go_command

- Motivation
- Configuration versus convention
- Go's conventions
- Getting started with the go command
- Limitations
- More information

The Go distribution includes a command, named "`go`", that automates the downloading, building, installation, and testing of Go packages and commands. This document talks about why we wrote a new command, what it is, what it's not, and how to use it.

Motivation

You might have seen early Go talks in which Rob Pike jokes that the idea for Go arose while waiting for a large Google server to compile. That really was the motivation for Go: to build a language that worked well for building the large software that Google writes and runs. It was clear from the start that such a language must provide a way to express dependencies between code libraries clearly, hence the package grouping and the explicit import blocks. It was also clear from the start that you might want arbitrary syntax for describing the code being imported; this is why import paths are string literals.

An explicit goal for Go from the beginning was to be able to build Go code using only the information found in the source itself, not needing to write a makefile or one of the many modern replacements for makefiles. If Go needed a configuration file to explain how to build your program, then Go would have failed.

At first, there was no Go compiler, and the initial development focused on building one and then building libraries for it. For expedience, we postponed the automation of building Go code by using make and writing makefiles. When compiling a single package involved multiple invocations of the Go compiler, we even used a program to write the makefiles for us. You can find it if you dig through the repository history.

The purpose of the new go command is our return to this ideal, that Go programs should compile without configuration or additional effort on the part of the developer beyond writing the necessary import statements.

Configuration versus convention

The way to achieve the simplicity of a configuration-free system is to establish conventions. The system works only to the extent that those conventions are followed. When we first launched Go, many people published packages that had to be installed in certain places, under certain names, using certain build tools, in order to be used. That's understandable: that's the way it works in most other languages. Over the last few years we consistently reminded people about the `goinstall` command (now replaced by `go get`) and its conventions: first, that the import path is derived in a known way from the URL of the source code; second, that the place to store the sources in the local file system is derived in a known way from the import path; third, that each directory in a source tree corresponds to a single package; and fourth, that the package is built using only information in the source code. Today, the vast majority of packages follow these conventions. The Go ecosystem is simpler and more powerful as a result.

We received many requests to allow a makefile in a package directory to provide just a little extra configuration beyond what's in the source code. But that would have introduced new rules. Because we did not accede to such requests, we were able to write the go command and eliminate our use of make or any other build system.

It is important to understand that the go command is not a general build tool. It cannot be configured and it does not attempt to build anything but Go packages. These are important simplifying assumptions: they simplify not only the implementation but also, more important, the use of the tool itself.

Go's conventions

The `go` command requires that code adheres to a few key, well-established conventions.

First, the import path is derived in a known way from the URL of the source code. For Bitbucket, GitHub, Google Code, and Launchpad, the root directory of the repository is identified by the repository's main URL, without the `https://` prefix. Subdirectories are named by adding to that path. For example, the source code of the Google logging package `glog` is obtained by running

git clone https://github.com/golang/glog\

and thus the import path of the glog package is "`github.com/golang/glog`".

These paths are on the long side, but in exchange we get an automatically managed name space for import paths and the ability for a tool like the go command to look at an unfamiliar import path and deduce where to obtain the source code.

Second, the place to store sources in the local file system is derived in a known way from the import path, specifically `$GOPATH/src/<import-path>`. If unset, `$GOPATH` defaults to a subdirectory named `go` in the user's home directory. If `$GOPATH` is set to a list of paths, the go command tries `<dir>/src/<import-path>` for each of the directories in that list.

Each of those trees contains, by convention, a top-level directory named "`bin`", for holding compiled executables, and a top-level directory named "`pkg`", for holding compiled packages that can be imported, and the "`src`" directory, for holding package source files. Imposing this structure lets us keep each of these directory trees self-contained: the compiled form and the sources are always near each other.

These naming conventions also let us work in the reverse direction, from a directory name to its import path. This mapping is important for many of the go command's subcommands, as we'll see below.

Third, each directory in a source tree corresponds to a single package. By restricting a directory to a single package, we don't have to create hybrid import paths that specify first the directory and then the package within that directory. Also, most file management tools and UIs work on directories as fundamental units. Tying the fundamental Go unit—the package—to file system structure means that file system tools become Go package tools. Copying, moving, or deleting a package corresponds to copying, moving, or deleting a directory.

