Overview

The Workspace proposal aims to improve the workflow of developers who often make changes in multiple modules. If the proposal is accepted, developers no longer need to modify the `go.mod` file with replace directives that ultimately shouldn't be checked in, and to manage sets of replaces for each "root" module of their build. Further, `go.work` provides configuration for tools that intend to operate on multiple modules at the same time, for example `gopls`.

Please read the Workflows section of the Workspace proposal design document for more information about the workflows the proposal aims to address and how it addresses them. If you have a multi-module workflow that you'd like improved, please give feedback on the proposal issue on how the prototype does or does not help.

For those of you who are more visually oriented, here's a video:

Go Workspaces Proposal Demo Video

Setting up the Go command

To try the prototype, download the `gotip` command, which allows you to use alternate versions of the `go` command, including this prototype. Do so by running

```
go install golang.org/dl/gotip@latest
```

if you're on Go 1.16 or later. For older versions of Go, install the tool using the old `go get way`: `go get golang.org/dl/gotip`.

`gotip` needs to be initialized with a version of `go` to run for you. That's done by running `gotip download`. In our case, we'll download the `dev.cmdgo` branch of the Go repository, which holds the source for the workspace prototype:

```
gotip download dev.cmdgo
```

This will download and compile the code for the prototype and make it available from the `gotip` command.

`gotip` can then be used exactly like the go command. For example you can run `gotip build ...` or `gotip run ...`.

The workspace file format

Workspaces are defined using `go.work` files. If your current directory contains a `go.work` file (or it's under a directory that does) the Go command will be put into workspace mode when doing a build (this is similar to how a `go.mod` file defines the scope of a module). When in a workspace, instead of using the closest containing `go.mod` to figure out what module you're in, the `go` command uses the closest containing `go.work` to figure out what *workspace* you're in. All the modules specified in that workspace become your main modules! You can build all of that code from any directory in the workspace.

Let's look at a `go.work` file:

```
go 1.17

directory (
    ./baz // foo.org/bar/baz
    ./tools // golang.org/x/tools
)

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

The syntax is intentionally similar to `go.mod`'s syntax. We have three directives: the `go` directive, the `directory` directive, and the `replace` directive. The `go` and `replace` directives are there for backwards compatibility reasons (see the proposal doc for more details), behave similarly to their counterparts in `go.mod`, and have the same syntax. The `directory` directive allows you to specify which modules you're working in (by the directory containing the `go.mod` file. These are the modules that become the main modules in the workspace.

Setting up `gopls`

You'll need to add a bit of configuration to test out `gopls` with the workspace proposal prototype. You'll need to tell your editor to use an alternate version of Go.

If you've already got VS Code set up with Gopls, You'll need to add the following entry to your settings.json: `{     ...     "go.alternateTools": {         "go": "gotip"     }     ... }` so that it knows to use `gotip` as the go tool.

For other editors see the configuration section of the gopls docs for information for how to set an alternate `go` tool.

Known issues with the prototype

- Replace directives are not yet supported in the `go.work` file. # Giving Feedback

Feel free to post all feedback on the issue #45713. Alternatively, you can send Michael Matloob a slack message, an email at their last name at golang dot org, or send a message at their last name on twitter.