Go Modules Reference

Table of Contents

Introduction

Modules are how Go manages dependencies.

This document is a detailed reference manual for Go's module system. For an introduction to creating Go projects, see How to Write Go Code. For information on using modules, migrating projects to modules, and other topics, see the blog series starting with Using Go Modules.

Modules, packages, and versions

A *module* is a collection of packages that are released, versioned, and distributed together. Modules may be downloaded directly from version control repositories or from module proxy servers.

A module is identified by a module path, which is declared in a `go.mod` file, together with information about the module's dependencies. The *module root directory* is the directory that contains the `go.mod` file. The *main module* is the module containing the directory where the `go` command is invoked.

Each *package* within a module is a collection of source files in the same directory that are compiled together. A *package path* is the module path joined with the subdirectory containing the package (relative to the module root). For example, the module `"golang.org/x/net"` contains a package in the directory `"html"`. That package's path is `"golang.org/x/net/html"`.

Module paths

A *module path* is the canonical name for a module, declared with the `module` directive in the module's `go.mod` file. A module's path is the prefix for package paths within the module.

A module path should describe both what the module does and where to find it. Typically, a module path consists of a repository root path, a directory within the repository (usually empty), and a major version suffix (only for major version 2 or higher).

- The *repository root path* is the portion of the module path that corresponds to the root directory of the version control repository where the module is developed. Most modules are defined in their repository's root directory, so this is usually the entire path. For example, `golang.org/x/net` is the repository root path for the module of the same name. See Finding a repository for a module path for information on how the `go` command locates a repository using HTTP requests derived from a module path.
- If the module is not defined in the repository's root directory, the *module subdirectory* is the part of the module path that names the directory, not including the major version suffix. This also serves as a prefix for semantic version tags. For example, the module `golang.org/x/tools/gopls` is in the `gopls` subdirectory of the repository with root path `golang.org/x/tools`, so it has the module subdirectory `gopls`. See Mapping versions to commits and Module directories within a repository.
- If the module is released at major version 2 or higher, the module path must end with a major version suffix like `/v2`. This may or may not be part of the subdirectory name.

For example, the module with path `golang.org/x/repo/sub/v2` could be in the `/sub` or `/sub/v2` subdirectory of the repository `golang.org/x/repo`.

If a module might be depended on by other modules, these rules must be followed so that the `go` command can find and download the module. There are also several lexical restrictions on characters allowed in module paths.

A module that will never be fetched as a dependency of any other module may use any valid package path for its module path, but must take care not to collide with paths that may be used by the module's dependencies or the Go standard library. The Go standard library uses package paths that do not contain a dot in the first path element, and the `go` command does not attempt to resolve such paths from network servers. The paths `example` and `test` are reserved for users: they will not be used in the standard library and are suitable for use in self-contained modules, such as those defined in tutorials or example code or created and manipulated as part of a test.

Versions

A *version* identifies an immutable snapshot of a module, which may be either a release or a pre-release. Each version starts with the letter `v`, followed by a semantic version. See Semantic Versioning 2.0.0 for details on how versions are formatted, interpreted, and compared.

To summarize, a semantic version consists of three non-negative integers (the major, minor, and patch versions, from left to right) separated by dots. The patch version may be followed by an optional pre-release string starting with a hyphen. The pre-release string or patch version may be followed by a build metadata string starting with a plus. For example, `v0.0.0`, `v1.12.134`, `v8.0.5-pre`, and `v2.0.9+meta` are valid versions.

Each part of a version indicates whether the version is stable and whether it is compatible with previous versions.

- The major version must be incremented and the minor and patch versions must be set to zero after a backwards incompatible change is made to the module's public interface or documented functionality, for example, after a package is removed.
- The minor version must be incremented and the patch version set to zero after a backwards compatible change, for example, after a new function is added.
- The patch version must be incremented after a change that does not affect the module's public interface, such as a bug fix or optimization.
- The pre-release suffix indicates a version is a pre-release. Pre-release versions sort before the corresponding release versions. For example, `v1.2.3-pre` comes before `v1.2.3`.
- The build metadata suffix is ignored for the purpose of comparing versions. Tags with build metadata are ignored in version control repositories, but build metadata is preserved in versions specified in `go.mod` files. The suffix `+incompatible` denotes a version released before migrating to modules version major version 2 or later (see Compatibility with non-module repositories).

A version is considered unstable if its major version is 0 or it has a pre-release suffix. Unstable versions are not subject to compatibility requirements. For example, `v0.2.0` may not be compatible with `v0.1.0`, and `v1.5.0-beta` may not be compatible with `v1.5.0`.

Go may access modules in version control systems using tags, branches, or revisions that don't follow these conventions. However, within the main module, the `go` command will automatically convert revision names that don't follow this standard into canonical versions. The `go` command will also remove build metadata suffixes (except for `+incompatible`) as part of this process. This may result in a pseudo-version, a pre-release version that encodes a revision identifier (such as a Git commit hash) and a timestamp from a version control system. For example, the command `go get golang.org/x/net@daa7c041` will convert the commit hash `daa7c041` into the pseudo-version `v0.0.0-20191109021931-daa7c04131f5`. Canonical versions are required outside the main module, and the `go` command will report an error if a non-canonical version like `master` appears in a `go.mod` file.

Pseudo-versions

A *pseudo-version* is a specially formatted pre-release version that encodes information about a specific revision in a version control repository. For example, `v0.0.0-20191109021931-daa7c04131f5` is a pseudo-version.

Pseudo-versions may refer to revisions for which no semantic version tags are available. They may be used to test commits before creating version tags, for example, on a development branch.

Each pseudo-version has three parts:

- A base version prefix (`vX.0.0` or `vX.Y.Z-0`), which is either derived from a semantic version tag that precedes the revision or `vX.0.0` if there is no such tag.
- A timestamp (`yyyymmddhhmmss`), which is the UTC time the revision was created. In Git, this is the commit time, not the author time.
- A revision identifier (`abcdefabcdef`), which is a 12-character prefix of the commit hash, or in Subversion, a zero-padded revision number.

Each pseudo-version may be in one of three forms, depending on the base version. These forms ensure that a pseudo-version compares higher than its base version, but lower than the next tagged version.

- `vX.0.0-yyyymmddhhmmss-abcdefabcdef` is used when there is no known base version. As with all versions, the major version X must match the module's major version suffix.
- `vX.Y.Z-pre.0.yyyymmddhhmmss-abcdefabcdef` is used when the base version is a pre-release version like `vX.Y.Z-pre`.
- `vX.Y.(Z+1)-0.yyyymmddhhmmss-abcdefabcdef` is used when the base version is a release version like `vX.Y.Z`. For example, if the base version is `v1.2.3`, a pseudo-version might be `v1.2.4-0.20191109021931-daa7c04131f5`.

More than one pseudo-version may refer to the same commit by using different base versions. This happens naturally when a lower version is tagged after a pseudo-version is written.

These forms give pseudo-versions two useful properties:

- Pseudo-versions with known base versions sort higher than those versions but lower than other pre-release for later versions.
- Pseudo-versions with the same base version prefix sort chronologically.

The `go` command performs several checks to ensure that module authors have control over how pseudo-versions are compared with other versions and that pseudo-versions refer to revisions that are actually part of a module's commit history.

- If a base version is specified, there must be a corresponding semantic version tag that is an ancestor of the revision described by the pseudo-version. This prevents developers from bypassing minimal version selection using a pseudo-version that compares higher than all tagged versions like `v1.999.999-99999999999999-daa7c04131f5`.
- The timestamp must match the revision's timestamp. This prevents attackers from flooding module proxies with an unbounded number of otherwise identical pseudo-versions. This also prevents module consumers from changing the relative ordering of versions.
- The revision must be an ancestor of one of the module repository's branches or tags. This prevents attackers from referring to unapproved changes or pull requests.

Pseudo-versions never need to be typed by hand. Many commands accept a commit hash or a branch name and will translate it into a pseudo-version (or tagged version if available) automatically. For example:

```
go get example.com/mod@master
go list -m -json example.com/mod@abcd1234
```

Major version suffixes

Starting with major version 2, module paths must have a *major version suffix* like `/v2` that matches the major version. For example, if a module has the path `example.com/mod` at `v1.0.0`, it must have the path `example.com/mod/v2` at version `v2.0.0`.

Major version suffixes implement 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.

By definition, packages in a new major version of a module are not backwards compatible with the corresponding packages in the previous major version. Consequently, starting with v2, packages need new import paths. This is accomplished by adding a major version suffix to the module path. Since the module path is a prefix of the import path for each package

within the module, adding the major version suffix to the module path provides a distinct import path for each incompatible version.

Major version suffixes are not allowed at major versions `v0` or `v1`. There is no need to change the module path between `v0` and `v1` because `v0` versions are unstable and have no compatibility guarantee. Additionally, for most modules, `v1` is backwards compatible with the last `v0` version; a `v1` version acts as a commitment to compatibility, rather than an indication of incompatible changes compared with `v0`.

As a special case, modules paths starting with `gopkg.in/` must always have a major version suffix, even at `v0` and `v1`. The suffix must start with a dot rather than a slash (for example, `gopkg.in/yaml.v2`).

Major version suffixes let multiple major versions of a module coexist in the same build. This may be necessary due to a diamond dependency problem. Ordinarily, if a module is required at two different versions by transitive dependencies, the higher version will be used. However, if the two versions are incompatible, neither version will satisfy all clients. Since incompatible versions must have different major version numbers, they must also have different module paths due to major version suffixes. This resolves the conflict: modules with distinct suffixes are treated as separate modules, and their packages—even packages in same subdirectory relative to their module roots—are distinct.

Many Go projects released versions at `v2` or higher without using a major version suffix before migrating to modules (perhaps before modules were even introduced). These versions are annotated with a `+incompatible` build tag (for example, `v2.0.0+incompatible`). See Compatibility with non-module repositories for more information.

Resolving a package to a module

When the `go` command loads a package using a package path, it needs to determine which module provides the package.

The `go` command starts by searching the build list for modules with paths that are prefixes of the package path. For example, if the package `example.com/a/b` is imported, and the module `example.com/a` is in the build list, the `go` command will check whether `example.com/a` contains the package, in the directory `b`. At least one file with the `.go` extension must be present in a directory for it to be considered a package. Build constraints are not applied for this purpose. If exactly one module in the build list provides the package, that module is used. If no modules provide the package or if two or more modules provide the package, the `go` command reports an error. The `-mod=mod` flag instructs the `go` command to attempt to find new modules providing missing packages and to update `go.mod` and `go.sum`. The `go get` and `go mod tidy` commands do this automatically.

When the `go` command looks up a new module for a package path, it checks the `GOPROXY` environment variable, which is a comma-separated list of proxy URLs or the keywords `direct` or `off`. A proxy URL indicates the `go` command should contact a module proxy using the `GOPROXY` protocol. `direct` indicates that the `go` command should communicate

with a version control system. `off` indicates that no communication should be attempted. The `GOPRIVATE` and `GONOPROXY` environment variables can also be used to control this behavior.

For each entry in the `GOPROXY` list, the `go` command requests the latest version of each module path that might provide the package (that is, each prefix of the package path). For each successfully requested module path, the `go` command will download the module at the latest version and check whether the module contains the requested package. If one or more modules contain the requested package, the module with the longest path is used. If one or more modules are found but none contain the requested package, an error is reported. If no modules are found, the `go` command tries the next entry in the `GOPROXY` list. If no entries are left, an error is reported.

For example, suppose the `go` command is looking for a module that provides the package `golang.org/x/net/html`, and `GOPROXY` is set to `https://corp.example.com,https://proxy.golang.org`. The `go` command may make the following requests:

- To `https://corp.example.com/` (in parallel):
    - Request for latest version of `golang.org/x/net/html`
    - Request for latest version of `golang.org/x/net`
    - Request for latest version of `golang.org/x`
    - Request for latest version of `golang.org`
- To `https://proxy.golang.org/`, if all requests to `https://corp.example.com/` have failed with 404 or 410:
    - Request for latest version of `golang.org/x/net/html`
    - Request for latest version of `golang.org/x/net`
    - Request for latest version of `golang.org/x`
    - Request for latest version of `golang.org`

After a suitable module has been found, the `go` command will add a new requirement with the new module's path and version to the main module's `go.mod` file. This ensures that when the same package is loaded in the future, the same module will be used at the same version. If the resolved package is not imported by a package in the main module, the new requirement will have an `// indirect` comment.

`go.mod` files

A module is defined by a UTF-8 encoded text file named `go.mod` in its root directory. The `go.mod` file is line-oriented. Each line holds a single directive, made up of a keyword followed by arguments. For example:

```
module example.com/my/thing

go 1.12
```

```
require example.com/other/thing v1.0.2
require example.com/new/thing/v2 v2.3.4
exclude example.com/old/thing v1.2.3
replace example.com/bad/thing v1.4.5 => example.com/good/thing v1.4.5
retract [v1.9.0, v1.9.5]
```

The leading keyword can be factored out of adjacent lines to create a block, like in Go imports.

```
require (
    example.com/new/thing/v2 v2.3.4
    example.com/old/thing v1.2.3
)
```

The `go.mod` file is designed to be human readable and machine writable. The `go` command provides several subcommands that change `go.mod` files. For example, `go get` can upgrade or downgrade specific dependencies. Commands that load the module graph will automatically update `go.mod` when needed. `go mod edit` can perform low-level edits. The `golang.org/x/mod/modfile` package can be used by Go programs to make the same changes programmatically.

A `go.mod` file is required for the main module, and for any replacement module specified with a local file path. However, a module that lacks an explicit `go.mod` file may still be required as a dependency, or used as a replacement specified with a module path and version; see Compatibility with non-module repositories.

Lexical elements

When a `go.mod` file is parsed, its content is broken into a sequence of tokens. There are several kinds of tokens: whitespace, comments, punctuation, keywords, identifiers, and strings.

White space consists of spaces (U+0020), tabs (U+0009), carriage returns (U+000D), and newlines (U+000A). White space characters other than newlines have no effect except to separate tokens that would otherwise be combined. Newlines are significant tokens.

Comments start with `//` and run to the end of a line. `/* */` comments are not allowed.

Punctuation tokens include `(`, `)`, and `=>`.

Keywords distinguish different kinds of directives in a `go.mod` file. Allowed keywords are `module`, `go`, `require`, `replace`, `exclude`, and `retract`.

Identifiers are sequences of non-whitespace characters, such as module paths or semantic versions.

Strings are quoted sequences of characters. There are two kinds of strings: interpreted strings beginning and ending with quotation marks (", U+0022) and raw strings beginning and ending with grave accents (`, U+0060). Interpreted strings may contain escape

sequences consisting of a backslash (\, U+005C) followed by another character. An escaped quotation mark (\") does not terminate an interpreted string. The unquoted value of an interpreted string is the sequence of characters between quotation marks with each escape sequence replaced by the character following the backslash (for example, \" is replaced by ", \n is replaced by n). In contrast, the unquoted value of a raw string is simply the sequence of characters between grave accents; backslashes have no special meaning within raw strings.

Identifiers and strings are interchangeable in the `go.mod` grammar.

Module paths and versions

Most identifiers and strings in a `go.mod` file are either module paths or versions.

A module path must satisfy the following requirements:

- The path must consist of one or more path elements separated by slashes (/, U+002F). It must not begin or end with a slash.
- Each path element is a non-empty string made of up ASCII letters, ASCII digits, and limited ASCII punctuation (–, ., _, and ~).
- A path element may not begin or end with a dot (., U+002E).
- The element prefix up to the first dot must not be a reserved file name on Windows, regardless of case (`CON`, `com1`, `NuL`, and so on).
- The element prefix up to the first dot must not end with a tilde followed by one or more digits (like `EXAMPL~1.COM`).

If the module path appears in a `require` directive and is not replaced, or if the module paths appears on the right side of a `replace` directive, the `go` command may need to download modules with that path, and some additional requirements must be satisfied.

- The leading path element (up to the first slash, if any), by convention a domain name, must contain only lower-case ASCII letters, ASCII digits, dots (., U+002E), and dashes (–, U+002D); it must contain at least one dot and cannot start with a dash.
- For a final path element of the form /vN where N looks numeric (ASCII digits and dots), N must not begin with a leading zero, must not be /v1, and must not contain any dots.
    - For paths beginning with `gopkg.in/`, this requirement is replaced by a requirement that the path follow the gopkg.in service's conventions.

Versions in `go.mod` files may be canonical or non-canonical.

A canonical version starts with the letter v, followed by a semantic version following the Semantic Versioning 2.0.0 specification. See Versions for more information.

Most other identifiers and strings may be used as non-canonical versions, though there are some restrictions to avoid problems with file systems, repositories, and module proxies. Non-canonical versions are only allowed in the main module's `go.mod` file. The `go` command

will attempt to replace each non-canonical version with an equivalent canonical version when it automatically updates the `go.mod` file.

In places where a module path is associated with a version (as in `require`, `replace`, and `exclude` directives), the final path element must be consistent with the version. See Major version suffixes.

Grammar

`go.mod` syntax is specified below using Extended Backus-Naur Form (EBNF). See the Notation section in the Go Language Specification for details on EBNF syntax.

```
GoMod = { Directive } .
Directive = ModuleDirective |
            GoDirective |
            RequireDirective |
            ExcludeDirective |
            ReplaceDirective |
            RetractDirective .
```

Newlines, identifiers, and strings are denoted with `newline`, `ident`, and `string`, respectively.

Module paths and versions are denoted with `ModulePath` and `Version`.

```
ModulePath = ident | string . /* see restrictions above */
Version = ident | string .    /* see restrictions above */
```

`module` directive

A `module` directive defines the main module's path. A `go.mod` file must contain exactly one `module` directive.

```
ModuleDirective = "module" ( ModulePath | "(" newline ModulePath newline ")" ) ne
```

Example:

```
module golang.org/x/net
```

Deprecation

A module can be marked as deprecated in a block of comments containing the string `Deprecated:` (case-sensitive) at the beginning of a paragraph. The deprecation message starts after the colon and runs to the end of the paragraph. The comments may appear immediately before the `module` directive or afterward on the same line.

Example:

```
// Deprecated: use example.com/mod/v2 instead.
module example.com/mod
```

Since Go 1.17, `go list -m -u` checks for information on all deprecated modules in the build list. `go get` checks for deprecated modules needed to build packages named on the command line.

When the `go` command retrieves deprecation information for a module, it loads the `go.mod` file from the version matching the `@latest` version query without considering retractions or exclusions. The `go` command loads the list of retracted versions from the same `go.mod` file.

To deprecate a module, an author may add a `// Deprecated:` comment and tag a new release. The author may change or remove the deprecation message in a higher release.

A deprecation applies to all minor versions of a module. Major versions higher than `v2` are considered separate modules for this purpose, since their major version suffixes give them distinct module paths.

Deprecation messages are intended to inform users that the module is no longer supported and to provide migration instructions, for example, to the latest major version. Individual minor and patch versions cannot be deprecated; `retract` may be more appropriate for that.

go directive

A `go` directive indicates that a module was written assuming the semantics of a given version of Go. The version must be a valid Go version, such as `1.9`, `1.14`, or `1.21rc1`.

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.

If the `go` directive is missing, `go 1.16` is assumed.

```
GoDirective = "go" GoVersion newline .
GoVersion = string | ident .  /* valid release version; see above */
```

Example:

```
go 1.14
```

`toolchain` directive

A `toolchain` directive declares a suggested Go toolchain to use with a module. The suggested Go toolchain's version cannot be less than the required Go version declared in the `go` directive. The `toolchain` directive only has an effect when the module is the main module and the default toolchain's version is less than the suggested toolchain's version.

For reproducibility, the `go` command writes its own toolchain name in a `toolchain` line any time it is updating the `go` version in the `go.mod` file (usually during `go get`).

