Task Dependencies: ant

Steven J Zeil

February 25, 2013


Outline


ant

ant is a build manager based upon a task dependency graph expressed in an XML file

- **ant** devised by James Davidson of Sun, contributed to Apache project (along with what would eventually become TomCat), released in 2000
- Quickly became a standard tool for Java projects
  - slower to move into other arenas


What's Wrong with make?

ant is actually an acronym for Another Neat Tool.
But why do we need "another" tool for build management?

- **make** works by issuing command to **/bin/sh**
  - That's not portable.
- The commands that people write into their makefile rules are generally not portable either:
  - Commands themselves are system-dependent (e.g., **mkdir**, **cp**, **chmod**
  - Paths are system-dependent (`/` versus `\`, legtal characters, quoting rules)
  - Path lists are system-dependent (`:` versus `;`)


Other Criticisms

- Some feel that **make** is too low-level with tis focus on individual files
  - Some will feel that **ant** is too high-level


Other Criticisms

- Some feel that **make** is too low-level with tis focus on individual files
  - Some will feel that **ant** is too high-level
  - But this is the apparent rationale for moving the focus from file dependencies to task dependencies.

Other Criticisms

- Some feel that **make** is too low-level with tis focus on individual files
  - Some will feel that **ant** is too high-level
  - But this is the apparent rationale for moving the focus from file dependencies to task dependencies.
- The `makefile` syntax is arcane and hard to work with.


Other Criticisms

- Some feel that **make** is too low-level with tis focus on individual files
  - Some will feel that **ant** is too high-level
  - But this is the apparent rationale for moving the focus from file dependencies to task dependencies.
- The `makefile` syntax is arcane and hard to work with.
  - And XML syntax isn't?

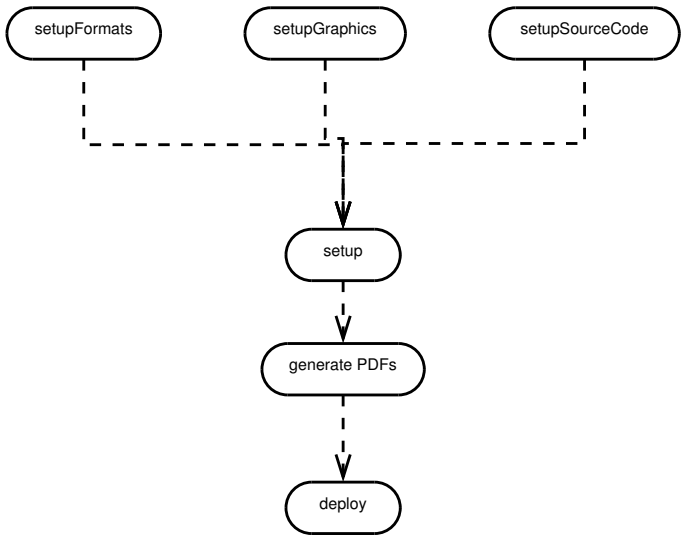

Outline I


ant

- **ant** looks for its instructions in a file named, by default, `build.xml`
- The **ant** command can name any target to be built, e.g.,

```
ant setup
```

- If no target is given, **ant** builds a target explicitly listed in `build.xml` as a default for the project.




ant Options

Some useful options:

-k, -keep-going "Keep going." Don't stop the build at the first failue, but continue building any required targets that do not depend on the one whose construction has failed.

-f *filename* Use *filename* instead of the default `build.xml`. Also `-file` or `-buildfile`

-D*property*=*value* Sets a property (similar to **make**'s variables)


Outline I


Build Files

The commandant build file is an XML file.

- The build file describes a project.
  - The project has a name and a default target.

```
<project name="382Website" default="deploy">
    <description>
        Extract Metadata Extractor - top level
    </description>
  ⋮
</project>
```


Targets

At its heart, a build file is a collection of *targets*.

- A target is an XML element and, as attributes, has a name and, optionally,
  - a list of dependencies
  - a condition
  - a human-readable description
- The target can contain multiple *tasks*, which contain the actual "commands" to get things done.

ant targets correspond, roughly, to **make**'s "artificial targets".


