Comparing ant and make as Java build tools
(Technology)

By GusherJizmac
Mon Sep 24th, 2001 at 08:44:18 PM EST

Recently, the tool ant has come to the forefront as a build tool for Java. It uses XML build files and contains many built-in commands to make it easy to work with Java code and Java projects. There are extensions to it to simplify developing web-based Java applications with tools like BEA Weblogic. Upon hearing about ant, many developers might first ask "Why not just use make?". This author has extensive experience with make, and with Java development, and the following article attempts to answer this question by offering a comparison of ant and make as build tools, under the assumption that Java applications are being developed. The conclusion drawn is that while ant has some powerful advantages, it is not as easily extensible, nor as flexible as make for most development situations and there isn't a strong argument in favor of using it.

Introduction

When my company first began serious Java application development (in the form of web-based, J2EE applications), my instinct was to create a Makefile system to handle the compilation of Java code, as well as some basic development environment setup procedures. This was done and included in our standard development procedure. Then, a while ago, I began reading about ant, which purported to be a better tool for automating builds in a Java environment. After reading about it, I decided to implement a simple J2EE application (using Tomcat and Oracle) using ant, instead of my company's Makefile system. This article documents my experience and impressions about using ant as a Java development tool. First, however, I provide a simple overview of the two tools.

make

If you aren't familiar with make (wherby I mean specifically GNU Make), it is a UNIX-based tool that reads a Makefile, which is a set of targets (with dependants) that get "made" when a target is out of date. What happens when a target is "made" is entirely up to the Makefile author. Usually, this means compiling source code into object or executable code, but it could be anything, including checkout of resources from source control or creating distributable archives. What happens during a make is typically calls to various shell commands. Since make originally was a UNIX program, make was designed assuming the full range of UNIX commands.

Make is very flexible, in that you have a wide latitude to define variables, manipulate those variables and pass them to any command available on your system. A simple Makefile might look like this:
FILE_BASE = hello
$(FILE_BASE) : $(FILE_BASE).o
gcc -o $(FILE_BASE) $(FILE_BASE).o

$(FILE_BASE).o : $(FILE_BASE).c
gcc -c $(FILE_BASE).c

Here, hello is a target, with hello.o being a dependant. hello.o is also a target itself, with hello.c as its dependant. In most cases, make assumes that targets and dependants are files. So, in this example, if hello has an earlier modification date than hello.o, it will run the associated command (namely gcc -o hello hello.o), which creates hello and brings it up to date. So, if you had a whole slew of source files, make allows you to only recompile those that changed, not the whole tree. For more about what you can do with make, see the links at the end of the article.

ant

ant is part of the Apache Jakarta project. Ant has a similar concept to make in terms of targets and dependants, but ant doesn't assume that targets and dependants are files by default. Ant also doesn't assume that the purpose of dependants are to bring targets up to date, it's merely a simple dependancy. Ant also makes use of built-in commands to implement targets, rather than assuming shell commands are available (although you can execute arbitrary shell commands). This makes ant run on any Java-supported platform. Many of the commands are specific to compiling or working with Java code. Since ant is written in Java, it has the added feature of running the Java compiler from within itself, which ends up making compilation quite fast. Ant relies on this fact, even, because the typical compilation target results in compiling every single java source it can find each time you run it. For users of make, this may sound horrendous, but the compilation step (for me) took the same amount of time to compile one class as it did for 50.