For details, see "Go toolchains".

```
ToolchainDirective = "toolchain" ToolchainName newline .
ToolchainName = string | ident .  /* valid toolchain name; see "Go toolchains" */
```

Example:

```
toolchain go1.21.0
```

godebug directive

A `godebug` directive declares a single GODEBUG setting to apply when this module is the main module. There can be more than one such line, and they can be factored. It is an error for the main module to name a GODEBUG key that does not exist. The effect of `godebug key=value` is as if every main package being compiled contained a source file that listed `//go:debug key=value`.

```
GodebugDirective = "godebug" ( GodebugSpec | "(" newline { GodebugSpec } ")" newl
GodebugSpec = GodebugKey "=" GodebugValue newline.
GodebugKey = GodebugChar { GodebugChar }.
GodebugValue = GodebugChar { GodebugChar }.
GodebugChar = any non-space character except , " ` ' (comma and quotes).
```

Example:

```
godebug default=go1.21
godebug (
    panicnil=1
    asynctimerchan=0
)
```

`require` directive

A `require` directive declares a minimum required version of a given module dependency. For each required module version, the `go` command loads the `go.mod` file for that version and incorporates the requirements from that file. Once all requirements have been loaded, the `go` command resolves them using minimal version selection (MVS) to produce the build list.

The `go` command automatically adds `// indirect` comments for some requirements. An `// indirect` comment indicates that no package from the required module is directly

imported by any package in the main module.

If the go directive specifies `go 1.16` or lower, the `go` command adds an indirect requirement when the selected version of a module is higher than what is already implied (transitively) by the main module's other dependencies. That may occur because of an explicit upgrade (`go get -u ./...`), removal of some other dependency that previously imposed the requirement (`go mod tidy`), or a dependency that imports a package without a corresponding requirement in its own `go.mod` file (such as a dependency that lacks a `go.mod` file altogether).

At `go 1.17` and above, the `go` command adds an indirect requirement for each module that provides any package imported (even indirectly) by a package or test in the main module or passed as an argument to `go get`. These more comprehensive requirements enable module graph pruning and lazy module loading.

```
RequireDirective = "require" ( RequireSpec | "(" newline { RequireSpec } ")" newl
RequireSpec = ModulePath Version newline .
```

Example:

```
require golang.org/x/net v1.2.3

require (
    golang.org/x/crypto v1.4.5 // indirect
    golang.org/x/text v1.6.7
)
```

`exclude` directive

An `exclude` directive prevents a module version from being loaded by the `go` command.

Since Go 1.16, if a version referenced by a `require` directive in any `go.mod` file is excluded by an `exclude` directive in the main module's `go.mod` file, the requirement is ignored. This may cause commands like `go get` and `go mod tidy` to add new requirements on higher versions to `go.mod`, with an `// indirect` comment if appropriate.

Before Go 1.16, if an excluded version was referenced by a `require` directive, the `go` command listed available versions for the module (as shown with `go list -m -versions`) and loaded the next higher non-excluded version instead. This could result in non-deterministic version selection, since the next higher version could change over time. Both release and pre-release versions were considered for this purpose, but pseudo-versions were not. If there were no higher versions, the `go` command reported an error.

`exclude` directives only apply in the main module's `go.mod` file and are ignored in other modules. See Minimal version selection for details.

```
ExcludeDirective = "exclude" ( ExcludeSpec | "(" newline { ExcludeSpec } ")" newl
ExcludeSpec = ModulePath Version newline .
```

Example:

```
exclude golang.org/x/net v1.2.3

exclude (
    golang.org/x/crypto v1.4.5
    golang.org/x/text v1.6.7
)
```

`replace` directive

A `replace` directive replaces the contents of a specific version of a module, or all versions of a module, with contents found elsewhere. The replacement may be specified with either another module path and version, or a platform-specific file path.

If a version is present on the left side of the arrow (=>), only that specific version of the module is replaced; other versions will be accessed normally. If the left version is omitted, all versions of the module are replaced.

If the path on the right side of the arrow is an absolute or relative path (beginning with `./` or `../`), it is interpreted as the local file path to the replacement module root directory, which must contain a `go.mod` file. The replacement version must be omitted in this case.

If the path on the right side is not a local path, it must be a valid module path. In this case, a version is required. The same module version must not also appear in the build list.

Regardless of whether a replacement is specified with a local path or module path, if the replacement module has a `go.mod` file, its `module` directive must match the module path it replaces.

`replace` directives only apply in the main module's `go.mod` file and are ignored in other modules. See Minimal version selection for details.

If there are multiple main modules, all main modules' `go.mod` files apply. Conflicting `replace` directives across main modules are disallowed, and must be removed or overridden in a replace in the `go.work` file.

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. A `replace` directive has no effect if the module version on the left side is not required.

```
ReplaceDirective = "replace" ( ReplaceSpec | "(" newline { ReplaceSpec } ")" newl
ReplaceSpec = ModulePath [ Version ] "=>" FilePath newline
            | ModulePath [ Version ] "=>" ModulePath Version newline .
FilePath = /* platform-specific relative or absolute file path */
```

Example:

```
replace golang.org/x/net v1.2.3 => example.com/fork/net v1.4.5

replace (
    golang.org/x/net v1.2.3 => example.com/fork/net v1.4.5
    golang.org/x/net => example.com/fork/net v1.4.5
    golang.org/x/net v1.2.3 => ./fork/net
    golang.org/x/net => ./fork/net
)
```

`retract` directive

A `retract` directive 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. Retracted versions should remain available in version control repositories and on module proxies to ensure that builds that depend on them are not broken. The word *retract* is borrowed from academic literature: a retracted research paper is still available, but it has problems and should not be the basis of future work.

When a module version is retracted, users will not upgrade to it automatically using `go get`, `go mod tidy`, or other commands. Builds that depend on retracted versions should continue to work, but users will be notified of retractions when they check for updates with `go list -m -u` or update a related module with `go get`.

To retract a version, a module author should add a `retract` directive to `go.mod`, then publish a new version containing that directive. The new version must be higher than other release or pre-release versions; that is, the `@latest` version query should resolve to the new version before retractions are considered. The `go` command loads and applies retractions from the version shown by `go list -m -retracted $modpath@latest` (where `$modpath` is the module path).

Retracted versions are hidden from the version list printed by `go list -m -versions` unless the `-retracted` flag is used. Retracted versions are excluded when resolving version queries like `@>=v1.2.3` or `@latest`.

A version containing retractions may retract itself. If the highest release or pre-release version of a module retracts itself, the `@latest` query resolves to a lower version after retracted versions are excluded.

As an example, consider a case where the author of module `example.com/m` publishes version `v1.0.0` accidentally. To prevent users from upgrading to `v1.0.0`, the author can add two `retract` directives to `go.mod`, then tag `v1.0.1` with the retractions.

```
retract (
    v1.0.0 // Published accidentally.
    v1.0.1 // Contains retractions only.
)
```

When a user runs `go get example.com/m@latest`, the `go` command reads retractions from `v1.0.1`, which is now the highest version. Both `v1.0.0` and `v1.0.1` are retracted, so the `go` command will upgrade (or downgrade!) to the next highest version, perhaps `v0.9.5`.

`retract` directives may be written with either a single version (like `v1.0.0`) or with a closed interval of versions with an upper and lower bound, delimited by `[` and `]` (like `[v1.1.0, v1.2.0]`). A single version is equivalent to an interval where the upper and lower bound are the same. Like other directives, multiple `retract` directives may be grouped together in a block delimited by `(` at the end of a line and `)` on its own line.

Each `retract` directive should have a comment explaining the rationale for the retraction, though this is not mandatory. The `go` command may display rationale comments in warnings about retracted versions and in `go list` output. A rationale comment may be written immediately above a `retract` directive (without a blank line in between) or afterward on the same line. If a comment appears above a block, it applies to all `retract` directives within the block that don't have their own comments. A rationale comment may span multiple lines.

```
RetractDirective = "retract" ( RetractSpec | "(" newline { RetractSpec } ")" newl
RetractSpec = ( Version | "[" Version "," Version "]" ) newline .
```

Examples:

- Retracting all versions between `v1.0.0` and `v1.9.9`:

```
retract v1.0.0
retract [v1.0.0, v1.9.9]
retract (
    v1.0.0
    [v1.0.0, v1.9.9]
)
```

- Returning to unversioned after prematurely released a version `v1.0.0`:

```
retract [v0.0.0, v1.0.1] // assuming v1.0.1 contains this retraction.
```

- Wiping out a module including all pseudo-versions and tagged versions:

```
retract [v0.0.0-0, v0.15.2]  // assuming v0.15.2 contains this retraction.
```

The `retract` directive was added in Go 1.16. Go 1.15 and lower will report an error if a `retract` directive is written in the main module's `go.mod` file and will ignore `retract` directives in `go.mod` files of dependencies.

Automatic updates

Most commands report an error if `go.mod` is missing information or doesn't accurately reflect reality. The `go get` and `go mod tidy` commands may be used to fix most of these problems. Additionally, the `–mod=mod` flag may be used with most module-aware commands

(`go build`, `go test`, and so on) to instruct the `go` command to fix problems in `go.mod` and `go.sum` automatically.

For example, consider this `go.mod` file:

```
module example.com/M

go 1.16

require (
    example.com/A v1
    example.com/B v1.0.0
    example.com/C v1.0.0
    example.com/D v1.2.3
    example.com/E dev
)

exclude example.com/D v1.2.3
```

The update triggered with `-mod=mod` rewrites non-canonical version identifiers to canonical semver form, so `example.com/A`'s `v1` becomes `v1.0.0`, and `example.com/E`'s `dev` becomes the pseudo-version for the latest commit on the `dev` branch, perhaps `v0.0.0-20180523231146-b3f5c0f6e5f1`.

The update modifies requirements to respect exclusions, so the requirement on the excluded `example.com/D v1.2.3` is updated to use the next available version of `example.com/D`, perhaps `v1.2.4` or `v1.3.0`.

The update removes redundant or misleading requirements. For example, if `example.com/A v1.0.0` itself requires `example.com/B v1.2.0` and `example.com/C v1.0.0`, then `go.mod`'s requirement of `example.com/B v1.0.0` is misleading (superseded by `example.com/A`'s need for `v1.2.0`), and its requirement of `example.com/C v1.0.0` is redundant (implied by `example.com/A`'s need for the same version), so both will be removed. If the main module contains packages that directly import packages from `example.com/B` or `example.com/C`, then the requirements will be kept but updated to the actual versions being used.

Finally, the update reformats the `go.mod` in a canonical formatting, so that future mechanical changes will result in minimal diffs. The `go` command will not update `go.mod` if only formatting changes are needed.

Because the module graph defines the meaning of import statements, any commands that load packages also use `go.mod` and can therefore update it, including `go build`, `go get`, `go install`, `go list`, `go test`, `go mod tidy`.

In Go 1.15 and lower, the `-mod=mod` flag was enabled by default, so updates were performed automatically. Since Go 1.16, the `go` command acts as if `-mod=readonly` were set instead: if any changes to `go.mod` are needed, the `go` command reports an error and suggests a fix.

Minimal version selection (MVS)

Go uses an algorithm called *Minimal version selection (MVS)* to select a set of module versions to use when building packages. MVS is described in detail in Minimal Version Selection by Russ Cox.

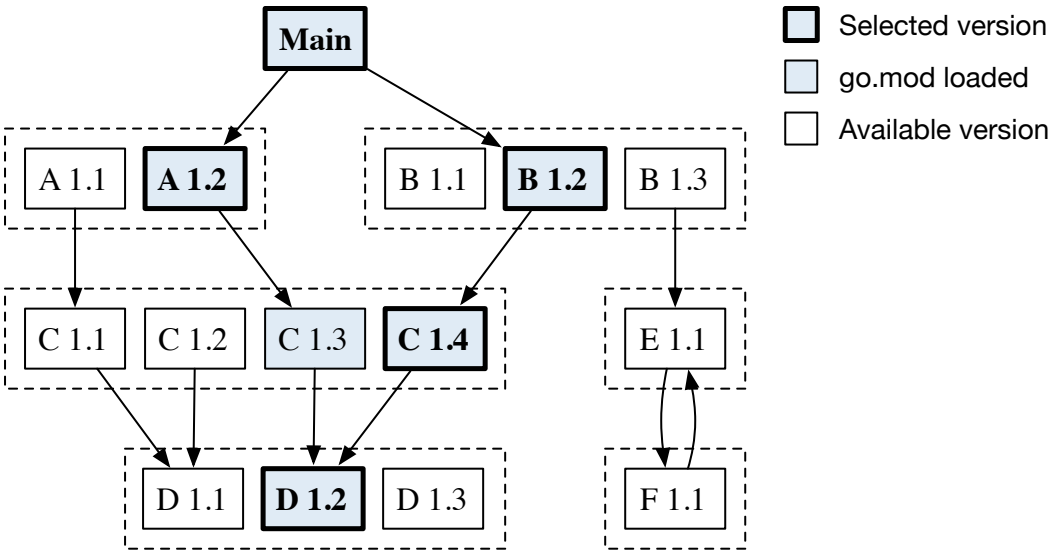
Conceptually, MVS operates on a directed graph of modules, specified with `go.mod` files. Each vertex in the graph represents a module version. Each edge represents a minimum required version of a dependency, specified using a `require` directive. The graph may be modified by `exclude` and `replace` directives in the `go.mod` file(s) of the main module(s) and by `replace` directives in the `go.work` file.

MVS produces the build list as output, the list of module versions used for a build.

MVS starts at the main modules (special vertices in the graph that have no version) and traverses the graph, tracking the highest required version of each module. At the end of the traversal, the highest required versions comprise the build list: they are the minimum versions that satisfy all requirements.

The build list may be inspected with the command `go list -m all`. Unlike other dependency management systems, the build list is not saved in a "lock" file. MVS is deterministic, and the build list doesn't change when new versions of dependencies are released, so MVS is used to compute it at the beginning of every module-aware command.

Consider the example in the diagram below. The main module requires module A at version 1.2 or higher and module B at version 1.2 or higher. A 1.2 and B 1.2 require C 1.3 and C 1.4, respectively. C 1.3 and C 1.4 both require D 1.2.



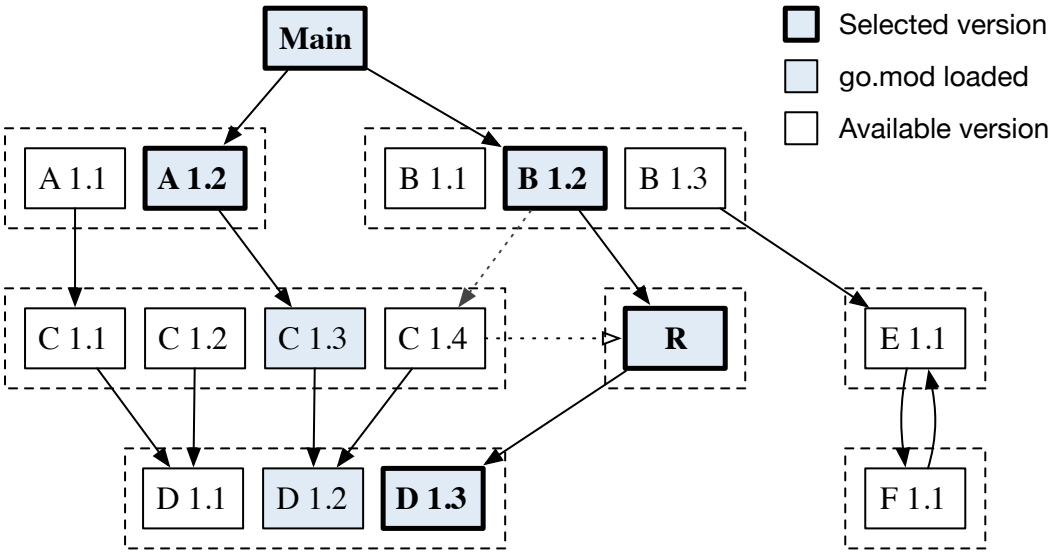
MVS visits and loads the `go.mod` file for each of the module versions highlighted in blue. At the end of the graph traversal, MVS returns a build list containing the bolded versions: A 1.2, B 1.2, C 1.4, and D 1.2. Note that higher versions of B and D are available but MVS does not select them, since nothing requires them.

Replacement

The content of a module (including its `go.mod` file) may be replaced using a `replace` directive in a main module's `go.mod` file or a workspace's `go.work` file. A `replace` directive may apply to a specific version of a module or to all versions of a module.

Replacements change the module graph, since a replacement module may have different dependencies than replaced versions.

Consider the example below, where C 1.4 has been replaced with R. R depends on D 1.3 instead of D 1.2, so MVS returns a build list containing A 1.2, B 1.2, C 1.4 (replaced with R), and D 1.3.

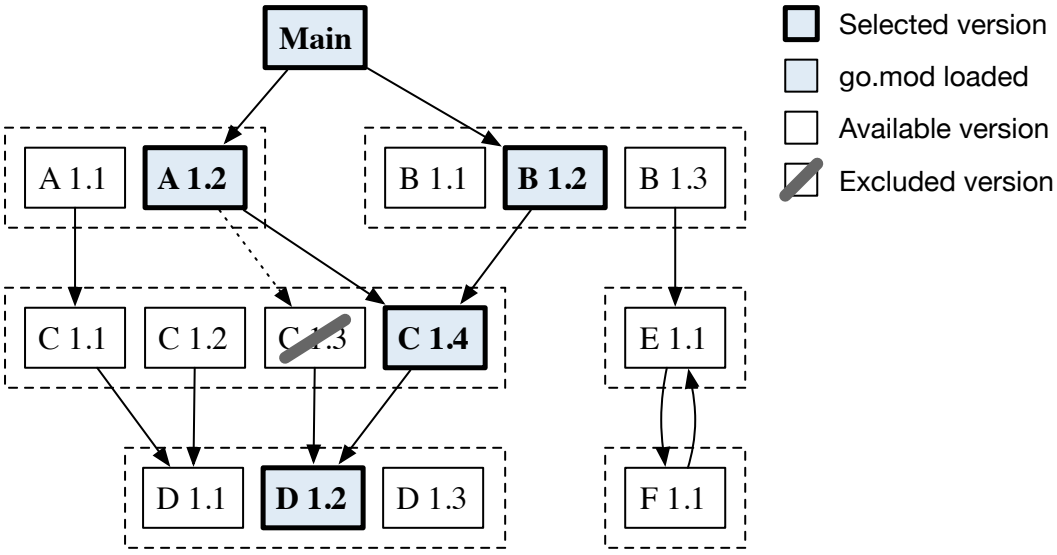


Exclusion

A module may also be excluded at specific versions using an `exclude` directive in the main module's `go.mod` file.

Exclusions also change the module graph. When a version is excluded, it is removed from the module graph, and requirements on it are redirected to the next higher version.

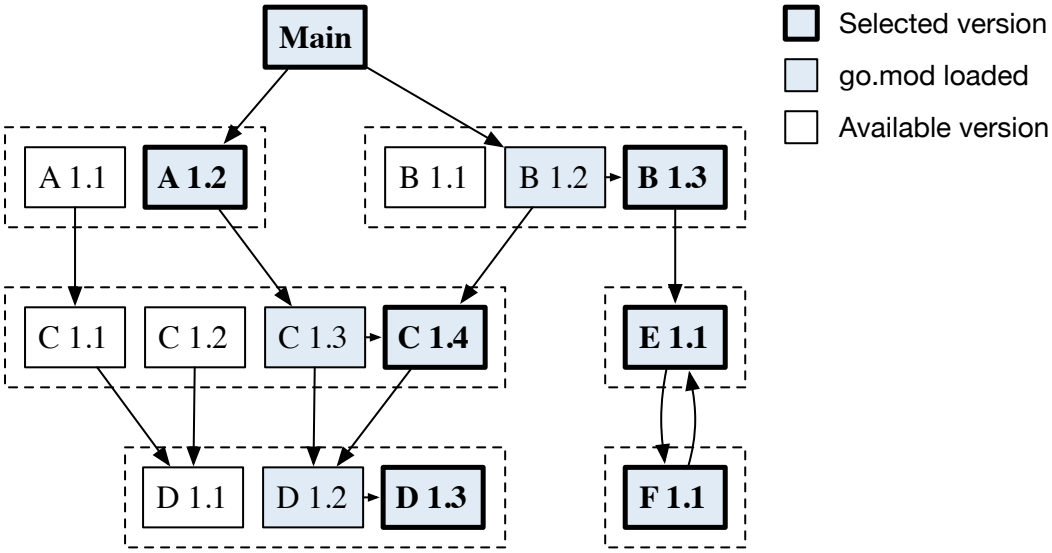
Consider the example below. C 1.3 has been excluded. MVS will act as if A 1.2 required C 1.4 (the next higher version) instead of C 1.3.



