Get Going With Go

Spring 2023

Contents

1 Introduction

Welcome to CS 138! This guide will get you started with the programming language Go. We'll go over getting setup (which can be a little tricky), organizing your code, formatting it, building and running it, as well as useful plugins for your favorite editors and IDEs.

We **highly** recommend getting set up correctly with Go before beginning on the assignments. Also, while this is a relatively long guide, don't feel like you have to read it all in one sitting. Treat it more like a reference document!

1.1 About Go

Go is an open-source programming language created by a team at Google (and other outside contributors). Go was initially started in 2007 by Robert Griesemer, Rob Pike, and Ken Thompson. Go is a systems language with roots in C, C++, and other languages. Version 1 of Go was released in 2012 and is under active development (v1.17 was released in August 2021). If you have more questions about Go's history there's a wonderful FAQ on their website which we urge you to check out.

2 Installation

To install go on your local machine, you should follow this instruction. Make sure you install version 1.19!

If you're on a Mac and use the Homebrew package manager, you can also simply run

```
brew install go@1.19
```

to install that version (or just `brew install go` to install the latest version).

If you're on Windows, refer to the above link!

IMPORTANT: As of Spring 2023, our codebase requires Go version 1.18+ for support for generics (we use version 1.19). The department machines only have up to Go version 1.17, so we ask that you do not use the department machines to run your projects until the Go version has been updated.

2.1 Go Modules

One of the hardest things for folks new to Go is structuring a Go project, as the Go compiler is very opinionated about it.

In the past, Go required all Go projects on your system to be located under one folder, specified by the GOPATH environment variable. The Go compiler would only build a project if source code was in a folder under `$GOPATH/src`, would install external libraries and dependencies under `$GOPATH/pkg`, and put compiled binaries under `$GOPATH/bin`.

In recent versions of Go, this dependence on GOPATH has been removed thanks to the advent of Go's module systems (also called just Go modules). Further, the community has standardized on one specific project layout structure that minimizes tensions. (Note that GOPATH is still used by Go internally, but is usually just left at its default value — typically `$HOME/go` on Unix systems and `%USERPROFILE%` on Windows).

At a very high level, the Go module system works by declaring your current folder a *module*. Modules can be thought of as virtual environments: they get their own private package management system, and all Go dependencies are contained inside the module.

To create a new project using Go modules, go to any folder on your system.

First, pick a globally unique name[1] for your module. Suppose you want to call your new module `example`.

Next, run `go mod init example`. This will create a `go.mod`.

Let's open up our `go.mod` file:

```
module example

go 1.19
```

Congratulations, you have made a Go module!

Now you can attempt to fetch packages. Try running

```
go install google.golang.org/protobuf/cmd/protoc-gen-go
```

This will install a Go package for you, making it available to your system to use. If you are familiar with `pip` from Python or `npm` from Node.js, it is very similar.

You will note that your `go.mod` has now been updated and looks like:

```
module example

go 1.19

require google.golang.org/protobuf v1.27.1 // indirect
```

Additionally, a file containing checksums for each downloaded package has been created at `go.sum`.

A tutorial on how to work with Go modules in more depth can be found here. In particular, we recommend looking at four aspects of the module system:

- The `go mod tidy` command. This cleans up unused packages and downloads newly added packages to the `go.mod` file (as you can manually edit that if you want to).
- You can control the **package version** of any package just by updating the version indicated in the `require` statement. Versions are somewhat haphazardly baked into the system — version numbers resolve to Git tags on the remote URL for the repository, and you cannot use version numbers with major version starting with 2 (owing to historical legacy). Go version numbers follow https://semver.org/semantic versioning.
- The `go clean -modcache` command. This removes cached packages, useful if you've updated the version for an already downloaded package and want to pull it in.
- The `replace` directive. This is a very powerful feature that allows you to *replace* what a module name gets resolved to. You would use the *replace* directive, for example, to point all imports of `foo` to a local directory. This allows you to transparently shim any package, as a way of trying out changes or behaviour. It is absolutely necessary when working with forks of a repository, as often all the import statements are pointing to the original source and you wish to avoid renaming all the statements (as some imports may depend on the source of the fork transitively, by depending on a package that depends on it).