Ant also uses an XML-based syntax for creating the buildfiles (although it is not strict XML in the sense that a DTD for all of ant cannot easily be made, and the syntax allows for a lot more flexibility than you might expect in an XML file). Following is an example:

```xml
<project name="foobar" default="compile" basedir=".">
  <description>Foobar!</description>
  <!-- set global properties for this build -->
  <property name="src" value="."/>
  <property name="build" value="site/foobar/war_file/WEB-INF/classes"/>
  <target name="init">
    <!-- Create the time stamp -->
    <tstamp/>
    <!-- Create the build directory structure used by compile -->
    <mkdir dir="${build}"/>
  </target>
  <target name="compile" depends="init">
    <!-- Compile the java code from ${src} into ${build} -->
    <javac debug="on" srcdir="${src}" destdir="${build}"/>
    <classpath>
      <pathelement path="${classpath}"/>
      <fileset dir="."/>
    </classpath>
  </target>
</project>
```
Here, compile is a target (and the default target), and it compiles the java source code. It is dependant on the init target, which ensures that the build directory exists, as well as creating a timestamp. Note that here, everytime that the compile target is executed, the init target will also be executed. These aren't files with timestamps that may become out of date; they are simply chained events that occur in a certain order. I should note that you can conditionally execute the commands of a target, based on execute-time criteria, much as you can in make. Additionally, you can write Java classes to create your own built-in commands, or you can use a special built-in command that lets you run an arbitrary command-line command.

Evaluation Method

I have many years of experience with GNU make, and worked on our company's Makefile system. I'm familiar with makes pros and cons, especially with respect to Java development, which I've been doing steadily for the past 3 years. This is obviously going to make me a little biased against ant, because I know much more about make. However, my first experience with make was to create platform independant (UNIX, Windows, OS/2) build-files for creating shared libraries, so I remember what it was like to have to learn a weird tool and get it to work in a general sense. Also, creating shared libraries (at least at the time) was not the trivial task that compilation is, and so I needed to learn make quite well to get it working right. My hope was that this experience would allow me to be objective with respect to ant.

I downloaded and installed ant, and set it up as my compilation tool for my project. Our company's Makefile system does the following things for the average developer:

- Maintain a list of 3rd party jars needed for the project.
- Checkout those jars if they are not checked out
- Compile any Java classes changed since the last compile
- "clean" functionality, which deletes all generated or checked-out files so you can do a "clean" build

Deployment of jars and classes (to the developer's Tomcat instance, not to a production server) is done by manipulating the arguments of the compilation command, so that the classes end up where tomcat is expecting them. My goal with ant was to create a buildfile that gave me this functionality. Doing so was very straightforward, save for configuring it to check out my jars only if they didn't exist. This required some more hunting and experimenting. Specifics on this appear below.

Once I had my ant file, I was off and developing and didn't really notice much of a difference between using make. With both systems I simply type a 'make' or 'ant' and my code compiles. The first time I build, my jars get checked out.

Comparison of the two Tools
My comparison will take two parts: The first will focus on the differences with using each tool for my specific task, namely writing a Java-based web application. The second part will discuss my more general impressions of ant vs. make as build tools, apart from my specific project (but still within the realm of writing Java applications).

**Using ant on my J2EE Web Application**

Boy does ant compile fast! The biggest difference I noticed was the speed at which ant compiled. I was quite distressed while creating my buildfile that ant would be compiling all the code every time, but after the first compile, I had to change to by output directory to verify that the classes were actually there, because I couldn't believe it had compiled all those sources so quickly. From a pure user perspective, however, ant didn't integrate with vim out of the box. With make (more specifically javac), you can have vim run your build and then step you to each line with syntax errors, because vim can parse the output of javac. Since ant reformats javac's output, vim wasn't able to do this. I assume that some tweaking with vim could alleviate this, but I didn't look into it. The reformatting (which amounts to indenting javac's output and prepending each line with a flag indicating that the output came from javac) made it harder to read the output of javac (probably because I was used to reading it direct from javac). I'm not sure if there is an option to disable this, but I couldn't find it.

I did have a somewhat difficult time getting ant to checkout my jar files only if they didn't exist. It required using the "uptodate" builtin command and creating a timestamped file. Perhaps there is a better way to do it, but the documentation wasn't clear. Accomplishing this in make was quite simple. I could make a target named for each jar (make allows for creating make-time targets dynamically, wheras ant can only have static targets defined at buildfile-creation), and then the command for that target is to check it out of CVS. The effect is that the checkout only happens when the file doesn't exist.