Upgrades

The `go get` command may be used to upgrade a set of modules. To perform an upgrade, the `go` command changes the module graph before running MVS by adding edges from visited versions to upgraded versions.

Consider the example below. Module B may be upgraded from 1.2 to 1.3, C may be upgraded from 1.3 to 1.4, and D may be upgraded from 1.2 to 1.3.



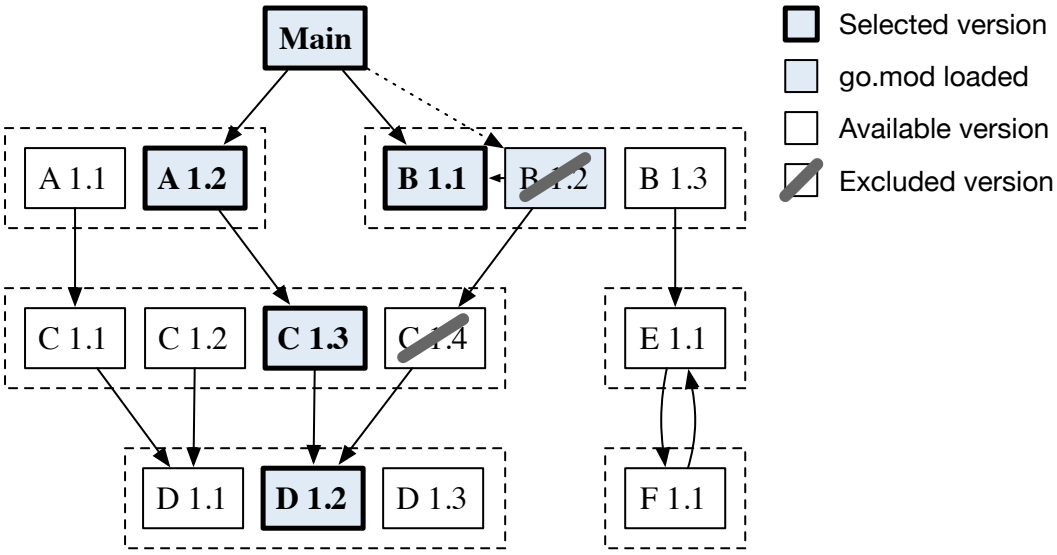
Upgrades (and downgrades) may add or remove indirect dependencies. In this case, E 1.1 and F 1.1 appear in the build list after the upgrade, since E 1.1 is required by B 1.3.

To preserve upgrades, the `go` command updates the requirements in `go.mod`. It will change the requirement on B to version 1.3. It will also add requirements on C 1.4 and D 1.3 with `// indirect` comments, since those versions would not be selected otherwise.

Downgrade

The `go get` command may also be used to downgrade a set of modules. To perform a downgrade, the `go` command changes the module graph by removing versions above the downgraded versions. It also removes versions of other modules that depend on removed versions, since they may not be compatible with the downgraded versions of their dependencies. If the main module requires a module version removed by downgrading, the requirement is changed to a previous version that has not been removed. If no previous version is available, the requirement is dropped.

Consider the example below. Suppose that a problem was found with C 1.4, so we downgrade to C 1.3. C 1.4 is removed from the module graph. B 1.2 is also removed, since it requires C 1.4 or higher. The main module's requirement on B is changed to 1.1.



`go get` can also remove dependencies entirely, using an `@none` suffix after an argument. This works similarly to a downgrade. All versions of the named module are removed from the module graph.

Module graph pruning

If the main module is at `go 1.17` or higher, the module graph used for minimal version selection includes only the *immediate* requirements for each module dependency that specifies `go 1.17` or higher in its own `go.mod` file, unless that version of the module is also (transitively) required by some *other* dependency at `go 1.16` or below. (The *transitive* dependencies of `go 1.17` dependencies are *pruned out* of the module graph.)

Since a `go 1.17` `go.mod` file includes a require directive for every dependency needed to build any package or test in that module, the pruned module graph includes all of the dependencies needed to `go build` or `go test` the packages in any dependency explicitly required by the main module. A module that is *not* needed to build any package or test in a given module cannot affect the run-time behavior of its packages, so the dependencies that are pruned out of the module graph would only cause interference between otherwise-unrelated modules.

Modules whose requirements have been pruned out still appear in the module graph and are still reported by `go list -m all`: their selected versions are known and well-defined, and packages can be loaded from those modules (for example, as transitive dependencies of tests loaded from other modules). However, since the `go` command cannot easily identify which dependencies of these modules are satisfied, the arguments to `go build` and `go test` cannot include packages from modules whose requirements have been pruned out. `go get` promotes the module containing each named package to an explicit dependency, allowing `go build` or `go test` to be invoked on that package.

Because Go 1.16 and earlier did not support module graph pruning, the full transitive closure of dependencies — including transitive `go 1.17` dependencies — is still included for each module that specifies `go 1.16` or lower. (At `go 1.16` and below, the `go.mod` file includes

only direct dependencies, so a much larger graph must be loaded to ensure that all indirect dependencies are included.)

The `go.sum` file recorded by `go mod tidy` for a module by default includes checksums needed by the Go version *one below* the version specified in its `go` directive. So a `go 1.17` module includes checksums needed for the full module graph loaded by Go 1.16, but a `go 1.18` module will include only the checksums needed for the pruned module graph loaded by Go 1.17. The `–compat` flag can be used to override the default version (for example, to prune the `go.sum` file more aggressively in a `go 1.17` module).

See the design document for more detail.

Lazy module loading

The more comprehensive requirements added for module graph pruning also enable another optimization when working within a module. If the main module is at `go 1.17` or higher, the `go` command avoids loading the complete module graph until (and unless) it is needed. Instead, it loads only the main module's `go.mod` file, then attempts to load the packages to be built using only those requirements. If a package to be imported (for example, a dependency of a test for a package outside the main module) is not found among those requirements, then the rest of the module graph is loaded on demand.

If all imported packages can be found without loading the module graph, the `go` command then loads the `go.mod` files for *only* the modules containing those packages, and their requirements are checked against the requirements of the main module to ensure that they are locally consistent. (Inconsistencies can arise due to version-control merges, hand-edits, and changes in modules that have been replaced using local filesystem paths.)

Workspaces

A *workspace* is a collection of modules on disk that are used as the main modules when running minimal version selection (MVS).

A workspace can be declared in a `go.work` file that specifies relative paths to the module directories of each of the modules in the workspace. When no `go.work` file exists, the workspace consists of the single module containing the current directory.

Most `go` subcommands that work with modules operate on the set of modules determined by the current workspace. `go mod init`, `go mod why`, `go mod edit`, `go mod tidy`, `go mod vendor`, and `go get` always operate on a single main module.

A command determines whether it is in a workspace context by first examining the `GOWORK` environment variable. If `GOWORK` is set to `off`, the command will be in a single-module context. If it is empty or not provided, the command will search the current working directory, and then successive parent directories, for a file `go.work`. If a file is found, the command will operate in the workspace it defines; otherwise, the workspace will include only the module containing the working directory. If `GOWORK` names a path to an existing file that ends in .work, workspace mode will be enabled. Any other value is an error. You can use the

`go env GOWORK` command to determine which `go.work` file the `go` command is using. `go env GOWORK` will be empty if the `go` command is not in workspace mode.

`go.work` files

A workspace is defined by a UTF-8 encoded text file named `go.work`. The `go.work` file is line oriented. Each line holds a single directive, made up of a keyword followed by arguments. For example:

```
go 1.18

use ./my/first/thing
use ./my/second/thing

replace example.com/bad/thing v1.4.5 => example.com/good/thing v1.4.5
```

As in `go.mod` files, a leading keyword can be factored out of adjacent lines to create a block.

```
use (
    ./my/first/thing
    ./my/second/thing
)
```

The `go` command provides several subcommands for manipulating `go.work` files. `go work init` creates new `go.work` files. `go work use` adds module directories to the `go.work` file. `go work edit` performs low-level edits. The `golang.org/x/mod/modfile` package can be used by Go programs to make the same changes programmatically.

The go command will maintain a `go.work.sum` file that keeps track of hashes used by the workspace that are not in collective workspace modules' go.sum files.

It is generally inadvisable to commit go.work files into version control systems, for two reasons:

- A checked-in `go.work` file might override a developer's own `go.work` file from a parent directory, causing confusion when their `use` directives don't apply.
- A checked-in `go.work` file may cause a continuous integration (CI) system to select and thus test the wrong versions of a module's dependencies. CI systems should generally not be allowed to use the `go.work` file so that they can test the behavior of the module as it would be used when required by other modules, where a `go.work` file within the module has no effect.

That said, there are some cases where committing a `go.work` file makes sense. For example, when the modules in a repository are developed exclusively with each other but not together with external modules, there may not be a reason the developer would want to use a different combination of modules in a workspace. In that case, the module author should ensure the individual modules are tested and released properly.

Lexical elements

Lexical elements in `go.work` files are defined in exactly the same way as for `go.mod files`.

Grammar

`go.work` syntax is specified below using Extended Backus-Naur Form (EBNF). See the Notation section in the Go Language Specification for details on EBNF syntax.

```
GoWork = { Directive } .
Directive = GoDirective |
            ToolchainDirective |
            UseDirective |
            ReplaceDirective .
```

Newlines, identifiers, and strings are denoted with `newline`, `ident`, and `string`, respectively.

Module paths and versions are denoted with `ModulePath` and `Version`. Module paths and versions are specified in exactly the same way as for `go.mod files`.

```
ModulePath = ident | string . /* see restrictions above */
Version = ident | string .    /* see restrictions above */
```

go directive

A `go` directive is required in a valid `go.work` file. The version must be a valid Go release version: a positive integer followed by a dot and a non-negative integer (for example, `1.18`, `1.19`).

The `go` directive indicates the go toolchain version with which the `go.work` file is intended to work. If changes are made to the `go.work` file format, future versions of the toolchain will interpret the file according to its indicated version.

A `go.work` file may contain at most one `go` directive.

```
GoDirective = "go" GoVersion newline .
GoVersion = string | ident .  /* valid release version; see above */
```

Example:

```
go 1.18
```

`toolchain` directive

A `toolchain` directive declares a suggested Go toolchain to use in a workspace. It only has an effect when the default toolchain is older than the suggested toolchain.

For details, see “Go toolchains”.

```
ToolchainDirective = "toolchain" ToolchainName newline .
ToolchainName = string | ident .  /* valid toolchain name; see “Go toolchains” */
```

Example:

```
toolchain go1.21.0
```

godebug directive

A `godebug` directive declares a single GODEBUG setting to apply when working in this workspace. The syntax and effect is the same as the `go.mod` file's `godebug` directive. When a workspace is in use, `godebug` directives in `go.mod` files are ignored.

`use` directive

A `use` adds a module on disk to the set of main modules in a workspace. Its argument is a relative path to the directory containing the module's `go.mod` file. A `use` directive does not add modules contained in subdirectories of its argument directory. Those modules may be added by the directory containing their `go.mod` file in separate `use` directives.

```
UseDirective = "use" ( UseSpec | "(" newline { UseSpec } ")" newline ) .
UseSpec = FilePath newline .
FilePath = /* platform-specific relative or absolute file path */
```

Example:

```
use ./mymod  // example.com/mymod

use (
    ../othermod
    ./subdir/thirdmod
)
```

`replace` directive

Similar to a `replace` directive in a `go.mod` file, a `replace` directive in a `go.work` file replaces the contents of a specific version of a module, or all versions of a module, with contents found elsewhere. A wildcard replace in `go.work` overrides a version-specific `replace` in a `go.mod` file.

`replace` directives in `go.work` files override any replaces of the same module or module version in workspace modules.

```
ReplaceDirective = "replace" ( ReplaceSpec | "(" newline { ReplaceSpec } ")" newl
ReplaceSpec = ModulePath [ Version ] "=>" FilePath newline
            | ModulePath [ Version ] "=>" ModulePath Version newline .
FilePath = /* platform-specific relative or absolute file path */
```

Example:

```
replace golang.org/x/net v1.2.3 => example.com/fork/net v1.4.5

replace (
    golang.org/x/net v1.2.3 => example.com/fork/net v1.4.5
    golang.org/x/net => example.com/fork/net v1.4.5
    golang.org/x/net v1.2.3 => ./fork/net
    golang.org/x/net => ./fork/net
)
```

Compatibility with non-module repositories

To ensure a smooth transition from GOPATH to modules, the `go` command can download and build packages in module-aware mode from repositories that have not migrated to modules by adding a `go.mod` file.

When the `go` command downloads a module at a given version directly from a repository, it looks up a repository URL for the module path, maps the version to a revision within the repository, then extracts an archive of the repository at that revision. If the module's path is equal to the repository root path, and the repository root directory does not contain a `go.mod` file, the `go` command synthesizes a `go.mod` file in the module cache that contains a `module` directive and nothing else. Since synthetic `go.mod` files do not contain `require` directives for their dependencies, other modules that depend on them may need additional `require` directives (with `// indirect` comments) to ensure each dependency is fetched at the same version on every build.

When the `go` command downloads a module from a proxy, it downloads the `go.mod` file separately from the rest of the module content. The proxy is expected to serve a synthetic `go.mod` file if the original module didn't have one.

`+incompatible` versions

A module released at major version 2 or higher must have a matching major version suffix on its module path. For example, if a module is released at `v2.0.0`, its path must have a `/v2` suffix. This allows the `go` command to treat multiple major versions of a project as distinct modules, even if they're developed in the same repository.

The major version suffix requirement was introduced when module support was added to the `go` command, and many repositories had already tagged releases with major version 2 or higher before that. To maintain compatibility with these repositories, the `go` command adds an `+incompatible` suffix to versions with major version 2 or higher without a `go.mod` file. `+incompatible` indicates that a version is part of the same module as versions with lower major version numbers; consequently, the `go` command may automatically upgrade to higher `+incompatible` versions even though it may break the build.

Consider the example requirement below:

```
require example.com/m v4.1.2+incompatible
```

The version `v4.1.2+incompatible` refers to the semantic version tag `v4.1.2` in the repository that provides the module `example.com/m`. The module must be in the repository root directory (that is, the repository root path must also be `example.com/m`), and a `go.mod` file must not be present. The module may have versions with lower major version numbers like `v1.5.2`, and the `go` command may upgrade automatically to `v4.1.2+incompatible` from those versions (see minimal version selection (MVS) for information on how upgrades work).

A repository that migrates to modules after version `v2.0.0` is tagged should usually release a new major version. In the example above, the author should create a module with the path `example.com/m/v5` and should release version `v5.0.0`. The author should also update imports of packages in the module to use the prefix `example.com/m/v5` instead of `example.com/m`. See Go Modules: v2 and Beyond for a more detailed example.

Note that the `+incompatible` suffix should not appear on a tag in a repository; a tag like `v4.1.2+incompatible` will be ignored. The suffix only appears in versions used by the `go` command. See Mapping versions to commits for details on the distinction between versions and tags.

Note also that the `+incompatible` suffix may appear on pseudo-versions. For example, `v2.0.1-20200722182040-012345abcdef+incompatible` may be a valid pseudo-version.

Minimal module compatibility

A module released at major version 2 or higher is required to have a major version suffix on its module path. The module may or may not be developed in a major version subdirectory within its repository. This has implications for packages that import packages within the module when building `GOPATH` mode.

Normally in `GOPATH` mode, a package is stored in a directory matching its repository's root path joined with its directory within the repository. For example, a package in the repository with root path `example.com/repo` in the subdirectory `sub` would be stored in `$GOPATH/src/example.com/repo/sub` and would be imported as `example.com/repo/sub`.

For a module with a major version suffix, one might expect to find the package `example.com/repo/v2/sub` in the directory `$GOPATH/src/example.com/repo/v2/sub`. This would require the module to be developed in the `v2` subdirectory of its repository. The `go` command supports this but does not require it (see Mapping versions to commits).

If a module is *not* developed in a major version subdirectory, then its directory in `GOPATH` will not contain the major version suffix, and its packages may be imported without the major version suffix. In the example above, the package would be found in the directory

`$GOPATH/src/example.com/repo/sub` and would be imported as `example.com/repo/sub`.

This creates a problem for packages intended to be built in both module mode and `GOPATH` mode: module mode requires a suffix, while `GOPATH` mode does not.

To fix this, *minimal module compatibility* was added in Go 1.11 and was backported to Go 1.9.7 and 1.10.3. When an import path is resolved to a directory in `GOPATH` mode:

- When resolving an import of the form `$modpath/$vn/$dir` where:
  - `$modpath` is a valid module path,
  - `$vn` is a major version suffix,
  - `$dir` is a possibly empty subdirectory,
- If all of the following are true:
  - The package `$modpath/$vn/$dir` is not present in any relevant `vendor` directory.
  - A `go.mod` file is present in the same directory as the importing file or in any parent directory up to the `$GOPATH/src` root,
  - No `$GOPATH[i]/src/$modpath/$vn/$suffix` directory exists (for any root `$GOPATH[i]`),
  - The file `$GOPATH[d]/src/$modpath/go.mod` exists (for some root `$GOPATH[d]`) and declares the module path as `$modpath/$vn`,
- Then the import of `$modpath/$vn/$dir` is resolved to the directory `$GOPATH[d]/src/$modpath/$dir`.

This rules allow packages that have been migrated to modules to import other packages that have been migrated to modules when built in `GOPATH` mode even when a major version subdirectory was not used.

Module-aware commands

Most `go` commands may run in *Module-aware mode* or *GOPATH mode*. In module-aware mode, the `go` command uses `go.mod` files to find versioned dependencies, and it typically loads packages out of the module cache, downloading modules if they are missing. In `GOPATH` mode, the `go` command ignores modules; it looks in `vendor` directories and in `GOPATH` to find dependencies.

As of Go 1.16, module-aware mode is enabled by default, regardless of whether a `go.mod` file is present. In lower versions, module-aware mode was enabled when a `go.mod` file was present in the current directory or any parent directory.

Module-aware mode may be controlled with the `GO111MODULE` environment variable, which can be set to `on`, `off`, or `auto`.

- If `GO111MODULE=off`, the `go` command ignores `go.mod` files and runs in `GOPATH` mode.
- If `GO111MODULE=on` or is unset, the `go` command runs in module-aware mode, even when no `go.mod` file is present. Not all commands work without a `go.mod` file: see Module commands outside a module.
- If `GO111MODULE=auto`, the `go` command runs in module-aware mode if a `go.mod` file is present in the current directory or any parent directory. In Go 1.15 and lower, this was the default behavior. `go mod` subcommands and `go install` with a version query run in module-aware mode even if no `go.mod` file is present.

In module-aware mode, `GOPATH` no longer defines the meaning of imports during a build, but it still stores downloaded dependencies (in `GOPATH/pkg/mod`; see Module cache) and installed commands (in `GOPATH/bin`, unless `GOBIN` is set).

Build commands

All commands that load information about packages are module-aware. This includes:

- `go build`
- `go fix`
- `go generate`
- `go install`
- `go list`
- `go run`
- `go test`
- `go vet`

When run in module-aware mode, these commands use `go.mod` files to interpret import paths listed on the command line or written in Go source files. These commands accept the following flags, common to all module commands.

- The `–mod` flag controls whether `go.mod` may be automatically updated and whether the `vendor` directory is used.
    - `–mod=mod` tells the `go` command to ignore the vendor directory and to automatically update `go.mod`, for example, when an imported package is not provided by any known module.
    - `–mod=readonly` tells the `go` command to ignore the `vendor` directory and to report an error if `go.mod` needs to be updated.
    - `–mod=vendor` tells the `go` command to use the `vendor` directory. In this mode, the `go` command will not use the network or the module cache.
    - By default, if the go version in `go.mod` is `1.14` or higher and a `vendor` directory is present, the `go` command acts as if `–mod=vendor` were used. Otherwise, the `go` command acts as if `–mod=readonly` were used.
    - `go get` rejects this flag as the purpose of the command is to modify dependencies, which is only allowed by `–mod=mod`.

