

Go Modules

This wiki page serves as a usage and troubleshooting guide.

- For tutorial blog posts, see [Using Go Modules](#).
- For a technical reference, see the [Go Modules Reference](#) (under development).

Go has included support for versioned modules as proposed [here](#) since 1.11. The initial prototype `vgo` was [announced](#) in February 2018. In July 2018, versioned modules [landed](#) in the main Go repository.

Since [Go 1.14](#), module support is considered ready for production use, and all users are encouraged to migrate to modules from other dependency management systems. If you are unable to migrate due to a problem in the Go toolchain, please ensure that the problem has an [open issue](#) filed. (If the issue is not on the Go1.16 milestone, please comment on why it prevents you from migrating so it can be prioritized appropriately). You can also provide an [experience report](#) for more detailed feedback.

Recent Changes

Go 1.16

See the [Go 1.16 release notes](#) for details.

- Module mode (`GOMODMODULE=on`) is the default in all cases
- Commands no longer modify `go.mod` / `go.sum` by default (`-mod=readonly`)
- `go install pkg@version` is the recommended way to globally install packages / executables
- `retract` is available in `go.mod`

Go 1.15

See the [Go 1.15 release notes](#) for details.

- The location of the module cache may now be set with the `GOMODCACHE` environment variable. The default value of `GOMODCACHE` is `GOPATH[0]/pkg/mod`, the location of the module cache before this change.
- A workaround is now available for Windows "Access is denied" errors in go commands that access the module cache, caused by external programs concurrently scanning the file system (see issue [#36568](#)). The workaround is not enabled by default because it is not safe to use when Go versions lower than 1.14.2 and 1.13.10 are running concurrently with the same module cache. It can be enabled by explicitly setting the environment variable `GODEBUG=modcacheunzipinplace=1`.

Go 1.14

See the [Go 1.14 release notes](#) for details.

- When the main module contains a top-level vendor directory and its `go.mod` file specifies `go 1.14` or higher, the go command now defaults to `-mod=vendor` for operations that accept that flag.
- `-mod=readonly` is now set by default when the `go.mod` file is read-only and no top-level vendor directory is present.
- `-modcacherw` is a new flag that instructs the go command to leave newly-created directories in the module cache at their default permissions rather than making them read-only.
- `-modfile=file` is a new flag that instructs the go command to read (and possibly write) an alternate `go.mod` file instead of the one in the module root directory.

- When module-aware mode is enabled explicitly (by setting `GO111MODULE=on`), most module commands have more limited functionality if no `go.mod` file is present.
- The `go` command now supports Subversion repositories in module mode.

Go 1.13

See the [Go 1.13 release notes](#) for details.

- The `go` tool now defaults to downloading modules from the public Go module mirror at <https://proxy.golang.org>, and also defaults to validating downloaded modules (regardless of source) against the public Go checksum database at <https://sum.golang.org>.
 - If you have private code, you most likely should configure the `GOPRIVATE` setting (such as `go env -w GOPRIVATE=*.corp.com,github.com/secret/repo`), or the more fine-grained variants `GONOPROXY` or `GONOSUMDB` that support less frequent use cases. See the [documentation](#) for more details.
- `GO111MODULE=auto` enables module-mode if any `go.mod` is found, even inside `GOPATH`. (Prior to Go 1.13, `GO111MODULE=auto` would never enable module-mode inside `GOPATH`).
- `go get` arguments have changed:
 - `go get -u` (without any arguments) now only upgrades the direct and indirect dependencies of your current *package*, and no longer examines your entire *module*.
 - `go get -u ./...` from your module root upgrades all the direct and indirect dependencies of your module, and now excludes test dependencies.
 - `go get -u -t ./...` is similar, but also upgrades test dependencies.
 - `go get` no longer supports `-m` (because it would have largely overlapped with `go get -d` due to other changes; you can usually replace `go get -m foo` with `go get -d foo`).

Table of Contents

The "Quick Start" and "New Concepts" sections are particularly important for someone who is starting to work with modules. The "How to..." sections cover more details on mechanics. The largest quantity of content on this page is in the FAQs answering more specific questions; it can be worthwhile to at least skim the FAQ one-liners listed here.

- [Quick Start](#)
 - [Example](#)
 - [Daily Workflow](#)
- [New Concepts](#)
 - [Modules](#)
 - [go.mod](#)
 - [Version Selection](#)
 - [Semantic Import Versioning](#)
- [How to Use Modules](#)
 - [How to Install and Activate Module Support](#)
 - [How to Define a Module](#)
 - [How to Upgrade and Downgrade Dependencies](#)
 - [How to Prepare for a Release \(All Versions\)](#)
 - [How to Prepare for a Release \(v2 or Higher\)](#)
 - [Publishing a Release](#)
- [Migrating to Modules](#)
- [Additional Resources](#)
- [Changes Since the Initial Vgo Proposal](#)

- [GitHub Issues](#)
- [FAQs](#)
 - [How are versions marked as incompatible?](#)
 - [When do I get old behavior vs. new module-based behavior?](#)
 - [Why does installing a tool via 'go get' fail with error 'cannot find main module'?](#)
 - [How can I track tool dependencies for a module?](#)
 - [What is the status of module support in IDEs, editors and standard tools like goimports, gorelease, etc.?](#)
- [FAQs — Additional Control](#)
 - [What community tooling exists for working with modules?](#)
 - [When should I use the 'replace' directive?](#)
 - [Can I work entirely outside of VCS on my local filesystem?](#)
 - [How do I use vendoring with modules? Is vendoring going away?](#)
 - [Are there "always on" module repositories and enterprise proxies?](#)
 - [Can I control when go.mod gets updated and when the go tools use the network to satisfy dependencies?](#)
 - [How do I use modules with CI systems such as Travis or CircleCI?](#)
 - [How do I download modules needed to build specific packages or tests?](#)
- [FAQs — go.mod and go.sum](#)
 - [Why does 'go mod tidy' record indirect and test dependencies in my 'go.mod'?](#)
 - [Is 'go.sum' a lock file? Why does 'go.sum' include information for module versions I am no longer using?](#)
 - [Should I still add a 'go.mod' file if I do not have any dependencies?](#)
 - [Should I commit my 'go.sum' file as well as my 'go.mod' file?](#)
- [FAQs — Semantic Import Versioning](#)
 - [Why must major version numbers appear in import paths?](#)
 - [Why are major versions v0, v1 omitted from import paths?](#)
 - [What are some implications of tagging my project with major version v0, v1, or making breaking changes with v2+?](#)
 - [Can a module consume a package that has not opted in to modules?](#)
 - [Can a module consume a v2+ package that has not opted into modules? What does '+incompatible' mean?](#)
 - [How are v2+ modules treated in a build if modules support is not enabled? How does "minimal module compatibility" work in 1.9.7+, 1.10.3+, and 1.11?](#)
 - [What happens if I create a go.mod but do not apply semver tags to my repository?](#)
 - [Can a module depend on a different version of itself?](#)
- [FAQs — Multi-Module Repositories](#)
 - [What are multi-module repositories?](#)
 - [Should I have multiple modules in a single repository?](#)
 - [Is it possible to add a module to a multi-module repository?](#)
 - [Is it possible to remove a module from a multi-module repository?](#)
 - [Can a module depend on an internal/ in another?](#)
 - [Can an additional go.mod exclude unnecessary content? Do modules have the equivalent of a .gitignore file?](#)
- [FAQs — Minimal Version Selection](#)
 - [Won't minimal version selection keep developers from getting important updates?](#)
- [FAQs — Possible Problems](#)
 - [What are some general things I can spot check if I am seeing a problem?](#)
 - [What can I check if I am not seeing the expected version of a dependency?](#)

- [Why am I getting an error 'cannot find module providing package foo'?](#)
- [Why does 'go mod init' give the error 'cannot determine module path for source directory'?](#)
- [I have a problem with a complex dependency that has not opted in to modules. Can I use information from its current dependency manager?](#)
- [How can I resolve "parsing go.mod: unexpected module path" and "error loading module requirements" errors caused by a mismatch between import paths vs. declared module identity?](#)
- [Why does 'go build' require gcc, and why are prebuilt packages such as net/http not used?](#)
- [Do modules work with relative imports like `import "../subdir"` ?](#)
- [Some needed files may not be present in populated vendor directory.](#)

Quick Start

Example

The details are covered in the remainder of this page, but here is a simple example of creating a module from scratch.

Create a directory outside of your GOPATH, and optionally initialize VCS:

```
$ mkdir -p /tmp/scratchpad/repo
$ cd /tmp/scratchpad/repo
$ git init -q
$ git remote add origin https://github.com/my/repo
```

Initialize a new module:

```
$ go mod init github.com/my/repo

go: creating new go.mod: module github.com/my/repo
```

Write your code:

```
$ cat <<EOF > hello.go
package main

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

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

Build and run:

```
$ go mod tidy
go: finding module for package rsc.io/quote
go: found rsc.io/quote in rsc.io/quote v1.5.2
$ go build -o hello
```

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

The `go.mod` file was updated to include explicit versions for your dependencies, where `v1.5.2` here is a [semver](#) tag:

```
$ cat go.mod
module github.com/my/repo

go 1.16

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

Daily Workflow

Prior to 1.16, no `go get` or `go mod tidy` was required prior to running `go build -o hello`. Implicit modification of `go.mod` and `go.sum` files was disabled by default in 1.16.

Your typical day-to-day workflow can be:

- Add import statements to your `.go` code as needed.
- Standard commands like `go build` or `go test` will automatically add new dependencies as needed to satisfy imports (updating `go.mod` and downloading the new dependencies).
- When needed, more specific versions of dependencies can be chosen with commands such as `go get foo@v1.2.3`, `go get foo@master` (`foo@default` with mercurial), `go get foo@e3702bed2`, or by editing `go.mod` directly.

A brief tour of other common functionality you might use:

- `go list -m all` — View final versions that will be used in a build for all direct and indirect dependencies ([details](#))
- `go list -u -m all` — View available minor and patch upgrades for all direct and indirect dependencies ([details](#))
- `go get -u ./...` or `go get -u=patch ./...` (from module root directory) — Update all direct and indirect dependencies to latest minor or patch upgrades (pre-releases are ignored) ([details](#))
- `go build ./...` or `go test ./...` (from module root directory) — Build or test all packages in the module ([details](#))
- `go mod tidy` — Prune any no-longer-needed dependencies from `go.mod` and add any dependencies needed for other combinations of OS, architecture, and build tags ([details](#))
- `replace` directive or `gohack` — Use a fork, local copy or exact version of a dependency ([details](#))
- `go mod vendor` — Optional step to create a `vendor` directory ([details](#))

After reading the next four sections on "New Concepts", you will have enough information to get started with modules for most projects. It is also useful to review the [Table of Contents](#) above (including the FAQ one-liners there) to familiarize yourself with the list of more detailed topics.

New Concepts

These sections provide a high-level introduction to the main new concepts. For more details and rationale, please see this 40-minute introductory [video by Russ Cox describing the philosophy behind the design](#), the [official proposal document](#), or the more detailed initial [vgo blog series](#).

Modules

A *module* is a collection of related Go packages that are versioned together as a single unit.

Modules record precise dependency requirements and create reproducible builds.

Most often, a version control repository contains exactly one module defined in the repository root. ([Multiple modules are supported in a single repository](#), but typically that would result in more work on an on-going basis than a single module per repository).

Summarizing the relationship between repositories, modules, and packages:

- A repository contains one or more Go modules.
- Each module contains one or more Go packages.
- Each package consists of one or more Go source files in a single directory.

Modules must be semantically versioned according to [semver](#), usually in the form `v(major).(minor).(patch)`, such as `v0.1.0`, `v1.2.3`, or `v1.5.0-rc.1`. The leading `v` is required. If using Git, [tag](#) released commits with their versions. Public and private module repositories and proxies are becoming available (see FAQ [below](#)).

go.mod

A module is defined by a tree of Go source files with a `go.mod` file in the tree's root directory. Module source code may be located outside of GOPATH. There are four directives: `module`, `require`, `replace`, `exclude`.

Here is an example `go.mod` file defining the module `github.com/my/thing`:

```
module github.com/my/thing

require (
    github.com/some/dependency v1.2.3
    github.com/another/dependency/v4 v4.0.0
)
```

A module declares its identity in its `go.mod` via the `module` directive, which provides the *module path*. The import paths for all packages in a module share the module path as a common prefix. The module path and the relative path from the `go.mod` to a package's directory together determine a package's import path.

For example, if you are creating a module for a repository `github.com/user/mymod` that will contain two packages with import paths `github.com/user/mymod/foo` and `github.com/user/mymod/bar`, then the first line in your `go.mod` file typically would declare your module path as `module github.com/user/mymod`, and the corresponding on-disk structure could be:

```
mymod
|-- bar
|   |-- bar.go
|-- foo
|   |-- foo.go
`-- go.mod
```

In Go source code, packages are imported using the full path including the module path. For example, if in our example above, we declared the module identity in `go.mod` as `module github.com/user/mymod`, a consumer could do:

```
import "github.com/user/mymod/bar"
```

This imports package `bar` from the module `github.com/user/mymod`.