Example of Targets

simplebuild.xml.listing

❶ The project has a name and default target

❷ A basic target. It is named "compile" and has a description (which may be picked up by some IDEs)

❸ This target has 3 tasks. It creates a directory, compiles Java source code, and prints a message when completed.

❹ This target illustrates both a dependency and a condition. The tasks within this target would not be executed if I invoked **ant** like this:

```
ant -Dtest.skip=1
```

However, the `unittest` task would still be considered to have succeeded, in the sense that tasks that depend on it would be allowed to run.


Task versus File Dependencies

ant targets correspond, roughly, to **make**'s "artificial targets".
So this build file
simplebuild.xml.listing is roughly equivalent to this makefile
simplemake.listing though a "real" makefile author would probably write this:
simplemake2.listing


Make Efficiency

If we do

```
make
make
```

The second command does not actually perform any steps.


Ant Efficiency

What happens if we do

```
ant
ant -Dskip.test=1
```

- Each of the tasks *is* executed, but
  - The javac task knows not to re-compile Java files with up-to-date class files
  - The javac task knows not to update Jar files that are newer than all of the files being added.
- So some level of incremental behavior gets built into many of the individual tasks.


Ant Efficiency

What happens if we do

```
ant
ant -Dskip.test=1
```

- Each of the tasks *is* executed, but
  - The `javac` task knows not to re-compile Java files with up-to-date class files
  - The `javac` task knows not to update Jar files that are newer than all of the files being added.
- So some level of incremental behavior gets built into many of the individual tasks.
- If we remove the `-Dskip.test=1`, however, the tests will be re-run.


Properties

Properties are named string values.

- Can be set from the command line or via a `<property` and a few other tasks
- Accessed as `${`*`propertyName`*`}`
- Properties are immutable: once set, attempts to re-assign their values are ignored
- By convention, properties names are grouped into faux hierarchies with '.'
  - e.g., `compile.src`, `compile.dest`, `compile.options`

The <property Task

Two basic modes:

- `<property name="compile.options" value="-g -O1"/>`
  Sets this property to "-g -O1"
- `<property name="compile.src" location="src/main/java"/>`
  Sets this property to the *absolute path* to the directory/file named.
- The / and \ characters are changed as necessary to conform to the OS on which **ant** is being run


Additional <property Variants

- `<property file="project.default.properties"/>`
  Loads property values from a file, written as a series of *property*=*value* lines

```
courseName=CS795
baseurl=https://secweb.cs.odu.edu/~zeil/cs795SD/s13
homeurl=https://secweb.cs.odu.edu/~zeil/cs795SD/s13/
email=zeil@cs.odu.edu
```

- `<property environment="env"/>`
  Copies the OS environment variables into the build state, prefaced by the indicated prefix
  - e.g., `${env.PATH}`


File Sets and Lists

- A file set is a collection of existing files
  - can be specified using wild cards
- A file list is a collection of files that may or may not exist
  - Must be specified explicitly without wild cards


File Sets

```
<fileset file="src/main.cpp"/>
<fileset dir="src"
    includes="main.cpp utility.h utility.cpp"/>
<fileset dir="src" includes="*.cpp,*.h"/>
```

- More commonly seen as a nested form

```
<fileset id="unitTests" dir="bin">
   <include name="**/Test*.class"/>
   <exclude name="**/*$*.class"/>
</fileset>
```

- The `id` in the prior example allows later references:

```
<fileset refid="unitTests"/>
```


File Lists

```
<filelist dir="src"
    files="main.cpp utilities.h utilities.cpp"/>
```

- Can also use `id` or `refid` attributes


Mappers

- Allow for a transformation of file names
- Some commands use a file set to describe inputs, then a mapper to describe outputs

```
<listset dir="src" includes="*.cpp"/>
<globmapper from="*.cpp" to="*.o"/>
```

would map each file in `src/*.cpp` to a corresponding `.o` file

```
<listset dir="bin" includes="**/Test*.java"/>
<packagemapper from="*.class" to="*"/>
```

would map a compiled unit test file `project/package/TestADT.class` to `project.package.TestADT`

- There are several other mappers as well


Selectors

Selectors provide more options for selecting file than simple include/exclude based on the names.

```
<fileset id="unitTestSrc" dir="src">
   <include name="**/Test*.java"/>
   <contains text="@Test" casesensitive="no"/>
</fileset>
```

(Our previous examples assumed that unit tests woule be identified by file name. Here we look instead for the JUnit4 @Test annotation.)