- The `-modcacherw` flag instructs the `go` command to create new directories in the module cache with read-write permissions instead of making them read-only. When this flag is used consistently (typically by setting `GOFLAGS=-modcacherw` in the environment or by running `go env -w GOFLAGS=-modcacherw`), the module cache may be deleted with commands like `rm -r` without changing permissions first. The `go clean -modcache` command may be used to delete the module cache, whether or not `-modcacherw` was used.
- The `-modfile=file.mod` flag instructs the `go` command to read (and possibly write) an alternate file instead of `go.mod` in the module root directory. The file's name must end with `.mod`. A file named `go.mod` must still be present in order to determine the module root directory, but it is not accessed. When `-modfile` is specified, an alternate `go.sum` file is also used: its path is derived from the `-modfile` flag by trimming the `.mod` extension and appending `.sum`.

Vendoring

When using modules, the `go` command typically satisfies dependencies by downloading modules from their sources into the module cache, then loading packages from those downloaded copies. *Vendoring* may be used to allow interoperation with older versions of Go, or to ensure that all files used for a build are stored in a single file tree.

The `go mod vendor` command constructs a directory named `vendor` in the main module's root directory containing copies of all packages needed to build and test packages in the main module. Packages that are only imported by tests of packages outside the main module are not included. As with `go mod tidy` and other module commands, build constraints except for `ignore` are not considered when constructing the `vendor` directory.

`go mod vendor` also creates the file `vendor/modules.txt` that contains a list of vendored packages and the module versions they were copied from. When vendoring is enabled, this manifest is used as a source of module version information, as reported by `go list -m` and `go version -m`. When the `go` command reads `vendor/modules.txt`, it checks that the module versions are consistent with `go.mod`. If `go.mod` has changed since `vendor/modules.txt` was generated, the `go` command will report an error. `go mod vendor` should be run again to update the `vendor` directory.

If the `vendor` directory is present in the main module's root directory, it will be used automatically if the `go` version in the main module's `go.mod` file is `1.14` or higher. To explicitly enable vendoring, invoke the `go` command with the flag `-mod=vendor`. To disable vendoring, use the flag `-mod=readonly` or `-mod=mod`.

When vendoring is enabled, build commands like `go build` and `go test` load packages from the `vendor` directory instead of accessing the network or the local module cache. The `go list -m` command only prints information about modules listed in `go.mod`. `go mod` commands such as `go mod download` and `go mod tidy` do not work differently when vendoring is enabled and will still download modules and access the module cache. `go get` also does not work differently when vendoring is enabled.

Unlike vendoring in GOPATH mode, the `go` command ignores vendor directories in locations other than the main module's root directory. Additionally, since vendor directories in other modules are not used, the `go` command does not include vendor directories when building module zip files (but see known bugs #31562 and #37397).

`go get`

Usage:

```
go get [-d] [-t] [-u] [build flags] [packages]
```

Examples:

```
# Upgrade a specific module.
$ go get golang.org/x/net

# Upgrade modules that provide packages imported by packages in the main module.
$ go get -u ./...

# Upgrade or downgrade to a specific version of a module.
$ go get golang.org/x/text@v0.3.2

# Update to the commit on the module's master branch.
$ go get golang.org/x/text@master

# Remove a dependency on a module and downgrade modules that require it
# to versions that don't require it.
$ go get golang.org/x/text@none

# Upgrade the minimum required Go version for the main module.
$ go get go

# Upgrade the suggested Go toolchain, leaving the minimum Go version alone.
$ go get toolchain

# Upgrade to the latest patch release of the suggested Go toolchain.
$ go get toolchain@patch
```

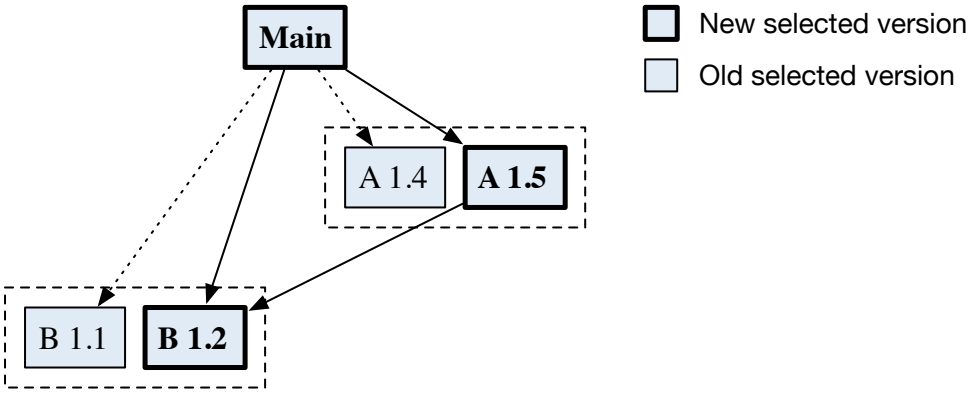
The `go get` command updates module dependencies in the `go.mod` file for the main module, then builds and installs packages listed on the command line.

The first step is to determine which modules to update. `go get` accepts a list of packages, package patterns, and module paths as arguments. If a package argument is specified, `go get` updates the module that provides the package. If a package pattern is specified (for example, `all` or a path with a `...` wildcard), `go get` expands the pattern to a set of packages, then updates the modules that provide the packages. If an argument names a module but not a package (for example, the module `golang.org/x/net` has no package in its root directory), `go get` will update the module but will not build a package. If no arguments are specified, `go get` acts as if `.` were specified (the package in the current directory); this may be used together with the –u flag to update modules that provide imported packages.

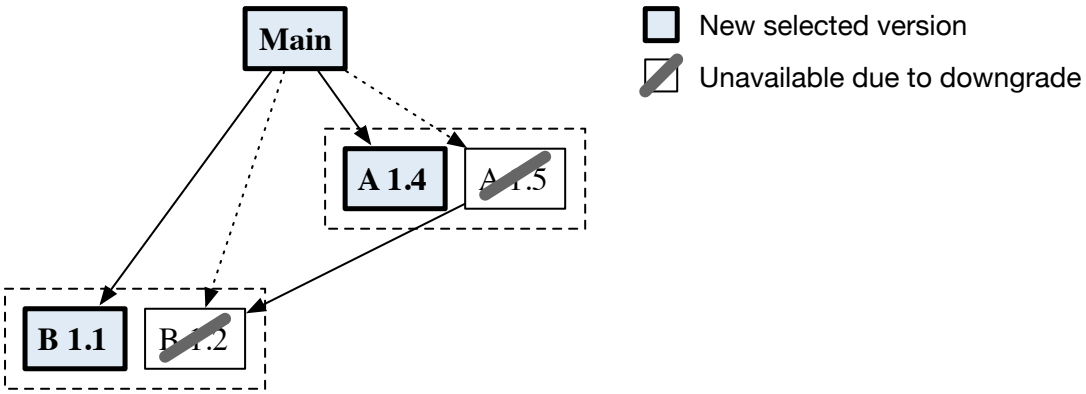
Each argument may include a *version query suffix* indicating the desired version, as in `go get golang.org/x/text@v0.3.0`. A version query suffix consists of an `@` symbol followed by a version query, which may indicate a specific version (`v0.3.0`), a version prefix (`v0.3`), a branch or tag name (`master`), a revision (`1234abcd`), or one of the special queries `latest`, `upgrade`, `patch`, or `none`. If no version is given, `go get` uses the `@upgrade` query.

Once `go get` has resolved its arguments to specific modules and versions, `go get` will add, change, or remove `require` directives in the main module's `go.mod` file to ensure the modules remain at the desired versions in the future. Note that required versions in `go.mod` files are *minimum versions* and may be increased automatically as new dependencies are added. See Minimal version selection (MVS) for details on how versions are selected and conflicts are resolved by module-aware commands.

Other modules may be upgraded when a module named on the command line is added, upgraded, or downgraded if the new version of the named module requires other modules at higher versions. For example, suppose module `example.com/a` is upgraded to version `v1.5.0`, and that version requires module `example.com/b` at version `v1.2.0`. If module `example.com/b` is currently required at version `v1.1.0`, `go get example.com/a@v1.5.0` will also upgrade `example.com/b` to `v1.2.0`.



Other modules may be downgraded when a module named on the command line is downgraded or removed. To continue the above example, suppose module `example.com/b` is downgraded to `v1.1.0`. Module `example.com/a` would also be downgraded to a version that requires `example.com/b` at version `v1.1.0` or lower.



A module requirement may be removed using the version suffix `@none`. This is a special kind of downgrade. Modules that depend on the removed module will be downgraded or removed

as needed. A module requirement may be removed even if one or more of its packages are imported by packages in the main module. In this case, the next build command may add a new module requirement.

If a module is needed at two different versions (specified explicitly in command line arguments or to satisfy upgrades and downgrades), `go get` will report an error.

After `go get` has selected a new set of versions, it checks whether any newly selected module versions or any modules providing packages named on the command line are retracted or deprecated. `go get` prints a warning for each retracted version or deprecated module it finds. `go list -m -u all` may be used to check for retractions and deprecations in all dependencies.

After `go get` updates the `go.mod` file, it builds the packages named on the command line. Executables will be installed in the directory named by the `GOBIN` environment variable, which defaults to `$GOPATH/bin` or `$HOME/go/bin` if the `GOPATH` environment variable is not set.

`go get` supports the following flags:

- The `-d` flag tells `go get` not to build or install packages. When `-d` is used, `go get` will only manage dependencies in `go.mod`. Using `go get` without `-d` to build and install packages is deprecated (as of Go 1.17). In Go 1.18, `-d` will always be enabled.
- The `-u` flag tells `go get` to upgrade modules providing packages imported directly or indirectly by packages named on the command line. Each module selected by `-u` will be upgraded to its latest version unless it is already required at a higher version (a pre-release).
- The `-u=patch` flag (not `-u patch`) also tells `go get` to upgrade dependencies, but `go get` will upgrade each dependency to the latest patch version (similar to the `@patch` version query).
- The `-t` flag tells `go get` to consider modules needed to build tests of packages named on the command line. When `-t` and `-u` are used together, `go get` will update test dependencies as well.
- The `-insecure` flag should no longer be used. It permits `go get` to resolve custom import paths and fetch from repositories and module proxies using insecure schemes such as HTTP. The `GOINSECURE` environment variable provides more fine-grained control and should be used instead.

Since Go 1.16, `go install` is the recommended command for building and installing programs. When used with a version suffix (like `@latest` or `@v1.4.6`), `go install` builds packages in module-aware mode, ignoring the `go.mod` file in the current directory or any parent directory, if there is one.

`go get` is more focused on managing requirements in `go.mod`. The `-d` flag is deprecated, and in Go 1.18, it will always be enabled.

`go install`

Usage:

```
go install [build flags] [packages]
```

Examples:

```
# Install the latest version of a program,
# ignoring go.mod in the current directory (if any).
$ go install golang.org/x/tools/gopls@latest

# Install a specific version of a program.
$ go install golang.org/x/tools/gopls@v0.6.4

# Install a program at the version selected by the module in the current directo
$ go install golang.org/x/tools/gopls

# Install all programs in a directory.
$ go install ./cmd/...
```

The `go install` command builds and installs the packages named by the paths on the command line. Executables (`main` packages) are installed to the directory named by the `GOBIN` environment variable, which defaults to `$GOPATH/bin` or `$HOME/go/bin` if the `GOPATH` environment variable is not set. Executables in `$GOROOT` are installed in `$GOROOT/bin` or `$GOTOOLDIR` instead of `$GOBIN`. Non-executable packages are built and cached but not installed.

Since Go 1.16, if the arguments have version suffixes (like `@latest` or `@v1.0.0`), `go install` builds packages in module-aware mode, ignoring the `go.mod` file in the current directory or any parent directory if there is one. This is useful for installing executables without affecting the dependencies of the main module.

To eliminate ambiguity about which module versions are used in the build, the arguments must satisfy the following constraints:

- Arguments must be package paths or package patterns (with "`...`" wildcards). They must not be standard packages (like `fmt`), meta-patterns (`std`, `cmd`, `all`), or relative or absolute file paths.
- All arguments must have the same version suffix. Different queries are not allowed, even if they refer to the same version.
- All arguments must refer to packages in the same module at the same version.
- Package path arguments must refer to `main` packages. Pattern arguments will only match `main` packages.
- No module is considered the main module.
    - If the module containing packages named on the command line has a `go.mod` file, it must not contain directives (`replace` and `exclude`) that would cause it to be interpreted differently if it were the main module.
    - The module must not require a higher version of itself.

- Vendor directories are not used in any module. (Vendor directories are not included in module zip files, so `go install` does not download them.)

See Version queries for supported version query syntax. Go 1.15 and lower did not support using version queries with `go install`.

If the arguments don't have version suffixes, `go install` may run in module-aware mode or GOPATH mode, depending on the `GO111MODULE` environment variable and the presence of a `go.mod` file. See Module-aware commands for details. If module-aware mode is enabled, `go install` runs in the context of the main module, which may be different from the module containing the package being installed.

`go list -m`

Usage:

```
go list -m [-u] [-retracted] [-versions] [list flags] [modules]
```

Example:

```
$ go list -m all
$ go list -m -versions example.com/m
$ go list -m -json example.com/m@latest
```

The `-m` flag causes `go list` to list modules instead of packages. In this mode, the arguments to `go list` may be modules, module patterns (containing the `...` wildcard), version queries, or the special pattern `all`, which matches all modules in the build list. If no arguments are specified, the main module is listed.

When listing modules, the `-f` flag still specifies a format template applied to a Go struct, but now a `Module` struct:

```
type Module struct {
    Path       string        // module path
    Version    string        // module version
    Versions   []string      // available module versions
    Replace    *Module       // replaced by this module
    Time       *time.Time    // time version was created
    Update     *Module       // available update (with -u)
    Main       bool          // is this the main module?
    Indirect   bool          // module is only indirectly needed by main module
    Dir        string        // directory holding local copy of files, if any
    GoMod      string        // path to go.mod file describing module, if any
    GoVersion  string        // go version used in module
    Retracted  []string      // retraction information, if any (with -retracted o
    Deprecated string        // deprecation message, if any (with -u)
    Error      *ModuleError  // error loading module
}

type ModuleError struct {
```

```
    Err string // the error itself
}
```

The default output is to print the module path and then information about the version and replacement if any. For example, `go list -m all` might print:

```
example.com/main/module
golang.org/x/net v0.1.0
golang.org/x/text v0.3.0 => /tmp/text
rsc.io/pdf v0.1.1
```

The `Module` struct has a `String` method that formats this line of output, so that the default format is equivalent to `-f '{{.String}}'`.

Note that when a module has been replaced, its `Replace` field describes the replacement module, and its `Dir` field is set to the replacement module's source code, if present. (That is, if `Replace` is non-nil, then `Dir` is set to `Replace.Dir`, with no access to the replaced source code.)

The `-u` flag adds information about available upgrades. When the latest version of a given module is newer than the current one, `list -u` sets the module's `Update` field to information about the newer module. `list -u` also prints whether the current selected version is retracted and whether the module is deprecated. The module's `String` method indicates an available upgrade by formatting the newer version in brackets after the current version. For example, `go list -m -u all` might print:

```
example.com/main/module
golang.org/x/old v1.9.9 (deprecated)
golang.org/x/net v0.1.0 (retracted) [v0.2.0]
golang.org/x/text v0.3.0 [v0.4.0] => /tmp/text
rsc.io/pdf v0.1.1 [v0.1.2]
```

(For tools, `go list -m -u -json all` may be more convenient to parse.)

The `-versions` flag causes `list` to set the module's `Versions` field to a list of all known versions of that module, ordered according to semantic versioning, lowest to highest. The flag also changes the default output format to display the module path followed by the space-separated version list. Retracted versions are omitted from this list unless the `-retracted` flag is also specified.

The `-retracted` flag instructs `list` to show retracted versions in the list printed with the `-versions` flag and to consider retracted versions when resolving version queries. For example, `go list -m -retracted example.com/m@latest` shows the highest release or pre-release version of the module `example.com/m`, even if that version is retracted. `retract` directives and deprecations are loaded from the `go.mod` file at this version. The `-retracted` flag was added in Go 1.16.

The template function `module` takes a single string argument that must be a module path or query and returns the specified module as a `Module` struct. If an error occurs, the result will be a `Module` struct with a non-nil `Error` field.

`go mod download`

Usage:

```
go mod download [-x] [-json] [-reuse=old.json] [modules]
```

Example:

```
$ go mod download
$ go mod download golang.org/x/mod@v0.2.0
```

The `go mod download` command downloads the named modules into the module cache. Arguments can be module paths or module patterns selecting dependencies of the main module or version queries of the form `path@version`. With no arguments, `download` applies to all dependencies of the main module.

The `go` command will automatically download modules as needed during ordinary execution. The `go mod download` command is useful mainly for pre-filling the module cache or for loading data to be served by a module proxy.

By default, `download` writes nothing to standard output. It prints progress messages and errors to standard error.

The `-json` flag causes `download` to print a sequence of JSON objects to standard output, describing each downloaded module (or failure), corresponding to this Go struct:

```
type Module struct {
    Path     string // module path
    Query    string // version query corresponding to this version
    Version  string // module version
    Error    string // error loading module
    Info     string // absolute path to cached .info file
    GoMod    string // absolute path to cached .mod file
    Zip      string // absolute path to cached .zip file
    Dir      string // absolute path to cached source root directory
    Sum      string // checksum for path, version (as in go.sum)
    GoModSum string // checksum for go.mod (as in go.sum)
    Origin   any    // provenance of module
    Reuse    bool   // reuse of old module info is safe
}
```

The `-x` flag causes `download` to print the commands `download` executes to standard error.

The -reuse flag accepts the name of file containing the JSON output of a previous 'go mod download -json' invocation. The go command may use this file to determine that a module is unchanged since the previous invocation and avoid redownloading it. Modules that are not

redownloaded will be marked in the new output by setting the Reuse field to true. Normally the module cache provides this kind of reuse automatically; the -reuse flag can be useful on systems that do not preserve the module cache.

`go mod edit`

Usage:

```
go mod edit [editing flags] [-fmt|-print|-json] [go.mod]
```

Example:

```
# Add a replace directive.
$ go mod edit -replace example.com/a@v1.0.0=./a

# Remove a replace directive.
$ go mod edit -dropreplace example.com/a@v1.0.0

# Set the go version, add a requirement, and print the file
# instead of writing it to disk.
$ go mod edit -go=1.14 -require=example.com/m@v1.0.0 -print

# Format the go.mod file.
$ go mod edit -fmt

# Format and print a different .mod file.
$ go mod edit -print tools.mod

# Print a JSON representation of the go.mod file.
$ go mod edit -json
```

The `go mod edit` command provides a command-line interface for editing and formatting `go.mod` files, for use primarily by tools and scripts. `go mod edit` reads only one `go.mod` file; it does not look up information about other modules. By default, `go mod edit` reads and writes the `go.mod` file of the main module, but a different target file can be specified after the editing flags.

The editing flags specify a sequence of editing operations.

- The `-module` flag changes the module's path (the `go.mod` file's module line).
- The `-go=version` flag sets the expected Go language version.
- The `-require=path@version` and `-droprequire=path` flags add and drop a requirement on the given module path and version. Note that `-require` overrides any existing requirements on `path`. These flags are mainly for tools that understand the module graph. Users should prefer `go get path@version` or `go get path@none`, which make other `go.mod` adjustments as needed to satisfy constraints imposed by other modules. See `go get`.
- The `-exclude=path@version` and `-dropexclude=path@version` flags add and drop an exclusion for the given module path and version. Note that `-exclude=path@version` is a no-op if that exclusion already exists.