`exclude` and `replace` directives only operate on the current ("main") module. `exclude` and `replace` directives in modules other than the main module are ignored when building the main module. The `replace` and `exclude` statements, therefore, allow the main module complete control over its own build, without also being subject to complete control by dependencies. (See FAQ [below](#) for a discussion of when to use a `replace` directive).

Version Selection

If you add a new import to your source code that is not yet covered by a `require` in `go.mod`, most `go` commands like `'go build'` and `'go test'` will automatically look up the proper module and add the *highest* version of that new direct dependency to your module's `go.mod` as a `require` directive. For example, if your new import corresponds to dependency M whose latest tagged release version is `v1.2.3`, your module's `go.mod` will end up with `require M v1.2.3`, which indicates module M is a dependency with allowed version `>= v1.2.3` (and `< v2`, given `v2` is considered incompatible with `v1`).

The *minimal version selection* algorithm is used to select the versions of all modules used in a build. For each module in a build, the version selected by minimal version selection is always the semantically *highest* of the versions explicitly listed by a `require` directive in the main module or one of its dependencies.

As an example, if your module depends on module A which has a `require D v1.0.0`, and your module also depends on module B which has a `require D v1.1.1`, then minimal version selection would choose `v1.1.1` of D to include in the build (given it is the highest listed `require` version). This selection of `v1.1.1` remains consistent even if sometime later a `v1.2.0` of D becomes available. This is an example of how the modules system provides 100% reproducible builds. When ready, the module author or user might choose to upgrade to the latest available version of D or choose an explicit version for D.

For a brief rationale and overview of the minimal version selection algorithm, [see the "High Fidelity Builds" section](#) of the official proposal, or see the [more detailed vgo blog series](#).

To see a list of the selected module versions (including indirect dependencies), use `go list -m all`.

See also the ["How to Upgrade and Downgrade Dependencies"](#) section below and the ["How are versions marked as incompatible?"](#) FAQ below.

Semantic Import Versioning

For many years, the official Go FAQ has included this advice on package versioning:

"Packages intended for public use should try to maintain backward 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."

The last sentence is especially important — if you break compatibility, you should change the import path of your package. With Go 1.11 modules, that advice is formalized into 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."

Recall [semver](#) requires a major version change when a v1 or higher package makes a backwards-incompatible change. The result of following both the import compatibility rule and semver is called *Semantic Import Versioning*, where the major version is included in the import path — this ensures the import path changes any time the major version increments due to a compatibility break.

As a result of Semantic Import Versioning, code opting in to Go modules **must comply with these rules**:

- Follow [semver](#). (An example VCS tag is `v1.2.3`).
- If the module is version v2 or higher, the major version of the module *must* be included as a `/vN` at the end of the module paths used in `go.mod` files (e.g., `module github.com/my/mod/v2`, `require github.com/my/mod/v2 v2.0.1`) and in the package import path (e.g., `import "github.com/my/mod/v2/mypkg"`). This includes the paths used in `go get` commands (e.g., `go get github.com/my/mod/v2@v2.0.1`). Note there is both a `/v2` and a `@v2.0.1` in that example. One way to think about it is that the module name now includes the `/v2`, so include `/v2` whenever you are using the module name).
- If the module is version v0 or v1, do *not* include the major version in either the module path or the import path.

In general, packages with different import paths are different packages. For example, `math/rand` is a different package than `crypto/rand`. This is also true if different import paths are due to different major versions appearing in the import path. Thus `example.com/my/mod/mypkg` is a different package than `example.com/my/mod/v2/mypkg`, and both may be imported in a single build, which among other benefits helps with diamond dependency problems and also allows a v1 module to be implemented in terms of its v2 replacement or vice versa.

See the "[Module compatibility and semantic versioning](#)" section of the `go` command documentation for more details on Semantic Import Versioning, and see <https://semver.org> for more about semantic versioning.

This section so far has been focused on code that has opted in to modules and imports other modules. However, putting major versions in import paths for v2+ modules could create incompatibilities with older versions of Go, or with code that has not yet opted in to modules. To help with this, there are three important transitional special cases or exceptions to the behavior and rules described above. These transitional exceptions will become less important over time as more packages opt in to modules.

Three Transitional Exceptions

1. `gopkg.in`

Existing code that uses import paths starting with `gopkg.in` (such as `gopkg.in/yaml.v1` and `gopkg.in/yaml.v2`) can continue to use those forms for their module paths and import paths even after opting in to modules.

2. '+incompatible' when importing non-module v2+ packages

A module can import a v2+ package that has not opted in to modules itself. A non-module v2+ package that has a valid v2+ [semver](#) tag will be recorded with a `+incompatible` suffix in the importing module's `go.mod` file. The `+incompatible` suffix indicates that even though the v2+ package has a valid v2+ [semver](#) tag such as `v2.0.0`, the v2+ package has not actively opted in to modules and hence that v2+ package is assumed to have *not* been created with an understanding of the implications of Semantic Import Versioning and how to use major versions in import paths. Therefore, when operating in [module mode](#), the `go` tool will treat a non-module v2+ package as an (incompatible) extension of the v1 version series of the package and assume the package has no awareness of Semantic Import Versioning, and the `+incompatible` suffix is an indication that the `go` tool is doing so.

3. "Minimal module compatibility" when module mode is not enabled

To help with backwards-compatibility, Go versions 1.9.7+, 1.10.3+ and 1.11 have been updated to make it easier for code built with those releases to be able to properly consume v2+ modules *without* requiring modification of pre-existing code. This behavior is called "minimal module compatibility", and it only takes effect when full [module mode](#) is disabled for the `go` tool, such as if such as you have set

`GO111MODULE=off` in Go 1.11, or are using Go versions 1.9.7+ or 1.10.3+. When relying on this "minimal module compatibility" mechanism in Go 1.9.7+, 1.10.3+ and 1.11, a package that has *not* opted in to modules would *not* include the major version in the import path for any imported v2+ modules. In contrast, a package that *has* opted in to modules *must* include the major version in the import path to import any v2+ modules (in order to properly import the v2+ module when the `go` tool is operating in full module mode with full awareness of Semantic Import Versioning).

For the exact mechanics required to release a v2+ module, please see the ["Releasing Modules \(v2 or Higher\)"](#) section below.

How to Use Modules

How to Install and Activate Module Support

To use modules, two install options are:

- [Install the latest Go 1.11 release.](#)
- [Install the Go toolchain from source](#) on the `master` branch.

Once installed, you can then activate module support in one of two ways:

- Invoke the `go` command in a directory outside of the `$GOPATH/src` tree, with a valid `go.mod` file in the current directory or any parent of it and the environment variable `GO111MODULE` unset (or explicitly set to `auto`).
- Invoke the `go` command with `GO111MODULE=on` environment variable set.

How to Define a Module

To create a `go.mod` for an existing project:

1. Navigate to the root of the module's source tree outside of GOPATH:

```
$ cd <project path outside $GOPATH/src> # e.g., cd ~/projects/hello
```

Note that outside of GOPATH, you do not need to set `GO111MODULE` to activate module mode.

Alternatively, if you want to work in your GOPATH:

```
$ export GO111MODULE=on # manually active module mode
$ cd $GOPATH/src/<project path> # e.g., cd
$GOPATH/src/you/hello
```

2. Create the initial module definition and write it to the `go.mod` file:

```
$ go mod init
```

This step converts from any existing `dep` `Gopkg.lock` file or any of the other [nine total supported dependency formats](#), adding require statements to match the existing configuration.

`go mod init` will often be able to use auxiliary data (such as VCS meta-data) to automatically determine the appropriate module path, but if `go mod init` states it can not automatically determine the module path, or if you need to otherwise override that path, you can supply the [module path](#) as an optional argument to `go mod init`, for example:

```
$ go mod init github.com/my/repo
```

Note that if your dependencies include v2+ modules, or if you are initializing a v2+ module, then after running `go mod init` you might also need to edit your `go.mod` and `.go` code to add `/vN` to import paths and module paths as described in the ["Semantic Import Versioning"](#) section above. This applies even if `go mod init` automatically converted your dependency information from `dep` or other dependency managers. (Because of this, after running `go mod init`, you typically should not run `go mod tidy` until you have successfully run `go build ./...` or similar, which is the sequence shown in this section).

3. Build the module. When executed from the root directory of a module, the `./...` pattern matches all the packages within the current module. `go build` will automatically add missing or unconverted dependencies as needed to satisfy imports for this particular build invocation:

```
$ go build ./...
```

4. Test the module as configured to ensure that it works with the selected versions:

```
$ go test ./...
```

5. (Optional) Run the tests for your module plus the tests for all direct and indirect dependencies to check for incompatibilities:

```
$ go test all
```

Prior to tagging a release, see the ["How to Prepare for a Release"](#) section below.

For more information on all of these topics, the primary entry point to the official modules documentation is [available on golang.org](#).

How to Upgrade and Downgrade Dependencies

Day-to-day upgrading and downgrading of dependencies should be done using 'go get', which will automatically update the `go.mod` file. Alternatively, you can edit `go.mod` directly.

In addition, go commands like 'go build', 'go test', or even 'go list' will automatically add new dependencies as needed to satisfy imports (updating `go.mod` and downloading the new dependencies).

To upgrade a dependency to the latest version:

```
go get example.com/package
```

To upgrade a dependency *and all its dependencies* to the latest version:

```
go get -u example.com/package
```

To view available minor and patch upgrades for all direct and indirect dependencies:

```
go list -u -m all
```

To view available minor and patch upgrades *only* for the direct dependencies, run:

```
go list -u -f '{{if (and (not (or .Main .Indirect)) .Update)}}{{.Path}}: {{.Version}}  
-> {{.Update.Version}}{{end}}}' -m all 2> /dev/null
```

To upgrade to the latest version for all direct and indirect dependencies of the current module, the following can be run from the module root directory:

- `go get -u ./...` to use the latest *minor or patch* releases (and add `-t` to also upgrade test dependencies)
- `go get -u=patch ./...` to use the latest *patch* releases (and add `-t` to also upgrade test dependencies)

`go get foo` updates to the latest version of `foo`. `go get foo` is equivalent to `go get foo@latest` — in other words, `@latest` is the default if no `@` version is specified.

In this section, "latest" is the latest version with a [semver](#) tag, or the latest known commit if there are no semver tags. Prerelease tags are not selected as "latest" unless there are no other semver tags on the repository ([details](#)).

A common mistake is thinking `go get -u foo` solely gets the latest version of `foo`. In actuality, the `-u` in `go get -u foo` or `go get -u foo@latest` means to *also* get the latest versions for *all* of the direct and indirect dependencies of `foo`. A common starting point when upgrading `foo` is instead to do `go get foo` or `go get foo@latest` without a `-u` (and after things are working, consider `go get -u=patch foo`, `go get -u=patch`, `go get -u foo`, or `go get -u`).

To upgrade or downgrade to a more specific version, 'go get' allows version selection to be overridden by adding an `@version` suffix or "[module query](#)" to the package argument, such as `go get foo@v1.6.2`, `go get foo@e3702bed2`, or `go get foo@'<v1.6.2'`.

Using a branch name such as `go get foo@master` (`foo@default` with mercurial) is one way to obtain the latest commit regardless of whether or not it has a semver tag.

In general, module queries that do not resolve to a semver tag will be recorded as [pseudo-versions](#) in the `go.mod` file.

See the "[Module-aware go get](#)" and "[Module queries](#)" sections of the `go` command documentation for more information on the topics here.

Modules are capable of consuming packages that have not yet opted into modules, including recording any available semver tags in `go.mod` and using those semver tags to upgrade or downgrade. Modules can also consume packages that do not yet have any proper semver tags (in which case they will be recorded using pseudo-versions in `go.mod`).

After upgrading or downgrading any dependencies, you may then want to run the tests again for all packages in your build (including direct and indirect dependencies) to check for incompatibilities:

```
$ go test all
```

How to Prepare for a Release

Releasing Modules (All Versions)

Best practices for creating a release of a module are expected to emerge as part of the initial modules experiment. Many of these might end up being automated by a [future 'go release' tool](#).

Some current suggested best practices to consider prior to tagging a release:

- Run `go mod tidy` to possibly prune any extraneous requirements (as described [here](#)) and also ensure your current `go.mod` reflects all possible build tags/OS/architecture combinations (as described [here](#)).
 - In contrast, other commands like `go build` and `go test` will not remove dependencies from `go.mod` that are no longer required and only update `go.mod` based on the current build invocation's tags/OS/architecture.
- Run `go test all` to test your module (including running the tests for your direct and indirect dependencies) as a way of validating that the currently selected package versions are compatible.
 - The number of possible version combinations is exponential in the number of modules, so in general, you cannot expect your dependencies to have tested against all possible combinations of their dependencies.
 - As part of the modules work, `go test all` has been [re-defined to be more useful](#): to include all the packages in the current module plus all the packages they depend on through a sequence of one or more imports while excluding packages that don't matter in the current module.
- Ensure your `go.sum` file is committed along with your `go.mod` file. See [FAQ below](#) for more details and rationale.