- Other selectors replicate several of the tests from the classic Unix **find** command


Path Sets

Used to specify a sequence of paths, usually to be searched.

```
<classpath>
    <pathelement path="${env.CLASSPATH}"/>
    <fileset dir="target/classes">
       <include name="**/*.class"/>
    </fileset>
    <filelist refid="third-party_jars"/>
</classpath>
```


Referencing Path Sets

- For reason unclear to me, you cannot name classpaths and re-use them directly, but must do it this way

```
<path name="test.compile.classpath">
    <pathelement path="${env.CLASSPATH}"/>
    <fileset dir="target/classes">
       <include name="**/*.class"/>
    </fileset>
    <filelist refid="third-party_jars"/>
</path>
   ⋮
<classpath refid="test.compile.classpath"/>
```


Filters

Filters are used to modify the outputs of some commands by performing various substitutions:

```
<copy file="../../templates/@{format}.tex"
      tofile="${doc}-@{format}.ltx">
  <filterset>
    <filter token="doc" value="${doc}"/>
    <filter token="relPath" value="${relPath}"/>
    <filter token="format" value="@{format}"/>
  </filterset>
</copy>
```

A filter set replaces tokens like @doc@ by a string, in this case the value of the property ${doc}


Filter Chains

Filter chains offer a variety of more powerful options, e.g.,

```
<loadfile property="doctitle" srcfile="${doc}.info.tex">
  <filterchain>
    <linecontains>
      <contains value="\title{"/>
    </linecontains>
    <tokenfilter>
      <replaceregex pattern=" *\\title[{]([^}]*)[}]"
                    replace="\1"/>
    </tokenfilter>
  </filterchain>
</loadfile>
```

- `loadfile` loads an entire file into a property
- The filter extracts the contents of a LaTeX `\title{...}` command


Tasks

The Ant Manual has a good breakdown on these.

- Consistent with their XML structure, tasks can be parameterized via attributes or nested XML attributes
  - Sometimes you can do the same thing either way.
- Look at:
  - File tasks: copy, delete, mkdir, move, fixcrlf, sync
  - Compile tasks: javac, depend
  - Archive, documentation, testing tasks
  - Execution tasks: java, exec, apply


Extending Ant

- Ant has a built-in macro capability
- More powerful extension is accomplished by adding Java classes, mapped onto task names:

```
<project name="code2html" default="build">

  <taskdef classpath="JFlex.jar"
           classname="JFlex.anttask.JFlexTask"
           name="jflex" />
     ⋮

  <target name="generateSource">
    <mkdir dir="src/main/java"/>
    <jflex file="src/main/jflex/code2html.flex"
           destdir="src/main/java"/>
    <jflex file="src/main/jflex/code2tex.flex"
                destdir="src/main/java"/>
       ⋮
```


Finding Extensions

- Many Java-oriented tools (e.g. JFlex) come with an ant task as part of the package.
- Other are contributed by users of the tool, (e.g. LaTeX)
- Some general-purpose Ant libraries.
  e.g., antcontrib adds
  - C/C++ compilation
  - If and For-loop
  - outofdate (a make-like file dependency wrapper)
  - enhanced property tasks (e.g., URL encoding)


Outline I


Code Annotation Tool

The steps involved in building this tool are:

1. Run the program **jflex** on each file in `src/main/jflex`, generating a pair of .java files that get placed in `src/main/java`
2. Compile the Java files in `src/main/java`, placing the results in `target/classes`
3. Compile the Java files in `src/test/java` (using the `target/classes` compilation results, placing the results in `target/test-classes`.
4. Run the JUnit tests in `target/test-classes`.
5. If all tests pass, package the compiled classes in `target/classes` into a .jar file.


The Code Annotation Tool Build I

codeAnnotation.build.listing
This is a fairly "stock" build for a Java project.
Nonetheless, there are multiple steps that would not be handled by typical IDE build managers.

