformed into a minimal version selection build list construction problem. It's delightful that we've replaced an NP-complete problem with an NL-complete problem, but there's little practical value to knowing that: being in NL only guarantees a polynomial-time solution, and we already have a linear-time one.)

Excluding Modules

Minimal version selection always selects the minimal (oldest) module version that satisfies the overall requirements of a build. If that version is buggy in some way, an upgrade or downgrade operation can modify the top-level target's requirements list to force selection of a different version.

It can also be useful to record explicitly that the version is buggy, to avoid reintroducing it in any future upgrade or downgrade operations. But we want to do that in a way that keeps the uniqueness and minimality properties of the previous section, so we must use constraints that are both Horn and dual-Horn formulas. That means build constraints can only be unconditional positive assertions (X: X must be installed), unconditional negative assertions ($\neg$ Y: Y must not be installed), and positive implications (X $\rightarrow$ Z, equivalently $\neg$ X $\vee$ Z: if X is installed, then Z must be installed). Negative implications (X $\rightarrow \neg$ Y, equivalently $\neg$ X $\vee \neg$ Y: if X is installed, then Y must *not* be installed) cannot be added as constraints without breaking the form. Module exclusions must therefore be unconditional: they must be decided independent of selections made during build list construction.

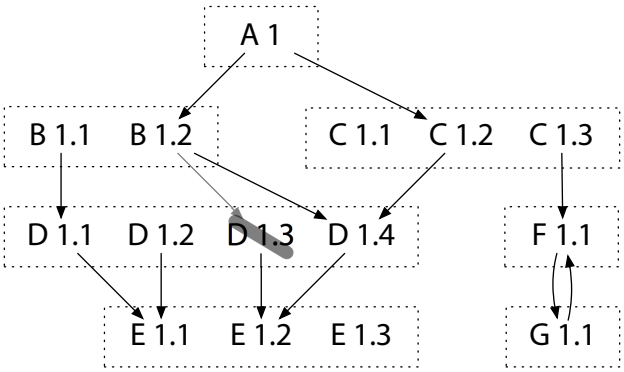
What we *can* do is allow a module to declare its own *local* list of excluded module versions. By local, I mean that the list is consulted only for builds with-

in that module; a larger build using the module only as a dependency would ignore the exclusion list. In our example, if A's build consulted D 1.3's list, then the exact set of exclusions would depend on whether the build selected, say, D 1.3 or D 1.4, making the exclusions conditional and leading to an NP-complete search problem. Only the top-level module is guaranteed to be in the build, so only the top-level module's exclusion list is used. Note that it would be fine to consult exclusion lists from other sources, such as a global exclusion list loaded over the network, as long as the decision to use the list is made before the build begins and the list content does not depend on which modules are selected during the build.

Despite all the focus on making exclusions unconditional, it might seem like we already have conditional exclusions: C 1.2 requires D 1.4 and so implicitly excludes D 1.3. But our algorithms do not treat this as an exclusion. When Algorithm 1 runs, it adds both D 1.3 (for B) and D 1.4 (for C) to the rough build list, along with their minimum requirements. The final simplification pass removes D 1.3 only because D 1.4 is present. The difference here between declaring an incompatibility and declaring a minimum requirement is critical. Declaring that C 1.2 must not be built with D 1.3 only describes how to fail. Declaring that C 1.2 must be built with D 1.4 instead describes how to succeed.

Exclusions then must be unconditional. Knowing that fact is important, but it does not tell us exactly how to implement exclusions. A simple answer is to add exclusions as the build constraints, with clauses like "D 1.3 must not be installed." Unfortunately, adding that clause alone would make modules that require D 1.3, like B 1.2, uninstallable. We need to express somehow that B 1.2 can choose D 1.4. The simple way to do that is to revise the build constraint, changing "B 1.2 $\rightarrow$ D 1.3" to "B 1.2 $\rightarrow$ D 1.3 $\vee$ D 1.4" and in general allowing all future versions of D. But that clause (equivalently, $\neg$ B 1.2 $\vee$ D 1.3 $\vee$ D 1.4) has two positive literals, making the overall build formula not a Horn formula anymore. It is still a dual-Horn formula, so we can still define a linear-time build list construction, but that construction—and therefore the question of how to perform an upgrade—would no longer be guaranteed to have a unique, minimal answer.

Instead of implementing exclusions as new build constraints, we can implement them by changing existing ones. That is, we can modify the requirements graph, just as we did for upgrades and downgrades. If a specific module is excluded, then we can remove it from the module requirement graph but also change any existing requirements on that module to require the next newer version instead. For example, if we excluded D 1.3, then we'd also update B 1.2 to require D 1.4:



If the latest version of a module is removed, then any modules requiring that version also need to be removed, as in the downgrade algorithm. For example, if G 1.1 were removed, then C 1.3 would need to be removed as well.

Once the exclusions have been applied to the module requirement graph, the algorithms proceed as before.

Replacing Modules

During development of A, suppose we find a bug in D 1.4, and we want to test a potential fix. We need some way to replace D 1.4 in our build with an unreleased copy U. We can allow a module to declare this as a replacement: "proceed as if D 1.4's source code and requirements have been replaced by U's."

Like exclusions, replacements can be implemented by modifying the module requirement graph in a preprocessing step, not by adding complexity to the algorithms that process the graph. Also like exclusions, the replacement list is local to one module. The build of A consults the replacement list from A but not from B 1.2, C 1.2, or any of the other modules in the build. This avoids making replacements conditional, which would be difficult to implement, and it also avoids the possibility of conflicting replacements: what if B 1.2 and C 1.2 specify different replacements for E 1.2? More generally, keeping exclusions and replacements local to one module limits the control that module exerts on other builds.

Who Controls Your Build?

The dependencies of a top-level module must be given some control over the top-level build. B 1.2 needs to be able to make sure it is built with D 1.3 or later, not with D 1.2. Otherwise we end up with the current `go get`'s stale dependency failure mode.

At the same time, for builds to remain predictable and understandable, we cannot give dependencies arbitrary, fine-grained control over the top-level build. That leads to conflicts and surprises. For example, suppose B declares that it requires an even version of D, while C declares that it requires a prime version of D. D is frequently updated and is up to D 1.99. Using B or C in isolation, it's always possible to use a relatively recent version of D (D 1.98 or D 1.97, respectively). But when A uses both B and C, the build silently selects the much older (and buggier) D 1.2 instead. That's an extreme example, but it raises the question: why should the authors of B and C be given such extreme control over A's build? As I write this post, there is an open bug report that the Kubernetes Go client declares a requirement on a specific, two-year-old version of `gopkg.in/yaml.v2`. When a developer tried to use a new feature of that YAML library in a program that already used the Kubernetes Go client, even after attempting to upgrade to the latest possible version, code using the new feature failed to compile, because "latest" had been constrained by the Kubernetes requirement. In this case, the use of a two-year-old YAML library version may be entirely reasonable within the context of the Kubernetes code base, and clearly the Kubernetes authors should have complete control over their own builds, but that level of control does not make sense to extend to other developers' builds.

In the design of module requirements, exclusions, and replacements, I've tried to balance the competing concerns of allowing dependencies enough control to ensure a succesful build without allowing them so much control that they harm the build. Minimum requirements combine without conflict, so it is feasible (even easy) to gather them from all dependencies. But exclusions and replacements can and do conflict, so we allow them to be specified only by the top-level module.

A module author is therefore in complete control of that module's build when it is the main program being built, but not in complete control of other users' builds that depend on the module. I believe this distinction will make minimal version selection scale to much larger, more distributed code bases than existing systems.