[1] It is typically a best practice to use the remote URL from which the project can be downloaded e.g. a module might get named `github.com/user/example`. This is important because it allows other Go modules to download your module. For self-contained Go code which will never be downloaded by other modules (which includes all the projects you will be working on in this class), any globally unique name is fine.

3 Code Organization

A repository of Go code can contain one or more packages:

- A package is a collection of related `.go` files, usually implementing one particular algorithm or utility.
- Most packages are designed to be shared, and will be imported into other files. Within packages, you can decide which types and functions are "exported," meaning they will be available to code written outside of this package. In contrast, "unexported" types and functions are only available for use within the package.
- Code in the special package `main` is designed to be compiled into an executable binary, usually a command-line program run by end users.
- Note that a project or repository can have several `main` packages. Each of them may exist in different directories, and will compile to different binaries (named after the directory). This is a common pattern in projects with multiple executables or CLI programs (`client` and `server`, for example.)
- See the *Building and Running* section for information on building packages and binaries.

Additionally, there are some standards on how to structure any Go project:

- The top-level `cmd/` folder is meant to house any `main` packages. You can have multiple unrelated `main` packages in separate subfolders.
- The top-level `pkg/` folder is meant to house packages used by other applications (such as those in `cmd/` or remote code) — that is, public packages. Typically, you may put public library code within this module anyone can download.
- The top-level `internal/` folder is meant to house packages used only within the module (including those in `cmd/`) — that is, private packages. This folder is unique in that the privacy is guaranteed by the compiler — attempts to import code from another repository's `internal` folder will result in an error during compilation.
- Golang developers discourage the use of `src/` folders. More standards are discussed and found here.

Deciding when to put code within `pkg/` or `internal/` depends on your project's usecases and needs. For all of the projects in this course, we have opted to place core code within `pkg/`, on the premise that Go compiler restrictions on package imports are one less thing to worry about as you learn to master Go — real code in a real production setting may choose differently.

Note that in this class, for each project (except for Puddlestore) we provide the stencil code for each runnable binary that you will want to generate. So you don't have to worry too much about the structure of your packages and code, and can focus on implementing the systems we ask you to implement.

4 Learning Go

The best way to pick up the basics of Go is to walk through the Tour of Go.

In particular, look out for the difference between slices and arrays (hint: you should almost always use slices in Go), exported vs. unexported names, channels, and goroutines.

If you prefer a more hands-on approach to learning Go, the Go by Example series is a great set of practical code examples in Go representing some of the language's unique features.

Current TAs of the class have had great results when learning Go by simply looking at the corresponding example when they got to a part of the project code that required a particular feature they hadn't seen before.

4.1 Standard Library

Once you get up-and-running with Go's syntax and actually start writing code, the most comprehensive reference for Go's standard library can be found here.

Every package, along with all of their functions and types are very well documented in these pages, including type signature and a description of what each function and type does.

4.2 Getting Help

Links to the Go forums, mailing list, Slack, and FAQ can be found on the help page.

If you get confused about a particular function you come across in CS138, or if you need to find out how to do something you haven't seen before, you can always search for "golang [your question]". Note that typing "golang" instead of "go" will usually give better results, since a lot of things on the Internet are called "go" (like HBO Go!).

The TAs can also help you with Go questions. We offer regularly scheduled TA office hours throughout the week. Check the calendar page on website for more details.

5 Building and Running

There are a few main commands you should be aware of to build and run Go code:

- `go build`

  When you want to compile a `main` package into an executable (such as compiling your CoinLite binary), navigate to the directory containing the `main` package, and run `go build`. This will compile the binary, link any imports, and place the executable into your current working directory.

  When you want to compile a non-`main` package, navigating to the directory containing the package and running `go build` will compile all the files, link any imports, and exit without any generated files. This is useful to see if your package compiles without errors or warnings.

- `go run`

  When you want to compile and quickly run a `main` package, but not save the resulting binary executable, run it with `go run ./path/to/file.go`, where `file.go` is a file in the `main` package.

- `go install`

  When you want to compile a `main` package and add the generated binary to `$GOPATH/bin` (which still exists even after using Go modules), run `go install /path/to/main/package`. This means you can simply run the binary from anywhere, such as `$ coinlite -mode ... -addr ...`, since the binary can be found from your `$PATH`.

