go.mod file reference

Table of Contents

Each Go module is defined by a go.mod file that describes the module's properties, including its dependencies on other modules and on versions of Go.

These properties include:

- The current module's **module path**. This should be a location from which the module can be downloaded by Go tools, such as the module code's repository location. This serves as a unique identifier, when combined with the module's version number. It is also the prefix of the package path for all packages in the module. For more about how Go locates the module, see the Go Modules Reference.
- The minimum **version of Go** required by the current module.
- A list of minimum versions of other **modules required** by the current module.
- Instructions, optionally, to **replace** a required module with another module version or a local directory, or to **exclude** a specific version of a required module.

Go generates a go.mod file when you run the go mod init command. The following example creates a go.mod file, setting the module's module path to example/mymodule:

```
$ go mod init example/mymodule
```

Use go commands to manage dependencies. The commands ensure that the requirements described in your go.mod file remain consistent and the content of your go.mod file is valid. These commands include the go get and go mod tidy and go mod edit commands.

For reference on go commands, see Command go. You can get help from the command line by typing go help *command-name*, as with go help mod tidy.

See also

- Go tools make changes to your go.mod file as you use them to manage dependencies. For more, see Managing dependencies.
- For more details and constraints related to go.mod files, see the Go modules reference.

Example

A go.mod file includes directives as shown in the following example. These are described elsewhere in this topic.

```
module example.com/mymodule

go 1.14

require (
    example.com/othermodule v1.2.3
    example.com/thismodule v1.2.3
    example.com/thatmodule v1.2.3
)

replace example.com/thatmodule => ../thatmodule
exclude example.com/thismodule v1.3.0
```

module

Declares the module's module path, which is the module's unique identifier (when combined with the module version number). The module path becomes the import prefix for all packages the module contains.

For more, see module directive in the Go Modules Reference.

Syntax

```
module module-path
```

module-path
: The module's module path, usually the repository location from which the module can be downloaded by Go tools. For module versions v2 and later, this value must end with the major version number, such as /v2.

Examples

The following examples substitute example.com for a repository domain from which the module could be downloaded.

- Module declaration for a v0 or v1 module:

  ```
  module example.com/mymodule
  ```

- Module path for a v2 module:

  ```
  module example.com/mymodule/v2
  ```

Notes

The module path must uniquely identify your module. For most modules, the path is a URL where the go command can find the code (or a redirect to the code). For modules that won't ever be downloaded directly, the module path can be just some name you control that will ensure uniqueness. The prefix example/ is also reserved for use in examples like these.

For more details, see Managing dependencies.

In practice, the module path is typically the module source's repository domain and path to the module code within the repository. The go command relies on this form when downloading module versions to resolve dependencies on the module user's behalf.

Even if you're not at first intending to make your module available for use from other code, using its repository path is a best practice that will help you avoid having to rename the module if you publish it later.

If at first you don't know the module's eventual repository location, consider temporarily using a safe substitute, such as the name of a domain you own or a name you control (such as your company name), along with a path following from the module's name or source directory. For more, see Managing dependencies.

For example, if you're developing in a stringtools directory, your temporary module path might be <company-name>/stringtools, as in the following example, where *company-name* is your company's name:

```
go mod init <company-name>/stringtools
```

go

Indicates that the module was written assuming the semantics of the Go version specified by the directive.

For more, see go directive in the Go Modules Reference.

Syntax

```
go minimum-go-version
```

minimum-go-version
    The minimum version of Go required to compile packages in this module.

Examples

- Module must run on Go version 1.14 or later:

  ```
  go 1.14
  ```

Notes

The go directive sets the minimum version of Go required to use this module. Before Go 1.21, the directive was advisory only; now it is a mandatory requirement: Go toolchains refuse to use modules declaring newer Go versions.

The go directive is an input into selecting which Go toolchain to run. See "Go toolchains" for details.

