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.