Introduction to Software Development Tooling

Lecture Notes: Build Systems

Lecture 1

Module overview

Welcome to module three! You now know how to use Git to keep track of your code and share it with others, but code isn't very useful unless you can run it! In this module, you'll learn about *build systems*, a type of tool that automates the process of *building* code—that is, converting it into a form that can be run and/or distributed[1]. You can think of build systems as supercharged shell scripts: like shell scripts, their purpose is to encapsulate a complex and intricate sequence of steps within a single command that's easy to remember and is guaranteed to produce the same result each time it's run.

Build systems differ from the shell in a key conceptual way, though: when you run a script, you tell the shell exactly what commands to run and in what order. But when you run a build, you tell the build system only what results you want and let it decide what commands are needed to produce those results. This may seem like a matter of semantics (and to some degree it is—a complex enough shell script could in theory do exactly what a build system does), but it has a huge real-world effect on how efficiently you can work.

To illustrate, consider a complex project like the Linux kernel or Google Chrome, both of which number in the millions of lines of code. If you were to build such a project using a script that simply compiled every source file, you'd have to wait hours to test even the tiniest change, even if that change didn't touch 99% of the source files! With a build system, those hours become seconds because the system knows the purpose of each file it produces and how those files flow into each other. As a result, it can simply skip steps it knows haven't changed!

What problem do build systems solve?

Compiling software is an error-prone and slow process. Writing `g++` every time—whether that be typing it out manually, using the up arrow keys to go through your search history, or using `Ctrl`-`r`—is not the way to go. Build systems automate this process: they let you write the command (or series of commands) once, name this process, and then use that name thereafter. Then you get to check in your build configuration to source control that your teammates may use it too.

Not only is the typing slow, but the actual compilation step is slow as well. Compilers were designed to produce fast-running code, not to compile quickly. Build systems and compilers can work together to help you re-compile only what's needed instead of compiling every file every time. You can specify the relationships between files (e.g. "`main.cpp` uses features from `network.cpp` and `types.h`") and let the compiler figure out on its own what needs to be rebuilt. This kind of incremental build is *much* faster.

Last, engineering projects tend to have much larger scope, more files, and more external dependencies than, for example, introductory course CS projects do. So while you may be able to get away with re-typing the same command and building everything each time for your courses, this might not be the case in a larger project. This could be the difference between a twelve-hour build and a two-minute (or even instant) build.

How does do they solve it?

Build systems tend to operate around once core principle: specify end results and what their component parts are and then let the tooling do the repetitive work. This is called *declarative* programming. You, the programmer, don't need to figure out in what order to go and run a bunch of build commands. The build tool does it for you.

Build systems do this by assembling a *dependency graph* of all the different components of your software project. When you describe that " `fileA` uses features from `fileB` ", the build system infers that any time `fileB` is modified, all the users of it—direct and indirect—must be re-built. Like Git, build systems tend not to watch the filesystem. Instead, they divine what the state of the project is with every command.

Some build systems use modification time (m-time) of files to determine when they were last modified. Some build systems use content hashing, like Git does, to only rebuild when the contents of the file change. (The implication here, which is important, is that m-time might change even if the content does not.)

Build systems can be closely tied to specific languages, or general task runners

"Build systems" is a generic term that refers to any piece of software used to make building software easier. You could, for example, write an elaborate shell script that functions as the build system for a given project. Shell scripting is a generic solution because it's not wed to any particular project language. You could use it to build your C, C++, Rust, etc project. This is the sort of "null example" of a build system because—as you found out—writing shell scripts is hard and you would be hard-pressed to solve a lot of your problems other than repetitive typing.

Other generic build systems include Make, Just, Pants, Bazel, and Buck. They are task runners that build dependency graphs and detect modifications based either on m-time or content changes. They were designed to be generic enough to be used in any project.

Sometimes, though, you might want features from your programming language of choice to be exposed in the build system. For example, the programming language Rust has a notion of software packages called *crates* and its own build system, Cargo, knows how to find and build them. Cargo has been programmed to understand how the Rust compiler works. There is a lot of value in the fact that the Rust ecosystem has unified around one build system and project description language, TOML[2]. The same can be said for JavaScript with NPM/Yarn, Python with pip/setuptools, Go with `go build`, and Haskell with Cabal and Stack—though unfortunately there is an ecosystem split between the two tools.

Why will we focus on Make in this class?

We will focus on Make in this class. Make is a general-purpose build system: while it was designed to build C projects, it has no problem building C++, Java, or any other projects the same way. As such, it is widely used in the real world and available on Linux, BSD, and macOS machines. If you were to pick a hammer for all of your nails, Make is a good choice.

Lecture 2

See the Make documentation which is very thorough and helpful.

Targets and recipes

Targets are the core of a Makefile. They are the results you need to eventually build; the names of the files on disk you want to see by the end of the build process[3]. In the example below, a Makefile has a target called `mybinary`—the name preceding the colon (`:`).

```
mybinary:
    gcc main.c
    mv a.out mybinary
```

It also has *recipes* to build that target: `gcc main.c`. The recipes section consists of a list of shell commands that are run in succession[4] to build the target. In this case, the two recipes— `gcc main.c` and `mv a.out mybinary` —will run one after another.

