CS5220 Advanced Topics in Web Programming

Understand Build and Build Tools

Chengyu Sun
California State University, Los Angeles

Build

Source Code

Metadata

Resources

Libraries

Formats for execution, distribution, and/or deployment:

`EXE, JAR, WAR, ZIP, APK …`

Common Build Tasks

- Preprocessing
- Compilation
- Postprocessing
- Testing
- Packaging
- Distribution
- Deployment

Build Example

- The Code
  - `mean.c`
  - `mean.h`
  - `main.c`
- Adopted from the example at http://www.adp-gmbh.ch/cpp/gcc/create_lib.html

Build Example – Requirements

- Compile `mean.c` into an object file
- Create a static library `libmean.a` and place it under `/lib` folder
- Create a dynamic library `libmean.so` and place it under the `/lib` folder
- Create an executable `main-static` by compiling `main.c` and link it to the static library
- Create an executable `main-dynamic` by compiling `main.c` and link it to the dynamic library
- Remove all generated files (for a clean build)

Build Example – Makefile …

```
all: main-static main-dynamic

mean.o : mean.c
        gcc -fPIC -c mean.c

lib/libmean.a : mean.o
        [ -d lib ] || mkdir lib
        ar rcs lib/libmean.a mean.o

lib/libmean.so : mean.o
        [ -d lib ] || mkdir lib
        gcc -shared -o lib/libmean.so  mean.o
… …
```

… Build Example – Makefile

… …

```
main-static : main.c lib/libmean.a
        gcc main.c -static -o main-static -L lib -lmean

main-dynamic : main.c lib/libmean.so
        gcc main.c -o main-dynamic -L lib -lmean

clean:
        rm -rf lib *.o main-static main-dynamic
```

Understand Makefile

Rule

```
lib/libmean.a : mean.o
        [ -d lib ] || mkdir lib
        ar rcs lib/libmean.a mean.o
```

- **Target**: the name of a file or operation
- **Prerequisites**: the names of files or targets
- **Recipe**: actions to achieve the target (i.e. output) from the prerequisites (i.e. input)

Using Make

```
make [target]
```

- For example:
  - `make`
    - First target is considered the default target (usually named "all")
  - `make main-static`
  - `make clean`

About Make and Makefile

- `Makefile` → *build script*
  - Consists of small "programs" (i.e. recipes) that implement various build tasks (i.e. targets)
- Make → *build tool*
  - Execute the build script intelligently and efficiently

Why Not Just Use an IDE …

- Can your IDE do *everything* you want?
  - Deploy a web application to a remote server
  - Generate source code from some metadata files
  - Create a zip package of selected files for homework submission
  - …

… Why Not Just Use an IDE

- A build tool provides a "*mini programming language*" that can be used to implement any build task
- IDEs with integrated build tools
  - Visual Studio with NMake
  - Eclipse with Maven
  - Android Studio with Gradle

The Main Problem of Make

- Make uses system commands and shell scripting as its “programming language”

Platform-dependent

Ant

- A build tool for Java
  - Platform-independent
  - Designed for Java compiler
- Build file `build.xml` uses XML syntax
- Developed by James Duncan Davidson, who also wrote Tomcat

Ant Build Example: CSNS

```
<project name="csns" basedir="." default="build">
    <target name="init">
        <mkdir dir="${build.dir}" />
        <copy todir="${web.dir}/WEB-INF">
        … …
    </target>
    <target name="build" depends="init">
        <javac srcdir="${src.dir}" destdir="${class.dir}" />
    </target>
    <target name="clean">
        <delete dir="${build.dir}" />
        … …
    </target>
</project>
```

Ant vs. Make

	Ant	Make
Build Task	`<target>`	Target
Build Task Dependency	`depends`	Prerequisites
Basic Action	Tasks like `<javac>`, `<mkdir>`, `<copy>`, and `<delete>`	System commands like `javac`, `mkdir`, `cp`, and `rm`
"Programming Language"	Special tasks like `<parallel>`, `<sequential>`, `<condition>`, and `<retry>`	Shell Script

The Main Problems of Ant (and Make)

- No (library) dependency management
  - Difficult to obtain, keep track of, and update libraries
- Every build is different
  - Difficult to learn a build process

Maven

- A build tool for Java (or a *project management* tool as claimed by its developers)
  - Project Object Model (POM)
  - Project Lifecycles
  - Dependency Management
  - Plugin Framework

A Simple Maven Example

pom.xml

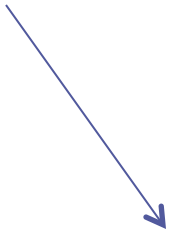
```
<project>
    <modelVersion>4.0.0</modelVersion>
    <groupId>edu.calstatela.cs520</groupId>
    <artifactId>maven-example</artifactId>
    <version>1.0</version>
</project>
```

Run:

```
mvn compile
mvn package
```

pom.xml and modelVersion

- `pom.xml` is a *complete* description of the project
- `modelVersion` is the version of the "language" in which the description is written



Project Object Model (POM)

Maven Coordinates

- `groupId`
  - Name of the company, organization, team etc., usually using the reverse URL naming convention
- `artifactId`
  - A unique name for the project under groupId
- `version`
- `packaging`, default: jar
- `classifier`

Maven coordinates uniquely identifies a project.