Releasing Modules (v2 or Higher)

If you are releasing a v2 or higher module, please first review the discussion in the ["Semantic Import Versioning"](#) section above, which includes why major versions are included in the module path and import path for v2+ modules, as well as how Go versions 1.9.7+ and 1.10.3+ have been updated to simplify that transition.

Note that if you are adopting modules for the first time for a pre-existing repository or set of packages that have already been tagged `v2.0.0` or higher before adopting modules, then the [recommended best practice](#) is to increment the major version when first adopting modules. For example, if you are the author of `foo`, and the latest tag for the `foo` repository is `v2.2.2`, and `foo` has not yet adopted modules, then the best practice would be to use `v3.0.0` for the first release of `foo` to adopt modules (and hence the first release of `foo` to contain a `go.mod` file). Incrementing the major version in this case provides greater clarity to consumers of `foo`, allows for additional non-module patches or minor releases on the v2 series of `foo` if needed, and provides a strong signal for a module-based consumer of `foo` that different major versions result if you do `import "foo"` and a corresponding `require foo v2.2.2+incompatible`, vs. `import "foo/v3"` and a corresponding `require foo/v3 v3.0.0`. (Note that this advice regarding incrementing the major version when first adopting modules does *not* apply to pre-existing repos or packages whose latest versions are `v0.x.x` or `v1.x.x`).

There are two alternative mechanisms to release a v2 or higher module. Note that with both techniques, the new module release becomes available to consumers when the module author pushes the new tags. Using the example of creating a `v3.0.0` release, the two options are:

1. **Major branch:** Update the `go.mod` file to include a `/v3` at the end of the module path in the `module` directive (e.g., `module github.com/my/module/v3`). Update import statements within the module to

also use `/v3` (e.g., `import "github.com/my/module/v3/mypkg"`). Tag the release with `v3.0.0` .

- Go versions 1.9.7+, 1.10.3+, and 1.11 are able to properly consume and build a v2+ module created using this approach without requiring updates to consumer code that has not yet opted in to modules (as described in the ["Semantic Import Versioning"](#) section above).
- A community tool github.com/marwan-at-work/mod helps automate this procedure. See the [repository](#) or the [community tooling FAQ](#) below for an overview.
- To avoid confusion with this approach, consider putting the `v3.*.*` commits for the module on a separate v3 branch.
- **Note:** creating a new branch is *not* required. If instead you have been previously releasing on master and would prefer to tag `v3.0.0` on master, that is a viable option. (However, be aware that introducing an incompatible API change in `master` can cause issues for non-modules users who issue a `go get -u` given the `go` tool is not aware of [semver](#) prior to Go 1.11 or when [module mode](#) is not enabled in Go 1.11+).
- Pre-existing dependency management solutions such as `dep` currently can have problems consuming a v2+ module created in this way. See for example [dep#1962](#).

2. **Major subdirectory:** Create a new `v3` subdirectory (e.g., `my/module/v3`) and place a new `go.mod` file in that subdirectory. The module path must end with `/v3` . Copy or move the code into the `v3` subdirectory. Update import statements within the module to also use `/v3` (e.g., `import "github.com/my/module/v3/mypkg"`). Tag the release with `v3.0.0` .

- This provides greater backwards-compatibility. In particular, Go versions older than 1.9.7 and 1.10.3 are also able to properly consume and build a v2+ module created using this approach.
- A more sophisticated approach here could exploit type aliases (introduced in Go 1.9) and forwarding shims between major versions residing in different subdirectories. This can provide additional compatibility and allow one major version to be implemented in terms of another major version but would entail more work for a module author. An in-progress tool to automate this is `goforward` . Please see [here](#) for more details and rationale, along with a functioning initial version of `goforward` .
- Pre-existing dependency management solutions such as `dep` should be able to consume a v2+ module created in this way.

See <https://research.swtch.com/vgo-module> for a more in-depth discussion of these alternatives.

Publishing a release

A new module version may be published by pushing a tag to the repository that contains the module source code. The tag is formed by concatenating two strings: a *prefix* and a *version*.

The *version* is the semantic import version for the release. It should be chosen by following the rules of [semantic import versioning](#).

The *prefix* indicates where a module is defined within a repository. If the module is defined at the root of the repository, the prefix is empty, and the tag is just the version. However, in [multi-module repositories](#), the prefix distinguishes versions for different modules. The prefix is the directory within the repository where the module is defined. If the repository follows the major subdirectory pattern described above, the prefix does not include the major version suffix.

For example, suppose we have a module `example.com/repo/sub/v2` , and we want to publish version `v2.1.6` . The repository root corresponds to `example.com/repo` , and the module is defined in

`sub/v2/go.mod` within the repository. The prefix for this module is `sub/`. The full tag for this release should be `sub/v2.1.6`.

Migrating to Modules

This section attempts to briefly enumerate the major decisions to be made when migrating to modules as well as list other migration-related topics. References are generally provided to other sections for more details.

This material is primarily based on best practices that have emerged from the community as part of the modules experiment; this is, therefore, a work-in-progress section that will improve as the community gains more experience.

Summary:

- The modules system is designed to allow different packages in the overall Go ecosystem to opt-in at different rates.
- Packages that are already on version v2 or higher have more migration considerations, primarily due to the implications of [Semantic Import versioning](#).
- New packages and packages on v0 or v1 have substantially fewer considerations when adopting modules.
- Modules defined with Go 1.11 can be used by older Go versions (although the exact Go versions depend on the strategy used by the main module and its dependencies, as outlined below).

Migration topics:

Automatic Migration from Prior Dependency Managers

- `go mod init` automatically translates the required information from [dep, glide, govendor, godep, and 5 other pre-existing dependency managers](#) into a `go.mod` file that produces the equivalent build.
- If you are creating a v2+ module, be sure your `module` directive in the converted `go.mod` includes the appropriate `/vN` (e.g., `module foo/v3`).
- Note that if you are importing v2+ modules, you might need to do some manual adjustments after an initial conversion in order to add `/vN` to the `require` statements that `go mod init` generates after translating from a prior dependency manager. See the ["How to Define a Module"](#) section above for more details.
- In addition, `go mod init` will not edit your `.go` code to add any required `/vN` to import statements. See the ["Semantic Import versioning"](#) and ["Releasing Modules \(v2 or Higher\)"](#) sections above for the required steps, including some options around community tools to automate the conversion.

Providing Dependency Information to Older Versions of Go and Non-Module Consumers

- Older versions of Go understand how to consume a vendor directory created by `go mod vendor`, as do Go 1.11 and 1.12+ when module mode is disabled. Therefore, vendoring is one way for a module to provide dependencies to older versions of Go that do not fully understand modules, as well as to consumers that have not enabled modules themselves. See the [vendoring FAQ](#) and the `go` command [documentation](#) for more details.

Updating Pre-Existing Install Instructions

- Pre-modules, it is common for install instructions to include `go get -u foo`. If you are publishing a module `foo`, consider dropping the `-u` in instructions for modules-based consumers.
 - `-u` asks the `go` tool to upgrade all the direct and indirect dependencies of `foo`.
 - A module consumer might choose to run `go get -u foo` later, but there are more benefits of ["High Fidelity Builds"](#) if `-u` is not part of the initial install instructions. See ["How to Upgrade and Downgrade Dependencies"](#) for more details.
 - `go get -u foo` does still work, and can still be a valid choice for install instructions.

- In addition, `go get foo` is not strictly needed for a module-based consumer.
 - Simply adding an import statement `import "foo"` is sufficient. (Subsequent commands like `go build` or `go test` will automatically download `foo` and update `go.mod` as needed).
- Module-based consumers will not use a `vendor` directory by default.
 - When module mode is enabled in the `go` tool, `vendor` is not strictly required when consuming a module (given the information contained in `go.mod` and the cryptographic checksums in `go.sum`), but some pre-existing install instructions assume the `go` tool will use `vendor` by default. See the [vendoring FAQ](#) for more details.
- Install instructions that include `go get foo/...` might have issues in some cases (see discussion in [#27215](#)).

Avoid Breaking Existing Import Paths

A module declares its identity in its `go.mod` via the `module` directive, such as `module github.com/my/module`. All packages within the module must be imported by any module-aware consumer with import paths that match the module's declared module path (either exactly for a root package, or with the module path as a prefix of the import path). The `go` command reports an `unexpected module path` error if there is a mismatch between an import path vs. the corresponding module's declared module path.

When adopting modules for a pre-existing set of packages, care should be taken to avoid breaking existing import paths used by existing consumers, unless you are incrementing your major version when adopting modules.

For example, if your pre-existing README has been telling consumers to use `import "gopkg.in/foo.v1"`, and if you then adopt modules with a v1 release, your initial `go.mod` should almost certainly read `module gopkg.in/foo.v1`. If you wanted to move away from using `gopkg.in`, that would be a breaking change for your current consumers. One approach would be to change to something like `module github.com/repo/foo/v2` if you later move to v2.

Note that module paths and import paths are case-sensitive. Changing a module from `github.com/Sirupsen/logrus` to `github.com/sirupsen/logrus`, for example, is a breaking change for consumers, even if GitHub automatically forwards from one repository name to the new repository name.

After you have adopted modules, changing your module path in your `go.mod` is a breaking change.

Overall, this is similar to the pre-modules enforcement of a canonical import path via ["import path comments"](#), which are also sometimes called "import pragmas" or "import path enforcement". As an example, the package `go.uber.org/zap` is currently hosted at `github.com/uber-go/zap`, but uses an import path comment [next to the package declaration](#) that triggers an error for any pre-modules consumer using the wrong github-based import path:

```
package zap // import "go.uber.org/zap"
```

Import path comments are obsoleted by the `go.mod` file's module statement.

Incrementing the Major Version When First Adopting Modules with v2+ Packages

- If you have packages that have already been tagged v2.0.0 or higher before adopting modules, then the recommended best practice is to increment the major version when first adopting modules. For example, if you are on `v2.0.1` and have not yet adopted modules, then you would use `v3.0.0` for the first release that adopts modules. See the ["Releasing Modules \(v2 or Higher\)"](#) section above for more details.

v2+ Modules Allow Multiple Major Versions Within a Single Build

- If a module is on v2 or higher, an implication is that multiple major versions can be in a single build (e.g., `foo` and `foo/v3` might end up in a single build).
 - This flows naturally from the rule that "packages with different import paths are different packages".
 - When this happens, there will be multiple copies of package-level state (e.g., package-level state for `foo` and package-level state for `foo/v3`) as well as each major version will run its own `init` function.
 - This approach helps with multiple aspects of the modules system, including helping with diamond dependency problems, gradual migration to new versions within large codebases, and allowing a major version to be implemented as a shim around a different major version.
- See the "Avoiding Singleton Problems" section of <https://research.swtch.com/vgo-import> or [#27514](#) for some related discussion.

Modules Consuming Non-Module Code

- Modules are capable of consuming packages that have not yet opted into modules, with the appropriate package version information recorded in the importing module's `go.mod`. Modules can consume packages that do not yet have any proper semver tags. See FAQ [below](#) for more details.
- Modules can also import a v2+ package that has not opted into modules. It will be recorded with a `+incompatible` suffix if the imported v2+ package has valid semver tags. See FAQ [below](#) for more details.

Non-Module Code Consuming Modules

- **Non-module code consuming v0 and v1 modules:**
 - Code that has not yet opted in to modules can consume and build v0 and v1 modules (without any requirement related to the Go version used).
- **Non-module code consuming v2+ modules:**
 - Go versions 1.9.7+, 1.10.3+ and 1.11 have been updated so that code built with those releases can properly consume v2+ modules without requiring modification of pre-existing code as described in the ["Semantic Import versioning"](#) and ["Releasing Modules \(v2 or Higher\)"](#) sections above.
 - Go versions prior to 1.9.7 and 1.10.3 can consume v2+ modules if the v2+ module was created following the "Major subdirectory" approach outlined in the ["Releasing Modules \(v2 or Higher\)"](#) section.

Strategies for Authors of Pre-Existing v2+ Packages

For authors of pre-existing v2+ packages considering opting in to modules, one way to summarize the alternative approaches is as a choice between three top-level strategies. Each choice then has follow-on decisions and variations (as outlined above). These alternative top-level strategies are:

1. Require clients to use Go versions 1.9.7+, 1.10.3+, or 1.11+.

The approach uses the "Major Branch" approach and relies on the "minimal module awareness" that was backported to 1.9.7 and 1.10.3. See the ["Semantic Import versioning"](#) and ["Releasing Modules \(v2 or Higher\)"](#) sections above for more details.