- The `-replace=old[@v]=new[@v]` flag adds a replacement of the given module path and version pair. If the @v in `old@v` is omitted, a replacement without a version on the left side is added, which applies to all versions of the old module path. If the @v in `new@v` is omitted, the new path should be a local module root directory, not a module path. Note that `-replace` overrides any redundant replacements for `old[@v]`, so omitting @v will drop replacements for specific versions.
- The `-dropreplace=old[@v]` flag drops a replacement of the given module path and version pair. If the @v is provided, a replacement with the given version is dropped. An existing replacement without a version on the left side may still replace the module. If the @v is omitted, a replacement without a version is dropped.
- The `-retract=version` and `-dropretract=version` flags add and drop a retraction for the given version, which may be a single version (like `v1.2.3`) or an interval (like `[v1.1.0,v1.2.0]`). Note that the `-retract` flag cannot add a rationale comment for the `retract` directive. Rationale comments are recommended and may be shown by `go list -m -u` and other commands.

The editing flags may be repeated. The changes are applied in the order given.

`go mod edit` has additional flags that control its output.

- The `-fmt` flag reformats the `go.mod` file without making other changes. This reformatting is also implied by any other modifications that use or rewrite the `go.mod` file. The only time this flag is needed is if no other flags are specified, as in `go mod edit -fmt`.
- The `-print` flag prints the final `go.mod` in its text format instead of writing it back to disk.
- The `-json` flag prints the final `go.mod` in JSON format instead of writing it back to disk in text format. The JSON output corresponds to these Go types:

```
type Module struct {
    Path    string
    Version string
}

type GoMod struct {
    Module  ModPath
    Go      string
    Require []Require
    Exclude []Module
    Replace []Replace
    Retract []Retract
}

type ModPath struct {
    Path       string
    Deprecated string
}

type Require struct {
    Path     string
```

```
    Version  string
    Indirect bool
}

type Replace struct {
    Old Module
    New Module
}

type Retract struct {
    Low       string
    High      string
    Rationale string
}
```

Note that this only describes the `go.mod` file itself, not other modules referred to indirectly. For the full set of modules available to a build, use `go list -m -json all`. See `go list -m`.

For example, a tool can obtain the `go.mod` file as a data structure by parsing the output of `go mod edit -json` and can then make changes by invoking `go mod edit` with `-require`, `-exclude`, and so on.

Tools may also use the package `golang.org/x/mod/modfile` to parse, edit, and format `go.mod` files.

go mod graph

Usage:

```
go mod graph [-go=version]
```

The `go mod graph` command prints the module requirement graph (with replacements applied) in text form. For example:

```
example.com/main example.com/a@v1.1.0
example.com/main example.com/b@v1.2.0
example.com/a@v1.1.0 example.com/b@v1.1.1
example.com/a@v1.1.0 example.com/c@v1.3.0
example.com/b@v1.1.0 example.com/c@v1.1.0
example.com/b@v1.2.0 example.com/c@v1.2.0
```

Each vertex in the module graph represents a specific version of a module. Each edge in the graph represents a requirement on a minimum version of a dependency.

`go mod graph` prints the edges of the graph, one per line. Each line has two space-separated fields: a module version and one of its dependencies. Each module version is identified as a string of the form `path@version`. The main module has no `@version` suffix, since it has no version.

The `-go` flag causes `go mod graph` to report the module graph as loaded by the given Go version, instead of the version indicated by the `go` directive in the `go.mod` file.

See Minimal version selection (MVS) for more information on how versions are chosen. See also `go list -m` for printing selected versions and `go mod why` for understanding why a module is needed.

`go mod init`

Usage:

```
go mod init [module-path]
```

Example:

```
go mod init
go mod init example.com/m
```

The `go mod init` command initializes and writes a new `go.mod` file in the current directory, in effect creating a new module rooted at the current directory. The `go.mod` file must not already exist.

`init` accepts one optional argument, the module path for the new module. See Module paths for instructions on choosing a module path. If the module path argument is omitted, `init` will attempt to infer the module path using import comments in `.go` files, vendoring tool configuration files, and the current directory (if in `GOPATH`).

If a configuration file for a vendoring tool is present, `init` will attempt to import module requirements from it. `init` supports the following configuration files.

- `GLOCKFILE` (Glock)
- `Godeps/Godeps.json` (Godeps)
- `Gopkg.lock` (dep)
- `dependencies.tsv` (godeps)
- `glide.lock` (glide)
- `vendor.conf` (trash)
- `vendor.yml` (govend)
- `vendor/manifest` (gvt)
- `vendor/vendor.json` (govendor)

Vendoring tool configuration files can't always be translated with perfect fidelity. For example, if multiple packages within the same repository are imported at different versions, and the repository only contains one module, the imported `go.mod` can only require the module at one version. You may wish to run `go list -m all` to check all versions in the build list, and `go mod tidy` to add missing requirements and to drop unused requirements.

`go mod tidy`

Usage:

```
go mod tidy [-e] [-v] [-go=version] [-compat=version]
```

`go mod tidy` ensures that the `go.mod` file matches the source code in the module. It adds any missing module requirements necessary to build the current module's packages and dependencies, and it removes requirements on modules that don't provide any relevant packages. It also adds any missing entries to `go.sum` and removes unnecessary entries.

The `-e` flag (added in Go 1.16) causes `go mod tidy` to attempt to proceed despite errors encountered while loading packages.

The `-v` flag causes `go mod tidy` to print information about removed modules to standard error.

`go mod tidy` works by loading all of the packages in the main module and all of the packages they import, recursively. This includes packages imported by tests (including tests in other modules). `go mod tidy` acts as if all build tags are enabled, so it will consider platform-specific source files and files that require custom build tags, even if those source files wouldn't normally be built. There is one exception: the `ignore` build tag is not enabled, so a file with the build constraint `// +build ignore` will not be considered. Note that `go mod tidy` will not consider packages in the main module in directories named `testdata` or with names that start with `.` or `_` unless those packages are explicitly imported by other packages.

Once `go mod tidy` has loaded this set of packages, it ensures that each module that provides one or more packages has a `require` directive in the main module's `go.mod` file or — if the main module is at `go 1.16` or below — is required by another required module. `go mod tidy` will add a requirement on the latest version of each missing module (see Version queries for the definition of the `latest` version). `go mod tidy` will remove `require` directives for modules that don't provide any packages in the set described above.

`go mod tidy` may also add or remove `// indirect` comments on `require` directives. An `// indirect` comment denotes a module that does not provide a package imported by a package in the main module. (See the `require` directive for more detail on when `// indirect` dependencies and comments are added.)

If the `-go` flag is set, `go mod tidy` will update the `go` directive to the indicated version, enabling or disabling module graph pruning and lazy module loading (and adding or removing indirect requirements as needed) according to that version.

By default, `go mod tidy` will check that the selected versions of modules do not change when the module graph is loaded by the Go version immediately preceding the version indicated in the `go` directive. The versioned checked for compatibility can also be specified explicitly via the `-compat` flag.

`go mod vendor`

Usage:

```
go mod vendor [-e] [-v] [-o]
```

The `go mod vendor` command constructs a directory named `vendor` in the main module's root directory that contains copies of all packages needed to support builds and tests of packages in the main module. Packages that are only imported by tests of packages outside the main module are not included. As with `go mod tidy` and other module commands, build constraints except for `ignore` are not considered when constructing the `vendor` directory.

When vendoring is enabled, the `go` command will load packages from the `vendor` directory instead of downloading modules from their sources into the module cache and using packages those downloaded copies. See Vendoring for more information.

`go mod vendor` also creates the file `vendor/modules.txt` that contains a list of vendored packages and the module versions they were copied from. When vendoring is enabled, this manifest is used as a source of module version information, as reported by `go list -m` and `go version -m`. When the `go` command reads `vendor/modules.txt`, it checks that the module versions are consistent with `go.mod`. If `go.mod` changed since `vendor/modules.txt` was generated, `go mod vendor` should be run again.

Note that `go mod vendor` removes the `vendor` directory if it exists before re-constructing it. Local changes should not be made to vendored packages. The `go` command does not check that packages in the `vendor` directory have not been modified, but one can verify the integrity of the `vendor` directory by running `go mod vendor` and checking that no changes were made.

The `-e` flag (added in Go 1.16) causes `go mod vendor` to attempt to proceed despite errors encountered while loading packages.

The `-v` flag causes `go mod vendor` to print the names of vendored modules and packages to standard error.

The `-o` flag (added in Go 1.18) causes `go mod vendor` to output the vendor tree at the specified directory instead of `vendor`. The argument can be either an absolute path or a path relative to the module root.

`go mod verify`

Usage:

```
go mod verify
```

`go mod verify` checks that dependencies of the main module stored in the module cache have not been modified since they were downloaded. To perform this check, `go mod verify` hashes each downloaded module `.zip` file and extracted directory, then compares those hashes with a hash recorded when the module was first downloaded. `go mod verify` checks each module in the build list (which may be printed with `go list -m all`).

If all the modules are unmodified, `go mod verify` prints "all modules verified". Otherwise, it reports which modules have been changed and exits with a non-zero status.

Note that all module-aware commands verify that hashes in the main module's `go.sum` file match hashes recorded for modules downloaded into the module cache. If a hash is missing from `go.sum` (for example, because the module is being used for the first time), the `go` command verifies its hash using the checksum database (unless the module path is matched by `GOPRIVATE` or `GONOSUMDB`). See Authenticating modules for details.

In contrast, `go mod verify` checks that module `.zip` files and their extracted directories have hashes that match hashes recorded in the module cache when they were first downloaded. This is useful for detecting changes to files in the module cache *after* a module has been downloaded and verified. `go mod verify` does not download content for modules not in the cache, and it does not use `go.sum` files to verify module content. However, `go mod verify` may download `go.mod` files in order to perform minimal version selection. It will use `go.sum` to verify those files, and it may add `go.sum` entries for missing hashes.

go mod why

Usage:

```
go mod why [-m] [-vendor] packages...
```

`go mod why` shows a shortest path in the import graph from the main module to each of the listed packages.

The output is a sequence of stanzas, one for each package or module named on the command line, separated by blank lines. Each stanza begins with a comment line starting with # giving the target package or module. Subsequent lines give a path through the import graph, one package per line. If the package or module is not referenced from the main module, the stanza will display a single parenthesized note indicating that fact.

For example:

```
$ go mod why golang.org/x/text/language golang.org/x/text/encoding
# golang.org/x/text/language
rsc.io/quote
rsc.io/sampler
golang.org/x/text/language

# golang.org/x/text/encoding
(main module does not need package golang.org/x/text/encoding)
```

The `-m` flag causes `go mod why` to treat its arguments as a list of modules. `go mod why` will print a path to any package in each of the modules. Note that even when `-m` is used, `go mod why` queries the package graph, not the module graph printed by `go mod graph`.

The `-vendor` flag causes `go mod why` to ignore imports in tests of packages outside the main module (as `go mod vendor` does). By default, `go mod why` considers the graph of

packages matched by the `all` pattern. This flag has no effect after Go 1.16 in modules that declare `go 1.16` or higher (using the go directive in `go.mod`), since the meaning of `all` changed to match the set of packages matched by `go mod vendor`.

`go version -m`

Usage:

```
go version [-m] [-v] [file ...]
```

Example:

```
# Print Go version used to build go.
$ go version

# Print Go version used to build a specific executable.
$ go version ~/go/bin/gopls

# Print Go version and module versions used to build a specific executable.
$ go version -m ~/go/bin/gopls

# Print Go version and module versions used to build executables in a directory.
$ go version -m ~/go/bin/
```

`go version` reports the Go version used to build each executable file named on the command line.

If no files are named on the command line, `go version` prints its own version information.

If a directory is named, `go version` walks that directory, recursively, looking for recognized Go binaries and reporting their versions. By default, `go version` does not report unrecognized files found during a directory scan. The `-v` flag causes it to report unrecognized files.

The `-m` flag causes `go version` to print each executable's embedded module version information, when available. For each executable, `go version -m` prints a table with tab-separated columns like the one below.

```
$ go version -m ~/go/bin/goimports
/home/jrgopher/go/bin/goimports: go1.14.3
        path    golang.org/x/tools/cmd/goimports
        mod     golang.org/x/tools      v0.0.0-20200518203908-8018eb2c26ba      h
        dep     golang.org/x/mod        v0.2.0          h1:KU7oHjnv3XNWfa5COkzUi
        dep     golang.org/x/xerrors    v0.0.0-20191204190536-9bdfabe68543      h
```

The format of the table may change in the future. The same information may be obtained from `runtime/debug.ReadBuildInfo`.

The meaning of each row in the table is determined by the word in the first column.

- **path**: the path of the `main` package used to build the executable.
- **mod**: the module containing the `main` package. The columns are the module path, version, and sum, respectively. The main module has the version `(devel)` and no sum.
- **dep**: a module that provided one or more packages linked into the executable. Same format as `mod`.
- **=>**: a replacement for the module on the previous line. If the replacement is a local directory, only the directory path is listed (no version or sum). If the replacement is a module version, the path, version, and sum are listed, as with `mod` and `dep`. A replaced module has no sum.

`go clean -modcache`

Usage:

```
go clean [-modcache]
```

The `-modcache` flag causes `go clean` to remove the entire module cache, including unpacked source code of versioned dependencies.

This is usually the best way to remove the module cache. By default, most files and directories in the module cache are read-only to prevent tests and editors from unintentionally changing files after they've been authenticated. Unfortunately, this causes commands like `rm -r` to fail, since files can't be removed without first making their parent directories writable.

The `-modcacherw` flag (accepted by `go build` and other module-aware commands) causes new directories in the module cache to be writable. To pass `-modcacherw` to all module-aware commands, add it to the `GOFLAGS` variable. `GOFLAGS` may be set in the environment or with `go env -w`. For example, the command below sets it permanently:

```
go env -w GOFLAGS=-modcacherw
```

`-modcacherw` should be used with caution; developers should be careful not to make changes to files in the module cache. `go mod verify` may be used to check that files in the cache match hashes in the main module's `go.sum` file.

Version queries

Several commands allow you to specify a version of a module using a *version query*, which appears after an `@` character following a module or package path on the command line.

Examples:

```
go get example.com/m@latest
go mod download example.com/m@master
go list -m -json example.com/m@e3702bed2
```

A version query may be one of the following:

- A fully-specified semantic version, such as `v1.2.3`, which selects a specific version. See Versions for syntax.
- A semantic version prefix, such as `v1` or `v1.2`, which selects the highest available version with that prefix.
- A semantic version comparison, such as `<v1.2.3` or `>=v1.5.6`, which selects the nearest available version to the comparison target (the lowest version for > and >=, and the highest version for < and <=).
- A revision identifier for the underlying source repository, such as a commit hash prefix, revision tag, or branch name. If the revision is tagged with a semantic version, this query selects that version. Otherwise, this query selects a pseudo-version for the underlying commit. Note that branches and tags with names matched by other version queries cannot be selected this way. For example, the query `v2` selects the latest version starting with `v2`, not the branch named `v2`.
- The string `latest`, which selects the highest available release version. If there are no release versions, `latest` selects the highest pre-release version. If there are no tagged versions, `latest` selects a pseudo-version for the commit at the tip of the repository's default branch.
- The string `upgrade`, which is like `latest` except that if the module is currently required at a higher version than the version `latest` would select (for example, a pre-release), `upgrade` will select the current version.
- The string `patch`, which selects the latest available version with the same major and minor version numbers as the currently required version. If no version is currently required, `patch` is equivalent to `latest`. Since Go 1.16, `go get` requires a current version when using `patch` (but the `–u=patch` flag does not have this requirement).

Except for queries for specific named versions or revisions, all queries consider available versions reported by `go list –m –versions` (see `go list –m`). This list contains only tagged versions, not pseudo-versions. Module versions disallowed by `exclude` directives in the main module's `go.mod` file are not considered. Versions covered by `retract` directives in the `go.mod` file from the `latest` version of the same module are also ignored except when the `–retracted` flag is used with `go list –m` and except when loading `retract` directives.

Release versions are preferred over pre-release versions. For example, if versions `v1.2.2` and `v1.2.3–pre` are available, the `latest` query will select `v1.2.2`, even though `v1.2.3–pre` is higher. The `<v1.2.4` query would also select `v1.2.2`, even though `v1.2.3–pre` is closer to `v1.2.4`. If no release or pre-release version is available, the `latest`, `upgrade`, and `patch` queries will select a pseudo-version for the commit at the tip of the repository's default branch. Other queries will report an error.

Module commands outside a module

Module-aware Go commands normally run in the context of a main module defined by a `go.mod` file in the working directory or a parent directory. Some commands may be run in module-aware mode without a `go.mod` file, but most commands work differently or report an error when no `go.mod` file is present.

See Module-aware commands for information on enabling and disabling module-aware mode.

Command	Behavior

`go build`
`go doc`
`go fix`
`go fmt`
`go generate`
`go install`
`go list`
`go run`
`go test`
`go vet`
	Only packages in the standard library and packages specified as `.go` files on the command line can be loaded, imported, and built. Packages from other modules cannot be built, since there is no place to record module requirements and ensure deterministic builds.

`go get`	Packages and executables may be built and installed as usual. Note that there is no main module when `go get` is run without a `go.mod` file, so `replace` and `exclude` directives are not applied.
`go list -m`	Explicit version queries are required for most arguments, except when the `-versions` flag is used.
`go mod download`	Explicit version queries are required for most arguments.
`go mod edit`	An explicit file argument is required.

`go mod graph`
`go mod tidy`
`go mod vendor`
`go mod verify`
`go mod why`
	These commands require a `go.mod` file and will report an error if one is not present.

`go work init`

Usage:

```
go work init [moddirs]
```

Init initializes and writes a new go.work file in the current directory, in effect creating a new workspace at the current directory.

go work init optionally accepts paths to the workspace modules as arguments. If the argument is omitted, an empty workspace with no modules will be created.

Each argument path is added to a use directive in the go.work file. The current go version will also be listed in the go.work file.

`go work edit`

Usage:

```
go work edit [editing flags] [go.work]
```

The `go work edit` command provides a command-line interface for editing `go.work`, for use primarily by tools or scripts. It only reads `go.work`; it does not look up information about the modules involved. If no file is specified, Edit looks for a `go.work` file in the current directory and its parent directories

The editing flags specify a sequence of editing operations.

- The `-fmt` flag reformats the go.work file without making other changes. This reformatting is also implied by any other modifications that use or rewrite the `go.work` file. The only time this flag is needed is if no other flags are specified, as in 'go work edit `-fmt`'.
- The `-use=path` and `-dropuse=path` flags add and drop a use directive from the `go.work` file's set of module directories.
- The `-replace=old[@v]=new[@v]` flag adds a replacement of the given module path and version pair. If the @v in `old@v` is omitted, a replacement without a version on the left side is added, which applies to all versions of the old module path. If the @v in new@v is omitted, the new path should be a local module root directory, not a module path. Note that `-replace` overrides any redundant replacements for `old[@v]`, so omitting @v will drop existing replacements for specific versions.
- The `-dropreplace=old[@v]` flag drops a replacement of the given module path and version pair. If the @v is omitted, a replacement without a version on the left side is dropped.
- The `-go=version` flag sets the expected Go language version.

The editing flags may be repeated. The changes are applied in the order given.

`go work edit` has additional flags that control its output

- The -print flag prints the final go.work in its text format instead of writing it back to go.mod.
- The -json flag prints the final go.work file in JSON format instead of writing it back to go.mod. The JSON output corresponds to these Go types:

```
type Module struct {
    Path    string
    Version string
}

type GoWork struct {
    Go        string
```

```
    Directory []Directory
    Replace   []Replace
}

type Use struct {
    Path       string
    ModulePath string
}

type Replace struct {
    Old Module
    New Module
}
```

`go work use`

Usage:

```
go work use [-r] [moddirs]
```

The `go work use` command provides a command-line interface for adding directories, optionally recursively, to a `go.work` file.

A `use` directive will be added to the `go.work` file for each argument directory listed on the command line `go.work` file, if it exists on disk, or removed from the `go.work` file if it does not exist on disk.

The `-r` flag searches recursively for modules in the argument directories, and the use command operates as if each of the directories were specified as arguments.