**Using ant to make a full-blown development system**

My company's Makefile system supports (or will support) additional functionality, including tagging source files, committing class files for distribution, running checks on source files before tagging, and other things needed for a sophisticated development environment. Implementing these is fairly straightforward, because make and shell programming is very similar to LISP programming, where code and data are interchangeable, and there is a high degree of dynamicism. This allows for a general-purpose Makefile that needs to know little or nothing about the files on which it operates. My checkout example above indicates what I mean. A better example is if I want to make a "tag" target, that tags all source files as "stage", so they can be pushed to a staging environment. Before tagging, I need to make sure that each file has been committed to CVS. I can't imagine how to do this with ant without writing a builtin command or shell script (or, worse yet, hard-coding the java source file names in the buildfile). I'm sure it can be done, but in make, it is very simple. Since you can construct targets and commands dynamically at make-time, I can write a generic make target that does this check and fails if any file isn't committed to CVS. Being able to do this easily is, in my opinion, is a
pretty basic feature that a build system should have. Ant could support this with a more robust variable assignment mechanism and with a way to create dynamic targets, or a way to treat property values as lists for iteration.

Integrating ant with other systems also seems to be quite cumbersome. You basically have two choices to integrate ant with other systems (e.g. clear case if you don't use CVS): writing a java class, or using the <Exec...> builtin command. Writing a Java class seems quite cumbersome, and often Java is not suited to executing system-level tasks involved with running command-line tools typically provided by configuration management tool vendors. Using the builtin command is also somewhat cumbersome, as it is not as flexible as just running the command through a shell. You are basically specifying all the parameters to pass to the java method that executes external commands, and as mentioned before, this is not always appropriate for executing command line tasks.

Even if this is done (as many have done with providing builtin commands for BEA Weblogic), your commands won't work if the interface changes. For example, BEA radically changed their EJB deployment procedure between version 4.5 and 5.1. An ant builtin task would have to be rewritten, recompiled and redistributed to accommodate this. Another example is the change between javadoc for java 1.1 and for java2. If such a change occurs again, ant will not easily be able to take advantage of the better featureset. This is because with ant, you would have to reimplement your builtin task. With make, you can easily change your Makefile.

Additionally, ant doesn't have nearly the documentation, developer knowledge, or proven value that make has. I know that whatever needs my organization will have (including developing non-Java applications), make will be able to support them. I cannot say the same for ant.

That said, ant has some good things going for it. For one, if you are a developer that knows neither make nor ant, you will be up and compiling javacode a lot faster with ant. ant essentially provides a builtin Makefile for compiling and running Java code, and that's pretty good. Make was designed with compiling C code in mind, and getting it to work with Java requires some non-obvious make coding that does require some prior knowledge of make to do. Additionally, ant compiles incredibly fast. While I didn't try ant on a real sized project (e.g. 1000s of classes), for my one-person development effort, it was much faster than make (of course, you could have make call ant for compiling). Finally, ant is truly cross-platform. Ant will run on Max OS 9 and VMS. Make will not (without additional software installed). In make's favor, I should point out that Cygwin provides free versions of make and all other GNU tools for windows, including bash, and that the vast amount of Java coders in the world are using either UNIX or Windows. So, for most Java coders, make is platform independent (although it does require an extra package for Windows).

**Summary**

OK, I don't want to fill this article up with anecdotal evidence or, worse yet, ranting. So, I'll try to capture my thoughts in the following pros/cons table:
• ant

**Pros**
- Fast Compiles.
- Easy to get up and running for compilation.
- True platform independance.
- Good error reporting of buildfile syntax errors.

**Cons**
- Property value assignment very simple and inflexible.
- Cannot create dynamic targets at build-time, making complex build tasks difficult to write.
- Compiling every source file every time might be a problem for large systems.
- Integration with other systems limited to Java classes or executing command line through Java.
- Wrapping commands in "builtin tasks" exposes them to breakage when underlying tasks change.
- Sparse documentation, not widely used
- Use for non-Java projects not realistic without more builtin commands.