2. Allow clients to use even older Go versions like Go 1.8.

This approach uses the "Major Subdirectory" approach and involves creating a subdirectory such as `/v2` or `/v3`. See the ["Semantic Import versioning"](#) and ["Releasing Modules \(v2 or Higher\)"](#) sections above for

more details.

3. Wait on opting into modules.

In this strategy, things continue to work with client code that has opted into modules as well as with client code that has not opted into modules. As time goes by, Go versions 1.9.7+, 1.10.3+, and 1.11+ will be out for an increasingly longer time period, and at some point in the future, it becomes more natural or client-friendly to require Go versions 1.9.7+/1.10.3+/1.11+, and at that point in time, you can implement strategy 1 above (requiring Go versions 1.9.7+, 1.10.3+, or 1.11+) or even strategy 2 above (though if you are ultimately going to go with strategy 2 above in order to support older Go versions like 1.8, then that is something you can do now).

Additional Resources

Documentation and Proposal

- Official documentation:
 - Latest [HTML documentation for modules on golang.org](https://golang.org/doc/modules).
 - Run `go help modules` for more about modules. (This is the main entry point for modules topics via `go help`)
 - Run `go help mod` for more about the `go mod` command.
 - Run `go help module-get` for more about the behavior of `go get` when in module-aware mode.
 - Run `go help goproxy` for more about the module proxy, including a pure file-based option via a `file:///` URL.
- The initial ["Go & Versioning"](#) series of blog posts on `vgo` by Russ Cox (first posted February 20, 2018)
- Official [golang.org blog post introducing the proposal](#) (March 26, 2018)
 - This provides a more succinct overview of the proposal than the full `vgo` blog series, along with some of the history and process behind the proposal
- Official [Versioned Go Modules Proposal](#) (last updated March 20, 2018)

Introductory Material

- Introductory 40-minute video ["The Principles of Versions in Go"](#) from GopherCon Singapore by Russ Cox (May 2, 2018)
 - Succinctly covers the philosophy behind the design of versioned Go modules, including the three core principles of "Compatibility", "Repeatability", and "Cooperation"
- Example based 35-minute introductory video ["What are Go modules and how do I use them?"](#) ([slides](#)) by Paul Jolly (August 15, 2018)
- Introductory blog post ["Taking Go Modules for a Spin"](#) by Dave Cheney (July 14, 2018)
- Introductory [Go Meetup slides on modules](#) by Chris Hines (July 16, 2018)
- Introductory 30-minute video ["Intro to Go Modules and SemVer"](#) by Francesc Campoy (Nov 15, 2018)

Additional Material

- Blog post ["Using Go modules with vendor support on Travis CI"](#) by Fatih Arslan (August 26, 2018)
- Blog post ["Go Modules and CircleCI"](#) by Todd Keech (July 30, 2018)
- Blog post ["The vgo proposal is accepted. Now what?"](#) by Russ Cox (May 29, 2018)
 - Includes a summary of what it means that versioned modules are currently an experimental opt-in feature
- Blog post on [how to build go from tip and start using go modules](#) by Carolyn Van Slyck (July 16, 2018)

Changes Since the Initial Vgo Proposal

As part of the proposal, prototype, and beta processes, there have been over 400 issues created by the overall community. Please continue to supply feedback.

Here is a partial list of some of the larger changes and improvements, almost all of which were primarily based on feedback from the community:

- Top-level vendor support was retained rather than vgo-based builds ignoring vendor directories entirely ([discussion](#), [CL](#))
- Backported minimal module-awareness to allow older Go versions 1.9.7+ and 1.10.3+ to more easily consume modules for v2+ projects ([discussion](#), [CL](#))
- Allowed vgo to use v2+ tags by default for pre-existing packages did not yet have a go.mod (recent update in related behavior described [here](#))
- Added support via command `go get -u=patch` to update all transitive dependencies to the latest available patch-level versions on the same minor version ([discussion](#), [documentation](#))
- Additional control via environmental variables (e.g., GOFLAGS in [#26585](#), [CL](#))
- Finer grain control on whether or not go.mod is allowed to be updated, how vendor directory is used, and whether or not network access is allowed (e.g., `-mod=readonly`, `-mod=vendor`, `GOPROXY=off`; related [CL](#) for a recent change)
- Added more flexible replace directives ([CL](#))
- Added additional ways to interrogate modules (for human consumption, as well as for better editor / IDE integration)
- The UX of the go CLI has continued to be refined based on experiences so far (e.g., [#26581](#), [CL](#))
- Additional support for warming caches for use cases such as CI or docker builds via `go mod download` ([#26610](#))
- **Most likely:** better support for installing specific versions of programs to GOBIN ([#24250](#))

GitHub Issues

- [Currently open module issues](#)
- [Closed module issues](#)
- [Closed vgo issues](#)
- Submit a [new module issue](#) using 'cmd/go:' as the prefix

FAQs

How are versions marked as incompatible?

The `require` directive allows any module to declare that it should be built with version `>= x.y.z` of a dependency D (which may be specified due to incompatibilities with version `< x.y.z` of module D). Empirical data suggests [this is the dominant form of constraints used in `dep` and `cargo`](#). In addition, the top-level module in the build can `exclude` specific versions of dependencies or `replace` other modules with different code. See the full proposal for [more details and rationale](#).

One of the key goals of the versioned modules proposal is to add a common vocabulary and semantics around versions of Go code for both tools and developers. This lays a foundation for future capabilities to declare additional forms of incompatibilities, such as possibly:

- declaring deprecated versions as [described](#) in the initial `vgo` blog series
- declaring pair-wise incompatibility between modules in an external system as discussed for example [here](#) during the proposal process

- declaring pair-wise incompatible versions or insecure versions of a module after a release has been published. See for example the on-going discussion in [#24031](#) and [#26829](#)

When do I get old behavior vs. new module-based behavior?

In general, modules are opt-in for Go 1.11, so by design old behavior is preserved by default.

Summarizing when you get the old 1.10 status quo behavior vs. the new opt-in modules-based behavior:

- Inside GOPATH — defaults to old 1.10 behavior (ignoring modules)
- Outside GOPATH while inside a file tree with a `go.mod` — defaults to modules behavior
- GO111MODULE environment variable:
 - unset or `auto` — default behavior above
 - `on` — force module support on regardless of directory location
 - `off` — force module support off regardless of directory location

Why does installing a tool via `go get` fail with error `cannot find main module`?

This occurs when you have set `GO111MODULE=on`, but are not inside of a file tree with a `go.mod` when you run `go get`.

The simplest solution is to leave `GO111MODULE` unset (or equivalently explicitly set to `GO111MODULE=auto`), which avoids this error.

Recall one of the primary reason modules exist is to record precise dependency information. This dependency information is written to your current `go.mod`. If you are not inside of a file tree with a `go.mod` but you have told the `go get` command to operate in module mode by setting `GO111MODULE=on`, then running `go get` will result in the error `cannot find main module` because there is no `go.mod` available to record dependency information.

Solution alternatives include:

1. Leave `GO111MODULE` unset (the default, or explicitly set `GO111MODULE=auto`), which results in friendlier behavior. This will give you Go 1.10 behavior when you are outside of a module and hence will avoid `go get` reporting `cannot find main module`.
2. Leave `GO111MODULE=on`, but as needed disable modules temporarily and enable Go 1.10 behavior during `go get`, such as via `GO111MODULE=off go get example.com/cmd`. This can be turned into a simple script or shell alias such as `alias oldget='GO111MODULE=off go get'`
3. Create a temporary `go.mod` file that is then discarded. This has been automated by a [simple shell script](#) by [@rogppeppe](#). This script allows version information to optionally be supplied via `vgoget example.com/cmd[@version]`. (This can be a solution for avoiding the error `cannot use path@version syntax in GOPATH mode`).
4. `gobin` is a module-aware command to install and run main packages. By default, `gobin` installs/runs main packages without first needing to manually create a module, but with the `-m` flag it can be told to use an existing module to resolve dependencies. Please see the `gobin` [README](#) and [FAQ](#) for details and additional use cases.
5. Create a `go.mod` you use to track your globally installed tools, such as in `~/global-tools/go.mod`, and `cd` to that directory prior to running `go get` or `go install` for any globally installed tools.

6. Create a `go.mod` for each tool in separate directories, such as `~/tools/gorename/go.mod` and `~/tools/goimports/go.mod`, and `cd` to that appropriate directory prior to running `go get` or `go install` for the tool.

This current limitation will be resolved. However, the primary issue is that modules are currently opt-in, and a full solution will likely wait until `GO111MODULE=on` becomes the default behavior. See [#24250](#) for more discussion, including this comment:

This clearly must work eventually. The thing I'm not sure about is exactly what this does as far as the version is concerned: does it create a temporary module root and go.mod, do the install, and then throw it away? Probably. But I'm not completely sure, and for now, I didn't want to confuse people by making vgo do things outside go.mod trees. Certainly, the eventual go command integration has to support this.

This FAQ has been discussing tracking *globally* installed tools.

If instead, you want to track the tools required by a *specific* module, see the next FAQ.

How can I track tool dependencies for a module?

If you:

- want to use a go-based tool (e.g. `stringer`) while working on a module, and
- want to ensure that everyone is using the same version of that tool while tracking the tool's version in your module's `go.mod` file

then one currently recommended approach is to add a `tools.go` file to your module that includes import statements for the tools of interest (such as `import _ "golang.org/x/tools/cmd/stringer"`), along with a `//go:build tools` build constraint. The import statements allow the `go` command to precisely record the version information for your tools in your module's `go.mod`, while the `//go:build tools` build constraint prevents your normal builds from actually importing your tools.

For a concrete example of how to do this, please see this ["Go Modules by Example" walkthrough](#).

A discussion of the approach along with an earlier concrete example of how to do this is in [this comment in #25922](#).

The brief rationale (also from [#25922](#)):

I think the tools.go file is, in fact, the best practice for tool dependencies, certainly for Go 1.11.

I like it because it does not introduce new mechanisms.

It simply reuses existing ones.

What is the status of module support in IDEs, editors and standard tools like goimports, gorename, etc?

Support for modules is starting to land in editors and IDEs.

For example:

- **GoLand**: currently has full support for modules outside and inside GOPATH, including completion, syntax analysis, refactoring, navigation as described [here](#).
- **VS Code**: work is complete, MS recommending modules over GOPATH, the former tracking issue ([#1532](#)) has been closed. Documentation is available in the [VS Code module repository](#).
- **Atom with go-plus**: tracking issue is [#761](#).

- **vim with vim-go**: initial support for syntax highlighting and formatting `go.mod` has [landed](#). Broader support tracked in [#1906](#).
- **emacs with go-mode.el**: tracking issue in [#237](#).

The status of other tools such as goimports, guru, gorelease and similar tools is being tracked in an umbrella issue [#24661](#). Please see that umbrella issue for latest status.

Some tracking issues for particular tools include:

- **gocode**: tracking issue in [mdempsky/gocode/#46](#). Note that `nsf/gocode` is recommending people migrate off of `nsf/gocode` to `mdempsky/gocode`.
- **go-tools** (tools by dominikh such as staticcheck, megacheck, gosimple): sample tracking issue [dominikh/go-tools#328](#).

In general, even if your editor, IDE or other tools have not yet been made module aware, much of their functionality should work with modules if you are using modules inside GOPATH and do `go mod vendor` (because then the proper dependencies should be picked up via GOPATH).

The full fix is to move programs that load packages off of `go/build` and onto `golang.org/x/tools/go/packages`, which understands how to locate packages in a module-aware manner. This will likely eventually become `go/packages`.

FAQs — Additional Control

What community tooling exists for working with modules?

The community is starting to build tooling on top of modules. For example:

- [github.com/rogpeppe/gohack](#)
 - A new community tool to automate and greatly simplify `replace` and multi-module workflows, including allowing you to easily modify one of your dependencies
 - For example, `gohack example.com/some/dependency` automatically clones the appropriate repository and adds the necessary `replace` directives to your `go.mod`
 - Remove all gohack replace statements with `gohack undo`
 - The project is continuing to expand to make other module-related workflows easier
- [github.com/marwan-at-work/mod](#)
 - Command line tool to automatically upgrade/downgrade major versions for modules
 - Automatically adjusts `go.mod` files and related import statements in go source code
 - Helps with upgrades, or when first opting into modules with a v2+ package
- [github.com/akyoto/mgit](#)
 - Lets you view & control semver tags of all of your local projects
 - Shows untagged commits and lets you tag them all at once (`mgit -tag +0.0.1`)
- [github.com/goware/modvendor](#)
 - Helps copy additional files into the `vendor/` folder, such as shell scripts, .cpp files, .proto files, etc.
- [github.com/psampaz/go-mod-outdated](#)
 - Displays outdated dependencies in a human friendly way
 - Provides a way to filter indirect dependencies and dependencies without updates
 - Provides a way to break CI pipelines in case of outdated dependencies
- [github.com/oligot/go-mod-upgrade](#)

- Update outdated Go dependencies interactively

When should I use the replace directive?