The go directive affects use of new language features:

- For packages within the module, the compiler rejects use of language features introduced after the version specified by the go directive. For example, if a module has the directive go 1.12, its packages may not use numeric literals like 1_000_000, which were introduced in Go 1.13.
- If an older Go version builds one of the module's packages and encounters a compile error, the error notes that the module was written for a newer Go version. For example, suppose a module has go 1.13 and a package uses the numeric literal 1_000_000. If that package is built with Go 1.12, the compiler notes that the code is written for Go 1.13.

The go directive also affects the behavior of the go command:

- At go 1.14 or higher, automatic vendoring may be enabled. If the file vendor/modules.txt is present and consistent with go.mod, there is no need to explicitly use the -mod=vendor flag.
- At go 1.16 or higher, the all package pattern matches only packages transitively imported by packages and tests in the main module. This is the same set of packages retained by go mod vendor since modules were introduced. In lower versions, all also includes tests of packages imported by packages in the main module, tests of those packages, and so on.
- At go 1.17 or higher:
  - The go.mod file includes an explicit require directive for each module that provides any package transitively imported by a package or test in the main module. (At go 1.16 and lower, an indirect dependency is included only if minimal version selection would otherwise select a different version.) This extra information enables module graph pruning and lazy module loading.
  - Because there may be many more // indirect dependencies than in previous go versions, indirect dependencies are recorded in a separate block within the go.mod file.
  - go mod vendor omits go.mod and go.sum files for vendored dependencies. (That allows invocations of the go command within subdirectories of vendor to identify the correct main module.)
  - go mod vendor records the go version from each dependency's go.mod file in vendor/modules.txt.
- At go 1.21 or higher:

  - The go line declares a required minimum version of Go to use with this module.
  - The go line must be greater than or equal to the go line of all dependencies.
  - The go command no longer attempts to maintain compatibility with the previous older version of Go.
  - The go command is more careful about keeping checksums of go.mod files in the go.sum file.

A go.mod file may contain at most one go directive. Most commands will add a go directive with the current Go version if one is not present.

toolchain

Declares a suggested Go toolchain to use with this module. Only takes effect when the module is the main module and the default toolchain is older than the suggested toolchain.

For more see "Go toolchains" and toolchain directive in the Go Modules Reference.

Syntax

```
toolchain toolchain-name
```

toolchain-name
: The suggested Go toolchain's name. Standard toolchain names take the form go*V* for a Go version *V*, as in go1.21.0 and go1.18rc1. The special value default disables automatic toolchain switching.

Examples

- Suggest using Go 1.21.0 or newer:

  ```
  toolchain go1.21.0
  ```

Notes

See "Go toolchains" for details about how the toolchain line affects Go toolchain selection.

godebug

Indicates the default GODEBUG settings to be applied to the main packages of this module. These override any toolchain defaults, and are overridden by explicit //go:debug lines in main packages.

Syntax

```
godebug debug-key=debug-value
```

debug-key
: The name of the setting to be applied. A list of settings and the versions they were introduced in can be found at GODEBUG History.

debug-value
: The value provided to the setting. If not otherwise specified, 0 to disable and 1 to enable the named behavior.

Examples

- Use the new 1.23 asynctimerchan=0 behavior:

  ```
  godebug asynctimerchan=0
  ```

- Use the default GODEBUGs from Go 1.21, but the old panicnil=1 behavior:

  ```
  godebug (
      default=go1.21
      panicnil=1
  )
  ```

Notes

GODEBUG settings only apply for builds of main packages and test binaries in the current module. They have no effect when a module is used as a dependency.

See “Go, Backwards Compatibility, and GODEBUG” for details on backwards compatibility.

require

Declares a module as a dependency of the current module, specifying the minimum version of the module required.

For more, see require directive in the Go Modules Reference.

Syntax

```
require module-path module-version
```

module-path
: The module's module path, usually a concatenation of the module source's repository domain and the module name. For module versions v2 and later, this value must end with the major version number, such as /v2.

module-version
: The module's version. This can be either a release version number, such as v1.2.3, or a Go-generated pseudo-version number, such as v0.0.0-20200921210052-fa0125251cc4.