1. Not all tasks need to be within targets.
   Properties are usually not, but this is an example of a more "active" task - it copies the **ant** output into a log file.
2. Establishes the `<jflex` tag, provided in `JFlex.jar`
3. Includes an XML file of additional **ant** commands.
   - The name of the file loaded includes the operating system name `${os.name}`, so this allows for customization.
     build-Linux.paths.listing
     build-Windows7.paths.listing
   - **ant** has various tasks for including other files into a build


The Code Annotation Tool Build II

- `loadfile`: loads properties in plain-text form
- `include`: shown here, loads XML **ant** instructions
- `import`: Similar to `include`, but `import` allows the imported targets to be overridden by the importer. `include` does not The manual page for `import` has a good example showing the difference.

❹ In this target we use the `jflex` tag which was loaded as an extension earlier.
This creates several Java files in `src/main/java`.

❺ All Java source in `src/main/java` is compiled, placing the resulting `.class` files in `target/classes`

- Note the classpath, which was included from our OS-dependent path files.

❻ All Java source in `src/test/java` is compiled, placing the resulting `.class` files in `target/test-classes`


The Code Annotation Tool Build III

- The destination directory is distinct to make it easier to later package up the "real" deliverable classes, omitting the test drivers.
- The classpath here is different, because it must include both the code we have already compiled and the JUnit package.

❼ Run the tests and generate a basic summary report.

❽ Package up the application into a Jar file, selecting one of the 4 executables as the default to be run when the jar file is lauchned by double-clicking or via `java -jar`

❾ A typical clean-up task,

- Note that this target does not depend on the others. It is intended to be invoked separately.
- The task is made simpler be keeping the compilation binaries in a separate directory.


The Code Annotation Tool Build IV

- In larger projects, I might have done the same with the JFlex-generated sources, so that they would be cleaned as well.


Managing Subprojects with ant

Typical key ideas:

- Gather common structures into a build file that can be included or imported by each subproject.
- Create a top-level build file that
  - Performs project-wide initialization
  - Distributes common targets to individual subproject builds


A Top-Level Build I

topProject-build.listing

❶ A macro declaration

- This macro takes a parameter, "target"
- It uses the task `subant` to invoke **ant** recursively on that target (`@target`) for each `build.xml` file that it finds in a subdirectory (at any depth)
  - except for subdirectories of `templates`

❷ This task is used for simulate **make**-like file dependencies. It deletes the target files if any of them are older than any of the source files.

- Still a bit more coarse-grained than **make**

❸ Sets a property to true/false depending on whether a file exists.

- The file is the same one used as the target of the prior `dependset`


A Top-Level Build II

❹ Part of the project-wide setup, skipped if the earlier `dependset` decided that the target file was already up-to-date.

❺ The main target for project-wide setup.

❻ The "build" target is issued to individual sub-projects using the earlier macro.

- This is the default target.
- Once we are satisfied with a build, we can issue a new **ant** command to perform either of the next two options.

❼ After all the builds are done, we could build a zip file of the results.

- The date and time of the build is included in the zip file name.

❽ Or, instead of the zip file, we might sync the results with another directory (a website).

Extend or Exec?

As we move further from common Java project structures, the problem arises of how to issue commands for which we have no readily available **ant** task.

- Write our own task as a java class


Extend or Exec?

As we move further from common Java project structures, the problem arises of how to issue commands for which we have no readily available **ant** task.

- Write our own task as a java class
  - A lot of work, particularly for a one-off


Extend or Exec?

As we move further from common Java project structures, the problem arises of how to issue commands for which we have no readily available **ant** task.

- Write our own task as a java class
  - A lot of work, particularly for a one-off
- If the desired command is actually a Java program, use the `java` task to launch it.


Extend or Exec?

As we move further from common Java project structures, the problem arises of how to issue commands for which we have no readily available **ant** task.

- Write our own task as a java class
  - A lot of work, particularly for a one-off
- If the desired command is actually a Java program, use the `java` task to launch it.
  - Can be reasonably portable


Extend or Exec?

As we move further from common Java project structures, the problem arises of how to issue commands for which we have no readily available **ant** task.