As described in the ['go.mod' concepts section above](#), `replace` directives provide additional control in the top-level `go.mod` for what is actually used to satisfy a dependency found in the Go source or `go.mod` files, while `replace` directives in modules other than the main module are ignored when building the main module.

The `replace` directive allows you to supply another import path that might be another module located in VCS (GitHub or elsewhere), or on your local filesystem with a relative or absolute file path. The new import path from the `replace` directive is used without needing to update the import paths in the actual source code.

`replace` allows the top-level module control over the exact version used for a dependency, such as:

- `replace example.com/some/dependency => example.com/some/dependency v1.2.3`

`replace` also allows the use of a forked dependency, such as:

- `replace example.com/some/dependency => example.com/some/dependency-fork v1.2.3`

You can also reference branches, for example:

- `replace example.com/some/dependency => example.com/some/dependency-fork master`

One sample use case is if you need to fix or investigate something in a dependency, you can have a local fork and add something like the following in your top-level `go.mod` :

- `replace example.com/original/import/path => /your/forked/import/path`

`replace` also can be used to inform the go tooling of the relative or absolute on-disk location of modules in a multi-module project, such as:

- `replace example.com/project/foo => ../foo`

Note: if the right-hand side of a `replace` directive is a filesystem path, then the target must have a `go.mod` file at that location. If the `go.mod` file is not present, you can create one with `go mod init .`

In general, you have the option of specifying a version to the left of the `=>` in a `replace` directive, but typically it is less sensitive to change if you omit that (e.g., as done in all of the `replace` examples above).

A `require` directive is needed for each `replace` directive of a direct dependency. When replacing a dependency from a filesystem path, the version of the corresponding `require` directive is essentially ignored; in this case, the pseudoversion `v0.0.0` is a good choice to make this clear, e.g. `require example.com/module v0.0.0` .

You can confirm you are getting your expected versions by running `go list -m all` , which shows you the actual final versions that will be used in your build including taking into account `replace` statements.

See the ['go mod edit' documentation](#) for more details.

github.com/rogppe/gohack makes these types of workflows much easier, especially if your goal is to have mutable checkouts of dependencies of a module. See the [repository](#) or the immediately prior FAQ for an overview.

See the next FAQ for the details of using `replace` to work entirely outside of VCS.

Can I work entirely outside of VCS on my local filesystem?

Yes. VCS is not required.

This is very simple if you have a single module you want to edit at a time outside of VCS (and you either have only one module in total, or if the other modules reside in VCS). In this case, you can place the file tree containing the single `go.mod` in a convenient location. Your `go build`, `go test` and similar commands will work even if your single module is outside of VCS (without requiring any use of `replace` in your `go.mod`).

If you want to have multiple inter-related modules on your local disk that you want to edit at the same time, then `replace` directives are one approach. Here is a sample `go.mod` that uses a `replace` with a relative path to point the `hello` module at the on-disk location of the `goodbye` module (without relying on any VCS):

```
module example.com/me/hello

require (
    example.com/me/goodbye v0.0.0
)

replace example.com/me/goodbye => ../goodbye
```

A small runnable example is shown in this [thread](#).

How do I use vendoring with modules? Is vendoring going away?

The initial series of `vgo` blog posts did propose dropping vendoring entirely, but [feedback](#) from the community resulted in retaining support for vendoring.

In brief, to use vendoring with modules:

- `go mod vendor` resets the main module's vendor directory to include all packages needed to build and test all of the module's packages based on the state of the `go.mod` files and Go source code.
- By default, `go` commands like `go build` ignore the vendor directory when in module mode.
- The `-mod=vendor` flag (e.g., `go build -mod=vendor`) instructs the `go` commands to use the main module's top-level vendor directory to satisfy dependencies. The `go` commands in this mode therefore ignore the dependency descriptions in `go.mod` and assume that the vendor directory holds the correct copies of dependencies. Note that only the main module's top-level vendor directory is used; vendor directories in other locations are still ignored.
- Some people will want to routinely opt-in to vendoring by setting a `GOFLAGS=-mod=vendor` environment variable.

Older versions of Go such as 1.10 understand how to consume a vendor directory created by `go mod vendor`, as do Go 1.11 and 1.12+ when [module mode](#) is disabled. Therefore, vendoring is one way for a module to provide dependencies to older versions of Go that do not fully understand modules, as well as to consumers that have not enabled modules themselves.

If you are considering using vendoring, it is worthwhile to read the ["Modules and vendoring"](#) and ["Make vendored copy of dependencies"](#) sections of the tip documentation.

Are there "always on" module repositories and enterprise proxies?

Publicly hosted "always on" immutable module repositories and optional privately hosted proxies and repositories are becoming available.

For example:

- proxy.golang.org - Official project - Run by [Google](#) - The default Go module proxy built by the Go team.
- proxy.golang.com.cn - China proxy project - Run by [China Golang Contributor Club](#) - China Go module proxy.
- mirrors.tencent.com/go - Commercial project - Run by [Tencent Cloud](#) - A Go module proxy alternate.
- mirrors.aliyun.com/goproxy - Commercial project - Run by [Alibaba Cloud](#) - A Go module proxy alternate.
- goproxy.cn - Open source project - Run by [Qiniu Cloud](#) - The most trusted Go module proxy in China.
- goproxy.io - Open source project - Run by [China Golang Contributor Club](#) - A global proxy for Go modules.
- [Athens](#) - Open source project - Self-hosted - A Go module datastore and proxy.
- athens.azurefd.net - Open source project - A hosted module proxy running Athens.
- [Goproxy](#) - Open source project - Self-hosted - A minimalist Go module proxy handler.
- [THUMBAL](#) - Open source project - Self-hosted - Go mod proxy server and Go vanity import path server.

Note that you are not required to run a proxy. Rather, the go tooling in 1.11 has added optional proxy support via [GOPROXY](#) to enable more enterprise use cases (such as greater control), and also to better handle situations such as "GitHub is down" or people deleting GitHub repositories.

Can I control when go.mod gets updated and when the go tools use the network to satisfy dependencies?

By default, a command like `go build` will reach out to the network as needed to satisfy imports.

Some teams will want to disallow the go tooling from touching the network at certain points, or will want greater control regarding when the go tooling updates `go.mod`, how dependencies are obtained, and how vendoring is used.

The go tooling provides a fair amount of flexibility to adjust or disable these default behaviors, including via `-mod=readonly`, `-mod=vendor`, `GOFLAGS`, `GOPROXY=off`, `GOPROXY=file:///filesystem/path`, `go mod vendor`, and `go mod download`.

The details on these options are spread throughout the official documentation. One community attempt at a consolidated overview of knobs related to these behaviors is [here](#), which includes links to the official documentation for more information.

How do I use modules with CI systems such as Travis or CircleCI?

The simplest approach is likely just setting the environment variable `GO111MODULE=on`, which should work with most CI systems.

However, it can be valuable to run tests in CI on Go 1.11 with modules enabled as well as disabled, given some of your users will not have yet opted in to modules themselves. Vendoring is also a topic to consider.

The following two blog posts cover these topics more concretely:

- ["Using Go modules with vendor support on Travis CI"](#) by Fatih Arslan
- ["Go Modules and CircleCI"](#) by Todd Keech

How do I download modules needed to build specific packages or tests?

The `go mod download` command (or equivalently, `go mod download all`) downloads all modules in the build list (as reported by `go list -m all`). Many of these modules aren't needed to build packages in the main module, since the full build list contains things like test dependencies and tool dependencies for other modules. Consequently, Docker images prepared with `go mod download` may be larger than necessary.

Instead, consider using `go list`. For example, `go list ./...` will download the modules needed to build the packages `./...` (the set of packages in the main module, when run from the module root directory).

To download test dependencies as well, use `go list -test ./...`.

By default, `go list` will only consider dependencies needed for the current platform. You can set `GOOS` and `GOARCH` to make `go list` consider another platform, for example, `GOOS=linux GOARCH=amd64 go list ./...`. The `-tags` flag may also be used to select packages with specific build tags.

This technique may be less necessary in the future when lazy module loading is implemented (see [#36460](#)), since the module pattern `all` will include fewer modules.

FAQs — go.mod and go.sum

Why does 'go mod tidy' record indirect and test dependencies in my 'go.mod'?

The modules system records precise dependency requirements in your `go.mod`. (For more details, see the [go.mod concepts](#) section above or the [go.mod tip documentation](#)).

`go mod tidy` updates your current `go.mod` to include the dependencies needed for tests in your module — if a test fails, we must know which dependencies were used in order to reproduce the failure.

`go mod tidy` also ensures your current `go.mod` reflects the dependency requirements for all possible combinations of OS, architecture, and build tags (as described [here](#)). In contrast, other commands like `go build` and `go test` only update `go.mod` to provide the packages imported by the requested packages under the current `GOOS`, `GOARCH`, and build tags (which is one reason `go mod tidy` might add requirements that were not added by `go build` or similar).

If a dependency of your module does not itself have a `go.mod` (e.g., because the dependency has not yet opted in to modules itself), or if its `go.mod` file is missing one or more of its dependencies (e.g., because the module author did not run `go mod tidy`), then the missing transitive dependencies will be added to *your* module's requirements, along with an `// indirect` comment to indicate that the dependency is not from a direct import within your module.

Note that this also means that any missing test dependencies from your direct or indirect dependencies will also be recorded in your `go.mod`. (An example of when this is important: `go test all` runs the tests of *all* direct and indirect dependencies of your module, which is one way to validate that your current combination of versions work together. If a test fails in one of your dependencies when you run `go test all`, it is important to have a complete set of test dependency information recorded so that you have reproducible `go test all` behavior).

Another reason you might have `// indirect` dependencies in your `go.mod` file is if you have upgraded (or downgraded) one of your indirect dependencies beyond what is required by your direct dependencies, such as if you ran `go get -u` or `go get foo@1.2.3`. The go tooling needs a place to record those new versions, and it does so in your `go.mod` file (and it does not reach down into your dependencies to modify *their* `go.mod` files).

In general, the behaviors described above are part of how modules provide 100% reproducible builds and tests by recording precise dependency information.

If you are curious as to why a particular module is showing up in your `go.mod`, you can run `go mod why -m <module>` to [answer](#) that question. Other useful tools for inspecting requirements and versions include `go mod graph` and `go list -m all`.

Is 'go.sum' a lock file? Why does 'go.sum' include information for module versions I am no longer using?

No, `go.sum` is not a lock file. The `go.mod` files in a build provide enough information for 100% reproducible builds.

For validation purposes, `go.sum` contains the expected cryptographic checksums of the content of specific module versions. See the [FAQ below](#) for more details on `go.sum` (including why you typically should check in `go.sum`) as well as the "[Module downloading and verification](#)" section in the tip documentation.

In addition, your module's `go.sum` records checksums for all direct and indirect dependencies used in a build (and hence your `go.sum` will frequently have more modules listed than your `go.mod`).

Should I commit my 'go.sum' file as well as my 'go.mod' file?

Typically your module's `go.sum` file should be committed along with your `go.mod` file.

- `go.sum` contains the expected cryptographic checksums of the content of specific module versions.
- If someone clones your repository and downloads your dependencies using the `go` command, they will receive an error if there is any mismatch between their downloaded copies of your dependencies and the corresponding entries in your `go.sum`.
- In addition, `go mod verify` checks that the on-disk cached copies of module downloads still match the entries in `go.sum`.
- Note that `go.sum` is not a lock file as used in some alternative dependency management systems. (`go.mod` provides enough information for reproducible builds).
- See very brief [rationale here](#) from Filippo Valsorda on why you should check in your `go.sum`. See the "[Module downloading and verification](#)" section of the tip documentation for more details. See possible future extensions being discussed for example in [#24117](#) and [#25530](#).

Should I still add a 'go.mod' file if I do not have any dependencies?

Yes. This supports working outside of GOPATH, helps communicate to the ecosystem that you are opting in to modules, and in addition the `module` directive in your `go.mod` serves as a definitive declaration of the identity of your code (which is one reason why import comments might eventually be deprecated). Of course, modules are purely an opt-in capability in Go 1.11.

FAQs — Semantic Import Versioning

Why must major version numbers appear in import paths?

Please see the discussion on the Semantic Import Versioning and the import compatibility rule in the "[Semantic Import Versioning](#)" concepts section above. See also the [blog post announcing the proposal](#), which talks more about the motivation and justification for the import compatibility rule.

Why are major versions v0, v1 omitted from import paths?"

Please see the question "Why are major versions v0, v1 omitted from import paths?" in the earlier [FAQ from the official proposal discussion](#).

What are some implications of tagging my project with major version v0, v1, or making breaking changes with v2+?