Examples

- Requiring a released version v1.2.3:

  ```
  require example.com/othermodule v1.2.3
  ```

- Requiring a version not yet tagged in its repository by using a pseudo-version number generated by Go tools:

  ```
  require example.com/othermodule v0.0.0-20200921210052-fa0125251cc4
  ```

Notes

When you run a go command such as go get, Go inserts require directives for each module containing imported packages. When a module isn't yet tagged in its repository, Go assigns a pseudo-version number it generates when you run the command.

You can have Go require a module from a location other than its repository by using the replace directive.

For more about version numbers, see Module version numbering.

For more about managing dependencies, see the following:

- Adding a dependency
- Getting a specific dependency version
- Discovering available updates
- Upgrading or downgrading a dependency
- Synchronizing your code's dependencies

tool

Adds a package as a dependency of the current module, and makes it available to run with `go tool` when the current working directory is within this module.

Syntax

```
tool package-path
```

package-path
: The tool's package path, a concatenation of the module containing the tool and the (possibly empty) path to the package implementing the tool within the module.

Examples

- Declaring a tool implemented in the current module:

  ```
  module example.com/mymodule

  tool example.com/mymodule/cmd/mytool
  ```

- Declaring a tool implemented in a separate module:

  ```
  module example.com/mymodule

  tool example.com/atool/cmd/atool

  require example.com/atool v1.2.3
  ```

Notes

You can use `go tool` to run tools declared in your module by fully qualified package path or, if there is no ambiguity, by the last path segment. In the first example above you could run `go tool mytool` or `go tool example.com/mymodule/cmd/mytool`.

In workspace mode, you can use `go tool` to run a tool declared in any workspace module.

Tools are built using the same module graph as the module itself. A require directive is needed to select the version of the module that implements the tool. Any replace directives, or exclude directives also apply to the tool and its dependencies.

For more information see Tool dependencies.

replace

Replaces the content of a module at a specific version (or all versions) with another module version or with a local directory. Go tools will use the replacement path when resolving the dependency.

For more, see replace directive in the Go Modules Reference.

Syntax

```
replace module-path [module-version] => replacement-path [replacement-version]
```

module-path
: The module path of the module to replace.

module-version
: Optional. A specific version to replace. If this version number is omitted, all versions of the module are replaced with the content on the right side of the arrow.

replacement-path
: The path at which Go should look for the required module. This can be a module path or a path to a directory on the file system local to the replacement module. If this is a module path, you must specify a *replacement-version* value. If this is a local path, you may not use a *replacement-version* value.

replacement-version
: The version of the replacement module. The replacement version may only be specified if *replacement-path* is a module path (not a local directory).

Examples

- Replacing with a fork of the module repository

  In the following example, any version of example.com/othermodule is replaced with the specified fork of its code.

  ```
  require example.com/othermodule v1.2.3

  replace example.com/othermodule => example.com/myfork/othermodule v1.2.3-fix
  ```

  When you replace one module path with another, do not change import statements for packages in the module you’re replacing.

  For more on using a forked copy of module code, see Requiring external module code from your own repository fork.

- Replacing with a different version number

  The following example specifies that version v1.2.3 should be used instead of any other version of the module.

  ```
  require example.com/othermodule v1.2.2

  replace example.com/othermodule => example.com/othermodule v1.2.3
  ```

  The following example replaces module version v1.2.5 with version v1.2.3 of the same module.

  ```
  replace example.com/othermodule v1.2.5 => example.com/othermodule v1.2.3
  ```

- Replacing with local code

  The following example specifies that a local directory should be used as a replacement for all versions of the module.

  ```
  require example.com/othermodule v1.2.3

  replace example.com/othermodule => ../othermodule
  ```

  The following example specifies that a local directory should be used as a replacement for v1.2.5 only.

  ```
  require example.com/othermodule v1.2.5

  replace example.com/othermodule v1.2.5 => ../othermodule
  ```

  For more on using a local copy of module code, see Requiring module code in a local directory.

Notes

Use the replace directive to temporarily substitute a module path value with another value when you want Go to use the other path to find the module's source. This has the effect of redirecting Go's search for the module to the replacement's location. You needn't change package import paths to use the replacement path.