Fourth, each package is built using only the information present in the source files. This makes it much more likely that the tool will be able to adapt to changing build environments and conditions. For example, if we allowed extra configuration such as compiler flags or command line recipes, then that configuration would need to be updated each time the build tools changed; it would also be inherently tied to the use of a specific toolchain.

Getting started with the go command

Finally, a quick tour of how to use the go command. As mentioned above, the default `$GOPATH` on Unix is `$HOME/go`. We'll store our programs there. To use a different location, you can set `$GOPATH`; see How to Write Go Code for details.

We first add some source code. Suppose we want to use the indexing library from the codesearch project along with a left-leaning red-black tree. We can install both with the "`go get`" subcommand:

```
$ go get github.com/google/codesearch/index
$ go get github.com/petar/GoLLRB/llrb
$\
```

Both of these projects are now downloaded and installed into `$HOME/go`, which contains the two directories `src/github.com/google/codesearch/index/` and `src/github.com/petar/GoLLRB/llrb/`, along with the compiled packages (in `pkg/`) for those libraries and their dependencies.

Because we used version control systems (Mercurial and Git) to check out the sources, the source tree also contains the other files in the corresponding repositories, such as related packages. The "`go list`" subcommand lists the import paths corresponding to its arguments, and the pattern "`./...`" means start in the current directory ("`./`") and find all packages below that directory ("`...`"):

```
$ cd $HOME/go/src
$ go list ./...
github.com/google/codesearch/cmd/cgrep
github.com/google/codesearch/cmd/cindex
github.com/google/codesearch/cmd/csearch
github.com/google/codesearch/index
github.com/google/codesearch/regexp
github.com/google/codesearch/sparse
github.com/petar/GoLLRB/example
github.com/petar/GoLLRB/llrb
$\
```

We can also test those packages:

```
$ go test ./...
?       github.com/google/codesearch/cmd/cgrep  [no test files]
?       github.com/google/codesearch/cmd/cindex [no test files]
?       github.com/google/codesearch/cmd/csearch    [no test files]
ok      github.com/google/codesearch/index  0.203s
ok      github.com/google/codesearch/regexp 0.017s
?       github.com/google/codesearch/sparse [no test files]
?       github.com/petar/GoLLRB/example         [no test files]
ok      github.com/petar/GoLLRB/llrb            0.231s
$\
```

If a go subcommand is invoked with no paths listed, it operates on the current directory:

```
$ cd github.com/google/codesearch/regexp
```

```
$ go list
github.com/google/codesearch/regexp
$ go test -v
=== RUN   TestNstateEnc
--- PASS: TestNstateEnc (0.00s)
=== RUN   TestMatch
--- PASS: TestMatch (0.00s)
=== RUN   TestGrep
--- PASS: TestGrep (0.00s)
PASS
ok      github.com/google/codesearch/regexp 0.018s
$ go install
$\
```

That "`go install`" subcommand installs the latest copy of the package into the pkg directory. Because the go command can analyze the dependency graph, "`go install`" also installs any packages that this package imports but that are out of date, recursively.

Notice that "`go install`" was able to determine the name of the import path for the package in the current directory, because of the convention for directory naming. It would be a little more convenient if we could pick the name of the directory where we kept source code, and we probably wouldn't pick such a long name, but that ability would require additional configuration and complexity in the tool. Typing an extra directory name or two is a small price to pay for the increased simplicity and power.

Limitations

As mentioned above, the go command is not a general-purpose build tool. In particular, it does not have any facility for generating Go source files *during* a build, although it does provide `go generate`, which can automate the creation of Go files *before* the build. For more advanced build setups, you may need to write a makefile (or a configuration file for the build tool of your choice) to run whatever tool creates the Go files and then check those generated source files into your repository. This is more work for you, the package author, but it is significantly less work for your users, who can use "`go get`" without needing to obtain and build any additional tools.

More information

For more information, read How to Write Go Code and see the go command documentation.

□□□ #2
□ □□□ □□□ 2026-10-31 13:51:00 UTC
□ □□□ □□□ 2026-10-31 13:51:01 UTC