In response to a comment about "*k8s does minor releases but changes the Go API in each minor release*", Russ Cox made the following [response](#) that highlights some implications for picking v0, v1, vs. frequently making breaking changes with v2, v3, v4, etc. with your project:

I don't fully understand the k8s dev cycle etc, but I think generally the k8s team needs to decide/confirm what they intend to guarantee to users about stability and then apply version numbers accordingly to express that.

- *To make a promise about API compatibility (which seems like the best user experience!) then start doing that and use 1.X.Y.*
- *To have the flexibility to make backwards-incompatible changes in every release but allow different parts of a large program to upgrade their code on different schedules, meaning different parts can use different major versions of the API in one program, then use X.Y.0, along with import paths like k8s.io/client/vX/foo.*
- *To make no promises about API compatible and also require every build to have only one copy of the k8s libraries no matter what, with the implied forcing of all parts of a build to use the same version even if not all of them are ready for it, then use 0.X.Y.*

On a related note, Kubernetes has some atypical build approaches (currently including custom wrapper scripts on top of godep), and hence Kubernetes is an imperfect example for many other projects, but it will likely be an interesting example as [Kubernetes moves towards adopting Go 1.11 modules](#).

Can a module consume a package that has not opted in to modules?

Yes.

If a repository has not opted in to modules but has been tagged with valid [semver](#) tags (including the required leading `v`), then those semver tags can be used in a `go get` , and a corresponding semver version will be recorded in the importing module's `go.mod` file. If the repository does not have any valid semver tags, then the repository's version will be recorded with a "[pseudo-version](#)" such as `v0.0.0-20171006230638-a6e239ea1c69` (which includes a timestamp and a commit hash, and which are designed to allow a total ordering across versions recorded in `go.mod` and to make it easier to reason about which recorded versions are "later" than another recorded version).

For example, if the latest version of package `foo` is tagged `v1.2.3` but `foo` has not itself opted in to modules, then running `go get foo` or `go get foo@v1.2.3` from inside module M will be recorded in module M's `go.mod` file as:

```
require foo v1.2.3
```

The `go` tool will also use available semver tags for a non-module package in additional workflows (such as `go list -u=patch` , which upgrades the dependencies of a module to available patch releases, or `go list -u -m all` , which shows available upgrades, etc.).

Please see the next FAQs for additional details related to v2+ packages that have not opted in to modules.

Can a module consume a v2+ package that has not opted into modules? What does '+incompatible' mean?

Yes, a module can import a v2+ package that has not opted into modules, and if the imported v2+ package has a valid [semver](#) tag, it will be recorded with a `+incompatible` suffix.

Additional Details

Please be familiar with the material in the "[Semantic Import Versioning](#)" section above.

It is helpful to first review some core principles that are generally useful but particularly important to keep in mind when thinking about the behavior described in this FAQ.

The following core principles are *always* true when the `go` tool is operating in module mode (e.g.,

```
GO111MODULE=on
```

);

1. A package's import path defines the identity of the package.
 - Packages with *different* import paths are treated as *different* packages.
 - Packages with the *same* import path are treated as the *same* package (and this is true *even if* the VCS tags say the packages have different major versions).
2. An import path without a `/vN` is treated as a v1 or v0 module (and this is true *even if* the imported package has not opted in to modules and has VCS tags that say the major version is greater than 1).
3. The module path (such as `module foo/v2`) declared at the start of a module's `go.mod` file is both:
 - the definitive declaration of that module's identity
 - the definitive declaration of how that module must be imported by consuming code

As we will see in the next FAQ, these principles are not always true when the `go` tool is *not* in module mode, but these principles are always true when the `go` tool is in module mode.

In short, the `+incompatible` suffix indicates that principle 2 above is in effect when the following are true:

- an imported package has not opted in to modules, and
- its VCS tags say the major version is greater than 1, and
- principle 2 is overriding the VCS tags – the import path without a `/vN` is treated as a v1 or v0 module (even though the VCS tags say otherwise)

When the `go` tool is in module mode, it will assume a non-module v2+ package has no awareness of Semantic Import Versioning and treat it as an (incompatible) extension of the v1 version series of the package (and the `+incompatible` suffix is an indication that the `go` tool is doing so).

Example

Suppose:

- `oldpackage` is a package that predates the introduction of modules
- `oldpackage` has never opted in to modules (and hence does not have a `go.mod` itself)
- `oldpackage` has a valid semver tag `v3.0.1`, which is its latest tag

In this case, running for example `go get oldpackage@latest` from inside module M will record the following in module M's `go.mod` file:

```
require oldpackage v3.0.1+incompatible
```

Note that there is no `/v3` used at the end of `oldpackage` in the `go get` command above or in the recorded `require` directive – using `/vN` in module paths and import paths is a feature of [Semantic Import Versioning](#), and `oldpackage` has not signaled its acceptance and understanding of Semantic Import Versioning given `oldpackage` has not opted into modules by having a `go.mod` file within `oldpackage` itself. In other words, even though `oldpackage` has a [semver](#) tag of `v3.0.1`, `oldpackage` is not granted the rights and responsibilities of [Semantic Import Versioning](#) (such as using `/vN` in import paths) because `oldpackage` has not yet stated its desire to do so.

The `+incompatible` suffix indicates that the `v3.0.1` version of `oldpackage` has not actively opted in to modules, and hence the `v3.0.1` version of `oldpackage` is assumed to *not* understand Semantic Import

Versioning or how to use major versions in import paths. Therefore, when operating in [module mode](#), the `go` tool will treat the non-module `v3.0.1` version of `oldpackage` as an (incompatible) extension of the `v1` version series of `oldpackage` and assume that the `v3.0.1` version of `oldpackage` has no awareness of Semantic Import Versioning, and the `+incompatible` suffix is an indication that the `go` tool is doing so.

The fact that the `v3.0.1` version of `oldpackage` is considered to be part of the `v1` release series according to Semantic Import Versioning means for example that versions `v1.0.0`, `v2.0.0`, and `v3.0.1` are all always imported using the same import path:

```
import "oldpackage"
```

Note again that there is no `/v3` used at the end of `oldpackage`.

In general, packages with different import paths are different packages. In this example, given versions `v1.0.0`, `v2.0.0`, and `v3.0.1` of `oldpackage` would all be imported using the same import path, they are therefore treated by a build as the same package (again because `oldpackage` has not yet opted in to Semantic Import Versioning), with a single copy of `oldpackage` ending up in any given build. (The version used will be the semantically highest of the versions listed in any `require` directives; see ["Version Selection"](#)).

If we suppose that later a new `v4.0.0` release of `oldpackage` is created that adopts modules and hence contains a `go.mod` file, that is the signal that `oldpackage` now understands the rights and responsibilities of Semantic Import Versioning, and hence a module-based consumer would now import using `/v4` in the import path:

```
import "oldpackage/v4"
```

and the version would be recorded as:

```
require oldpackage/v4 v4.0.0
```

`oldpackage/v4` is now a different import path than `oldpackage`, and hence a different package. Two copies (one for each import path) would end up in a module-aware build if some consumers in the build have `import "oldpackage/v4"` while other consumers in the same build have `import "oldpackage"`. This is desirable as part of the strategy to allow gradual adoption of modules. In addition, even after modules are out of their current transitional phase, this behavior is also desirable to allow gradual code evolution over time with different consumers upgrading at different rates to newer versions (e.g., allowing different consumers in a large build to choose to upgrade at different rates from `oldpackage/v4` to some future `oldpackage/v5`).

How are v2+ modules treated in a build if modules support is not enabled? How does "minimal module compatibility" work in 1.9.7+, 1.10.3+, and 1.11?

When considering older Go versions or Go code that has not yet opted in to modules, Semantic Import Versioning has significant backwards-compatibility implications related to v2+ modules.

As described in the ["Semantic Import Versioning"](#) section above:

- a module that is version v2 or higher must include a `/vN` in its own module path declared in its `go.mod`.
- a module-based consumer (that is, code that has opted in to modules) must include a `/vN` in the import path to import a v2+ module.

However, the ecosystem is expected to proceed at varying paces of adoption for modules and Semantic Import Versioning.

As described in more detail in the ["How to Release a v2+ Module"](#) section, in the "Major Subdirectory" approach, the author of a v2+ module creates subdirectories such as `mymodule/v2` or `mymodule/v3` and moves or copies the appropriate packages underneath those subdirectories. This means the traditional import path logic (even in older Go releases such as Go 1.8 or 1.7) will find the appropriate packages upon seeing an import statement such as `import "mymodule/v2/mypkg"`. Hence, packages residing in a "Major Subdirectory" v2+ module will be found and used even if modules support is not enabled (whether that is because you are running Go 1.11 and have not enabled modules, or because you are running a older version like Go 1.7, 1.8, 1.9 or 1.10 that does not have full module support). Please see the ["How to Release a v2+ Module"](#) section for more details on the "Major Subdirectory" approach.

The remainder of this FAQ is focused on the "Major Branch" approach described in the ["How to Release a v2+ Module"](#) section. In the "Major Branch" approach, no `/vN` subdirectories are created and instead the module version information is communicated by the `go.mod` file and by applying semver tags to commits (which often will be on `master`, but could be on different branches).

In order to help during the current transitional period, "minimal module compatibility" was [introduced](#) to Go 1.11 to provide greater compatibility for Go code that has not yet opted in to modules, and that "minimal module compatibility" was also backported to Go 1.9.7 and 1.10.3 (where those versions are effectively always operating with full module mode disabled given those older Go versions do not have full module support).

The primary goals of "minimal module compatibility" are:

1. Allow older Go versions 1.9.7+ and 1.10.3+ to be able to more easily compile modules that are using Semantic Import Versioning with `/vN` in import paths, and provide that same behavior when [module mode](#) is disabled in Go 1.11.
2. Allow old code to be able to consume a v2+ module without requiring that old consumer code to immediately change to using a new `/vN` import path when consuming a v2+ module.
3. Do so without relying on the module author to create `/vN` subdirectories.

Additional Details – "Minimal Module Compatibility"

"Minimal module compatibility" only takes effect when full [module mode](#) is disabled for the `go` tool, such as if you have set `GOL11MODULE=off` in Go 1.11, or are using Go versions 1.9.7+ or 1.10.3+.

When a v2+ module author has *not* created `/v2` or `/vN` subdirectories and you are instead relying on the "minimal module compatibility" mechanism in Go 1.9.7+, 1.10.3+ and 1.11:

- A package that has *not* opted in to modules would *not* include the major version in the import path for any imported v2+ modules.
- In contrast, a package that *has* opted in to modules *must* include the major version in the import path to import any v2+ modules.
 - If a package has opted in to modules, but does not include the major version in the import path when importing a v2+ modules, it will not import a v2+ version of that module when the `go` tool is operating in full module mode. (A package that has opted in to modules is assumed to "speak" Semantic Import Versioning. If `foo` is a module with v2+ versions, then under Semantic Import Versioning saying `import "foo"` means import the v1 Semantic Import Versioning series of `foo`).
- The mechanism used to implement "minimal module compatibility" is intentionally very narrow:
 - The entirety of the logic is – when operating in GOPATH mode, an unresolvable import statement containing a `/vN` will be tried again after removing the `/vN` if the import statement is inside

code that has opted in to modules (that is, import statements in `.go` files within a tree with a valid `go.mod` file).

- The net effect is that an import statement such as `import "foo/v2"` within code that lives inside of a module will still compile correctly in GOPATH mode in 1.9.7+, 1.10.3+ and 1.11, and it will resolve as if it said `import "foo"` (without the `/v2`), which means it will use the version of `foo` that resides in your GOPATH without being confused by the extra `/v2`.
- "Minimal module compatibility" does not affect anything else, including it does not affect paths used in the `go` command line (such as arguments to `go get` or `go list`).
- This transitional "minimal module awareness" mechanism purposefully breaks the rule of "packages with different import paths are treated as different packages" in pursuit a very specific backwards-compatibility goal – to allow old code to compile unmodified when it is consuming a v2+ module. In slightly more detail:
 - It would be a more burdensome for the overall ecosystem if the only way for old code to consume a v2+ module was to first change the old code.
 - If we are not modifying old code, then that old code must work with pre-module import paths for v2+ modules.
 - On the other hand, new or updated code opting in to modules must use the new `/vN` import for v2+ modules.
 - The new import path is not equal to old import path, yet both are allowed to work in a single build, and therefore we have two different functioning import paths that resolve to the same package.
 - For example, when operating in GOPATH mode, `import "foo/v2"` appearing in module-based code resolves to the same code residing in your GOPATH as `import "foo"`, and the build ends up with one copy of `foo` – in particular, whatever version is on disk in GOPATH. This allows module-based code with `import "foo/v2"` to compile even in GOPATH mode in 1.9.7+, 1.10.3+ and 1.11.
- In contrast, when the `go` tool is operating in full module mode:
 - There are no exceptions to the rule "packages with different import paths are different packages" (including vendoring has been refined in full module mode to also adhere to this rule).
 - For example, if the `go` tool is in full module mode and `foo` is a v2+ module, then `import "foo"` is asking for a v1 version of `foo` vs. `import "foo/v2"` is asking for a v2 version of `foo`.