- Write our own task as a java class
  - A lot of work, particularly for a one-off
- If the desired command is actually a Java program, use the `java` task to launch it.
  - Can be reasonably portable
  - But some **ant** purists still find this objectionable


Extend or Exec?

As we move further from common Java project structures, the problem arises of how to issue commands for which we have no readily available **ant** task.

- Write our own task as a java class
  - A lot of work, particularly for a one-off
- If the desired command is actually a Java program, use the `java` task to launch it.
  - Can be reasonably portable
  - But some **ant** purists still find this objectionable
- Use the `exec` or `apply` tasks to simply run a command or non-Java program


Extend or Exec?

As we move further from common Java project structures, the problem arises of how to issue commands for which we have no readily available **ant** task.

- Write our own task as a java class
  - A lot of work, particularly for a one-off
- If the desired command is actually a Java program, use the `java` task to launch it.
  - Can be reasonably portable
  - But some **ant** purists still find this objectionable
- Use the `exec` or `apply` tasks to simply run a command or non-Java program
  - Portability becomes much more of an issue


Mitigating Factors

Although the community has contributed numerous extension tasks,

- Some have steep learning curves and/or tricky setup
  - e.g., `cc` task from ant-contrib
- Others may be incomplete or buggy

This can drive a project to use `exec`/`apply` even if **tasks** purportedly exist


Generating Course Website PDFs

Each document is a subproject with a build file like this:

```
<project name="docs" default="build">

  <import file="../../commonBuild.xml"/>

  <target name="makeSources" depends="properties">
    <docformat format="slides"/>
    <docformat format="web"/>
    <docformat format="printable"/>
    <docformat format="10-4x3"/>
    <docformat format="10-16x10"/>
    <docformat format="7-4x3"/>
    <docformat format="7-16x10"/>
  </target>
```

Outline I


Limitations of Eclipse Builder

- Cannot run code-proprocessing (e.g., JFlex)
- An Eclipse project is oriented towards producing a single output product (program, library, . . .)
  - With C++ projects, a problem if you have a "real" product (e.g., a library) and a set of test drivers, each of which yields a distinct program executable.
  - Java projects have fewer problems (because executables don't need separate processing), but what it you are planning to generate both
    - a binary distribution jar, and
    - a source distribution jar


Project Dependencies

Eclipse supports the idea of projects that depend on other projects, so you could do

- project1 produces the binary distribution jar


Project Dependencies

Eclipse supports the idea of projects that depend on other projects, so you could do

- project1 produces the binary distribution jar
- project2 depends on project1 and produces a source distribution jar
  - These projects must reside together in a known relative path from one another
  - project2 is not automatically rebuild if project1 has changed


Project Dependencies

Eclipse supports the idea of projects that depend on other projects, so you could do

- project1 produces the binary distribution jar
- project2 depends on project1 and produces a source distribution jar
  - These projects must reside together in a known relative path from one another
  - project2 is not automatically rebuild if project1 has changed
- Does not scale well.
  - For C++ are you going to have a distinct project for each test driver?


Eclipse/Ant Integration

Eclipse is generally **ant**-friendly.

- Drop a `build.xml` file into a project and Eclipse will recognize it.
  - Right-clicking on it will bring up options to run it, or to configure how to run it
    - including the selection of the target
    - some preference given to targets with descriptions
- Once **ant** has been run, the "Run Last Tool" button defaults to re-running it.
- But the default build is still Eclipse's default build manager
  - For projects with elaborate classpaths, requires keeping both the Eclipse project description and the build file up-to-date and consistent.
  - Pre-compilation steps (e.g., tools that generate source code) are not re-run automatically when needed.


Eclipse Builders

Eclipse supports mutliple, plug-able *builders*.

- Open Project Properties and go to "Builders"
  - In a typical java project, you have just the "Java Builder"
  - Click new to see options.
    In this case, select "Ant Builder".
  - Fill in the main screen. Leave "Arguments" blank.
  - Go to the Targets tab. Select appropriate targets for
    Clean: Menu selection `Project->clean`
    Manual build: What you want done after explicitly requesting a build
    Auto build: What you want done after a file has been saved/changed
- Return to the Builders list and uncheck the "Java Builder"