

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
```

A TOUR OF VERSIONED GO (VGO)

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
$
```

A TOUR OF VERSIONED GO (VGO)

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
$
```

A TOUR OF VERSIONED GO (VGO)

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`:

A TOUR OF VERSIONED GO (VGO)

```
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](https://github.com/golang/issue), and please include “x/vgo:” at the start of the title. More posts tomorrow. Thanks, and have fun!