What happens if I create a `go.mod` but do not apply semver tags to my repository?

[semver](#) is a foundation of the modules system. In order to provide the best experience for consumers, module authors are encouraged to apply semver VCS tags (e.g., `v0.1.0` or `v1.2.3-rc.1`), but semver VCS tags are not strictly required:

1. Modules are required to follow the *semver specification* in order for the `go` command to behave as documented. This includes following the semver specification regarding how and when breaking changes are allowed.
2. Modules that do not have semver VCS tags are recorded by consumers using a semver version in the form of a [pseudo-version](#). Typically this will be a v0 major version, unless the module author constructed a v2+ module following the ["Major Subdirectory"](#) approach.
3. Therefore, modules that do not apply semver VCS tags and have not created a "Major Subdirectory" are effectively declaring themselves to be in the semver v0 major version series, and a module-based consumer will treat them as having a semver v0 major version.

Can a module depend on a different version of itself?

A module can depend on a different major version of itself: by-and-large, this is comparable to depending on a different module. This can be useful for different reasons, including to allow a major version of a module to be implemented as a shim around a different major version.

In addition, a module can depend on a different major version of itself in a cycle, just as two completely different modules can depend on each other in a cycle.

However, if you are not expecting a module to depend on a different version of itself, it can be a sign of a mistake. For example, .go code intending to import a package from a v3 module might be missing the required `/v3` in the import statement. That mistake can manifest as a v3 module depending on the v1 version of itself.

If you are surprised to see a module to depend on a different version of itself, it can be worthwhile to review the ["Semantic Import Versioning"](#) section above along with the FAQ ["What can I check if I am not seeing the expected version of a dependency?"](#).

It continues to be a constraint that two *packages* may not depend on each other in a cycle.

FAQS — Multi-Module Repositories

What are multi-module repositories?

A multi-module repository is a repository that contains multiple modules, each with its own go.mod file. Each module starts at the directory containing its go.mod file, and contains all packages from that directory and its subdirectories recursively, excluding any subtree that contains another go.mod file.

Each module has its own version information. Version tags for modules below the root of the repository must include the relative directory as a prefix. For example, consider the following repository:

```
my-repo
|-- foo
|   |-- rop
|       |-- go.mod
```

The tag for version 1.2.3 of module "my-repo/foo/rop" is "foo/rop/v1.2.3".

Typically, the path for one module in the repository will be a prefix of the others. For example, consider this repository:

```
my-repo
|-- bar
|-- foo
|   |-- rop
|   |-- yut
|-- go.mod
|-- mig
|   |-- go.mod
|   |-- vub
```


 Fig. A top-level module's path is a prefix of another module's path.

Fig. A top-level module's path is a prefix of another module's path.

This repository contains two modules. However, the module "my-repo" is a prefix of the path of the module "my-repo/mig".

Should I have multiple modules in a single repository?

Adding modules, removing modules, and versioning modules in such a configuration require considerable care and deliberation, so it is almost always easier and simpler to manage a single-module repository rather than multiple modules in an existing repository.

Russ Cox commented in [#26664](#):

For all but power users, you probably want to adopt the usual convention that one repo = one module. It's important for long-term evolution of code storage options that a repo can contain multiple modules, but it's almost certainly not something you want to do by default.

Two examples of how multi-modules can be more work:

- `go test ./...` from the repository root will no longer test everything in the repository
- you might need to routinely manage the relationship between the modules via `replace` directives.

However, there is additional nuance beyond those two examples. Please read the FAQs in this [sub-section](#) carefully if you are considering having multiple modules in a single repository.

Two example scenarios where it can make sense to have more than one `go.mod` in a repository:

1. if you have usage examples where the examples themselves have a complex set of dependencies (e.g., perhaps you have a small package but include an example of using your package with kubernetes). In that case, it can make sense for your repository to have an `example` or `_example` directory with its own `go.mod`, such as shown [here](#).
2. if you have a repository with a complex set of dependencies, but you have a client API with a smaller set of dependencies. In some cases, it might make sense to have an `api` or `clientapi` or similar directory with its own `go.mod`, or to separate out that `clientapi` into its own repository.

However, for both of those cases, if you are considering creating a multi-module repository for performance or download size for a large set of indirect dependencies, you are strongly encouraged to first try with a GOPROXY, which will be enabled by default in Go 1.13. Using a GOPROXY mostly equals any performance benefits or dependency download size benefits that might otherwise come from creating a multi-module repository.

Is it possible to add a module to a multi-module repository?

Yes. However, there are two classes of this problem:

The first class: the package to which the module is being added to is not in version control yet (a new package). This case is straightforward: add the package and the `go.mod` in the same commit, tag the commit, and push.

The second class: the path at which the module is being added is in version control and contains one or more existing packages. This case requires a considerable amount of care. To illustrate, consider again the following repository (now in a github.com location to simulate the real-world better):

```
github.com/my-repo
|-- bar
|-- foo
|   |-- rop
|   `-- yut
```

```
|-- go.mod
`-- mig
    |-- vub
```

Consider adding module "github.com/my-repo/mig". If one were to follow the same approach as above, the package /my-repo/mig could be provided by two different modules: the old version of "github.com/my-repo", and the new, standalone module "github.com/my-repo/mig". If both modules are active, importing "github.com/my-repo/mig" would cause an "ambiguous import" error at compile time.

The way to get around this is to make the newly-added module depend on the module it was "carved out" from, at a version after which it was carved out.

Let's step through this with the above repository, assuming that "github.com/my-repo" is currently at v1.2.3:

1. Add github.com/my-repo/mig/go.mod:

```
cd path-to/github.com/my-repo/mig
go mod init github.com/my-repo/mig

# Note: if "my-repo/mig" does not actually depend on "my-repo", add a blank
# import.
# Note: version must be at or after the carve-out.
go mod edit -require github.com/myrepo@v1.3
```

2. `git commit`

3. `git tag v1.3.0`

4. `git tag mig/v1.0.0`

5. Next, let's test these. We can't `go build` or `go test` naively, since the go commands would try to fetch each dependent module from the module cache. So, we need to use replace rules to cause `go` commands to use the local copies:

```
cd path-to/github.com/my-repo/mig
go mod edit -replace github.com/my-repo@v1.3.0=../
go test ./...
go mod edit -dropreplace github.com/my-repo@v1.3.0
```

6. `git push origin master v1.3.0 mig/v1.0.0` push the commit and both tags

Note that in the future golang.org/issue/28835 should make the testing step a more straightforward experience.

Note also that code was removed from module "github.com/my-repo" between minor versions. It may seem strange to not consider this a major change, but in this instance the transitive dependencies continue to provide compatible implementations of the removed packages at their original import paths.

Is it possible to remove a module from a multi-module repository?

Yes, with the same two cases and similar steps as above.

Can a module depend on an internal/ in another?

Yes. Packages in one module are allowed to import internal packages from another module as long as they share the same path prefix up to the internal/ path component. For example, consider the following repository:

```
my-repo
|-- foo
|   `-- go.mod
|-- go.mod
`-- internal
```

Here, package foo can import /my-repo/internal as long as module "my-repo/foo" depends on module "my-repo". Similarly, in the following repository:

```
my-repo
|-- foo
|   `-- go.mod
`-- internal
    `-- go.mod
```

Here, package foo can import my-repo/internal as long as module "my-repo/foo" depends on module "my-repo/internal". The semantics are the same in both: since my-repo is a shared path prefix between my-repo/internal and my-repo/foo, package foo is allowed to import package internal.

Can an additional go.mod exclude unnecessary content? Do modules have the equivalent of a .gitignore file?

One additional use case for having multiple `go.mod` files in a single repository is if the repository has files that should be pruned from a module. For example, a repository might have very large files that are not needed for the Go module, or a multi-language repository might have many non-Go files.

An empty `go.mod` in a directory will cause that directory and all of its subdirectories to be excluded from the top-level Go module.

If the excluded directory does not contain any `.go` files, no additional steps are needed beyond placing the empty `go.mod` file. If the excluded directory does contain `.go` files, please first carefully review the other FAQs in [this multi-module repository section](#).

FAQs — Minimal Version Selection

Won't minimal version selection keep developers from getting important updates?

Please see the question "Won't minimal version selection keep developers from getting important updates?" in the earlier [FAQ from the official proposal discussion](#).

FAQs — Possible Problems

What are some general things I can spot check if I am seeing a problem?

- Double-check that modules are enabled by running `go env` to confirm it does not show an empty value for the read-only `GOMOD` variable.
 - Note: you never set `GOMOD` as a variable because it is effectively read-only debug output that `go env` outputs.

- If you are setting `GO111MODULE=on` to enable modules, double-check that it is not accidentally the plural `GO111MODULES=on`. (People sometimes naturally include the `s` because the feature is often called "modules").
- If vendoring is expected to be used, check that the `-mod=vendor` flag is being passed to `go build` or similar, or that `GOFLAGS=-mod=vendor` is set.
 - Modules by default ignore the `vendor` directory unless you ask the `go` tool to use `vendor`.
- It is frequently helpful to check `go list -m all` to see the list of actual versions selected for your build
 - `go list -m all` usually gives you more detail compared to if you were to instead just look at a `go.mod` file.
- If running `go get foo` fails in some way, or if `go build` is failing on a particular package `foo`, it often can be helpful to check the output from `go get -v foo` or `go get -v -x foo`:
 - In general, `go get` will often provide more a detailed error message than `go build`.
 - The `-v` flag to `go get` asks to print more verbose details, though be mindful that certain "errors" such as 404 errors *might* be expected based on how a remote repository was configured.
 - If the nature of the problem is still not clear, you can also try the more verbose `go get -v -x foo`, which also shows the git or other VCS commands being issued. (If warranted, you can often execute the same git commands outside of the context of the `go` tool for troubleshooting purposes).
- You can check to see if you are using a particularly old git version
 - Older versions of git were a common source of problems for the `vgo` prototype and Go 1.11 beta, but much less frequently in the GA 1.11.
- The module cache in Go 1.11 can sometimes cause various errors, primarily if there were previously network issues or multiple `go` commands executing in parallel (see [#26794](#), which is addressed for Go 1.12). As a troubleshooting step, you can copy `$GOPATH/pkg/mod` to a backup directory (in case further investigation is warranted later), run `go clean -modcache`, and then see whether the original problem persists.
- If you are using Docker, it can be helpful to check if you can reproduce the behavior outside of Docker (and if the behavior only occurs in Docker, the list of bullets above can be used as a starting point to compare results between inside Docker vs. outside).

The error you are currently examining might be a secondary issue caused by not having the expected version of a particular module or package in your build. Therefore, if the cause of a particular error is not obvious, it can be helpful to spot check your versions as described in the next FAQ.

What can I check if I am not seeing the expected version of a dependency?

1. A good first step is to run `go mod tidy`. There is some chance this might resolve the issue, but it will also help put your `go.mod` file into a consistent state with respect to your `.go` source code, which will help make any subsequent investigation easier. (If `go mod tidy` itself changes the versions of a dependency in a way you don't expect, first read [this FAQ on 'go mod tidy'](#). If that does not explain it, you can try resetting your `go.mod` and then run `go list -mod=readonly all`, which might give a more specific message about whatever was requiring a change to its version).
2. The second step usually should be to check `go list -m all` to see the list of actual versions selected for your build. `go list -m all` shows you the final selected versions, including for indirect dependencies and after resolving versions for any shared dependencies. It also shows the outcome of any `replace` and `exclude` directives.

3. A good next step can be to examine the output of `go mod graph` or `go mod graph | grep <module-of-interest>`. `go mod graph` prints the module requirement graph (including taking into account replacements). Each line in the output has two fields: the first column is a consuming module, and the second column is one of that module's requirements (including the version required by that consuming module). This can be a quick way to see which modules are requiring a particular dependency, including when your build has a dependency that has different required versions from different consumers in your build (and if that is the case, it is important to be familiar with the behavior described in the ["Version Selection"](#) section above).

`go mod why -m <module>` can also be useful here, although it is typically more useful for seeing why a dependency is included at all (rather than why a dependency ends up with a particular version).

`go list` provides many more variations of queries that can be useful to interrogate your modules if needed. One example is the following, which will show the exact versions used in your build excluding test-only dependencies:

```
go list -deps -f '{{with .Module}}{{.Path}} {{.Version}}{{end}}' ./... | sort -u
```

A more detailed set of commands and examples for interrogating your modules can be seen in a runnable "Go Modules by Example" [walkthrough](#).

One cause of unexpected versions can be due to someone having created an invalid or unexpected `go.mod` file that was not intended, or a related mistake (for example: a `v2.0.1` version of module might have incorrectly declared itself to be `module foo` in its `go.mod` without the required `/v2`; an import statement in `.go` code intended to import a `v3` module might be missing the required `/v3`; a `require` statement in a `go.mod` for a `v4` module might be missing the required `/v4`). Therefore, if the cause of a particular issue you are seeing is not obvious, it can be worthwhile to first re-read the material in the ["go.mod"](#) and ["Semantic Import Versioning"](#) sections above (given these include important rules that modules must follow) and then take a few minutes to spot check the most relevant `go.mod` files and import statements.