Why Not Just Use Project Name

- Rage-quit: Coder unpublished 17 lines of JavaScript and “broke the Internet”

Convention Over Configuration

- Systems, libraries, and frameworks should assume *reasonable defaults*.

See the Effect POM tab of pom.xml in Eclipse for all the "defaults".

Default Directory Structure

- `src/main/java`
- `src/main/resources` for files that should be placed under classpath
- `src/main/webapp` for web applications
- `src/test/java`
- `target`

How To Build a Maven Project

- Q: what happens when you run `mvn compile`?
- A: Maven will go through each phase of the build lifecycle up to the `compile` phase, and run the operations associated with each phase.

Build Lifecycle

- The process for building and distributing a project
- A build lifecycle consists of a number of steps called phases.

http://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html#Lifecycle_Reference

About Lifecycle Phases

- Not all projects utilize all the phases. For a project, most phases can be *empty*, i.e. there are no *operations* associated with them.

Example: `mvn compile`

Phase	Operation(s)
validate	
initialize	
generate-sources	
process-sources	
generate-resources	
process-resources	`resources`
compile	`compile`

Goals and Plugins

- Goals, a.k.a. Mojos, are *operations* provided by Maven plugins

Some Maven Plugins

- resources
- compiler
- surefire
- jar, war

http://maven.apache.org/plugins/index.html

Example of Using a Plugin

```
<build><plugins><plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>2.3.2</version>
    <executions><execution>
            <id>default-compile</id>
            <phase>compile</phase>
            <goals>
              <goal>compile</goal>
            </goals>
            <configuration>
                <target>1.7</target>
            </configuration>
    </execution></executions>
</plugin></plugins></build>
```

About The Plugin Example

- A plugin is uniquely identified by its coordinates just like any other project
- Goals are associated (i.e. *bound*) to a build lifecycle phase
- The behavior of a goal can be customized with additional parameters in the <configuration> section

Run a Maven Build

```
mvn <phase>
```

- Maven will go through each build lifecycle phase up to the specified phase
- In each phase, execute the goals bound to that phase

Run a Maven Build in Eclipse

- Right click on the project then select `Run As → Maven Build …`
- Give the build a name
- Enter the phase name for `Goals`
- Click `Run`

Maven vs. Ant and Make ...

	Ant	Make	Maven
Build Task	`<target>`	Target	??
Build Task Dependency	`depends`	Prerequisites	??
Basic Action	Ant tasks	System commands	??
“Programming Language”	Special Ant tasks	Shell Script	??

… Maven vs. Ant and Make

- Tradeoff between standardization and flexibility
  - Maven: standardized project description (i.e. POM) and build process (i.e. build lifecycles)
  - Ant/Make: “program” any build process
- Dependency Management
  - Maven wins!

Dependency Management

- A dependency of a project is a library that the project depends on
- Adding a dependency to a project is as simple as adding the coordinates of the library to `pom.xml`
- Maven *automatically downloads the library from an online repository* and store it locally for future use

Dependency Example

```
<dependencies>
    <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>javax.servlet-api</artifactId>
        <version>3.0.1</version>
    </dependency>
</dependencies>
```

- Add a dependency to `pom.xml`
- Add a dependency in Eclipse

More About Dependency Management

- Dependencies of a dependency are automatically included
- Dependency conflicts are automatically resolved

Node.js

- Standalone JavaScript with some additional language features and libraries
- Very popular in web development
  - Asynchronous, event-driven design makes it efficient to process high-volume of web requests
  - Single language (i.e. JavaScript) for both backend and frontend

NPM (Node Package Manager)

- A dependency manager for Node.js projects with some build support
- A repository and registry for JavaScript packages

Default package.json

```
{
  "name": "proj",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}
```

Add Dependency to Project

```
npm install <package>
```

- Example: add the `request` dependency to the project

```
"dependencies": {
   "request": "^2.83.0"
 }
```

Semantic Versioning (Semver)

Version number: `X.Y.Z`

- `X` – Major Release: breaks backward compatibility
- `Y` – Minor Release: add new features and doesn't break old ones
- `Z` – Patch release: bugs fixes and minor changes

^ and ~ in Version Numbers

- ^ : can be updated to any future minor release
- ~ : can be updated to any future patch release

Build Support in NPM …

```
{
  "name": "proj",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}
```

… Build Support in NPM …

- NPM has a number of *commands* corresponding to common build tasks like `test`, `start`, `stop`, `pack`, and `publish`
- Similar to Maven build cycle *phases*, each NPM command may have stages (e.g. `pretest`, `test`, `posttest`) in which actions can be performed

… Build Support in NPM

- Like Make, NPM relies on system commands and external tools for basic actions
- For complex build tasks, the choices are either programming them in JavaScript, or use additional build tools like Grunt or Gulp

Readings

- GNU Make - https://www.gnu.org/software/make/
- Ant - http://ant.apache.org/
- Maven - https://maven.apache.org/
- npm - https://www.npmjs.com/