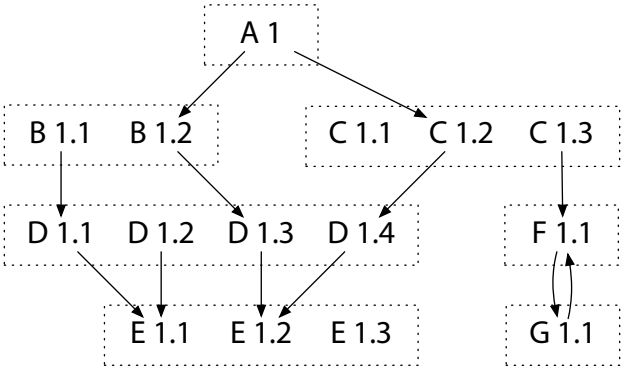
High-Fidelity Builds

Let's return now to the question of high-fidelity builds.

At the start of the post we saw that, using `go get` to build A, it was possible to use dependencies different than the ones A's author had used, without a good reason. I called this as a low-fidelity build, because it is a poor reproduction of the original build of A. Using minimal version selection, builds are instead high-fidelity. The module requirements, which are included with the module's source code, uniquely determine how to build it directly. The user's build of A will match the author's build exactly: a reproducible build. But high-fidelity means more.

Having a reproducible build is generally understood to be a binary property, for a whole-program build: a user's build is exactly the same the author's, or it isn't. What about when building a library module as part of a larger program? It would be helpful for a user's build of a library to match the author's whenever possible. Then the user runs the same code (including dependencies) that the author developed and tested with. In a larger project, of course, it may be impossible for a user's build of a library to match the author's build exactly. Another part of that build may force the use of a newer dependency, making the user's build of the library deviate from the author's build. Let's refer to a build as high-fidelity when it deviates from the author's own build only to satisfy a requirement elsewhere in the build.

Consider again our original example:



In this example, the build of A combines B 1.2 and D 1.4, even though B's author was using D 1.3. That change is necessary because A also uses C 1.2, which requires D 1.4. The build of A is still a high-fidelity build of B 1.2: it deviates by using D 1.4, but only because it must. In contrast, if the build used E 1.3, as `go get -u`, Dep, and Cargo typically do, that build would be low-fidelity: it deviates unnecessarily.

Minimal version selection provides high-fidelity builds by using the oldest version available that meets the requirements. The release of a new version has no effect on the build. 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. The lock file ensures reproducible builds for whole programs, but it is ignored for library modules; the Cargo FAQ explains that this is "precisely because a library should **not** be deterministically recompiled for all users of the library." It's true that a perfect reproduction is not always possible, but by giving up entirely, the Cargo approach admits unnecessary deviation from the library author's builds. That is, it delivers low-fidelity builds. In our example, when A first adds B 1.2 or C 1.2 to its build, Cargo will

see that they require E 1.2 or later and will choose E 1.3. Until directed otherwise, however, it seems better to continue to build with E 1.2, as the authors of B and C did. Using the oldest allowed version also eliminates the redundancy of having two different files (manifest and lock) that both specify which modules versions to use.

Automatically using newer versions also makes it easy for minimum requirements to be wrong. Suppose we start working on A using B 1.1, the latest version at the time, and we record that A requires only B 1.1. But then B 1.2 comes out and we start using it in our own builds and lock file, without updating the manifest. At this point there is no longer any development or testing of A with B 1.1. We may start using features or depending on bug fixes from B 1.2, but now A incorrectly lists its minimum requirement as B 1.1. If users always also choose newer versions than the minimum requirement, then there is not much harm done: they'll use B 1.2 as well. But when the system does try to use the declared minimum, it will break. For example, when a user attempts a limited update of A, the system cannot see that updating to B 1.2 is also required. More generally, whenever the minimum versions (in the manifest) and the built versions (in the lock file) differ, why should we believe that building with the minimum versions will produce a working library? To try to detect this problem, Cargo developers have proposed that `cargo publish` try a build with the minimum versions of all dependencies before publishing. That will detect when A starts using a new feature in B 1.2—building with B 1.1 will fail—but it will not detect when A starts depending on a new bug fix.

The fundamental problem is that preferring the newest allowed version of a module during version selection produces a low-fidelity build. Lock files are a partial solution, targeting whole-program builds; additional build checks like in `cargo publish` are also a partial solution. A more complete solution is to use the version of the module the author did. That makes a user's build as close as possible to the author's build: a high-fidelity build.

Upgrade Speed

Given that minimal version selection takes the minimum allowed version of each dependency, it's easy to think that this would lead to use of very old copies of packages, which in turn might lead to unnecessary bugs or security problems. In practice, however, I think the opposite will happen, because the minimum allowed version is the *maximum* of all the constraints, so the one lever of control made available to all modules in a build is the ability to force the use of a newer version of a dependency than would otherwise be used. I expect that users of minimal version selection will end up with programs that are almost as up-to-date as their friends using more aggressive systems like Cargo.

For example, suppose you are writing a program that depends on a handful of other modules, all of which depend on some very common module, like `gopkg.in/yaml.v2`. Your program's build will use the *newest* YAML version among the ones requested by your module and that handful of dependencies. Even just one conscientious dependency can force your build to update many other dependencies. This is the opposite of the Kubernetes Go client problem I mentioned earlier.

If anything, minimal version selection would instead suffer the opposite problem, that this "max of the minimums" answer serves as a ratchet that forces dependencies forward too quickly. But I think in practice dependencies will move forward at just the right speed, which ends up being just the right amount slower than Cargo and friends.

Upgrade Timing

A key feature of minimal version selection is that upgrade do not happen until a developer asks for them to happen. You don't get an untested version of a module unless you asked for that module to be upgraded.

For example, in Cargo, if package B depends on package C 2.9 and you add B to your build, you don't get C 2.9. You get the latest allowed version at that moment, maybe C 2.15. Maybe C 2.15 was released just a few minutes ago and the author hasn't yet been told about an important bug. That's too bad for you and your build. On the other hand, in minimal version selection, module B's `go.mod` file will list the exact version of C that B's author developed and tested with. You'll get that version. Or maybe some other module in your program developed and tested with a newer version of C. Then you'll get that version. But you will never get a version of C that some module in the program did not explicitly request in its `go.mod` file. This should mean you only ever get a version of C that worked for someone else, not the very latest version that maybe hasn't worked for anyone.

To be clear, my purpose here is not to pick on Cargo, which I think is a very well-designed system. I'm using Cargo here as an example of a model that many developers are familiar with, to try to convey what would be different in minimal version selection.

Minimality

I call this system minimal version selection because the system as a whole appears to be minimal: I don't see how to remove anything more without breaking it. Some people will undoubtedly say that too much has been removed already, but so far it seems perfectly able to handle the real-world cases I've examined. We'll find out more by experimenting with the `vgo` prototype.

The key to minimal version selection is its preference for the minimum allowed version of a module. When I compared `go get -u`'s "upgrade everything to latest" approach to Cargo's "manifest and lock" approach in the context of a system that can rely on the import compatibility rule, I realized that both manifest and lock exist for the same purpose: to work around the "upgrade everything to latest" default behavior. The manifest describes which newer versions are unneeded, and the lock describes which newer versions are unwanted. Instead, why not change the default? Use the minimum version allowed, typically the exact version the author used, and leave timing of upgrades completely to user control. This approach leads to reproducible builds without lock files, and more generally to high-fidelity builds that deviate from the author's own build only when required.

More than anything else, I wanted to find a version selection algorithm that was understandable. Predictable. Boring. Where other systems instead seem to optimize for displays of raw flexibility and power, minimal version selection aims to be invisible. I hope it succeeds.

Reproducible, Verifiable, Verified Builds

Go & Versioning, Part 5

Russ Cox
February 21, 2018

research.swtch.com/vgo-repro

Once both Go developers and tools share a vocabulary around package versions, it's relatively straightforward to add support in the toolchain for reproducible, verifiable, and verified builds. In fact, the basics are already in the vgo prototype.