This is a bit of a silly example since `gcc` has a `-o` option to specify the output filename, but there are certainly situations where multiple shell commands might be necessary.

Dependency relations

Let's look again at the Makefile from above:

```
mybinary:
    gcc main.c
    mv a.out mybinary
```

We can run this once with `make mybinary` and then we will have a `mybinary` on disk. During the course of development, though, we will likely make a change to `main.c` and try to rebuild with `make mybinary`. Make will give an unhelpful response and do nothing:

```
$ make mybinary
gcc main.c
mv a.out mybinary
$ vim main.c
...
$ make mybinary
make: 'mybinary' is up to date.
$
```

This is because there is no way for Make to know about the implicit relationship between `mybinary` and `main.c`. As far as Make is concerned, the contents of the recipes are opaque shell commands; it does not attempt to introspect on them, nor does it have an innate idea of what `gcc` means. You, the programmer, have to specify the relationship manually.

In order to instruct Make to rebuild `mybinary` when `main.c` is modified, add `main.c` as a *dependency* of `mybinary`:

```
mybinary: main.c
    gcc main.c
    mv a.out mybinary
```

Now Make will look at the m-time of both `main.c` and `mybinary`. If `main.c` is newer, it will rebuild `mybinary`.

It is possible to specify more than one dependency per target. If `main.c` also included a header, `myheader.h`, you should add `myheader.h` to the dependencies for `main.c`. Otherwise the binary and the header will be out of sync. For libraries use headers to specify interfaces, this is bad. At best, you might get a wrong answer. At worst, a crash or memory corruption.

Notes on split compilation (object files)

For languages like C that have a notion of split compilation, Make is even more useful. If you have, say, 10 C files that all need to be compiled together, it is possible to run:

```
$ gcc file0.c file1.c ... file9.c -o mybinary
$
```

This will compile each file and then link the results together at the end into `mybinary`.

Unfortunately, this compilation process will throw all of the intermediate results away every time. If you only change `file2.c` and nothing else, you will still end up building all of the other C files again[5]. For this reason, it is possible to compile each file into a corresponding *object file* and then link those together:

```
$ gcc -c file0.c file1.c ... file9.c
$ gcc file0.o file1.o ... file9.o -o mybinary
$
```

Now if you change `file2.c` , you need only re-run `gcc -c file2.c` and the linking step, which together should be much faster than recompiling everything.

It's hard to keep track of this manually, so we can write Make recipes to handle this for us:

```
mybinary: file0.o file1.o file2.o  # and so on
    gcc file0.o file1.o file2.o -o mybinary

file0.o: file0.c
    gcc -c file0.c

file1.o: file1.c
    gcc -c file1.c

file2.o: file2.c
    gcc -c file2.c
```

Now it is possible to modify any one C file and have the binary rebuilt automatically with the least amount of steps.

You will notice that all of this typing is getting cumbersome. We will talk about a solution to this repetition later!

The dependency graph

A DAG, just like Git uses!

```
┌────────────────────┐
│mybinary            │
└┬────────┬────────┬─┘
┌▽──────┐┌▽──────┐┌▽──────┐
│file0.o││file1.o││file2.o│
└┬──────┘└┬──────┘└┬──────┘
┌▽──────┐┌▽──────┐┌▽──────┐
│file0.c││file1.c││file2.c│
└───────┘└───────┘└───────┘
```

```
$ cat Makefile
mybinary: mybinary
        touch mybinary
$ make mybinary
make: Circular mybinary <- mybinary dependency dropped.
$
```

What happens when you type `make`

- Make reads `Makefile`
- determines target(s) to execute
- builds DAG/topo sort

- execute in order (potentially in parallel)

Why is a Makefile better than a shell script?

Using your knowledge from module 1 (CLI), it would possible to compile your projects using a simple shell script that builds every file every time. This is a fine solution, to a point; if the number of files or the compilation time for individual files grows, your build script will get slower over time.

You might get fancy and add some features to compare m-times in your shell script. Maybe you add a function called `build_if_newer` and get 60% of the functionality of Make. This will work. But now you have to reason about an ever-growing shell script and if it perfectly implements your ideal build semantics. And Make will do it better still—Make already has built-in parallelism. Does your shell script?

Make has been around a long time and its performance and correctness are well understood. Its interface is well understood, too; people who know Make are at home in most other projects that use Make. Not so for a custom shell script.

---

1. If you're used to C++, this probably brings to mind compiling your code with `g++`, `clang++`, or the like. And for C++ and other compiled languages, compilation is indeed a core step of any build. But that doesn't mean that there's no such thing as a "build" for other languages or even that compilation is the only important part of a C++ build. There are lots of other things you might need to do to produce a runnable program: you might generate HTML documentation or man pages from Markdown source code; you might compress and bundle resources like graphics and audio; you might remove whitespace from code written in a scripting language to reduce its size. All these things can be part of a build, and with a build system you can ensure they happen perfectly, every time. ↩

2. See Metcalfe's law for more info. ↩

3. Not always; sometimes there are "phony" targets that are names for batches of recipes and do not produce files. ↩

4. They are run in different shell processes, so state like variables and working directory do not persist between commands (`cd` won't work). ↩

5. If you have a bunch of C files that include a header, you *will* want to rebuild them if you change the header. This is because C's `include` mechanism is a textual copy&paste process early in the compilation pipeline. ↩