`go work sync`

Usage:

```
go work sync
```

The `go work sync` command syncs the workspace's build list back to the workspace's modules.

The workspace's build list is the set of versions of all the (transitive) dependency modules used to do builds in the workspace. `go work sync` generates that build list using the Minimal Version Selection (MVS) algorithm, and then syncs those versions back to each of modules specified in the workspace (with `use` directives).

Once the workspace build list is computed, the `go.mod` file for each module in the workspace is rewritten with the dependencies relevant to that module upgraded to match the workspace build list. Note that Minimal Version Selection guarantees that the build list's version of each module is always the same or higher than that in each workspace module.

Module proxies

GOPROXY protocol

A *module proxy* is an HTTP server that can respond to GET requests for paths specified below. The requests have no query parameters, and no specific headers are required, so even a site serving from a fixed file system (including a `file://` URL) can be a module proxy.

Successful HTTP responses must have the status code 200 (OK). Redirects (3xx) are followed. Responses with status codes 4xx and 5xx are treated as errors. The error codes 404 (Not Found) and 410 (Gone) indicate that the requested module or version is not available on the proxy, but it may be found elsewhere. Error responses should have content type `text/plain` with `charset` either `utf-8` or `us-ascii`.

The `go` command may be configured to contact proxies or source control servers using the `GOPROXY` environment variable, which accepts a list of proxy URLs. The list may include the keywords `direct` or `off` (see Environment variables for details). List elements may be separated by commas (`,`) or pipes (`|`), which determine error fallback behavior. When a URL is followed by a comma, the `go` command falls back to later sources only after a 404 (Not Found) or 410 (Gone) response. When a URL is followed by a pipe, the `go` command falls back to later sources after any error, including non-HTTP errors such as timeouts. This error handling behavior lets a proxy act as a gatekeeper for unknown modules. For example, a proxy could respond with error 403 (Forbidden) for modules not on an approved list (see Private proxy serving private modules).

The table below specifies queries that a module proxy must respond to. For each path, `$base` is the path portion of a proxy URL,`$module` is a module path, and `$version` is a version. For example, if the proxy URL is `https://example.com/mod`, and the client is requesting the `go.mod` file for the module `golang.org/x/text` at version `v0.3.2`, the client would send a GET request for `https://example.com/mod/golang.org/x/text/@v/v0.3.2.mod`.

To avoid ambiguity when serving from case-insensitive file systems, the `$module` and `$version` elements are case-encoded by replacing every uppercase letter with an exclamation mark followed by the corresponding lower-case letter. This allows modules `example.com/M` and `example.com/m` to both be stored on disk, since the former is encoded as `example.com/!m`.

Path	Description
`$base/$module/@v/list`	Returns a list of known versions of the given module in plain text, one per line. This list should not include pseudo-versions.
`$base/$module/@v/$version.info`	Returns JSON-formatted metadata about a specific version of a module. The response must be a JSON object that corresponds to the Go data structure below:

Path	Description

	<pre>type Info struct {
	    Version string // version string
	    Time time.Time // commit time
	}</pre>
	The `Version` field is required and must contain a valid, canonical version (see Versions). The `$version` in the request path does not need to be the same version or even a valid version; this endpoint may be used to find versions for branch names or revision identifiers. However, if `$version` is a canonical version with a major version compatible with `$module`, the `Version` field in a successful response must be the same.
	
	The `Time` field is optional. If present, it must be a string in RFC 3339 format. It indicates the time when the version was created.
	
	More fields may be added in the future, so other names are reserved.

`$base/$module/@v/$version.mod`	Returns the `go.mod` file for a specific version of a module. If the module does not have a `go.mod` file at the requested version, a file containing only a `module` statement with the requested module path must be returned. Otherwise, the original, unmodified `go.mod` file must be returned.
`$base/$module/@v/$version.zip`	Returns a zip file containing the contents of a specific version of a module. See Module zip files for details on how this zip file must be formatted.
`$base/$module/@latest`	Returns JSON-formatted metadata about the latest known version of a module in the same format as `$base/$module/@v/$version.info`. The latest version should be the version of the module that the `go` command should use if `$base/$module/@v/list` is empty or no listed version is suitable. This endpoint is optional, and module proxies are not required to implement it.

When resolving the latest version of a module, the `go` command will request `$base/$module/@v/list`, then, if no suitable versions are found, `$base/$module/@latest`. The `go` command prefers, in order: the semantically highest release version, the semantically highest pre-release version, and the chronologically most recent pseudo-version. In Go 1.12 and earlier, the `go` command considered pseudo-versions

in `$base/$module/@v/list` to be pre-release versions, but this is no longer true since Go 1.13.

A module proxy must always serve the same content for successful responses for `$base/$module/$version.mod` and `$base/$module/$version.zip` queries. This content is cryptographically authenticated using `go.sum` files and, by default, the checksum database.

The `go` command caches most content it downloads from module proxies in its module cache in `$GOPATH/pkg/mod/cache/download`. Even when downloading directly from version control systems, the `go` command synthesizes explicit `info`, `mod`, and `zip` files and stores them in this directory, the same as if it had downloaded them directly from a proxy. The cache layout is the same as the proxy URL space, so serving `$GOPATH/pkg/mod/cache/download` at (or copying it to) `https://example.com/proxy` would let users access cached module versions by setting `GOPROXY` to `https://example.com/proxy`.

Communicating with proxies

The `go` command may download module source code and metadata from a module proxy. The `GOPROXY` environment variable may be used to configure which proxies the `go` command may connect to and whether it may communicate directly with version control systems. Downloaded module data is saved in the module cache. The `go` command will only contact a proxy when it needs information not already in the cache.

The `GOPROXY` protocol section describes requests that may be sent to a `GOPROXY` server. However, it's also helpful to understand when the `go` command makes these requests. For example, `go build` follows the procedure below:

- Compute the build list by reading `go.mod` files and performing minimal version selection (MVS).
- Read the packages named on the command line and the packages they import.
- If a package is not provided by any module in the build list, find a module that provides it. Add a module requirement on its latest version to `go.mod`, and start over.
- Build packages after everything is loaded.

When the `go` command computes the build list, it loads the `go.mod` file for each module in the module graph. If a `go.mod` file is not in the cache, the `go` command will download it from the proxy using a `$module/@v/$version.mod` request (where `$module` is the module path and `$version` is the version). These requests can be tested with a tool like `curl`. For example, the command below downloads the `go.mod` file for `golang.org/x/mod` at version `v0.2.0`:

```
$ curl https://proxy.golang.org/golang.org/x/mod/@v/v0.2.0.mod
module golang.org/x/mod

go 1.12
```

```
require (
    golang.org/x/crypto v0.0.0-20191011191535-87dc89f01550
    golang.org/x/tools v0.0.0-20191119224855-298f0cb1881e
    golang.org/x/xerrors v0.0.0-20191011141410-1b5146add898
)
```

In order to load a package, the `go` command needs the source code for the module that provides it. Module source code is distributed in `.zip` files which are extracted into the module cache. If a module `.zip` is not in the cache, the `go` command will download it using a `$module/@v/$version.zip` request.

```
$ curl -O https://proxy.golang.org/golang.org/x/mod/@v/v0.2.0.zip
$ unzip -l v0.2.0.zip | head
Archive:  v0.2.0.zip
  Length      Date    Time    Name
---------  ---------- -----   ----
     1479  00-00-1980 00:00   golang.org/x/mod@v0.2.0/LICENSE
     1303  00-00-1980 00:00   golang.org/x/mod@v0.2.0/PATENTS
      559  00-00-1980 00:00   golang.org/x/mod@v0.2.0/README
       21  00-00-1980 00:00   golang.org/x/mod@v0.2.0/codereview.cfg
      214  00-00-1980 00:00   golang.org/x/mod@v0.2.0/go.mod
     1476  00-00-1980 00:00   golang.org/x/mod@v0.2.0/go.sum
     5224  00-00-1980 00:00   golang.org/x/mod@v0.2.0/gosumcheck/main.go
```

Note that `.mod` and `.zip` requests are separate, even though `go.mod` files are usually contained within `.zip` files. The `go` command may need to download `go.mod` files for many different modules, and `.mod` files are much smaller than `.zip` files. Additionally, if a Go project does not have a `go.mod` file, the proxy will serve a synthetic `go.mod` file that only contains a `module` directive. Synthetic `go.mod` files are generated by the `go` command when downloading from a version control system.

If the `go` command needs to load a package not provided by any module in the build list, it will attempt to find a new module that provides it. The section Resolving a package to a module describes this process. In summary, the `go` command requests information about the latest version of each module path that could possibly contain the package. For example, for the package `golang.org/x/net/html`, the `go` command would try to find the latest versions of the modules `golang.org/x/net/html`, `golang.org/x/net`, `golang.org/x/`, and `golang.org`. Only `golang.org/x/net` actually exists and provides that package, so the `go` command uses the latest version of that module. If more than one module provides the package, the `go` command will use the module with the longest path.

When the `go` command requests the latest version of a module, it first sends a request for `$module/@v/list`. If the list is empty or none of the returned versions can be used, it sends a request for `$module/@latest`. Once a version is chosen, the `go` command sends a `$module/@v/$version.info` request for metadata. It may then send `$module/@v/$version.mod` and `$module/@v/$version.zip` requests to load the `go.mod` file and source code.

```
$ curl https://proxy.golang.org/golang.org/x/mod/@v/list
v0.1.0
v0.2.0

$ curl https://proxy.golang.org/golang.org/x/mod/@v/v0.2.0.info
{"Version":"v0.2.0","Time":"2020-01-02T17:33:45Z"}
```

After downloading a `.mod` or `.zip` file, the `go` command computes a cryptographic hash and checks that it matches a hash in the main module's `go.sum` file. If the hash is not present in `go.sum`, by default, the `go` command retrieves it from the checksum database. If the computed hash does not match, the `go` command reports a security error and does not install the file in the module cache. The `GOPRIVATE` and `GONOSUMDB` environment variables may be used to disable requests to the checksum database for specific modules. The `GOSUMDB` environment variable may also be set to `off` to disable requests to the checksum database entirely. See Authenticating modules for more information. Note that version lists and version metadata returned for `.info` requests are not authenticated and may change over time.

Serving modules directly from a proxy

Most modules are developed and served from a version control repository. In direct mode, the `go` command downloads such a module with a version control tool (see Version control systems). It's also possible to serve a module directly from a module proxy. This is useful for organizations that want to serve modules without exposing their version control servers and for organizations that use version control tools the `go` command does not support.

When the `go` command downloads a module in direct mode, it first looks up the module server's URL with an HTTP GET request based on the module path. It looks for a `<meta>` tag with the name `go-import` in the HTML response. The tag's content must contain the repository root path, the version control system, and the URL, separated by spaces. See Finding a repository for a module path for details.

If the version control system is `mod`, the `go` command downloads the module from the given URL using the `GOPROXY` protocol.

For example, suppose the `go` command is attempting to download the module `example.com/gopher` at version `v1.0.0`. It sends a request to `https://example.com/gopher?go-get=1`. The server responds with an HTML document containing the tag:

```
<meta name="go-import" content="example.com/gopher mod https://modproxy.example.c
```

Based on this response, the `go` command downloads the module by sending requests for `https://modproxy.example.com/example.com/gopher/@v/v1.0.0.info`, `v1.0.0.mod`, and `v1.0.0.zip`.

Note that modules served directly from a proxy cannot be downloaded with `go get` in GOPATH mode.

Version control systems

The `go` command may download module source code and metadata directly from a version control repository. Downloading a module from a proxy is usually faster, but connecting directly to a repository is necessary if a proxy is not available or if a module's repository is not accessible to a proxy (frequently true for private repositories). Git, Subversion, Mercurial, Bazaar, and Fossil are supported. A version control tool must be installed in a directory in `PATH` in order for the `go` command to use it.

To download specific modules from source repositories instead of a proxy, set the `GOPRIVATE` or `GONOPROXY` environment variables. To configure the `go` command to download all modules directly from source repositories, set `GOPROXY` to `direct`. See Environment variables for more information.

Finding a repository for a module path

When the `go` command downloads a module in `direct` mode, it starts by locating the repository that contains the module.

If the module path has a VCS qualifier (one of `.bzr`, `.fossil`, `.git`, `.hg`, `.svn`) at the end of a path component, the `go` command will use everything up to that path qualifier as the repository URL. For example, for the module `example.com/foo.git/bar`, the `go` command downloads the repository at `example.com/foo.git` using git, expecting to find the module in the `bar` subdirectory. The `go` command will guess the protocol to use based on the protocols supported by the version control tool.

If the module path does not have a qualifier, the `go` command sends an HTTP `GET` request to a URL derived from the module path with a `?go-get=1` query string. For example, for the module `golang.org/x/mod`, the `go` command may send the following requests:

```
https://golang.org/x/mod?go-get=1 (preferred)
http://golang.org/x/mod?go-get=1  (fallback, only with GOINSECURE)
```

The `go` command follows redirects but otherwise ignores response status codes, so the server may respond with a 404 or any other error status. The `GOINSECURE` environment variable may be set to allow fallback and redirects to unencrypted HTTP for specific modules.

The server must respond with an HTML document containing a `<meta>` tag in the document's `<head>`. The `<meta>` tag should appear early in the document to avoid confusing the `go` command's restricted parser. In particular, it should appear before any raw JavaScript or CSS. The `<meta>` tag must have the form:

```
<meta name="go-import" content="root-path vcs repo-url">
```

`root-path` is the repository root path, the portion of the module path that corresponds to the repository's root directory. It must be a prefix or an exact match of the requested module

path. If it's not an exact match, another request is made for the prefix to verify the `<meta>` tags match.

`vcs` is the version control system. It must be one of the tools listed in the table below or the keyword `mod`, which instructs the `go` command to download the module from the given URL using the GOPROXY protocol. See Serving modules directly from a proxy for details.

`repo-url` is the repository's URL. If the URL does not include a scheme (either because the module path has a VCS qualifier or because the `<meta>` tag lacks a scheme), the `go` command will try each protocol supported by the version control system. For example, with Git, the `go` command will try `https://` then `git+ssh://`. Insecure protocols (like `http://` and `git://`) may only be used if the module path is matched by the `GOINSECURE` environment variable.

Name	Command	GOVCS default	Secure schemes
Bazaar	`bzr`	Private only	`https`, `bzr+ssh`
Fossil	`fossil`	Private only	`https`
Git	`git`	Public and private	`https`, `git+ssh`, `ssh`
Mercurial	`hg`	Public and private	`https`, `ssh`
Subversion	`svn`	Private only	`https`, `svn+ssh`

As an example, consider `golang.org/x/mod` again. The `go` command sends a request to `https://golang.org/x/mod?go-get=1`. The server responds with an HTML document containing the tag:

```
<meta name="go-import" content="golang.org/x/mod git https://go.googlesource.com/
```

From this response, the `go` command will use the Git repository at the remote URL `https://go.googlesource.com/mod`.

GitHub and other popular hosting services respond to `?go-get=1` queries for all repositories, so usually no server configuration is necessary for modules hosted at those sites.

After the repository URL is found, the `go` command will clone the repository into the module cache. In general, the `go` command tries to avoid fetching unneeded data from a repository. However, the actual commands used vary by version control system and may change over time. For Git, the `go` command can list most available versions without downloading commits. It will usually fetch commits without downloading ancestor commits, but doing so is sometimes necessary.

Mapping versions to commits

The `go` command may check out a module within a repository at a specific canonical version like `v1.2.3`, `v2.4.0-beta`, or `v3.0.0+incompatible`. Each module version should have a *semantic version tag* within the repository that indicates which revision should be checked out for a given version.

If a module is defined in the repository root directory or in a major version subdirectory of the root directory, then each version tag name is equal to the corresponding version. For example, the module `golang.org/x/text` is defined in the root directory of its repository, so the version `v0.3.2` has the tag `v0.3.2` in that repository. This is true for most modules.

If a module is defined in a subdirectory within the repository, that is, the module subdirectory portion of the module path is not empty, then each tag name must be prefixed with the module subdirectory, followed by a slash. For example, the module `golang.org/x/tools/gopls` is defined in the `gopls` subdirectory of the repository with root path `golang.org/x/tools`. The version `v0.4.0` of that module must have the tag named `gopls/v0.4.0` in that repository.

The major version number of a semantic version tag must be consistent with the module path's major version suffix (if any). For example, the tag `v1.0.0` could belong to the module `example.com/mod` but not `example.com/mod/v2`, which would have tags like `v2.0.0`.

A tag with major version `v2` or higher may belong to a module without a major version suffix if no `go.mod` file is present, and the module is in the repository root directory. This kind of version is denoted with the suffix `+incompatible`. The version tag itself must not have the suffix. See Compatibility with non-module repositories.

Once a tag is created, it should not be deleted or changed to a different revision. Versions are authenticated to ensure safe, repeatable builds. If a tag is modified, clients may see a security error when downloading it. Even after a tag is deleted, its content may remain available on module proxies.

Mapping pseudo-versions to commits

The `go` command may check out a module within a repository at a specific revision, encoded as a pseudo-version like `v1.3.2-0.20191109021931-daa7c04131f5`.

The last 12 characters of the pseudo-version (`daa7c04131f5` in the example above) indicate a revision in the repository to check out. The meaning of this depends on the version control system. For Git and Mercurial, this is a prefix of a commit hash. For Subversion, this is a zero-padded revision number.

Before checking out a commit, the `go` command verifies that the timestamp (`20191109021931` above) matches the commit date. It also verifies that the base version (`v1.3.1`, the version before `v1.3.2` in the example above) corresponds to a semantic version tag that is an ancestor of the commit. These checks ensure that module authors have full control over how pseudo-versions compare with other released versions.

See Pseudo-versions for more information.

Mapping branches and commits to versions

A module may be checked out at a specific branch, tag, or revision using a version query.

```
go get example.com/mod@master
```

The `go` command converts these names into canonical versions that can be used with minimal version selection (MVS). MVS depends on the ability to order versions unambiguously. Branch names and revisions can't be compared reliably over time, since they depend on repository structure which may change.

If a revision is tagged with one or more semantic version tags like `v1.2.3`, the tag for the highest valid version will be used. The `go` command only considers semantic version tags that could belong to the target module; for example, the tag `v1.5.2` would not be considered for `example.com/mod/v2` since the major version doesn't match the module path's suffix.

If a revision is not tagged with a valid semantic version tag, the `go` command will generate a pseudo-version. If the revision has ancestors with valid semantic version tags, the highest ancestor version will be used as the pseudo-version base. See Pseudo-versions.

Module directories within a repository

Once a module's repository has been checked out at a specific revision, the `go` command must locate the directory that contains the module's `go.mod` file (the module's root directory).

Recall that a module path consists of three parts: a repository root path (corresponding to the repository root directory), a module subdirectory, and a major version suffix (only for modules released at `v2` or higher).

For most modules, the module path is equal to the repository root path, so the module's root directory is the repository's root directory.

Modules are sometimes defined in repository subdirectories. This is typically done for large repositories with multiple components that need to be released and versioned independently. Such a module is expected to be found in a subdirectory that matches the part of the module's path after the repository root path. For example, suppose the module `example.com/monorepo/foo/bar` is in the repository with root path `example.com/monorepo`. Its `go.mod` file must be in the `foo/bar` subdirectory.

If a module is released at major version `v2` or higher, its path must have a major version suffix. A module with a major version suffix may be defined in one of two subdirectories: one with the suffix, and one without. For example, suppose a new version of the module above is released with the path `example.com/monorepo/foo/bar/v2`. Its `go.mod` file may be in either `foo/bar` or `foo/bar/v2`.

Subdirectories with a major version suffix are *major version subdirectories*. They may be used to develop multiple major versions of a module on a single branch. This may be unnecessary when development of multiple major versions proceeds on separate branches. However, major version subdirectories have an important property: in `GOPATH` mode, package import paths exactly match directories under `GOPATH/src`. The `go` command

provides minimal module compatibility in GOPATH mode (see Compatibility with non-module repositories), so major version subdirectories aren't always necessary for compatibility with projects built in GOPATH mode. Older tools that don't support minimal module compatibility may have problems though.