Since people sometimes disagree about the exact definitions of these terms, let's establish some basic terminology. For this post:

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

Vgo delivers reproducible builds by default. The resulting binaries are verifiable, in that they record versions of the exact source code that went into the build. And it is possible to configure your repository so that users rebuilding your software verify that their builds match yours, using cryptographic hashes, no matter how they obtain the dependencies.

Reproducible Builds

At the very least, we want to make sure that when you build my program, the build system decides to use the same versions of the code. Minimal version selection delivers this property by default. The `go.mod` file alone is enough to uniquely determine which module versions should be used for the build (assuming dependencies are available), and that decision is stable even as new versions of a module are introduced into the ecosystem. This differs from most other systems, which adopt new versions automatically and need to be constrained to yield reproducible builds. I covered this in the minimal version selection post, but it's an important, subtle detail, so I'll try to give a short reprise here.

To make this concrete, let's look at a few real packages from Cargo, Rust's package manager. To be clear, I am not picking on Cargo. I think Cargo is an example of the current state of the art in package managers, and there's much to learn from it. If we can make Go package management as smooth as Cargo's, I'll be happy. But I also think that it is worth exploring whether we would benefit from choosing a different default when it comes to version selection.

Cargo prefers maximum versions in the following sense. Over at crates.io, the latest `toml` is 0.4.5 as I write this post. It lists a dependency on `serde` 1.0 or later; the latest `serde` is 1.0.27. If you start a new project and add a dependency on `toml` 0.4.1 or later, Cargo has a choice to make. According to the constraints, any of 0.4.1, 0.4.2, 0.4.3, 0.4.4, or 0.4.5 would be acceptable. All other things being equal, Cargo prefers the newest acceptable version, 0.4.5. Similarly, any of `serde` 1.0.0 through 1.0.27 are acceptable, and Cargo chooses 1.0.27. These choices change as new versions are introduced. If `serde` 1.0.28 is released tonight and I add toml 0.4.5 to a project tomorrow, I'll get 1.0.28 instead of 1.0.27. As described so far, Cargo's builds are not repeatable. Cargo's (entirely reasonable) answer to this problem is to have not just a constraint file (the manifest, `Cargo.toml`) but also a list of the exact artifacts to use in the build (the

lock file, `Cargo.lock`). The lock file stops future upgrades; once it is written, your build stays on `serde` 1.0.27 even when 1.0.28 is released.

In contrast, minimal version selection prefers the minimum allowed version, which is the exact version requested by some `go.mod` in the project. That answer does not change as new versions are added. Given the same choices in the Cargo example, `vgo` would select `toml` 0.4.1 (what you requested) and then `serde` 1.0 (what `toml` requested). Those choices are stable, without a lock file. This is what I mean when I say that `vgo`'s builds are reproducible by default.

Verifiable Builds

Go binaries have long included a string indicating the version of Go they were built with. Last year I wrote a tool `rsc.io/goversion` that fetches that information from a given executable or tree of executables. For example, on my Ubuntu Linux laptop, I can look to see which system utilities are implemented in Go:

```
$ go get -u rsc.io/goversion
$ goversion /usr/bin
/usr/bin/containerd go1.8.3
/usr/bin/containerd-shim go1.8.3
/usr/bin/ctr go1.8.3
/usr/bin/go go1.8.3
/usr/bin/gofmt go1.8.3
/usr/bin/kbfsfuse go1.8.3
/usr/bin/kbnm go1.8.3
/usr/bin/keybase go1.8.3
/usr/bin/snap go1.8.3
/usr/bin/snapctl go1.8.3
$
```

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. Using our "hello, world" program from the tour:

```
$ go get -u rsc.io/goversion
$ goversion ./hello
./hello go1.10
$ goversion -m hello
./hello go1.10
    path  github.com/you/hello
    mod   github.com/you/hello  (devel)
    dep   golang.org/x/text     v0.0.0-20170915032832-14c0d48ead0c
    dep   rsc.io/quote          v1.5.2
    dep   rsc.io/sampler        v1.3.0
$
```

The main module, supposedly `github.com/you/hello`, has no version information, because it's the local development copy, not a specific version we downloaded. But if instead we build a command directly from a versioned module, then the listing does report versions for all modules:

```
$ vgo build -o hello2 rsc.io/hello
vgo: resolving import "rsc.io/hello"
vgo: finding rsc.io/hello (latest)
vgo: adding rsc.io/hello v1.0.0
vgo: finding rsc.io/hello v1.0.0
vgo: finding rsc.io/quote v1.5.1
vgo: downloading rsc.io/hello v1.0.0
```

```
$ goversion -m ./hello2
./hello2 go1.10
    path  rsc.io/hello
    mod   rsc.io/hello       v1.0.0
    dep   golang.org/x/text  v0.0.0-20170915032832-14c0d48ead0c
    dep   rsc.io/quote       v1.5.2
    dep   rsc.io/sampler     v1.3.0
$
```

When we do integrate versions into the main Go toolchain, we will add APIs to access this information from inside a running binary, just like `runtime.Version` provides access to the more limited Go version information.

For the purpose of attempting to reconstruct the binary, the information listed by `goversion -m` suffices: put the versions into a `go.mod` file and build the target named on the `path` line. But if the result is not the same binary, you might wonder about ways to narrow down what's different. What changed?

When vgo downloads each module, it computes a hash of the file tree corresponding to that module. That hash is also included in the binary, alongside the version information, and `goversion -mh` prints it:

```
$ goversion -mh ./hello
hello go1.10
    path  github.com/you/hello
    mod   github.com/you/hello  (devel)
    dep   golang.org/x/text     v0.0.0-20170915032832-14c0d48ead0c  h1:qgOY6WgZOaTkIIMiVjBQcw93ER...
    dep   rsc.io/quote          v1.5.2                             h1:w5fcysjrx7yqtD/aO+QwRjYZOK...
    dep   rsc.io/sampler        v1.3.1                             h1:F0c3J2nQCdk9ODsNhU3sElnvPI...
$ goversion -mh ./hello2
hello go1.10
    path  rsc.io/hello
    mod   rsc.io/hello       v1.0.0                             h1:CDmhdOARcor1WuRUvmE46PK91ahrS...
    dep   golang.org/x/text  v0.0.0-20170915032832-14c0d48ead0c  h1:qgOY6WgZOaTkIIMiVjBQcw93ERBE4...
    dep   rsc.io/quote       v1.5.2                             h1:w5fcysjrx7yqtD/aO+QwRjYZOKnaM...
    dep   rsc.io/sampler     v1.3.0                             h1:7uVkIFmeBqHfdjD+gZwtXXI+RODJ2...
$
```

The `h1:` prefix indicates which hash is being reported. Today, there is only "hash 1," a SHA-256 hash of a list of files and the SHA-256 hashes of their contents. If we need to update to a new hash later, the prefix will help us tell old from new hashes.

I must stress that these hashes are self-reported by the build system. If someone gives you a binary with certain hashes in its build information, there's no guarantee they are accurate. They are very useful information supporting a later verification, not a signature that can be trusted by themselves.

Verified Builds

An author distributing a program in source form might want to let users verify that they are building it with exactly the expected dependencies. We know vgo will make the same decisions about which versions of dependencies to use, but there is still the problem of mapping a version like v1.5.2 to an actual source tree. What if the author of v1.5.2 changes the tag to point at a different file tree? What if a malicious middlebox intercepts the download request and delivers a different zip file? What if the user has accidentally edited the source files in the local copy of v1.5.2? The vgo prototype supports this kind of verification too.