Use the exclude and replace directives to control build-time dependency resolution when building the current module. These directives are ignored in modules that depend on the current module.

The replace directive can be useful in situations such as the following:

- You're developing a new module whose code is not yet in the repository. You want to test with clients using a local version.
- You've identified an issue with a dependency, have cloned the dependency's repository, and you're testing a fix with the local repository.

Note that a replace directive alone does not add a module to the module graph. A require directive that refers to a replaced module version is also needed, either in the main module's go.mod file or a dependency's go.mod file. If you don't have a specific version to replace, you can use a fake version, as in the example below. Note that this will break modules that depend on your module, since replace directives are only applied in the main module.

```
require example.com/mod v0.0.0-replace

replace example.com/mod v0.0.0-replace => ./mod
```

For more on replacing a required module, including using Go tools to make the change, see:

- Requiring external module code from your own repository fork
- Requiring module code in a local directory

For more about version numbers, see Module version numbering.

exclude

Specifies a module or module version to exclude from the current module's dependency graph.

For more, see exclude directive in the Go Modules Reference.

Syntax

```
exclude module-path module-version
```

module-path
    The module path of the module to exclude.
module-version

The specific version to exclude.

Example

- Exclude example.com/theirmodule version v1.3.0

```
exclude example.com/theirmodule v1.3.0
```

Notes

Use the exclude directive to exclude a specific version of a module that is indirectly required but can't be loaded for some reason. For example, you might use it to exclude a version of a module that has an invalid checksum.

Use the exclude and replace directives to control build-time dependency resolution when building the current module (the main module you're building). These directives are ignored in modules that depend on the current module.

You can use the go mod edit command to exclude a module, as in the following example.

```
go mod edit -exclude=example.com/theirmodule@v1.3.0
```

For more about version numbers, see Module version numbering.

retract

Indicates that a version or range of versions of the module defined by go.mod should not be depended upon. A retract directive is useful when a version was published prematurely or a severe problem was discovered after the version was published.

For more, see retract directive in the Go Modules Reference.

Syntax

```
retract version // rationale
retract [version-low,version-high] // rationale
```

version
    A single version to retract.
version-low

Lower bound of a range of versions to retract.

version-high
: Upper bound of a range of versions to retract. Both *version-low* and *version-high* are included in the range.

rationale
: Optional comment explaining the retraction. May be shown in messages to the user.

Example

- Retracting a single version

```
retract v1.1.0 // Published accidentally.
```

- Retracting a range of versions

```
retract [v1.0.0,v1.0.5] // Build broken on some platforms.
```

Notes

Use the retract directive to indicate that a previous version of your module should not be used. Users will not automatically upgrade to a retracted version with `go get`, `go mod tidy`, or other commands. Users will not see a retracted version as an available update with `go list -m -u`.

Retracted versions should remain available so users that already depend on them are able to build their packages. Even if a retracted version is deleted from the source repository, it may remain available on mirrors such as proxy.golang.org. Users that depend on retracted versions may be notified when they run `go get` or `go list -m -u` on related modules.

The go command discovers retracted versions by reading retract directives in the go.mod file in the latest version of a module. The latest version is, in order of precedence:

1. Its highest release version, if any
2. Its highest pre-release version, if any
3. A pseudo-version for the tip of the repository's default branch.

When you add a retraction, you almost always need to tag a new, higher version so the command will see it in the latest version of the module.

You can publish a version whose sole purpose is to signal retractions. In this case, the new version may also retract itself.

For example, if you accidentally tag v1.0.0, you can tag v1.0.1 with the following directives:

```
retract v1.0.0 // Published accidentally.
retract v1.0.1 // Contains retraction only.
```

Unfortunately, once a version is published, it cannot be changed. If you later tag v1.0.0 at a different commit, the go command may detect a mismatched sum in go.sum or in the checksum database.

Retracted versions of a module do not normally appear in the output of go list -m -versions, but you can use the -retracted to show them. For more, see go list -m in the Go Modules Reference.