• make

**Pros**
- Incredibly flexible
- Run-time dynamicism simplifies creating complex tasks
- Powerful variable creation mechanism simplifies creating complex tasks
- Integration with other build and deployment tools is very simple through shell commands
- Assumes targets and dependants are files, which makes it easy to write targets that only do the work that is needed
- Extensive documentation, developer knowledge and proven track record as industry standard build tool
- Can be used for any type of project, even projects that mix languages and technologies.

**Cons**
- Requires UNIX command set installed to be effective.
- Requires some non-beginner level make coding to work effectively with Java compiler.
- Complex Makefiles can be difficult to debug if not architected properly.
- Typical UNIX user-unfriendly error messages increase learning curve.
- Full system compiles are slower because of multiple javac invocations (although this could be remedied by calling ant for compiles only :)

I've also come up with the following, that I believe characterizes the situations in which ant or make are warranted:

• Use ant if:
  - You are a single developer or one of a very small development group working a small Java-based application and you don't know make.
  - You are working on a project that will be developed or deployed on a platform other than UNIX or Windows
  - Your developers are all using their own build scripts and you need something fast and are not looking to build a large...
Use make if:
- You know make
- You are working on a medium to large sized project or on a project with more than a few developers.
- You need to integrate with a variety of configuration management tools
- You are building a medium to long-term solution for build and deployment that must handle a wide variety of tools and technologies

Conclusions

While ant is an interesting tool, I don't see it as a heavy-duty build tool for Java development. Furthermore, ant's overall design seems to be the opposite of make. ant tries to be all things to all people and reimplements functionality that's available and proven on UNIX. ant tries to have a command for everything. Looking at the list of builtin commands one can't help but get the feeling that ant was made by someone who did not understand make and didn't want to learn it, and slowly realized that there's a lot of functionality a build tool will need if it cannot rely on the UNIX command set. Comments on the ant homepage support this (e.g. "Makefiles are inherently evil") to an extent. make, on the other hand, is a framework for running arbitrary commands. True, make requires UNIX to be useful, but UNIX commands are available on almost all platforms where Java is being developed. Yes, make can be hard to learn, and it does have it's quirks, but it works and it works well for almost any situation. The make code to implement java compilation (ant's biggest draw, IMO) is quite simple (even if non-trival to derive). This is one time only operation. I think ant is great if you are a single developer who can't afford an IDE and don't want to mess with make. With ant, you'll be up and running quickly. For a Release Engineer or Lead Programmer on a project, though, make is a tried and tested tool that works just as well with Java as it does with C, and you know that it will do whatever you want it to do.

References

- Ant Homepage
- Ant Documentation
- Make Homepage
- Make Documentation
- Cygwin Homepage
- Java Homepage
- Tomcat Homepage

Before You Respond!!

I know that many of you who prefer ant will want to put me in my place and point out that because I don't know ant very well, I will be biased towards make. Aside from requesting that you post constructive comments on this issue instead of flames, I would ask that you keep the following in mind:

- Make does have a large base of users, and many will be less objective than I am without arguments more convincing than
"Makefiles are evil". The greatest tool in the world that no one uses is worthless.

- Do you know a lot about make and build systems? I do, and I've built a lot of Makefiles and used make to do a lot of things that would be needed in any **serious** build system. If you have something to say about ant's ability to complete with make at this level, I hope you have done something more than just compile java code and create a .war file
- Have you worked on a large Java project that lasted more than a few months and required more than a few developers? Have you worked on a project where non-programmers needed to create or modify files that were part of the overall build?
- I know that make and ant are not the only build tools around. I know that other build tools cost money, and I'm operating under the assumption that someone using these tools cannot or will not buy a build tool. I also don't have much experience with them, but I tried to write the article so that that wouldn't matter

OK, I just want comments to be constructive and useful. I don't purport to know everything and have no problem being set straight, as long as convincing arguments are made.

**Full discussion:** http://www.kuro5hin.org/story/2001/9/24/155525/350

All trademarks and copyrights on this page are owned by their respective companies. The Rest © 2000 - 2001 Kuro5hin.org Inc.