The final form may be somewhat different, but if you create a file named `go.modverify` next to `go.mod`, then builds will keep that file up-to-date with known hashes for specific versions of modules:

```
$ echo >go.modverify
$ vgo build
$ tcat go.modverify  # go get rsc.io/tcat, or use cat
golang.org/x/text  v0.0.0-20170915032832-14c0d48ead0c  h1:qgOY6WgZOaTkIIMiVjBQcw93ERBE4m30iBm00nk...
rsc.io/quote       v1.5.2                              h1:w5fcysjrx7yqtD/aO+QwRjYZOKnaM9Uh2b40tEl...
rsc.io/sampler     v1.3.0                              h1:7uVkIFmeBqHfdjD+gZwtXXI+RODJ2Wc4O7MPEh/...
$
```

The `go.modverify` file is a log of the hash of all versions ever encountered: lines are only added, never removed. If we update `rsc.io/sampler` to v1.3.1, then the log will now contain hashes for both versions:

```
$ vgo get rsc.io/sampler@v1.3.1
$ tcat go.modverify
golang.org/x/text  v0.0.0-20170915032832-14c0d48ead0c  h1:qgOY6WgZOaTkIIMiVjBQcw93ERBE4m30iBm00nk...
rsc.io/quote       v1.5.2                              h1:w5fcysjrx7yqtD/aO+QwRjYZOKnaM9Uh2b40tEl...
rsc.io/sampler     v1.3.0                              h1:7uVkIFmeBqHfdjD+gZwtXXI+RODJ2Wc4O7MPEh/...
rsc.io/sampler     v1.3.1                              h1:F0c3J2nQCdk9ODsNhU3sElnvPIxM/xV1c/qZuAe...
$
```

When `go.modverify` exists, `vgo` checks that all downloaded modules used in a given build are consistent with entries already in the file. For example, if we change the first digit of the `rsc.io/quote` hash from `w` to `v`:

```
$ vgo build
vgo: verifying rsc.io/quote v1.5.2: module hash mismatch
    downloaded:   h1:w5fcysjrx7yqtD/aO+QwRjYZOKnaM9Uh2b40tElTs3Y=
    go.modverify: h1:v5fcysjrx7yqtD/aO+QwRjYZOKnaM9Uh2b40tElTs3Y=
$
```

Or suppose we fix that one but then modify the v1.3.0 hash. Now our build succeeds, because v1.3.0 is not being used by the build, so its line is (correctly) ignored. But if we try to downgrade to v1.3.0, then the build verification will correctly begin failing:

```
$ vgo build
$ vgo get rsc.io/sampler@v1.3.0
vgo: verifying rsc.io/sampler v1.3.0: module hash mismatch
    downloaded:   h1:7uVkIFmeBqHfdjD+gZwtXXI+RODJ2Wc4O7MPEh/QiW4=
    go.modverify: h1:8uVkIFmeBqHfdjD+gZwtXXI+RODJ2Wc4O7MPEh/QiW4=
$
```

Developers who want to ensure that others rebuild their program with exactly the same sources they did can store a `go.modverify` in their repository. Then others building using the same repo will automatically get verified builds. For now, only the `go.modverify` in the top-level module of the build applies. But note that `go.modverify` lists all dependencies, including indirect dependencies, so the whole build is verified.

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. It is also useful to check that downloaded modules have not changed on the local machine since it was downloaded. This is less about security from attacks and more about avoiding mistakes. For example, because source file paths appear in stack traces, it's common to open those files when debugging. If you acciden-

tally (or, I suppose, intentionally) modify the file during the debugging session, it would be nice to be able to detect that later. The vgo `verify` command does this:

```
$ go get -u golang.org/x/vgo  # fixed a bug, sorry! :-)
$ vgo verify
all modules verified
$
```

If a source file changes, vgo `verify` notices:

```
$ echo >>$GOPATH/src/v/rsc.io/quote@v1.5.2/quote.go
$ vgo verify
rsc.io/quote v1.5.2: dir has been modified (/Users/rsc/src/v/rsc.io/quote@v1.5.2)
$
```

If we restore the file, all is well:

```
$ gofmt -w $GOPATH/src/v/rsc.io/quote@v1.5.2/quote.go
$ vgo verify
all modules verified
$
```

If cached zip files are modified after download, vgo `verify` notices that too, although I can't plausibly explain how that might happen:

```
$ zip $GOPATH/src/v/cache/rsc.io/quote/@v/v1.5.2.zip /etc/resolv.conf
  adding: etc/resolv.conf (deflated 36%)
$ vgo verify
rsc.io/quote v1.5.2: zip has been modified (/Users/rsc/src/v/cache/rsc.io/quote/@v/v1.5.2.zip)
$
```

Because vgo keeps the original zip file after unpacking it, if vgo `verify` decides that only one of the zip file and the directory tree have been modified, it could even print a diff of the two.

What's Next?

This is implemented already in vgo. You can try it out and use it. As with the rest of vgo, feedback about what doesn't work right (or works great) is appreciated.

The functionality presented here is more the start of something than a finished feature. A cryptographic hash of the file tree is a building block. The `go.modverify` built on top of it checks that developers all build a particular module with precisely the same dependencies, but there's no verification when downloading a new version of a module (unless someone else already added it to `go.modverify`), nor is there any sharing of expected hashes between modules.

The exact details of how to fix those two shortcomings are not obvious. It may make sense to allow some kind of cryptographic signatures of the file tree, and to verify that an upgrade finds a version signed with the same key as the previous version. Or it may make sense to adopt an approach along the lines of The Update Framework (TUF), although using their network protocols directly is not practical. Or, instead of using per-repo `go.modverify` logs, it might make sense to establish some kind of shared global log, a bit like Certificate Transparency, or to use a public identity server like Upspin. There are many avenues we might explore, but all this is getting a little ahead of ourselves. For now we are focused on successfully integrating versioning into the go command.

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, beacuse 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.

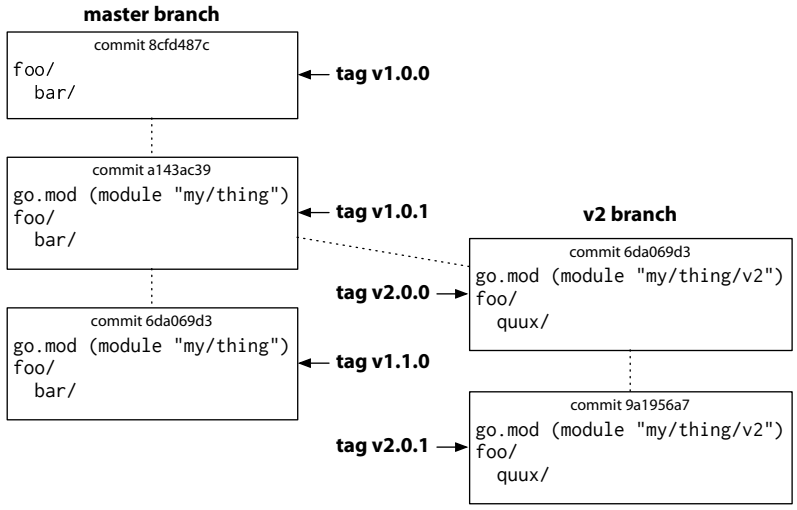
From Repository to Modules

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:

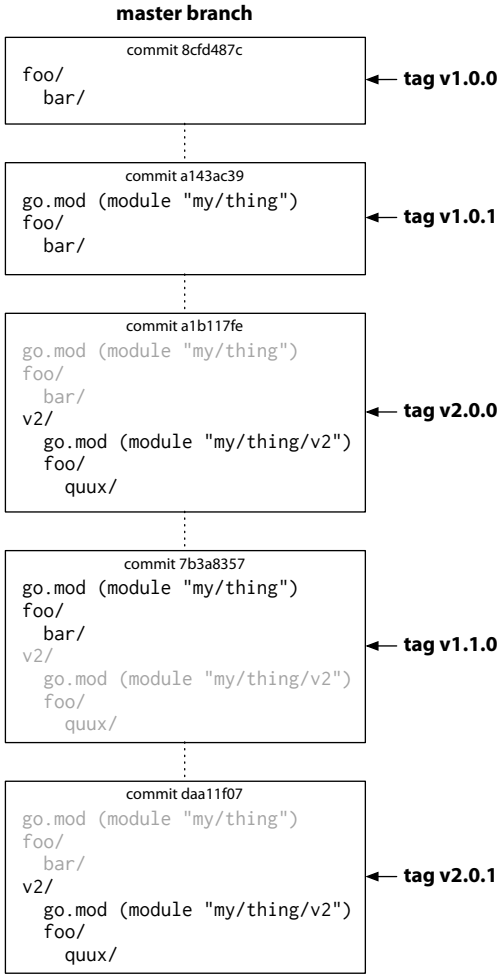


Go module using major branches

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 distinguishes 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 souce 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/gomodules/rsc">
Nothing to see here.
$
```

The `mod` server type indicates that modules are served in a file tree at that base URL. The relevant files at *storage.googleapis.com/gomodules/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.

Versioned Go Commands

Go & Versioning, Part 7

Russ Cox

February 23, 2018

research.swtch.com/vgo-cmd

What does it mean to add versioning to the go command? The overview post gave a preview, but the followup posts focused mainly on underlying details: the import compatibility rule, minimal version selection, and defining go modules. With those better understood, this post examines the details of how versioning affects the go command line and the reasons for those changes.

The major changes are:

- 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.
- The `go get` command will serve mainly to change which version of a package should be used in future build commands.
- The `go list` command will add access to module information.
- A new `go release` command will automate some of the work a module author should do when tagging a new release, such as checking API compatibility.
- 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.

All these changes are implemented in the `vgo` prototype.

Deciding exactly how a build system should work is hard. The introduction of new build caching in Go 1.10 prompted some important, difficult decisions about the meaning of go commands, and the introduction of versioning does too. Before I explain some of the decisions, I want to start by explaining a guiding principle that I've found helpful recently, which I call the isolation rule:

> *The result of a build command should depend only on the source files that are its logical inputs, never on hidden state left behind by previous build commands.)*
>
> *That is, what a command does in isolation—on a clean system loaded with only the relevant input source files—is what it should do all the time, no matter what else has happened on the system recently.*

To see the wisdom of this rule, let me retell an old build story and show how the isolation rule explains what happened.

An Old Build Story

Long ago, when compilers and computers were very slow, developers had scripts to build their whole programs from scratch, but if they were just modifying one source file, they might save time by manually recompiling just that file and then relinking the overall program, avoiding the cost of recompiling all the source files that hadn't changed. These manual incremental builds were fast but error-prone: if you forgot to recompile a source file that you'd modified, the link of the final executable would use an out-of-date object file, the executable would demonstrate buggy behavior, and you might spend a long time staring at the (correct!) source code looking for a bug that you'd already fixed.

Stu Feldman once explained what it was like in the early 1970s when he spent a few months working on a few-thousand-line Ratfor program:

> I would go home for dinner at six or so, recompile the whole world in the background, shut up, and then drive home. It would take through the drive home and through dinner for anything to happen. This is because I kept making the classic error of debugging a correct program, because you'd forget to compile the change.

Transliterated to modern C tools (instead of Ratfor), Feldman would work on a large program by first compiling it from scratch:

```
$ rm -f *.o && cc *.c && ld *.o
```

This build follows the isolation rule: starting from the same source files, it produces the same result, no matter what else has been run in that directory.

But then Feldman would make changes to specific source files and recompile only the modified ones, to save time:

```
$ cc r2.c r3.c r5.c && ld *.o
```

This incremental build does not follow the isolation rule. The correctness of the command depends on Feldman remembering which files they modified, and it's easy to forget one. But it was so much faster, everyone did it anyway, resorting to routines like Feldman's daily "build during dinner" to correct any mistakes.

Feldman continued:

> Then one day, Steve Johnson came storming into my office in his usual way, saying basically, "Goddamn it, I just spent the whole morning debugging a correct program, again. Why doesn't anybody do something like this? ..."

And that's the story of how Stu Feldman invented `make`.

`Make` was a major advance because it provided fast, incremental builds that followed the isolation rule. Isolation is important because it means the build is properly abstracted: only the source code matters. As a developer, you can make changes to source code and not even think about details like stale object files.

However, the isolation rule is never an absolute. There is always some area where it applies, which I call the abstraction zone. When you step out of the abstraction zone, you are back to needing to keep state in your head. For `make`, the abstraction zone is a single directory. If you are working on a program made up of libraries in multiple directories, traditional `make` is no help. Most Unix programs in the 1970s fit in a single directory, so it just wasn't important for `make` to provide isolation semantics in multi-directory builds.

Go Builds and the Isolation Rule

One way to view the history of design bug fixes in the `go` command is a sequence of steps extending its abstraction zone to better match developer expectations.

One of the advances of the `go` command was correct handling of source code spread across multiple directories, extending the abstraction zone beyond what `make` provided. Go programs are almost always spread across multiple directories, and when we used `make` it was very common to forget to install a package in one directory before trying to use it in another directory. We were all too familiar with "the classic error of debugging a correct program." But even after fixing that, there were still many ways to step out of the `go` command's abstraction zone, with unfortunate consequences.

To take one example, if you had multiple directory trees listed in GOPATH,

builds in one tree blindly assumed that installed packages in the others were up-to-date if present, but it would rebuild them if missing. This violation of the isolation rule caused no end of mysterious problems for projects using godep, which used a second GOPATH entry to simulate vendor directories. We fixed this in Go 1.5.

As another example, until very recently command-line flags were not part of the abstraction zone. If you start with a standard Go 1.9 distribution and run

```
$ go build hello.go
$ go install -a -gcflags=-N std
$ go build hello.go
```

the second `go build` command produces a different executable than the first. The first `hello` is linked against an optimized build of the Go and standard library, while the second `hello` is linked against an unoptimized standard library. This violation of the isolation rule led to widespread use of `go build -a` (always rebuild everything), to reestablish isolation semantics. We fixed this in Go 1.10.

In both cases, the `go` command was "working as designed." These were the kinds of details that we always kept mental track of when using other build systems, so it seemed reasonable to us not to abstract them away. In fact, when I designed the behavior, I thought it was feature that

```
$ go install -a -gcflags=-N std
$ go build hello.go
```

let you build an optimized `hello` against an unoptimized standard library, and I sometimes took advantage of that. But, on the whole, Go developers disagreed. They did not expect to, nor want to, keep mental track of that state. For me, the isolation rule is useful because it gives a simple test that helps me cut through any mental contamination left by years of using less capable build systems: every command should have only one meaning, no matter what other commands have preceded it.

The isolation rule implies that some commands may need to be made more complex, so one command can serve where two commands did before. For example, if you follow the isolation rule, how *do* you build an optimized `hello` against an unoptimized standard library? We answered this in Go 1.10 by extending the `-gcflags` argument to start with an optional pattern that controls which packages the flags affect. To build an optimized hello against an unoptimized standard library, `go build -gcflags=std=-N hello.go`.

The isolation rule also implies that previously context-dependent commands need to settle on one context-independent meaning. A good general rule seems to be to use the one meaning that developers are most familiar with. For example, a different variation of the flag problem is:

```
$ go build -gcflags=-N hello.go
$ rm -rf $GOROOT/pkg
$ go build -gcflags=-N hello.go
```

In Go 1.9, the first `go build` command builds an unoptimized hello against the preinstalled, optimized standard library. The second `go build` command finds no preinstalled standard library, so it rebuilds the standard library, and the `-gcflags` applies to all packages built during the command, so the result is an unoptimized hello built against an unoptimized standard library. For Go 1.10, we had to choose which meaning is the one true meaning.

Our original thought was that in the absence of a restricting pattern like `std=`, the `-gcflags=-N` should apply to all packages in the build, so that this command would always build an unoptimized hello against an unoptimized standard library. But most developers expect this command to apply the `-`

gcflags=-N only to the argument of go build, namely hello.go, because that's how it works in the common case, when you have *not* just deleted $GOROOT/pkg. We decided to preserve this expectation, defining that when no pattern is given, the flags apply only to the packages or files named on the build comamnd line. In Go 1.10, building hello.go with -gcflags=-N always builds an unoptimized hello against an optimized standard library, even if $GOROOT/pkg has been deleted and the standard library must be rebuilt on the spot. If you do want a completely unoptimized build, that's -gcflags=all=-N.

The isolation rule is also helpful for thinking through the design questions that arise in a versioned go command. Like in the flag decisions, some commands need to be made more capable. Others have multiple meanings now and must be reduced to a single meaning.

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).

Suppose I have a brand new Go 1.10 installation and I write this program to hello.go:

```
package main

import (
    "fmt"
    "rsc.io/quote"
)

func main() {
    fmt.Println(quote.Hello())
}
```

This fails:

```
$ go run hello.go
hello.go:5: import "rsc.io/quote": import not found
$
```

But this succeeds:

```
$ go get rsc.io/quote
$ go run hello.go
Hello, world.
$
```

I can explain this. After eight years of conditioning by use of goinstall and go get, it seemed obvious to me that this behavior was correct: go get downloads rsc.io/quote for us and stashes it away for use by future commands, so *of course* that must happen before go run. But I can explain the behavior of the optimization flag examples in the previous section too, and until a few months ago they also seemed obviously correct. After more thought, I now believe that any go command should be able to download versioned dependencies as needed. I changed my mind for a few reasons.

The first reason is the isolation rule. The fact that every other design mistake I've made in the go command violated the isolation rule strongly suggests that requiring a prepatory go get is a mistake too.

The second reason is that I've found it helpful to think of the downloaded versioned source code as living in a local cache that developers shouldn't need

to think about at all. If it's really a cache, cache misses can't be failures.

The third reason is the mental bookkeeping required. Today's go command expects developers to keep track of which packages are and are not downloaded, just as earlier go commands expected developers to keep track of which compiler flags had been used during the most recent package installs. As programs grow and as we add more precision about versioning, the mental burden will grow, even though the go command is already tracking the same information. For example, I think this hypothetical session is a suboptimal developer experience:

```
$ git clone https://github.com/rsc/hello
$ cd hello
$ go build
go: rsc.io/sampler(v1.3.1) not installed
$ go get
go: installing rsc.io/sampler(v1.3.1)
$ go build
$
```

If the command knows exactly what it needs, why make the user do it?

The fourth reason is that build systems in other languages already do this. When you check out a Rust repo and build it, `cargo build` automatically fetches the dependencies as part of the build, no questions asked.

The fifth reason is that downloading on demand allows downloading lazily, which in large programs may mean not downloading many dependencies at all. For example, the popular logging package `github.com/sirupsen/logrus` depends on `golang.org/x/sys`, but only when building on Solaris. The eventual `go.mod` file in `logrus` would list a specific version of `x/sys` as a dependency. When vgo sees `logrus` in a project, it will consult the `go.mod` file and determine which version satisfies an `x/sys` import. But all the users not building for Solaris will never see an `x/sys` import, so they can avoid the download of `x/sys` entirely. This optimization will become more important as the dependency graph grows.

I do expect resistance from developers who aren't yet ready to think about builds that download code on demand. We may need to make it possible to disable that with an environment variable, but downloads should be enabled by default.

Changing Versions (go get)

Plain go `get`, without `-u`, violates the command isolation rule and must be fixed. Today:

- If GOPATH is empty, go `get rsc.io/quote` downloads and builds the latest version of `rsc.io/quote` and its dependencies (for example, `rsc.io/sampler`).
- If there is already a `rsc.io/quote` in GOPATH, from a go `get` last year, then the new go `get` builds the old version.
- If `rsc.io/sampler` is already in GOPATH but `rsc.io/quote` is not, then go `get` downloads the latest `rsc.io/quote` and builds it against the old copy of `rsc.io/sampler`.

Overall, go `get` depends on the state of GOPATH, which breaks the command isolation rule. We need to fix that. Since go `get` has at least three meanings today, we have some latitude in defining new behavior. Today, `vgo get` fetches the latest version of the named modules but then the exact versions of any dependencies requested by those modules, subject to minimal version selec-

tion. For example, `vgo get rsc.io/quote` always fetches the latest version of `rsc.io/quote` and then builds it with the exact version of `rsc.io/sampler` that `rsc.io/quote` has requested.

Vgo also allows module versions to be specified on the command line:

```
$ vgo get rsc.io/quote@latest  # default
$ vgo get rsc.io/quote@v1.3.0
$ vgo get rsc.io/quote@'<v1.6' # finds v1.5.2
```

All of these also download (if not already cached) the specific version of `rsc.io/sampler` named in `rsc.io/quote`'s `go.mod` file. These commands modify the current module's `go.mod` file, and in that sense they do influence the operation of future commands. But that influence is through an explicit file that users are expected to know about and edit, not through hidden cache state. Note that if the version requested on the command line is earlier than the one already in `go.mod`, then `vgo get` does a downgrade, which will also downgrade other packages if needed, again following minimal version selection.

In contrast to plain `go get`, the `go get -u` command behaves the same no matter what the state of the GOPATH source cache: it downloads the latest copy of the named packages and the latest copy of all their dependencies. Since it follows the command isolation rule, we should keep the same behavior: `vgo get -u` upgrades the named modules to their latest versions and also upgrades all of their dependencies.

One idea that has come up in past few days is to introduce a mode halfway between `vgo get` (download the exact dependencies of the thing I asked for) and `vgo get -u` (download the latest dependencies). If we believe that authors are conscientious about being very careful with patch releases and only using them for critical, safe fixes, then it might make sense to have a `vgo get -p` that is like `vgo get` but then applies only patch-level upgrades. For example, if `rsc.io/quote` requires `rsc.io/sampler` v1.3.0 but v1.3.1 and v1.4.0 are also available, then `vgo get -p rsc.io/quote` would upgrade `rsc.io/sampler` to v1.3.1, not v1.4.0. If you think this would be useful, please let us know.

Of course, all the `vgo get` variants record the effect of their additions and upgrades in the `go.mod` file. In a sense, we've made these commands follow the isolation rule by introducing `go.mod` as an explicit, visible input replaces a previously implicit, hidden input: the state of the entire GOPATH.

Module Information (`go list`)

In addition to changing the versions being used, we need to provide some way to inspect the current ones. The `go list` command is already in charge of reporting useful information:

```
$ go list -f {{.Dir}} rsc.io/quote
/Users/rsc/src/rsc.io/quote
$ go list -f {{context.ReleaseTags}}
[go1.1 go1.2 go1.3 go1.4 go1.5 go1.6 go1.7 go1.8 go1.9 go1.10]
$
```

It probably makes sense to make module information available to the format template, and we should also provide shorthands for common operations like listing all the current module's dependencies. The `vgo` prototype already provides correct information for packages in dependency modules. For example:

```
$ vgo list -f {{.Dir}} rsc.io/quote
/Users/rsc/src/v/rsc.io/quote@v1.5.2
$
```

It also has a few shorthands. First, vgo `list -t` lists all available tagged versions of a module:

```
$ vgo list -t rsc.io/quote
rsc.io/quote
    v1.0.0
    v1.1.0
    v1.2.0
    v1.2.1
    v1.3.0
    v1.4.0
    v1.5.0
    v1.5.1
    v1.5.2
$
```

Second, vgo `list -m` lists the current module followed by its dependencies:

```
$ vgo list -m
MODULE                VERSION
github.com/you/hello  -
golang.org/x/text     v0.0.0-20170915032832-14c0d48ead0c
rsc.io/quote          v1.5.2
rsc.io/sampler        v1.3.0
$
```

Finally, vgo `list -m -u` adds a column showing the latest version of each module:

```
$ vgo list -m -u
MODULE                VERSION                              LATEST
github.com/you/hello  -                                    -
golang.org/x/text     v0.0.0-20170915032832-14c0d48ead0c  v0.0.0-20180208041248-4e4a3210bb54
rsc.io/quote          v1.5.2 (2018-02-14 10:44)            -
rsc.io/sampler        v1.3.0 (2018-02-13 14:05)            v1.99.99 (2018-02-13 17:20)
$
```

In the long term, these should be shorthands for more general support in the format template, so that other programs can obtain the information in other forms. Today they are just special cases.

Preparing New Versions (`go release`)

We want to encourage authors to issue tagged releases of their modules, so we need to make that as easy as possible. We intend to add a `go release` command that can take care of as much of the bookkeeping as needed. For example, it might:

- Check for backwards-incompatible type changes, compared to the previous release. We run a check like this when working on the Go standard library, and it is very helpful.
- Suggest whether this release should be a new point release or a new minor release (because there's new API or because many lines of code have changed). Or perhaps always suggest a new minor release unless the author asks for a point release, to keep a potential `go get -p` useful.

– Scan all source files in the module, even ones that aren't normally built, to make sure that all imports can be satisfied by the requirements listed in `go.mod`. Referring back to the example in the download section, this check would make sure that `logrus`'s `go.mod` lists `x/sys`.

As new best practices for releases arise, we can add them to `go release` so that authors always only have one step to check whether their module is ready for a new release.

Pattern matching

Most `go` commands take a list of packages as arguments, and that list can include patterns, like `rsc.io/...` (all packages with import paths beginning with `rsc.io/`), or `./...` (all packages in the current directory or subdirectories), or `all` (all packages). We need to check that these make sense in the new world of modules.

Originally, patterns did not treat vendor directories specially, so that if `github.com/you/hello/vendor/rsc.io/quote` existed, then `go test github.com/you/hello/...` matched and tested it, as did `go test ./...` when working in the `hello` source directory. The argument in favor of matching vendored code was that doing so avoided a special case and that it was actually useful to test your dependencies, as configured in your project, along with the rest of your project. The argument against matching vendored code was that many developers wanted an easy way to test just the code in their projects, assuming that dependencies have already been tested separately and are not changing. In Go 1.9, respecting that argument, we changed the `...` pattern not to walk into `vendor` directories, so that `go test github.com/you/hello/...` does not test vendored dependencies. This sets up nicely for `vgo`, which naturally would not match dependencies either, since they no longer live in a subdirectory of the main project. That is, there is no change in the behavior of `...` patterns when moving from `go` to `vgo`, because that change happened from Go 1.8 to Go 1.9 instead.

That leaves the pattern `all`. When we first wrote the `go` command, before `goinstall` and `go get`, it made sense to talk about building or testing "all packages." Today, it makes much less sense: most developers work in a GOPATH that has a mix of many different things, including many packages downloaded and forgotten about. I expect that almost no one runs commands like `go install all` or `go test all` anymore: it catches too many things that don't matter. The real problem is that `go test all` violates the isolation rule: its meaning depends on the implicit state of GOPATH set up by previous commands, so no one depends on its meaning anymore. In the `vgo` prototype, we have redefined `all` to have a single, consistent meaning: all the packages in the current module, plus all the packages they depend on through one a sequence of one or more imports.

The new `all` is exactly the packages a developer would need to test in order to sanity check that a particular combination of dependency versions work together, but it leaves out nearby packages that don't matter in the current module. For example, in the overview post, our `hello` module imported `rsc.io/quote` but not any other packages, and in particular not the buggy package `rsc.io/quote/buggy`. Running `go test all` in the `hello` module tests all packages in that module and then also `rsc.io/quote`. It omits `rsc.io/quote/buggy`, because that one is not needed, even indirectly, by the `hello` module, so it's irrelevant to test. This definition of `all` restores repeatability, and combined with Go 1.10's test caching, it should make `go test all` more useful than it ever has been.

Working outside GOPATH

If there can be multiple versions of a package with a given import path, then it no longer makes sense to require the active development version of that package to reside in a specific directory. What if I need to work on bug fixes for both v1.3 and v1.4 at the same time? Clearly it must be possible to check out modules in different locations. In fact, at that point there's no need to work in GOPATH at all.

GOPATH was doing three things: it defined the versions of dependencies (now in `go.mod`), it held the source code for those dependencies (now in a separate cache), and it provided a way to infer the import path for code in a particular directory (remove the leading `$GOPATH/src`). As long as we have some mechanism to decide the import path for the code in the current directory, we can stop requiring that developers work in GOPATH. That mechanism is the `go.mod` file's `module` directive. If I'm a directory named `buggy` and `../go.mod` says:

```
module "rsc.io/quote"
```

then my directory's import path must be `rsc.io/quote/buggy`.

The `vgo` prototype enables work outside GOPATH today, as the examples in the overview post showed. In fact, when inferring a `go.mod` from other dependency information, `vgo` will look for import comments in the current directory or subdirectories to try to get its bearings. For example, this worked even before Upspin had introduced a `go.mod` file:

```
$ cd $HOME
$ git clone https://github.com/upspin/upspin
$ cd upspin
$ vgo test -short ./...
```

The `vgo` command inferred from import comments that the module is named `upspin.io`, and it inferred a list of dependency version requirements from `Gopkg.lock`.

What's Next?

This is the last of my initial posts about the `vgo` design and prototype. There is more to work out, but inflicting 67 pages of posts on everyone seems like enough for one week.

I had planned to post a FAQ today and submit a Go proposal Monday, but I will be away next week after Monday. Rather than disappear for the first four days of official proposal discussion, I think I will post the proposal when I return. Please continue to ask questions on the mailing list threads or on these posts and to try the `vgo` prototype.

Thanks very much for all your interest and feedback so far. It's very important to me that we all work together to produce something that works well for Go developers and that is easy for us all to switch to.

Update, March 20, 2018: The official Go proposal is at https://golang.org/issue/24301, and the second comment on the issue will be the FAQ.

The vgo proposal is accepted. Now what?

Go & Versioning, Part 8

Russ Cox

May 29, 2018

research.swtch.com/vgo-accepted

Last week, the proposal review committee accepted the "vgo approach" elaborated on this blog in February and then summarized as proposal #24301. There has been some confusion about exactly what that means and what happens next.

In general, a Go proposal is a discussion about whether to adopt a particular approach and move on to writing, reviewing, and releasing a production implementation. Accepting a proposal does not mean the implementation is complete. (In some cases there is no implementation yet at all!) Accepting a proposal only means that we believe the design is appropriate and that the production implementation can proceed and be committed and released. Inevitably we find details that need adjustment during that process.

Vgo as it exists today is not the final implementation. It is a prototype to make the ideas concrete and to make it possible to experiment with the approach. Bugs and design flaws will necessarily be found and fixed as we move toward making it the official approach in the go command. For example, the original vgo prototype downloaded code from sites like GitHub using their APIs, for better efficiency and to avoid requiring users to have every possible version control system installed. Unfortunately, the GitHub API is far more restrictively rate-limited than plain `git` access, so the current vgo implementation has gone back to invoking `git`. Although we'd still like to move away from version control as the default mechanism for obtaining open source code, we won't do that until we have a viable replacement ready, to make any transition as smooth as possible.

More generally, the key reason for the vgo proposal is to add a common vocabulary and semantics around versions of Go code, so that developers and all kinds of tools can be precise when talking to each other about exactly which program should be built, run, or analyzed. Accepting the proposal is the beginning, not the end.

One thing I've heard from many people is that they want to start using vgo in their company or project but are held back by not having support for it in the toolchains their developers are using. The fact that vgo is integrated deeply into the go command, instead of being a separate vendor directory-writer, introduces a chicken-and-egg problem. To address that problem and make it as easy as possible for developers to try the vgo approach, we plan to include vgo functionality as an experimental opt-in feature in Go 1.11, with the hope of incorporating feedback and finalizing the feature for Go 1.12. (This rollout is analogous to how we included vendor directory functionality as an experimental opt-in feature in Go 1.5 and turned it on by default in Go 1.6.) We also plan to make minimal changes to legacy `go get` so that it can obtain and understand code written using vgo conventions. Those changes will be included in the next point release for Go 1.9 and Go 1.10.

One thing I've heard from zero people is that they wish my blog posts were longer. The original posts are quite dense and a number of important points are more buried than they should be. This post is the first of a series of much shorter posts to try to make focused points about specific details of the vgo design, approach, and process.

What is Software Engineering?

Go & Versioning, Part 9

Russ Cox
May 30, 2018

research.swtch.com/vgo-eng

Nearly all of Go's distinctive design decisions were aimed at making software engineering simpler and easier. We've said this often. The canonical reference is Rob Pike's 2012 article, "Go at Google: Language Design in the Service of Software Engineering." But what is software engineering?

> *Software engineering is what happens to programming*
> *when you add time and other programmers.*

Programming means getting a program working. You have a problem to solve, you write some Go code, you run it, you get your answer, you're done. That's programming, and that's difficult enough by itself. But what if that code has to keep working, day after day? What if five other programmers need to work on the code too? Then you start to think about version control systems, to track how the code changes over time and to coordinate with the other programmers. You add unit tests, to make sure bugs you fix are not reintroduced over time, not by you six months from now, and not by that new team member who's unfamiliar with the code. You think about modularity and design patterns, to divide the program into parts that team members can work on mostly independently. You use tools to help you find bugs earlier. You look for ways to make programs as clear as possible, so that bugs are less likely. You make sure that small changes can be tested quickly, even in large programs. You're doing all of this because your programming has turned into software engineering.

(This definition and explanation of software engineering is my riff on an original theme by my Google colleague Titus Winters, whose preferred phrasing is "software engineering is programming integrated over time." It's worth seven minutes of your time to see his presentation of this idea at CppCon 2017, from 8:17 to 15:00 in the video.)

As I said earlier, nearly all of Go's distinctive design decisions have been motivated by concerns about software engineering, by trying to accommodate time and other programmers into the daily practice of programming.

For example, most people think that we format Go code with `gofmt` to make code look nicer or to end debates among team members about program layout. But the most important reason for `gofmt` is that if an algorithm defines how Go source code is formatted, then programs, like `goimports` or `gorename` or `go fix`, can edit the source code more easily, without introducing spurious formatting changes when writing the code back. This helps you maintain code over time.

As another example, Go import paths are URLs. If code said `import "uuid"`, you'd have to ask which `uuid` package. Searching for `uuid` on godoc.org turns up dozens of packages. If instead the code says `import "github.com/pborman/uuid"`, now it's clear which package we mean. Using URLs avoids ambiguity and also reuses an existing mechanism for giving out names, making it simpler and easier to coordinate with other programmers.

Continuing the example, Go import paths are written in Go source files, not in a separate build configuration file. This makes Go source files self-contained, which makes it easier to understand, modify, and copy them. These decisions, and more, were all made with the goal of simplifying software engineering.

In later posts I will talk specifically about why versions are important for soft-

ware engineering and how software engineering concerns motivate the design changes from dep to vgo.

Why Add Versions To Go?

Go & Versioning, Part 10

Russ Cox

June 7, 2018

research.swtch.com/vgo-why-versions

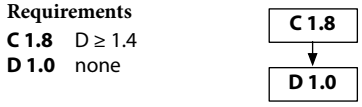
People sometimes ask me why we should add package versions to Go at all. Isn't Go doing well enough without versions? Usually these people have had a bad experience with versions in another language, and they associate versions with breaking changes. In this post, I want to talk a little about why we do need to add support for package versions to Go. Later posts will address why we won't encourage breaking changes.

The `go get` command has two failure modes caused by ignorance of versions: it can use code that is too old, and it can use code that is too new. For example, suppose we want to use a package D, so we run `go get D` with no packages installed yet. The `go get` command will download the latest copy of D (whatever `git clone` brings down), which builds successfully. To make our discussion easier, let's call that D version 1.0 and keep D's dependency requirements in mind (and in our diagrams). But remember that while we understand the idea of versions and dependency requirements, `go get` does not.

```
$ go get D
```

Now suppose that a month later, we want to use C, which happens to import D. We run `go get C`. The `go get` command downloads the latest copy of C, which happens to be C 1.8 and imports D. Since `go get` already has a downloaded copy of D, it uses that one instead of incurring the cost of a fresh download. Unfortunately, the build of C fails: C is using a new feature from D introduced in D 1.4, and `go get` is reusing D 1.0. The code is too old.

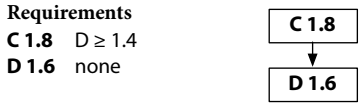
```
$ go get C
```



broken!

Next we try running `go get -u`, which downloads the latest copy of all the code involved, including code already downloaded.

```
$ go get -u C
```



(C 1.8 was tested with D 1.4 and has an unexpected incompatibility with D 1.6.)

broken!

Unfortunately, D 1.6 was released an hour ago and contains a bug that breaks C. Now the code is too new. Watching this play out from above, we know what `go get` needs to do: use D ≥ 1.4 but not D 1.6, so maybe D 1.4 or D 1.5. It's very difficult to tell `go get` that today, since it doesn't understand the concept of a package version.

Getting back to the original question in the post, *why add versions to Go?*

Because agreeing on a versioning system—a syntax for version identifiers, along with rules for how to order and interpret them—establishes a way for us to communicate more precisely with our tools, and with each other, about which copy of a package we mean. Versioning matters for correct builds, as we just saw, but it enables other interesting tools too.

For example, the obvious next step is to be able to list which versions of a package are being used in a given build and whether any of them have updates available. Generalizing that, it would be useful to have a tool that examines a list of builds, perhaps all the targets built at a given company, and assembles the same list. Such a list of versions can then feed into compliance checks, queries into bug databases, and so on. Embedding the version list in a built binary would even allow a program to make these checks on its own behalf while it runs. These all exist for other systems already, of course: I'm not claiming the ideas are novel. The point is that establishing agreement on a versioning system enables all these tools, which can even be built outside the language toolchain.

We can also move from query tools, which tell you about your code, to development tools, which update it for you. For example, an obvious next step is a tool to update a package's dependencies to their latest versions automatically whenever the package's tests and those of its dependencies continue to pass. Being able to describe versions might also enable tools that apply code cleanups. For example, having versions would let us write instructions "when using D version ≥ 1.4, replace the common client code idiom `x.Foo(1).Bar(2)` with `x.FooBar()`" that a tool like `go fix` could execute.

The goal of our work adding versions to the core Go toolchain—or, more generally, adding them to the shared working vocabulary of both Go developers and our tools—is to establish a foundation that helps with core issues like building working programs but also enables interesting external tools like these, and certainly others we haven't imagined yet.

If we're building a foundation for other tools, we should aim to make that foundation as versatile, strong, and robust as possible, to enable as many other tools as possible, with as little hindrance as possible to those tools. We're not just writing a single tool. We're defining the way all these tools will work together. This foundation is an API in the broad sense of something that programs must be written against. Like in any API, we want to choose a design that is powerful enough to enable many uses but at the same time simple, reliable, consistent, coherent, and predictable. Future posts will explore how vgo's design decisions aim for those properties.