Once the `go` command has found the module root directory, it creates a `.zip` file of the contents of the directory, then extracts the `.zip` file into the module cache. See File path and size constraints for details on what files may be included in the `.zip` file. The contents of the `.zip` file are authenticated before extraction into the module cache the same way they would be if the `.zip` file were downloaded from a proxy.

Module zip files do not include the contents of `vendor` directories or any nested modules (subdirectories that contain `go.mod` files). This means a module must take care not to refer to files outside its directory or in other modules. For example, `//go:embed` patterns must not match files in nested modules. This behavior may serve as a useful workaround in situations where files should not be included in a module. For example, if a repository has large files checked into a `testdata` directory, the module author could add an empty `go.mod` file in `testdata` so their users don't need to download those files. Of course, this may reduce coverage for users testing their dependencies.

Special case for LICENSE files

When the `go` command creates a `.zip` file for a module that is not in the repository root directory, if the module does not have a file named `LICENSE` in its root directory (alongside `go.mod`), the `go` command will copy the file named `LICENSE` from the repository root directory if it is present in the same revision.

This special case allows the same `LICENSE` file to apply to all modules within a repository. This only applies to files named `LICENSE` specifically, without extensions like `.txt`. Unfortunately, this cannot be extended without breaking cryptographic sums of existing modules; see Authenticating modules. Other tools and websites like pkg.go.dev may recognize files with other names.

Note also that the `go` command does not include symbolic links when creating module `.zip` files; see File path and size constraints. Consequently, if a repository does not have a `LICENSE` file in its root directory, authors may instead create copies of their license files in modules defined in subdirectories to ensure those files are included in module `.zip` files.

Controlling version control tools with GOVCS

The `go` command's ability to download modules with version control commands like `git` is critical to the decentralized package ecosystem, in which code can be imported from any server. It is also a potential security problem if a malicious server finds a way to cause the invoked version control command to run unintended code.

To balance the functionality and security concerns, the `go` command by default will only use `git` and `hg` to download code from public servers. It will use any known version control system to download code from private servers, defined as those hosting packages matching

the GOPRIVATE environment variable. The rationale behind allowing only Git and Mercurial is that these two systems have had the most attention to issues of being run as clients of untrusted servers. In contrast, Bazaar, Fossil, and Subversion have primarily been used in trusted, authenticated environments and are not as well scrutinized as attack surfaces.

The version control command restrictions only apply when using direct version control access to download code. When downloading modules from a proxy, the go command uses the GOPROXY protocol instead, which is always permitted. By default, the go command uses the Go module mirror (proxy.golang.org) for public modules and only falls back to version control for private modules or when the mirror refuses to serve a public package (typically for legal reasons). Therefore, clients can still access public code served from Bazaar, Fossil, or Subversion repositories by default, because those downloads use the Go module mirror, which takes on the security risk of running the version control commands using a custom sandbox.

The GOVCS variable can be used to change the allowed version control systems for specific modules. The GOVCS variable applies when building packages in both module-aware mode and GOPATH mode. When using modules, the patterns match against the module path. When using GOPATH, the patterns match against the import path corresponding to the root of the version control repository.

The general form of the GOVCS variable is a comma-separated list of `pattern:vcslist` rules. The pattern is a glob pattern that must match one or more leading elements of the module or import path. The vcslist is a pipe-separated list of allowed version control commands, or `all` to allow use of any known command, or `off` to allow nothing. Note that if a module matches a pattern with vcslist `off`, it may still be downloaded if the origin server uses the `mod` scheme, which instructs the go command to download the module using the GOPROXY protocol. The earliest matching pattern in the list applies, even if later patterns might also match.

For example, consider:

```
GOVCS=github.com:git,evil.com:off,*:git|hg
```

With this setting, code with a module or import path beginning with `github.com/` can only use `git`; paths on `evil.com` cannot use any version control command, and all other paths (`*` matches everything) can use only `git` or `hg`.

The special patterns `public` and `private` match public and private module or import paths. A path is private if it matches the GOPRIVATE variable; otherwise it is public.

If no rules in the GOVCS variable match a particular module or import path, the go command applies its default rule, which can now be summarized in GOVCS notation as `public:git|hg,private:all`.

To allow unfettered use of any version control system for any package, use:

```
GOVCS=*:all
```

To disable all use of version control, use:

```
GOVCS=*:off
```

The `go env -w` command can be used to set the `GOVCS` variable for future go command invocations.

`GOVCS` was introduced in Go 1.16. Earlier versions of Go may use any known version control tool for any module.

Module zip files

Module versions are distributed as `.zip` files. There is rarely any need to interact directly with these files, since the `go` command creates, downloads, and extracts them automatically from module proxies and version control repositories. However, it's still useful to know about these files to understand cross-platform compatibility constraints or when implementing a module proxy.

The `go mod download` command downloads zip files for one or more modules, then extracts those files into the module cache. Depending on `GOPROXY` and other environment variables, the `go` command may either download zip files from a proxy or clone source control repositories and create zip files from them. The `-json` flag may be used to find the location of download zip files and their extracted contents in the module cache.

The `golang.org/x/mod/zip` package may be used to create, extract, or check contents of zip files programmatically.

File path and size constraints

There are a number of restrictions on the content of module zip files. These constraints ensure that zip files can be extracted safely and consistently on a wide range of platforms.

- A module zip file may be at most 500 MiB in size. The total uncompressed size of its files is also limited to 500 MiB. `go.mod` files are limited to 16 MiB. `LICENSE` files are also limited to 16 MiB. These limits exist to mitigate denial of service attacks on users, proxies, and other parts of the module ecosystem. Repositories that contain more than 500 MiB of files in a module directory tree should tag module versions at commits that only include files needed to build the module's packages; videos, models, and other large assets are usually not needed for builds.
- Each file within a module zip file must begin with the prefix `$module@$version/` where `$module` is the module path and `$version` is the version, for example, `golang.org/x/mod@v0.3.0/`. The module path must be valid, the version must be valid and canonical, and the version must match the module path's major version suffix. See Module paths and versions for specific definitions and restrictions.
- File modes, timestamps, and other metadata are ignored.

- Empty directories (entries with paths ending with a slash) may be included in module zip files but are not extracted. The `go` command does not include empty directories in zip files it creates.
- Symbolic links and other irregular files are ignored when creating zip files, since they aren't portable across operating systems and file systems, and there's no portable way to represent them in the zip file format.
- Files within directories named `vendor` are ignored when creating zip files, since `vendor` directories outside the main module are never used.
- Files within directories containing `go.mod` files, other than the module root directory, are ignored when creating zip files, since they are not part of the module. The `go` command ignores subdirectories containing `go.mod` files when extracting zip files.
- No two files within a zip file may have paths equal under Unicode case-folding (see `strings.EqualFold`). This ensures that zip files can be extracted on case-insensitive file systems without collisions.
- A `go.mod` file may or may not appear in the top-level directory (`$module@$version/go.mod`). If present, it must have the name `go.mod` (all lowercase). Files named `go.mod` are not allowed in any other directory.
- File and directory names within a module may consist of Unicode letters, ASCII digits, the ASCII space character (U+0020), and the ASCII punctuation characters `!#$%& ()+,-.=@[]^_{}~`. Note that package paths may not contain all these characters. See `module.CheckFilePath` and `module.CheckImportPath` for the differences.
- A file or directory name up to the first dot must not be a reserved file name on Windows, regardless of case (`CON`, `com1`, `NuL`, and so on).

Private modules

Go modules are frequently developed and distributed on version control servers and module proxies that aren't available on the public internet. The `go` command can download and build modules from private sources, though it usually requires some configuration.

The environment variables below may be used to configure access to private modules. See Environment variables for details. See also Privacy for information on controlling information sent to public servers.

- `GOPROXY` — list of module proxy URLs. The `go` command will attempt to download modules from each server in sequence. The keyword `direct` instructs the `go` command to download modules from version control repositories where they're developed instead of using a proxy.
- `GOPRIVATE` — list of glob patterns of module path prefixes that should be considered private. Acts as a default value for `GONOPROXY` and `GONOSUMDB`.
- `GONOPROXY` — list of glob patterns of module path prefixes that should not be downloaded from a proxy. The `go` command will download matching modules from version control repositories where they're developed, regardless of `GOPROXY`.
- `GONOSUMDB` — list of glob patterns of module path prefixes that should not be checked using the public checksum database, sum.golang.org.

- `GOINSECURE` — list of glob patterns of module path prefixes that may be retrieved over HTTP and other insecure protocols.

These variables may be set in the development environment (for example, in a `.profile` file), or they may be set permanently with `go env -w`.

The rest of this section describes common patterns for providing access to private module proxies and version control repositories.

Private proxy serving all modules

A central private proxy server that serves all modules (public and private) provides the most control for administrators and requires the least configuration for individual developers.

To configure the `go` command to use such a server, set the following environment variables, replacing `https://proxy.corp.example.com` with your proxy URL and `corp.example.com` with your module prefix:

```
GOPROXY=https://proxy.corp.example.com
GONOSUMDB=corp.example.com
```

The `GOPROXY` setting instructs the `go` command to only download modules from `https://proxy.corp.example.com`; the `go` command will not connect to other proxies or version control repositories.

The `GONOSUMDB` setting instructs the `go` command not to use the public checksum database to authenticate modules with paths starting with `corp.example.com`.

A proxy running in this configuration will likely need read access to private version control servers. It will also need access to the public internet to download new versions of public modules.

There are several existing implementations of `GOPROXY` servers that may be used this way. A minimal implementation would serve files from a module cache directory and would use `go mod download` (with suitable configuration) to retrieve missing modules.

Private proxy serving private modules

A private proxy server may serve private modules without also serving publicly available modules. The `go` command can be configured to fall back to public sources for modules that aren't available on the private server.

To configure the `go` command to work this way, set the following environment variables, replacing `https://proxy.corp.example.com` with the proxy URL and `corp.example.com` with the module prefix:

```
GOPROXY=https://proxy.corp.example.com,https://proxy.golang.org,direct
GONOSUMDB=corp.example.com
```

The GOPROXY setting instructs the go command to try to download modules from `https://proxy.corp.example.com` first. If that server responds with 404 (Not Found) or 410 (Gone), the go command will fall back to `https://proxy.golang.org`, then to direct connections to repositories.

The GONOSUMDB setting instructs the go command not to use the public checksum database to authenticate modules whose paths start with `corp.example.com`.

Note that a proxy used in this configuration may still control access to public modules, even though it doesn't serve them. If the proxy responds to a request with an error status other than 404 or 410, the go command will not fall back to later entries in the GOPROXY list. For example, the proxy could respond with 403 (Forbidden) for a module with an unsuitable license or with known security vulnerabilities.

Direct access to private modules

The go command may be configured to bypass public proxies and download private modules directly from version control servers. This is useful when running a private proxy server is not feasible.

To configure the go command to work this way, set GOPRIVATE, replacing `corp.example.com` the private module prefix:

```
GOPRIVATE=corp.example.com
```

The GOPROXY variable does not need to be changed in this situation. It defaults to `https://proxy.golang.org,direct`, which instructs the go command to attempt to download modules from `https://proxy.golang.org` first, then fall back to a direct connection if that proxy responds with 404 (Not Found) or 410 (Gone).

The GOPRIVATE setting instructs the go command not to connect to a proxy or to the checksum database for modules starting with `corp.example.com`.

An internal HTTP server may still be needed to resolve module paths to repository URLs. For example, when the go command downloads the module `corp.example.com/mod`, it will send a GET request to `https://corp.example.com/mod?go-get=1`, and it will look for the repository URL in the response. To avoid this requirement, ensure that each private module path has a VCS suffix (like `.git`) marking the repository root prefix. For example, when the go command downloads the module `corp.example.com/repo.git/mod`, it will clone the Git repository at `https://corp.example.com/repo.git` or `ssh://corp.example.com/repo.git` without needing to make additional requests.

Developers will need read access to repositories containing private modules. This may be configured in global VCS configuration files like `.gitconfig`. It's best if VCS tools are configured not to need interactive authentication prompts. By default, when invoking Git, the go command disables interactive prompts by setting `GIT_TERMINAL_PROMPT=0`, but it respects explicit settings.

Passing credentials to private proxies

The `go` command supports HTTP basic authentication when communicating with proxy servers.

Credentials may be specified in a `.netrc` file. For example, a `.netrc` file containing the lines below would configure the `go` command to connect to the machine `proxy.corp.example.com` with the given username and password.

```
machine proxy.corp.example.com
login jrgopher
password hunter2
```

The location of the file may be set with the `NETRC` environment variable. If `NETRC` is not set, the `go` command will read `$HOME/.netrc` on UNIX-like platforms or `%USERPROFILE%\_netrc` on Windows.

Fields in `.netrc` are separated with spaces, tabs, and newlines. Unfortunately, these characters cannot be used in usernames or passwords. Note also that the machine name cannot be a full URL, so it's not possible to specify different usernames and passwords for different paths on the same machine.

Alternatively, credentials may be specified directly in `GOPROXY` URLs. For example:

```
GOPROXY=https://jrgopher:hunter2@proxy.corp.example.com
```

Use caution when taking this approach: environment variables may appear in shell history and in logs.

Passing credentials to private repositories

The `go` command may download a module directly from a version control repository. This is necessary for private modules if a private proxy is not used. See Direct access to private modules for configuration.

The `go` command runs version control tools like `git` when downloading modules directly. These tools perform their own authentication, so you may need to configure credentials in a tool-specific configuration file like `.gitconfig`.

To ensure this works smoothly, make sure the `go` command uses the correct repository URL and that the version control tool doesn't require a password to be entered interactively. The `go` command prefers `https://` URLs over other schemes like `ssh://` unless the scheme was specified when looking up the repository URL. For GitHub repositories specifically, the `go` command assumes `https://`.

For most servers, you can configure your client to authenticate over HTTP. For example, GitHub supports using OAuth personal access tokens as HTTP passwords. You can store HTTP passwords in a `.netrc` file, as when passing credentials to private proxies.

Alternatively, you can rewrite `https://` URLs to another scheme. For example, in `.gitconfig`:

```
[url "git@github.com:"]
    insteadOf = https://github.com/
```

For more information, see Why does “go get” use HTTPS when cloning a repository?

Privacy

The `go` command may download modules and metadata from module proxy servers and version control systems. The environment variable `GOPROXY` controls which servers are used. The environment variables `GOPRIVATE` and `GONOPROXY` control which modules are fetched from proxies.

The default value of `GOPROXY` is:

```
https://proxy.golang.org,direct
```

With this setting, when the `go` command downloads a module or module metadata, it will first send a request to `proxy.golang.org`, a public module proxy operated by Google (privacy policy). See `GOPROXY` protocol for details on what information is sent in each request. The `go` command does not transmit personally identifiable information, but it does transmit the full module path being requested. If the proxy responds with a 404 (Not Found) or 410 (Gone) status, the `go` command will attempt to connect directly to the version control system providing the module. See Version control systems for details.

The `GOPRIVATE` or `GONOPROXY` environment variables may be set to lists of glob patterns matching module prefixes that are private and should not be requested from any proxy. For example:

```
GOPRIVATE=*.corp.example.com,*.research.example.com
```

`GOPRIVATE` simply acts as a default for `GONOPROXY` and `GONOSUMDB`, so it’s not necessary to set `GONOPROXY` unless `GONOSUMDB` should have a different value. When a module path is matched by `GONOPROXY`, the `go` command ignores `GOPROXY` for that module and fetches it directly from its version control repository. This is useful when no proxy serves private modules. See Direct access to private modules.

If there is a trusted proxy serving all modules, then `GONOPROXY` should not be set. For example, if `GOPROXY` is set to one source, the `go` command will not download modules from other sources. `GONOSUMDB` should still be set in this situation.

```
GOPROXY=https://proxy.corp.example.com
GONOSUMDB=*.corp.example.com,*.research.example.com
```

If there is a trusted proxy serving only private modules, GONOPROXY should not be set, but care must be taken to ensure the proxy responds with the correct status codes. For example, consider the following configuration:

```
GOPROXY=https://proxy.corp.example.com,https://proxy.golang.org
GONOSUMDB=*.corp.example.com,*.research.example.com
```

Suppose that due to a typo, a developer attempts to download a module that doesn't exist.

```
go mod download corp.example.com/secret-product/typo@latest
```

The go command first requests this module from proxy.corp.example.com. If that proxy responds with 404 (Not Found) or 410 (Gone), the go command will fall back to proxy.golang.org, transmitting the secret-product path in the request URL. If the private proxy responds with any other error code, the go command prints the error and will not fall back to other sources.

In addition to proxies, the go command may connect to the checksum database to verify cryptographic hashes of modules not listed in go.sum. The GOSUMDB environment variable sets the name, URL, and public key of the checksum database. The default value of GOSUMDB is sum.golang.org, the public checksum database operated by Google (privacy policy). See Checksum database for details on what is transmitted with each request. As with proxies, the go command does not transmit personally identifiable information, but it does transmit the full module path being requested, and the checksum database cannot compute checksums for non-public modules.

The GONOSUMDB environment variable may be set to patterns indicating which modules are private and should not be requested from the checksum database. GOPRIVATE acts as a default for GONOSUMDB and GONOPROXY, so it's not necessary to set GONOSUMDB unless GONOPROXY should have a different value.

A proxy may mirror the checksum database. If a proxy in GOPROXY does this, the go command will not connect to the checksum database directly.

GOSUMDB may be set to off to disable use of the checksum database entirely. With this setting, the go command will not authenticate downloaded modules unless they're already in go.sum. See Authenticating modules.

Module cache

The *module cache* is the directory where the go command stores downloaded module files. The module cache is distinct from the build cache, which contains compiled packages and other build artifacts.

The default location of the module cache is $GOPATH/pkg/mod. To use a different location, set the GOMODCACHE environment variable.

The module cache has no maximum size, and the `go` command does not remove its contents automatically.

The cache may be shared by multiple Go projects developed on the same machine. The `go` command will use the same cache regardless of the location of the main module. Multiple instances of the `go` command may safely access the same module cache at the same time.

The `go` command creates module source files and directories in the cache with read-only permissions to prevent accidental changes to modules after they're downloaded. This has the unfortunate side-effect of making the cache difficult to delete with commands like `rm -rf`. The cache may instead be deleted with `go clean -modcache`. Alternatively, when the `-modcacherw` flag is used, the `go` command will create new directories with read-write permissions. This increases the risk of editors, tests, and other programs modifying files in the module cache. The `go mod verify` command may be used to detect modifications to dependencies of the main module. It scans the extracted contents of each module dependency and confirms they match the expected hash in `go.sum`.

The table below explains the purpose of most files in the module cache. Some transient files (lock files, temporary directories) are omitted. For each path, `$module` is a module path, and `$version` is a version. Paths ending with slashes (`/`) are directories. Capital letters in module paths and versions are escaped using exclamation points (`Azure` is escaped as `!azure`) to avoid conflicts on case-insensitive file systems.

Path	Description
`$module@$version/`	Directory containing extracted contents of a module `.zip` file. This serves as a module root directory for a downloaded module. It won't contain a `go.mod` file if the original module didn't have one.
`cache/download/`	Directory containing files downloaded from module proxies and files derived from version control systems. The layout of this directory follows the GOPROXY protocol, so this directory may be used as a proxy when served by an HTTP file server or when referenced with a `file://` URL.
`cache/download/$module/@v/list`	List of known versions (see GOPROXY protocol). This may change over time, so the `go` command usually fetches a new copy instead of re-using this file.
`cache/download/$module/@v/$version.info`	JSON metadata about the version. (see GOPROXY protocol). This may

Path	Description
	change over time, so the `go` command usually fetches a new copy instead of re-using this file.
`cache/download/$module/@v/$version.mod`	The `go.mod` file for this version (see GOPROXY protocol). If the original module did not have a `go.mod` file, this is a synthesized file with no requirements.
`cache/download/$module/@v/$version.zip`	The zipped contents of the module (see GOPROXY protocol and Module zip files).
`cache/download/$module/@v/$version.ziphash`	A cryptographic hash of the files in the `.zip` file. Note that the `.zip` file itself is not hashed, so file order, compression, alignment, and metadata don't affect the hash. When using a module, the `go` command verifies this hash matches the corresponding line in `go.sum`. The `go mod verify` command checks that the hashes of module `.zip` files and extracted directories match these files.
`cache/download/sumdb/`	Directory containing files downloaded from a checksum database (typically `sum.golang.org`).
`cache/vcs/`	Contains cloned version control repositories for modules fetched directly from their sources. Directory names are hex-encoded hashes derived from the repository type and URL. Repositories are optimized for size on disk. For example, cloned Git repositories are bare and shallow when possible.

