Get Going with Go

Spring 2018

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.

Note: 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 (v1.7 was released in August 2016). If you have more questions about Go's history there's a wonderful FAQ[1] on their website which we urge you to check out.

2 Installation

The department machines already have Go installed as a contrib project (v1.11) at `/contrib/bin/go1.11`, but in case you want to use your own machine, you can follow the official installation directions[2]. Note that the department machines also have go 1.7 installed, but this course uses 1.11. If you want to run go commands on the department machine, you need to type go1.11 instead of just go.

If you're on a Mac and use the Homebrew[3] package manager, you can simply run `brew install go` to install the latest version.

If you're on Windows, this blog post[4] by Wade Wegner is a great reference.

2.1 Setting up $GOPATH

The next step in setting up Go, and something students often find tricky, is setting up your `$GOPATH`. The creators of Go are very opinionated, and the approach they recommend is to store all of your Go source code (across all of your different projects) in one directory on your computer, called your `$GOPATH`.

You'll want to choose a directory to act as your `$GOPATH` on each system that you intend to use to work on CS138, including your CS account on the department file system.

If you think you're only going to use Go for CS138, then feel free to choose a CS138-specific directory as your `$GOPATH`, like `~/course/cs138/go`. Otherwise something like `~/go` works great! We recommend putting your path somewhere in your home directory for easy access. Windows users may use something like `C:\Users\<username>\go`.

2.2 Updating Environment Variables

Once you've decided what your `$GOPATH` will be, add the following two lines to a configuration file for the shell you're using. For `bash` (the default on the CS machines as well as on macOS), the file is `~/.bash_profile`.

```
export GOPATH=~/path/to/your/gopath
export PATH=$PATH:$GOPATH/bin
```

[1] `https://golang/doc/faq`
[2] `https://golang.org/doc/install`
[3] `http://brew.sh/`
[4] `http://www.wadewegner.com/2014/12/easy-go-programming-setup-for-windows/`

To see if this worked, open a new Terminal window and type `echo $GOPATH` and confirm that your `$GOPATH` prints to the screen. It's important to verify this, as an incorrect `$GOPATH` will lead to problems down the line.

Windows users will need to open `System > Advanced System Settings > Environment Variables`, then click `New` under `System Variables`, and set `GOPATH` to `C:\path\to\your\gopath`. You can see if this worked by typing `echo %GOPATH%` in your command line.

Now that this is set up, you can download a Go project along with all of its dependencies by simply running `go get github.com/<user>/<proj>`. Try it now with `go get github.com/golang/example/hello`.

This will download the source code of that project from GitHub, and put it in your `$GOPATH` along with any other packages that the project may depend on. If the project includes buildable binaries, running `go get` will also build and install those in `$GOPATH/bin`, which will make them runnable anywhere on your system. Try that by running `hello` in your Terminal. You should see the output `Hello, Go examples!`.

3 Code Organization

The `$GOPATH` contains source code, packages, and binary executables in `$GOPATH/src`, `$GOPATH/pkg`, and `$GOPATH/bin` respectively. You'll only have to edit the files in `$GOPATH/src`, as Go will generate packages and binaries for you when you compile your Go projects.

`$GOPATH/src` contains all your Go source code, and is divided up into folder structures that mirror the URL of the git repository that the code comes from. For instance, the repository `github.com/<user>/<proj>` will get installed to the directory `$GOPATH/src/github.com/<user>/<proj>`.

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.

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[5].

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[6] 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[7].

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[8].

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!).

5 Building and Running

There are three 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), run `go build /path/to/main/package/`. 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, `go build /path/to/package` 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.

[5] `https://tour.golang.org/welcome/1`
[6] `https://gobyexample.com`
[7] `https://golang.org/pkg/`
[8] `https://golang.org/help/`

- `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 `$GOPATH/bin`, run `go install /path/to/main/package`. This means you can simply run the binary from anywhere, such as `$ coinlite -mode ... -addr ...`, since the binary can be found from your `$PATH`.

- `go 1.11`

  Remember, when you use go on the department machines, you need to explicitly refer to go1.11 instead of just go. On the dept machines, go refers to go 1.7, which is not the version this course uses.

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

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[9].

[9] `https://golang.org/pkg/testing/`

6.1 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[10].

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`[11] 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.

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:

[10] `https://blog.golang.org/cover`
[11] `https://golang.org/cmd/gofmt/`

- Vim: vim-go
- Emacs: go-mode.el
- Atom: go-plus
- Sublime Text 3: GoSublime
- Eclipse: GoClipse
- Notepad++: npp-golang

You can also try Gogland, a new IDE from JetBrains (still in beta), that comes with these tools in-built.

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:

`http://cs.brown.edu/courses/cs138/s18/feedback.html`.