GOPATH variable

Go development using dependencies beyond the standard library is done using Go modules. When using Go modules, the GOPATH variable (which defaults to `$HOME/go` on Unix and `%USERPROFILE%\go` on Windows) is used for the following purposes:

- The `go install` command installs binaries to `$GOBIN`, which defaults to `$GOPATH/bin`.
- The `go get` command caches downloaded modules in `$GOMODCACHE`, which defaults to `$GOPATH/pkg/mod`.
- The `go get` command caches downloaded checksum database state in `$GOPATH/pkg/sumdb`.

See the go command documentation for full details about the GOPATH variable. The rest of this page concerns the GOPATH development mode, which is now deprecated.

GOPATH development mode

Before Go modules, Go development using dependencies used "GOPATH development mode," or "GOPATH mode" for short. In GOPATH mode, the `go` command used the GOPATH variable for the following purposes:

- The `go install` command installed binaries to `$GOBIN`, which defaults to `$GOPATH/bin`.
- The `go install` command installed the compiled package file for `import "example.com/y/z"` to `$GOPATH/pkg/example.com/y/z.a`.
- The `go get` command downloaded source code satisfying `import "example.com/y/z"` to `$GOPATH/src/example.com/y/z`.

Deprecating and removing GOPATH development mode

Go modules are the replacement for GOPATH development mode to add the concept of a package version throughout the Go ecosystem.

The transition from GOPATH development mode to Go modules has been gradual, spread across many Go releases:

- **Go 1.11 (August 2018)** introduced the `GO111MODULE` variable, which defaulted to `auto`. With `GO111MODULE=off`, the `go` command used GOPATH mode always. With `GO111MODULE=on`, the `go` command used module mode always. With `GO111MODULE=auto` (or leaving `GO111MODULE` unset), the `go` command decided the mode based on the current directory. If the current directory was *outside* `$GOPATH/src` and was within a source tree with a `go.mod` file in its root, then the `go` command used Go module mode. Otherwise the `go` command used GOPATH mode. This rule ensured that all commands run in `$GOPATH/src` were unaffected in `auto` mode but let users experiment with modules in other directories.

- **Go 1.13 (August 2019)** adjusted `GO111MODULE=auto` mode to remove the `$GOPATH/src` restriction: if a directory inside `$GOPATH/src` has a `go.mod` file, commands run in or below that directory now use module mode. This allows users to continue to organize their checked-out code in an import-based hierarchy but use modules for individual checkouts.
- **Go 1.16 (February 2021)** will change the default to `GO111MODULE=on`, using module mode always. That is, GOPATH mode will be disabled entirely by default. Users who need one to use GOPATH mode for one more release can set `GO111MODULE=auto` or `GO111MODULE=off` explicitly.
- **Go 1.NN (???)** will remove the `GO111MODULE` setting and GOPATH mode entirely, using module mode always.

Note that removing GOPATH development mode does *not* mean removing the GOPATH variable. It will still be used for the purposes listed at the top of this page.

FAQ

Is the GOPATH variable being removed?

No. The GOPATH variable (set in the environment or by `go env -w`) is **not** being removed. It will still be used to determine the default binary install location, module cache location, and checksum database cache location, as mentioned at the top of this page.

Can I still write code in GOPATH/src/import/path?

Yes. Many Go developers appreciate the structure that this convention provides and check out their module repositories into it. All your code needs to get started with modules is a `go.mod` file. See `go mod init`.

How can I compile one repo in GOPATH/src against changes made in another?

If you want to use unpublished changes in one module when building another, you can add a `replace` line to the other module's `go.mod`.

For example, if you have checked out `golang.org/x/website` and `golang.org/x/tools` to `$GOPATH/src/golang.org/x/website` and `$GOPATH/src/golang.org/x/tools`, then to make your local builds of `website` automatically use changes in `tools`, you would add this to `$GOPATH/src/golang.org/x/website/go.mod`:

```
replace golang.org/x/tools => ../tools
```

Of course, `replace` directives know nothing about `$GOPATH`. The same line would work fine if you had checked the two out into `$HOME/mycode/website` and `$HOME/mycode/tools`.

Why is GOPATH development mode being removed?

At its core, GOPATH development mode essentially supplies all those kinds of `replace` lines automatically, so that the code you build for dependencies is the code you happen to have checked out on your computer. That means your build is affected by old checkouts you happen to have lying around that you might have forgotten about. It means that the build you get on one machine can be different from another, even starting with the same version of the same top-level repo. And it means that the builds you get can be different from the ones another developer in the same project gets. Go modules address all these reproducibility concerns. The root cause of all these problems is that GOPATH mode does not have any concept of a package *version*.

In addition to reproducibility, Go modules provide a clear way to handle proxying and secure downloads. When you `git clone` a project and then grab its dependencies, those dependencies are being cryptographically checked (using the `go.sum` file) to make sure they're the same bits the original developer used. The only trusted part is the top-level `git clone`. Here again, this is only possible because Go modules, in contrast to GOPATH mode, have a concept of a package version.

And for future evolution of Go itself, modules clearly mark which version of the Go language a particular tree of files is written in. This makes it possible to disable problematic features—for example, `string(1)`, which many people think produces `"1"` but actually produces `"\x01"` (Ctrl-A)—in later versions of Go while keeping older programs building (because they are explicitly marked as having been written for the older version of Go).

There are more examples like these.

None of this is possible with GOPATH development mode as it exists today. We can't move the ecosystem forward and start really depending on these important properties of Go modules without retiring GOPATH mode.

(You might also ask: why not just add those things to GOPATH mode? The answer is: we did, and the result is Go modules.)

When was it decided to deprecate GOPATH development mode?

The original plan was to deprecate GOPATH mode in Go 1.13, but we wanted to take extra time to make modules even more robust for as many Go users as possible, so the deprecation was pushed back from that release. Discussion on issue #41330 and in the golang-tools group did not identify any remaining blockers for deprecating GOPATH, so it is now scheduled for Go 1.16, with removal in a future release, as stated in the timeline above.

What if I have more questions about moving from GOPATH development mode to Go modules?

See golang.org/help for a list of resources. If none of those are appropriate, feel free to file an issue here. We want everyone to be successful adopting Go modules.