Authenticating modules

When the `go` command downloads a module zip file or `go.mod` file into the module cache, it computes a cryptographic hash and compares it with a known value to verify the file hasn't changed since it was first downloaded. The `go` command reports a security error if a downloaded file does not have the correct hash.

For `go.mod` files, the `go` command computes the hash from the file content. For module zip files, the `go` command computes the hash from the names and contents of files within the archive in a deterministic order. The hash is not affected by file order, compression, alignment, and other metadata. See `golang.org/x/mod/sumdb/dirhash` for hash implementation details.

The `go` command compares each hash with the corresponding line in the main module's `go.sum` file. If the hash is different from the hash in `go.sum`, the `go` command reports a security error and deletes the downloaded file without adding it into the module cache.

If the `go.sum` file is not present, or if it doesn't contain a hash for the downloaded file, the `go` command may verify the hash using the checksum database, a global source of hashes for publicly available modules. Once the hash is verified, the `go` command adds it to `go.sum` and adds the downloaded file in the module cache. If a module is private (matched by the `GOPRIVATE` or `GONOSUMDB` environment variables) or if the checksum database is disabled (by setting `GOSUMDB=off`), the `go` command accepts the hash and adds the file to the module cache without verifying it.

The module cache is usually shared by all Go projects on a system, and each module may have its own `go.sum` file with potentially different hashes. To avoid the need to trust other modules, the `go` command verifies hashes using the main module's `go.sum` whenever it accesses a file in the module cache. Zip file hashes are expensive to compute, so the `go` command checks pre-computed hashes stored alongside zip files instead of re-hashing the files. The `go mod verify` command may be used to check that zip files and extracted directories have not been modified since they were added to the module cache.

go.sum files

A module may have a text file named `go.sum` in its root directory, alongside its `go.mod` file. The `go.sum` file contains cryptographic hashes of the module's direct and indirect dependencies. When the `go` command downloads a module `.mod` or `.zip` file into the module cache, it computes a hash and checks that the hash matches the corresponding hash in the main module's `go.sum` file. `go.sum` may be empty or absent if the module has no dependencies or if all dependencies are replaced with local directories using `replace` directives.

Each line in `go.sum` has three fields separated by spaces: a module path, a version (possibly ending with `/go.mod`), and a hash.

- The module path is the name of the module the hash belongs to.
- The version is the version of the module the hash belongs to. If the version ends with `/go.mod`, the hash is for the module's `go.mod` file only; otherwise, the hash is for the files within the module's `.zip` file.
- The hash column consists of an algorithm name (like `h1`) and a base64-encoded cryptographic hash, separated by a colon (`:`). Currently, SHA-256 (`h1`) is the only supported hash algorithm. If a vulnerability in SHA-256 is discovered in the future, support will be added for another algorithm (named `h2` and so on).

The `go.sum` file may contain hashes for multiple versions of a module. The `go` command may need to load `go.mod` files from multiple versions of a dependency in order to perform minimal version selection. `go.sum` may also contain hashes for module versions that aren't needed anymore (for example, after an upgrade). `go mod tidy` will add missing hashes and will remove unnecessary hashes from `go.sum`.

Checksum database

The checksum database is a global source of `go.sum` lines. The `go` command can use this in many situations to detect misbehavior by proxies or origin servers.

The checksum database allows for global consistency and reliability for all publicly available module versions. It makes untrusted proxies possible since they can't serve the wrong code without it going unnoticed. It also ensures that the bits associated with a specific version do not change from one day to the next, even if the module's author subsequently alters the tags in their repository.

The checksum database is served by sum.golang.org, which is run by Google. It is a Transparent Log (or "Merkle Tree") of `go.sum` line hashes, which is backed by Trillian. The main advantage of a Merkle tree is that independent auditors can verify that it hasn't been tampered with, so it is more trustworthy than a simple database.

The `go` command interacts with the checksum database using the protocol originally outlined in Proposal: Secure the Public Go Module Ecosystem.

The table below specifies queries that the checksum database must respond to. For each path, `$base` is the path portion of the checksum database URL, `$module` is a module path, and `$version` is a version. For example, if the checksum database URL is `https://sum.golang.org`, and the client is requesting the record for the module `golang.org/x/text` at version `v0.3.2`, the client would send a `GET` request for `https://sum.golang.org/lookup/golang.org/x/text@v0.3.2`.

To avoid ambiguity when serving from case-insensitive file systems, the `$module` and `$version` elements are case-encoded by replacing every uppercase letter with an exclamation mark followed by the corresponding lower-case letter. This allows modules `example.com/M` and `example.com/m` to both be stored on disk, since the former is encoded as `example.com/!m`.

Parts of the path surrounded by square brackets, like `[.p/$W]` denote optional values.

Path	Description
`$base/latest`	Returns a signed, encoded tree description for the latest log. This signed description is in the form of a note, which is text that has been signed by one or more server keys and can be verified using the server's public key. The tree description provides the size of the tree and the hash of the tree head at that

Path	Description
	size. This encoding is described in golang.org/x/mod/sumdb/tlog#FormatTree.
$base/lookup/$module@$version	Returns the log record number for the entry about $module at $version, followed by the data for the record (that is, the go.sum lines for $module at $version) and a signed, encoded tree description that contains the record.
$base/tile/$H/$L/$K[.p/$W]	Returns a [log tile](https://research.swtch.com/tlog#serving_tiles), which is a set of hashes that make up a section of the log. Each tile is defined in a two-dimensional coordinate at tile level $L, $Kth from the left, with a tile height of $H. The optional .p/$W suffix indicates a partial log tile with only $W hashes. Clients must fall back to fetching the full tile if a partial tile is not found.
$base/tile/$H/data/$K[.p/$W]	Returns the record data for the leaf hashes in /tile/$H/0/$K[.p/$W] (with a literal data path element).

If the go command consults the checksum database, then the first step is to retrieve the record data through the /lookup endpoint. If the module version is not yet recorded in the log, the checksum database will try to fetch it from the origin server before replying. This /lookup data provides the sum for this module version as well as its position in the log, which informs the client of which tiles should be fetched to perform proofs. The go command performs "inclusion" proofs (that a specific record exists in the log) and "consistency" proofs (that the tree hasn't been tampered with) before adding new go.sum lines to the main module's go.sum file. It's important that the data from /lookup should never be used without first authenticating it against the signed tree hash and authenticating the signed tree hash against the client's timeline of signed tree hashes.

Signed tree hashes and new tiles served by the checksum database are stored in the module cache, so the go command only needs to fetch tiles that are missing.

The go command doesn't need to directly connect to the checksum database. It can request module sums via a module proxy that mirrors the checksum database and supports the protocol above. This can be particularly helpful for private, corporate proxies which block requests outside the organization.

The GOSUMDB environment variable identifies the name of checksum database to use and optionally its public key and URL, as in:

```
GOSUMDB="sum.golang.org"
GOSUMDB="sum.golang.org+<publickey>"
GOSUMDB="sum.golang.org+<publickey> https://sum.golang.org"
```

The `go` command knows the public key of `sum.golang.org`, and also that the name `sum.golang.google.cn` (available inside mainland China) connects to the `sum.golang.org` checksum database; use of any other database requires giving the public key explicitly. The URL defaults to `https://` followed by the database name.

`GOSUMDB` defaults to `sum.golang.org`, the Go checksum database run by Google. See https://sum.golang.org/privacy for the service's privacy policy.

If `GOSUMDB` is set to `off`, or if `go get` is invoked with the `-insecure` flag, the checksum database is not consulted, and all unrecognized modules are accepted, at the cost of giving up the security guarantee of verified repeatable downloads for all modules. A better way to bypass the checksum database for specific modules is to use the `GOPRIVATE` or `GONOSUMDB` environment variables. See Private Modules for details.

The `go env -w` command can be used to set these variables for future `go` command invocations.

Environment variables

Module behavior in the `go` command may be configured using the environment variables listed below. This list only includes module-related environment variables. See `go help environment` for a list of all environment variables recognized by the `go` command.

Variable	Description
`GO111MODULE`	Controls whether the `go` command runs in module-aware mode or `GOPATH` mode. Three values are recognized: • `off`: the `go` command ignores `go.mod` files and runs in `GOPATH` mode. • `on` (or unset): the `go` command runs in module-aware mode, even when no `go.mod` file is present. • `auto`: the `go` command runs in module-aware mode if a `go.mod` file is present in the current directory or any parent directory. In Go 1.15 and lower, this was the default. See Module-aware commands for more information.
`GOMODCACHE`	The directory where the `go` command will store downloaded modules and related files. See Module cache for details on the structure of this directory. If `GOMODCACHE` is not set, it defaults to `$GOPATH/pkg/mod`.
`GOINSECURE`	Comma-separated list of glob patterns (in the syntax of Go's `path.Match`) of module path prefixes that may always be fetched in an insecure manner. Only applies to dependencies that are being fetched directly.

Variable	Description
	Unlike the `-insecure` flag on `go get`, `GOINSECURE` does not disable module checksum database validation. `GOPRIVATE` or `GONOSUMDB` may be used to achieve that.
`GONOPROXY`	Comma-separated list of glob patterns (in the syntax of Go's `path.Match`) of module path prefixes that should always be fetched directly from version control repositories, not from module proxies. If `GONOPROXY` is not set, it defaults to `GOPRIVATE`. See Privacy.
`GONOSUMDB`	Comma-separated list of glob patterns (in the syntax of Go's `path.Match`) of module path prefixes for which the `go` should not verify checksums using the checksum database. If `GONOSUMDB` is not set, it defaults to `GOPRIVATE`. See Privacy.
`GOPATH`	In `GOPATH` mode, the `GOPATH` variable is a list of directories that may contain Go code. In module-aware mode, the module cache is stored in the `pkg/mod` subdirectory of the first `GOPATH` directory. Module source code outside the cache may be stored in any directory. If `GOPATH` is not set, it defaults to the `go` subdirectory of the user's home directory.
`GOPRIVATE`	Comma-separated list of glob patterns (in the syntax of Go's `path.Match`) of module path prefixes that should be considered private. `GOPRIVATE` is a default value for `GONOPROXY` and `GONOSUMDB`. See Privacy. `GOPRIVATE` also determines whether a module is considered private for `GOVCS`.
`GOPROXY`	List of module proxy URLs, separated by commas (`,`) or pipes (`|`). When the `go` command looks up information about a module, it contacts each proxy in the list in sequence until it receives a successful response or a terminal error. A proxy may respond with a 404 (Not Found) or 410 (Gone) status to indicate the module is not available on that server. The `go` command's error fallback behavior is determined by the separator characters between URLs. If a proxy URL is followed by a comma, the `go` command falls back to the next URL after a 404 or 410 error; all other errors are considered terminal. If the proxy URL is followed by a pipe, the `go` command falls back to the next source after any error, including non-HTTP errors like timeouts.

Variable	Description

GOPROXY URLs may have the schemes `https`, `http`, or `file`. If a URL has no scheme, `https` is assumed. A module cache may be used directly as a file proxy:

```
GOPROXY=file://$(go env GOMODCACHE)/cache/download
```

Two keywords may be used in place of proxy URLs:

- `off`: disallows downloading modules from any source.
- `direct`: download directly from version control repositories instead of using a module proxy.

GOPROXY defaults to `https://proxy.golang.org,direct`. Under that configuration, the `go` command first contacts the Go module mirror run by Google, then falls back to a direct connection if the mirror does not have the module. See https://proxy.golang.org/privacy for the mirror's privacy policy. The GOPRIVATE and GONOPROXY environment variables may be set to prevent specific modules from being downloaded using proxies. See Privacy for information on private proxy configuration.

See Module proxies and Resolving a package to a module for more information on how proxies are used.

GOSUMDB

Identifies the name of the checksum database to use and optionally its public key and URL. For example:

```
GOSUMDB="sum.golang.org"
GOSUMDB="sum.golang.org+<publickey>"
GOSUMDB="sum.golang.org+<publickey> https://sum.golang.org"
```

The `go` command knows the public key of `sum.golang.org` and also that the name `sum.golang.google.cn` (available inside mainland China) connects to the `sum.golang.org` database; use of any other database requires giving the public key explicitly. The URL defaults to `https://` followed by the database name.

GOSUMDB defaults to `sum.golang.org`, the Go checksum database run by Google. See https://sum.golang.org/privacy for the service's privacy policy.

If GOSUMDB is set to `off` or if `go get` is invoked with the `-insecure` flag, the checksum database is not consulted, and all unrecognized modules are accepted, at the cost of giving up the security guarantee of verified repeatable downloads for all modules. A better way to bypass the checksum database for specific modules is to use the GOPRIVATE or GONOSUMDB environment variables.

Variable	Description
	See Authenticating modules and Privacy for more information.
GOVCS	Controls the set of version control tools the `go` command may use to download public and private modules (defined by whether their paths match a pattern in `GOPRIVATE`) or other modules matching a glob pattern. If `GOVCS` is not set, or if a module does not match any pattern in `GOVCS`, the `go` command may use `git` and `hg` for a public module, or any known version control tool for a private module. Concretely, the `go` command acts as if `GOVCS` were set to: `public:git|hg,private:all` See Controlling version control tools with `GOVCS` for a complete explanation.
GOWORK	The `GOWORK` environment variable instructs the `go` command to enter workspace mode using the provided [`go.work` file](#go-work-file) to define the workspace. If `GOWORK` is set to `off` workspace mode is disabled. This can be used to run the `go` command in single module mode: for example, `GOWORK=off go build .` builds the `.` package in single-module mode.`If `GOWORK` is empty, the `go` command will search for a `go.work` file as described in the [Workspaces](#workspaces) section.

Glossary

build constraint: A condition that determines whether a Go source file is used when compiling a package. Build constraints may be expressed with file name suffixes (for example, `foo_linux_amd64.go`) or with build constraint comments (for example, `// +build linux,amd64`). See Build Constraints.

build list: The list of module versions that will be used for a build command such as `go build`, `go list`, or `go test`. The build list is determined from the main module's `go.mod` file and `go.mod` files in transitively required modules using minimal version selection. The build list contains versions for all modules in the module graph, not just those relevant to a specific command.

canonical version: A correctly formatted version without a build metadata suffix other than `+incompatible`. For example, `v1.2.3` is a canonical version, but `v1.2.3+meta` is not.

current module: Synonym for main module.

deprecated module: A module that is no longer supported by its authors (though major versions are considered distinct modules for this purpose). A deprecated module is marked with a deprecation comment in the latest version of its `go.mod` file.

direct dependency: A package whose path appears in an `import` declaration in a `.go` source file for a package or test in the main module, or the module containing such a package. (Compare indirect dependency.)

direct mode: A setting of environment variables that causes the `go` command to download a module directly from a version control system, as opposed to a module proxy. `GOPROXY=direct` does this for all modules. `GOPRIVATE` and `GONOPROXY` do this for modules matching a list of patterns.

`go.mod` file: The file that defines a module's path, requirements, and other metadata. Appears in the module's root directory. See the section on `go.mod` files.

`go.work` file The file that defines the set of modules to be used in a workspace. See the section on `go.work` files

import path: A string used to import a package in a Go source file. Synonymous with package path.

indirect dependency: A package transitively imported by a package or test in the main module, but whose path does not appear in any `import` declaration in the main module; or a module that appears in the module graph but does not provide any package directly imported by the main module. (Compare direct dependency.)

lazy module loading: A change in Go 1.17 that avoids loading the module graph for commands that do not need it in modules that specify `go 1.17` or higher. See Lazy module loading.

main module: The module in which the `go` command is invoked. The main module is defined by a `go.mod` file in the current directory or a parent directory. See Modules, packages, and versions.

major version: The first number in a semantic version (`1` in `v1.2.3`). In a release with incompatible changes, the major version must be incremented, and the minor and patch versions must be set to 0. Semantic versions with major version 0 are considered unstable.

major version subdirectory: A subdirectory within a version control repository matching a module's major version suffix where a module may be defined. For example, the module `example.com/mod/v2` in the repository with root path `example.com/mod` may be defined in the repository root directory or the major version subdirectory `v2`. See Module directories within a repository.

major version suffix: A module path suffix that matches the major version number. For example, `/v2` in `example.com/mod/v2`. Major version suffixes are required at `v2.0.0` and later and are not allowed at earlier versions. See the section on Major version suffixes.

minimal version selection (MVS): The algorithm used to determine the versions of all modules that will be used in a build. See the section on Minimal version selection for details.

minor version: The second number in a semantic version (2 in `v1.2.3`). In a release with new, backwards compatible functionality, the minor version must be incremented, and the patch version must be set to 0.

module: A collection of packages that are released, versioned, and distributed together.

module cache: A local directory storing downloaded modules, located in `GOPATH/pkg/mod`. See Module cache.

module graph: The directed graph of module requirements, rooted at the main module. Each vertex in the graph is a module; each edge is a version from a `require` statement in a `go.mod` file (subject to `replace` and `exclude` statements in the main module's `go.mod` file).

module graph pruning: A change in Go 1.17 that reduces the size of the module graph by omitting transitive dependencies of modules that specify `go 1.17` or higher. See Module graph pruning.

module path: A path that identifies a module and acts as a prefix for package import paths within the module. For example, `"golang.org/x/net"`.

module proxy: A web server that implements the GOPROXY protocol. The `go` command downloads version information, `go.mod` files, and module zip files from module proxies.

module root directory: The directory that contains the `go.mod` file that defines a module.

module subdirectory: The portion of a module path after the repository root path that indicates the subdirectory where the module is defined. When non-empty, the module subdirectory is also a prefix for semantic version tags. The module subdirectory does not include the major version suffix, if there is one, even if the module is in a major version subdirectory. See Module paths.

package: A collection of source files in the same directory that are compiled together. See the Packages section in the Go Language Specification.

package path: The path that uniquely identifies a package. A package path is a module path joined with a subdirectory within the module. For example `"golang.org/x/net/html"` is the package path for the package in the module `"golang.org/x/net"` in the `"html"` subdirectory. Synonym of import path.

patch version: The third number in a semantic version (3 in `v1.2.3`). In a release with no changes to the module's public interface, the patch version must be incremented.

pre-release version: A version with a dash followed by a series of dot-separated identifiers immediately following the patch version, for example, `v1.2.3-beta4`. Pre-release versions are considered unstable and are not assumed to be compatible with other versions. A pre-release version sorts before the corresponding release version: `v1.2.3-pre` comes before `v1.2.3`. See also release version.

pseudo-version: A version that encodes a revision identifier (such as a Git commit hash) and a timestamp from a version control system. For example, `v0.0.0-20191109021931-daa7c04131f5`. Used for compatibility with non-module repositories and in other situations when a tagged version is not available.

release version: A version without a pre-release suffix. For example, `v1.2.3`, not `v1.2.3-pre`. See also pre-release version.

repository root path: The portion of a module path that corresponds to a version control repository's root directory. See Module paths.

retracted version: A version that should not be depended upon, either because it was published prematurely or because a severe problem was discovered after it was published. See `retract` directive.

semantic version tag: A tag in a version control repository that maps a version to a specific revision. See Mapping versions to commits.

selected version: The version of a given module chosen by minimal version selection. The selected version is the highest version for the module's path found in the module graph.

vendor directory: A directory named `vendor` that contains packages from other modules needed to build packages in the main module. Maintained with `go mod vendor`. See Vendoring.

version: An identifier for an immutable snapshot of a module, written as the letter `v` followed by a semantic version. See the section on Versions.

workspace: A collection of modules on disk that are used as the main modules when running minimal version selection (MVS). See the section on Workspaces