Usually, we recommend using `go install` while working on your projects. This is so that you can compile and generate multiple binaries at once, and use them instantly from anywhere. For instance, in a project that generates a binary for clients and a binary for servers, running `go install ./...` (note the ellipsis) will build every `main` package Go can find in the current directory downwards, and install them into `$GOPATH/bin`. Then you have immediate access from anywhere on your machine to those compiled binaries.

6 Testing and Benchmarking

You can write tests in Go by creating a file that ends with `_test.go`, and writing test functions within these files. Test functions are simply functions that begin with "Test" and take only one parameter, of type `*testing.T`.

You can run all the test functions for the current package using `go test`. Each test function will run once, and give you a chance to programmatically test the rest of your code. Examples of good tests include running different parts of your code, and verifying that the output or side effects from the code are consistent with what you expect.

In a distributed systems context, good tests should include firing up several servers that collectively make up the system, sending queries to this system, and most importantly taking down certain servers to see if your system is degrading as gracefully as you expect.

Note also that it is common practice to make each test file part of the package it is testing. So a file that tests the functionality of the `foo` package should itself be part of the `foo` package. Among other reasons, this is done so that the test file can test both the exported and the unexported functions of the package.

Full details on the Go testing package are available in the package docs.

6.1 Benchmarking

The `testing` library also supports running benchmarks on pieces of code. Similar to regular testing functions, benchmark functions are simply functions that begin with "Benchmark" and take only one parameter, of type `*testing.B`.

To run benchmarks, just run `go test -bench=.`

Benchmarks follow many of the same conventions and rules as regular testing functions. A few important differences:

- The code under performance must be run *multiple* times in a single test execution. The exact number of times is provided by `b.N`. Always wrap the code to measure in a for loop that runs at most `b.N` times.
- A timer begins running from the moment the function begins executing the inner loop. Control over the timer is provided by the `b.ResetTimer`, `b.StopTimer` and `b.StartTimer` methods.
- If code requires expensive setup prior to being run, you can use `b.CleanUp` to run teardown code automatically.

6.2 Checking Test Coverage

Go also provides a tool to check your current test coverage. If you're unfamiliar with the term, test coverage means the percentage of statements in your code that are executed when running the test suite. In general, you want to aim for test coverage around 80% in this class, keeping in mind that some boilerplate code (like printing out help messages when the user types in `help` into the CLI) probably don't need to be tested.

The best way to measure test coverage in Go is to visualize the coverage using the `go test -coverprofile=...` command. This generates HTML files that show visually, using green and red text, which lines of your code were run or not in your test suite. Every line of code that you write yourself (i.e. code not originally in the stencil), should be tested!

Details on the tool and how to use it are covered in this excellent blog post.

Note: Some IDE plugins are quite powerful and show test coverage in your code editor! Make sure to install them.

7 Recommended Tools

7.1 gofmt

Go provides a convenient command-line utility called `gofmt` that formats your code, altering indentation and spacing rules to generate code that fits the Go style guide. However, we recommend running `gofmt` on your code much more often (preferably each time you save) when working on your projects! It's a great, low-effort way of keeping your code looking manageable as you work on it. An easy way to do this is to config your editor to run `gofmt` on save. For example in vscode, this would require you to set `editor.formatOnSave = true`

Note: You **must** run `gofmt` on all your code before you handin. This is part of the style grade for each assignment.

7.2 godoc

Go provides the `godoc` command for quickly browsing package documentation. Running `go doc <package>` can show the documentation available in the source code of packages you have installed (often exactly what is in the online docs.) You can browse by package (`go doc fmt` or `go doc github.com/abiosoft/ishell`), or by a specific symbol (`go doc net.OpError.Temporary` or `go doc github.com/abiosoft/ishell.Shell`). You can even use it for your own code if you leave comments before functions, struct members, or other declarations.

7.3 IDE Plugins

Using plugins for Go for your favorite editor will **greatly** improve your workflow, and as such we **highly** recommend you install them. In particular, plugins exist to automatically run `gofmt` on your code each time you save, show test coverage inline, and lint your code for style. Here are some links:

- Vim: vim-go
- Emacs: go-mode.el
- VSCode: Go for Visual Studio Code
- Atom: go-plus
- Sublime Text 3: GoSublime
- Eclipse: GoClipse
- Notepad++: npp-golang

You can also try GoLand, an IDE from JetBrains, that comes with these tools in-built.

Feedback

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS1380 document by filling out the anonymous feedback form.