Forward Compatibility and Toolchain Management in Go 1.21 - The Go Programming Language

Russ Cox 14 August 2023

The Go Blog

Beyond Go 1.21's expanded commitment to backward compatibility, Go 1.21 also introduces better forward compatibility for Go code, meaning that Go 1.21 and later will take better care not to miscompile code that requires an even newer version of Go. Specifically, the `go` line in `go.mod` now specifies a minimum required Go toolchain version, while in previous releases it was a mostly unenforced suggestion.

To make it easier to keep up with these requirements, Go 1.21 also introduces toolchain management, so that different modules can use different Go toolchains just as they can use different versions of a required module. After installing Go 1.21, you'll never have to manually download and install a Go toolchain again. The `go` command can do it for you.

The rest of this post describes both of these Go 1.21 changes in more detail.

Forward Compatibility

Forward compatibility refers to what happens when a Go toolchain attempts to build Go code intended for a newer version of Go. If my program depends on a module M and needs a bug fix added in M v1.2.3, I can add `require M v1.2.3` to my `go.mod`, guaranteeing that my program won't be compiled against older versions of M. But if my program requires a particular version of Go, there hasn't been any way to express that: in particular, the `go.mod` `go` line did not express that.

For example, if I write code that uses the new generics added in Go 1.18, I can write `go 1.18` in my `go.mod` file, but that won't stop earlier versions of Go from trying to compile the code, producing errors like:

```
$ cat go.mod
go 1.18
module example

$ go version
go version go1.17

$ go build
```

```
# example
./x.go:2:6: missing function body
./x.go:2:7: syntax error: unexpected [, expecting (
note: module requires Go 1.18
$
```

The two compiler errors are misleading noise. The real problem is printed by the `go` command as a hint: the program failed to compile, so the `go` command points out the potential version mismatch.

In this example, we're lucky the build failed. If I write code that only runs correctly in Go 1.19 or later, because it depends on a bug fixed in that patch release, but I'm not using any Go 1.19-specific language features or packages in the code, earlier versions of Go will compile it and silently succeed.

Starting in Go 1.21, Go toolchains will treat the `go` line in `go.mod` not as a guideline but as a rule, and the line can list specific point releases or release candidates. That is, Go 1.21.0 understands that it cannot even build code that says `go 1.21.1` in its `go.mod` file, not to mention code that says much later versions like `go 1.22.0`.

The main reason we allowed older versions of Go to try to compile newer code was to avoid unnecessary build failures. It's very frustrating to be told that your version of Go is too old to build a program, especially if it might work anyway (maybe the requirement is unnecessarily conservative), and especially when updating to a newer Go version is a bit of a chore. To reduce the impact of enforcing the `go` line as a requirement, Go 1.21 adds toolchain management to the core distribution as well.

When you need a new version of a Go module, the `go` command downloads it for you. Starting in Go 1.21, when you need a newer Go toolchain, the `go` command downloads that for you too. This functionality is like Node's `nvm` or Rust's `rustup`, but built in to the core `go` command instead of being a separate tool.

If you are running Go 1.21.0 and you run a `go` command, say, `go build`, in a module with a `go.mod` that says `go 1.21.1`, the Go 1.21.0 `go` command will notice that you need Go 1.21.1, download it, and re-invoke that version's `go` command to finish the build. When the `go` command downloads and runs these other toolchains, it doesn't install them in your PATH or overwrite the current installation. Instead, it downloads them as Go modules, inheriting all the security and privacy benefits of modules, and then it runs them from the module cache.

There is also a new `toolchain` line in `go.mod` that specifies the minimum Go toolchain to use when working in a particular module. In contrast to the `go` line, `toolchain` does not impose a requirement on other modules. For example, a `go.mod` might say:

```
module m
go 1.21.0
toolchain go1.21.4
```

This says that other modules requiring `m` need to provide at least Go 1.21.0, but when we are working in `m` itself, we want an even newer toolchain, at least Go 1.21.4.

The `go` and `toolchain` requirements can be updated using `go get` like ordinary module requirements. For example, if you're using one of the Go 1.21 release candidates, you can start using Go 1.21.0 in a particular module by running:

```
go get go@1.21.0
```

That will download and run Go 1.21.0 to update the `go` line, and future invocations of the `go` command will see the line `go 1.21.0` and automatically re-invoke that version.

Or if you want to start using Go 1.21.0 in a module but leave the `go` line set to an older version, to help maintain compatibility with users of earlier versions of Go, you can update the `toolchain` line:

```
go get toolchain@go1.21.0
```

If you're ever wondering which Go version is running in a particular module, the answer is the same as before: run `go version`.

You can force the use of a specific Go toolchain version using the GOTOOLCHAIN environment variable. For example, to test code with Go 1.20.4:

```
GOTOOLCHAIN=go1.20.4 go test
```

Finally, a GOTOOLCHAIN setting of the form `version+auto` means to use `version` by default but allow upgrades to newer versions as well. If you have Go 1.21.0 installed, then when Go 1.21.1 is released, you can change your system default by setting a default GOTOOLCHAIN:

```
go env -w GOTOOLCHAIN=go1.21.1+auto
```

You'll never have to manually download and install a Go toolchain again. The `go` command will take care of it for you.

See "Go Toolchains" for more details.