Why am I getting an error 'cannot find module providing package foo'?

This is a general error message that can occur for several different underlying causes.

In some cases, this error is simply due to a mistyped path, so the first step likely should be to double-check for incorrect paths based on the details listed in the error message.

If you have not already done so, a good next step is often to try `go get -v foo` or `go get -v -x foo`:

- In general, `go get` will often provide more a detailed error message than `go build`.
- See the first troubleshooting FAQ in this section [above](#) for more details.

Some other possible causes:

- You might see the error `cannot find module providing package foo` if you have issued `go build` or `go build .` but do not have any `.go` source files in the current directory. If this is what you are encountering, the solution might be an alternative invocation such as `go build ./...` (where the `./...` expands out to match all the packages within the current module). See [#27122](#).
- The module cache in Go 1.11 can cause this error, including in the face of network issues or multiple `go` commands executing in parallel. This is resolved in Go 1.12. See the first troubleshooting FAQ in this section [above](#) for more details and possible corrective steps.

Why does 'go mod init' give the error 'cannot determine module path for source directory'?

`go mod init` without any arguments will attempt to guess the proper module path based on different hints such as VCS meta data. However, it is not expected that `go mod init` will always be able to guess the proper module path.

If `go mod init` gives you this error, those heuristics were not able to guess, and you must supply the module path yourself (such as `go mod init github.com/you/hello`).

I have a problem with a complex dependency that has not opted in to modules. Can I use information from its current dependency manager?

Yes. This requires some manual steps, but can be helpful in some more complex cases.

When you run `go mod init` when initializing your own module, it will automatically convert from a prior dependency manager by translating configuration files like `Gopkg.lock`, `glide.lock`, or `vendor.json` into a `go.mod` file that contains corresponding `require` directives. The information in a pre-existing `Gopkg.lock` file for example usually describes version information for all of your direct and indirect dependencies.

However, if instead you are adding a new dependency that has not yet opted in to modules itself, there is not a similar automatic conversion process from any prior dependency manager that your new dependency might have been using. If that new dependency itself has non-module dependencies that have had breaking changes, then in some cases that can cause incompatibility problems. In other words, a prior dependency manager of your new dependency is not automatically used, and that can cause problems with your indirect dependencies in some cases.

One approach is to run `go mod init` on your problematic non-module direct dependency to convert from its current dependency manager, and then use the `require` directives from the resulting temporary `go.mod` to populate or update the `go.mod` in your module.

For example, if `github.com/some/nonmodule` is a problematic direct dependency of your module that is currently using another dependency manager, you can do something similar to:

```
$ git clone -b v1.2.3 https://github.com/some/nonmodule /tmp/scratchpad/nonmodule
$ cd /tmp/scratchpad/nonmodule
$ go mod init
$ cat go.mod
```

The resulting `require` information from the temporary `go.mod` can be manually moved into the actual `go.mod` for your module, or you can consider using <https://github.com/rogppe/gomodmerge>, which is a community tool targeting this use case. In addition, you will want to add a `require` `github.com/some/nonmodule v1.2.3` to your actual `go.mod` to match the version that you manually cloned.

A concrete example of following this technique for docker is in this [#28489 comment](#), which illustrates getting a consistent set of versions of docker dependencies to avoid case sensitive issues between `github.com/sirupsen/logrus` vs. `github.com/Sirupsen/logrus`.

How can I resolve "parsing go.mod: unexpected module path" and "error loading module requirements" errors caused by a mismatch between import paths vs. declared module identity?

Why does this error occur?

In general, a module declares its identity in its `go.mod` via the `module` directive, such as `module example.com/m`. This is the "module path" for that module, and the `go` tool enforces consistency between that declared module path and the import paths used by any consumer. If a module's `go.mod` file reads `module example.com/m`, then a consumer must import packages from that module using import paths that start with that module path (e.g., `import "example.com/m"` or `import "example.com/m/sub/pkg"`).

The `go` command reports a `parsing go.mod: unexpected module path` fatal error if there is a mismatch between an import path used by a consumer vs. the corresponding declared module path. In addition, in some cases the `go` command will then report a more generic `error loading module requirements` error afterwards.

The most common cause of this error is if there was a name change (e.g., `github.com/Sirupsen/logrus` to `github.com/sirupsen/logrus`), or if a module was sometimes used via two different names prior to modules due to a vanity import path (e.g., `github.com/golang/sync` vs. the recommended `golang.org/x/sync`).

This can then cause problems if you have a dependency that is still being imported via an older name (e.g., `github.com/Sirupsen/logrus`) or a non-canonical name (e.g., `github.com/golang/sync`) but that dependency has subsequently adopted modules and now declares its canonical name in its `go.mod`. The error here can then trigger during an upgrade when the upgraded version of the module is found declaring a canonical module path that no longer matches the older import path.

Example problem scenario

- You are indirectly depending on `github.com/Quasilyte/go-consistent`.
- The project adopts modules, and then later changes its name to `github.com/quasilyte/go-consistent` (changing `Q` to lowercase `q`), which is a breaking change. GitHub forwards from the old name to the new name.
- You run `go get -u`, which attempts to upgrade all of your direct and indirect dependencies.
- `github.com/Quasilyte/go-consistent` is attempted to be upgraded, but the latest `go.mod` found now reads `module github.com/quasilyte/go-consistent`.
- The overall upgrade operation fails to complete, with error:

```
go: github.com/Quasilyte/go-consistent@v0.0.0-20190521200055-c6f3937de18c: parsing go.mod: unexpected module path "github.com/quasilyte/go-consistent" go get: error loading module requirements
```

Resolving

The most common form of the error is:

```
go: example.com/some/OLD/name@vX.Y.Z: parsing go.mod: unexpected module path "example.com/some/NEW/name"
```

If you visit the repository for `example.com/some/NEW/name` (from the right-side of the error), you can check the `go.mod` file for the latest release or `master` to see if it declares itself on the first line of the `go.mod` as `module example.com/some/NEW/name`. If so, that is a hint that you are seeing an "old module name" vs. "new module name" problem.

This remainder of this section focuses on resolving the "old name" vs. "new name" form of this the error by following these steps in sequence:

1. Check your own code to see if you are importing using `example.com/some/OLD/name`. If so, update your code to import using `example.com/some/NEW/name`.

2. If you received this error during an upgrade, you should try upgrading using the tip version of Go, which has more targeted upgrade logic ([#26902](#)) that can often sidestep this problem and also often has a better error message for this situation. Note that the `go get` arguments in tip / 1.13 are different than in 1.12. Example of obtaining tip and using it to upgrade your dependencies:

```
go get golang.org/dl/gotip && gotip download
gotip get -u all
gotip mod tidy
```

Because the problematic old import is often in an indirect dependency, upgrading with tip and then running `go mod tidy` can frequently upgrade you past the problematic version and then also remove the problematic version from your `go.mod` as no longer needed, which then puts you into a functioning state when you return to using Go 1.12 or 1.11 for day-to-day use. For example, see that approach work [here](#) to upgrade past `github.com/golang/lint` vs. `golang.org/x/lint` problems.

3. If you received this error while doing `go get -u foo` or `go get -u foo@latest`, try removing the `-u`. This will give you the set of dependencies used by `foo@latest` without upgrading the dependencies of `foo` past the versions that the author of `foo` likely verified as working when releasing `foo`. This can be important especially during this transitional time when some of the direct and indirect dependencies of `foo` might not yet have adopted [semver](#) or modules. (A common mistake is thinking `go get -u foo` solely gets the latest version of `foo`. In actuality, the `-u` in `go get -u foo` or `go get -u foo@latest` means to *also* get the latest versions for *all* of the direct and indirect dependencies of `foo`; that might be what you want, but it might not be especially if it is otherwise failing due to deep indirect dependencies).
4. If the steps above have not resolved the error, the next approach is slightly more complicated, but most often should work to resolve an "old name" vs. "new name" form of this error. This uses just information solely from the error message itself, plus some brief looking at some VCS history.
- 4.1. Go to the `example.com/some/NEW/name` repository
- 4.2. Determine when the `go.mod` file was introduced there (e.g., by looking at the blame or history view for the `go.mod`).
- 4.3. Pick the release or commit from *just before* the `go.mod` file was introduced there.
- 4.4. In your `go.mod` file, add a `replace` statement using the old name on both sides of the `replace` statement: `replace example.com/some/OLD/name => example.com/some/OLD/name <version-just-before-go.mod>`

Using our prior example where `github.com/Quasilyte/go-consistent` is the old name and `github.com/quasilyte/go-consistent` is the new name, we can see that the `go.mod` was first introduced there in commit [00c5b0cf371a](#). That repository is not using semver tags, so we will take the immediately prior commit [00dd7fb039e](#) and add it to the replace using the old uppercase Quasilyte name on both sides of the `replace`:

```
replace github.com/Quasilyte/go-consistent => github.com/Quasilyte/go-consistent
00dd7fb039e
```


This `replace` statement then enables us to upgrade past the problematic "old name" vs. "new name" mismatch by effectively preventing the old name from being upgraded to the new name in the presence of a `go.mod`. Usually, an upgrade via `go get -u` or similar can now avoid the error. If the upgrade completes, you can check to see if anyone is still importing the old name (e.g., `go mod graph | grep github.com/Quasilyte/go-consistent`) and if not, the `replace` can then be removed. (The reason this often works is because the upgrade itself can otherwise fail if an old problematic import path is used even though it might not be used in the final result if the upgrade had completed, which is tracked in [#30831](#)).

5. If the above steps have not resolved the problem, it might be because the problematic old import path is still in use by the latest version of one or more of your dependencies. In this case, it is important to identify who is still using the problematic old import path, and find or open an issue asking that the problematic importer change to using the now canonical import path. Using `gotip` in step 2. above might identify the problematic importer, but it does not do so in all cases, especially for upgrades ([#30661](#)). If it is unclear who is importing using the problematic old import path, you can usually find out by creating a clean module cache, performing the operation or operations that trigger the error, and then grepping for the old problematic import path within the module cache. For example:

```
export GOPATH=$(mktemp -d)
go get -u foo                # perform operation that generates the error of interest
cd $GOPATH/pkg/mod
grep -R --include="*.go" github.com/Quasilyte/go-consistent
```

6. If these steps are not sufficient to resolve the issue, or if you are a maintainer of a project that seems unable to remove references to an older problematic import path due to circular references, please see a much more detailed write-up of the problem on a separate [wiki page](#).

Finally, the above steps focus on how to resolve an underlying "old name" vs. "new name" problem. However, the same error message can also appear if a `go.mod` was placed in the wrong location or simply has the wrong module path. If that is the case, the importing that module should always fail. If you are importing a new module that you just created and has never been successfully imported before, you should check that the `go.mod` file is located correctly and that it has the proper module path that corresponds to that location. (The most common approach is a single `go.mod` per repository, with the single `go.mod` file placed in the repository root, and using the repository name as the module path declared in the `module` directive). See the ["go.mod"](#) section for more details.

Why does 'go build' require gcc, and why are prebuilt packages such as net/http not used?

In short:

Because the pre-built packages are non-module builds and can't be reused. Sorry. Disable cgo for now or install gcc.

This is only an issue when opting in to modules (e.g., via `GO111MODULE=on`). See [#26988](#) for additional discussion.

Do modules work with relative imports like `import "../subdir"` ?

No. See [#26645](#), which includes:

In modules, there finally is a name for the subdirectory. If the parent directory says "module m" then the subdirectory is imported as "m/subdir", no longer "../subdir".

Some needed files may not be present in populated vendor directory

Directories without `.go` files are not copied inside the `vendor` directory by `go mod vendor`. This is by design.

In short, setting aside any particular vendoring behavior – the overall model for go builds is that the files needed to build a package should be in the directory with the `.go` files.

Using the example of cgo – modifying C source code in other directories will not trigger a rebuild, and instead your build will use stale cache entries. The cgo documentation now [includes](#):

Note that changes to files in other directories do not cause the package to be recompiled, so all non-Go source code for the package should be stored in the package directory, not in subdirectories.

A community tool <https://github.com/goware/modvendor> allows you to easily copy a complete set of `.c`, `.h`, `.s`, `.proto` or other files from a module into the `vendor` directory. Although this can be helpful, some care must be taken to make sure your go build is being handled properly in general (regardless of vendoring) if you have files needed to build a package that are outside of the directory with the `.go` files.

See additional discussion in [#26366](#).

An alternative approach to traditional vendoring is to check in the module cache. It can end up with similar benefits as traditional vendoring and in some ways ends up with a higher fidelity copy. This approach is explained as a "Go Modules by Example" [walkthrough](#).