Hello World

Authors: David Maddison, jboner jboner

Introduction

This tutorial does not explain AOP, so if your new to the idea of AOP then please check out JavaWorld's series of articles to get you started.

What this tutorial will do is to try to walk you through a simple example on how you can write, define and weave in an aspect into your application.

Installation

Download the latest release and unzip it into the relevant location. This tutorial is based on the 2.0 version of AspectWerkz but works equally with 1.0 final.

The latest distribution can be found here.

After installation you need to set the `ASPECTWERKZ_HOME` environment variable to point to the installation directory. This is because quite a few of the scripts use this to find the required libraries. How this variable is set depends on you OS. Since I'm using Linux I've amended my `.bashrc file`, windows users could do this by using the control panel.

The Test Application

Now we've installed aspectwerkz, we need a test application into which to weave our aspects. As is the tradition, I'm going to use the standard HelloWorld application.

```
package testAOP;

public class HelloWorld {

    public static void main(String args[]) {
        HelloWorld world = new HelloWorld();
        world.greet();
    }

    public String greet() {
        System.out.println("Hello World!");
    }
}
```

This is simply a standard Java application, and can be compiled with `javac -d target HelloWorld.java`

Writing the Aspect

Next we need to develop the aspect which will contain the code to be weaved into our HelloWorld class. In this example I'm going to output a statement before and after the greet method is called.

```
package testAOP;

import
org.codehaus.aspectwerkz.joinpoint.JoinPoint
;

public class MyAspect {

    public void beforeGreeting(JoinPoint
joinPoint) {
        System.out.println("before
greeting...");
    }

    public void afterGreeting(JoinPoint
joinPoint) {
        System.out.println("after
greeting...");
    }
}
```

Notice the signature of the aspect methods. They need to take this `JoinPoint` argument otherwise the AspectWerkz weaver won't be able to identify the method when the aspect is weaved in, (and can leave you scratching your head as to why the weaving isn't working!).
(Note: for 2.0, specific optimizations can be applied by using the *StaticJoinPoint* interface or no interface at all. Please refer to the AspectWerkz 2.0 documentation)

To compile this aspect class you'll need to include the `aspectwerkz-0.10.jar` in the classpath, i.e.

```
javac -d target -classpath
$ASPECTWERKZ_HOME/lib/aspectwerkz-2.0.RC1.ja
r MyAspect.java
```

For AspectWerkz 1.0 final:

```
javac -d target -classpath
$ASPECTWERKZ_HOME/lib/aspectwerkz-1.0.jar
MyAspect.java
```

Defining the Aspect

At this point we have the test application and the actual aspect code, but we still need to tell AspectWerkz where to insert the aspect methods (*pointcuts*), and which aspect method to actual insert (*advice*).

Specifying pointcuts and advice can be done using either of (or a mixture of), the following methods.

Using an XML definition file

The XML definition file is just that, an XML file which specifies the pointcuts and advice using XML syntax. Here's one that will weave our MyAspect class into our HelloWorld program (`aop.xml`):

```
<aspectwerkz>
    <system id="AspectWerkzExample">
        <package name="testAOP">
            <aspect class="MyAspect">
                <pointcut name="greetMethod"
expression="execution(*
testAOP.HelloWorld.greet(..))"/>
                <advice
name="beforeGreeting" type="before"
bind-to="greetMethod"/>
                <advice name="afterGreeting"
type="after" bind-to="greetMethod"/>
            </aspect>
        </package>
    </system>
</aspectwerkz>
```

Most of this should be pretty straight forward, the main part being the aspect tag. Whilst I'm not going to explain every bit of this definition file, (I'll leave that up to the official documentation), I will explain a few important points.

When specifying the `pointcut` the name can be any label you like, it's only used to bind the `advice`. The expression should be any valid expression according to the Join point selection pattern language however you **MUST** make sure that the full package+class name is included in the pattern. If this isn't done, or if the pattern is slightly wrong, AspectWerkz won't be able to correctly identify the *greet method*.

In the `advice` tag, the `name` attribute should be the name of the method in the aspect class, (specified in the `aspect` tag), which you wish to insert at the specific joinpoint. Type is set to `before`, `after`, or `around`, depending on where exactly you wish to inser the method in relation to the joinpoint. `bind-to` specifies the name of the `pointcut` to which this `advice` will be bound.

This example identifies the `HelloWorld.greet()` method and assigns it the pointcut label `greetMethod`. It then inserts the `MyAspect.beforeGreeting` method just before `greet` is called, and `MyAspect.afterGreeting` just after the `greet` method returns.

Using Annotations

Annotations provide a way to add metadata to the actual aspect class, rather than specifying it in a seperate definition file. Aspect annotations are defined using JavaDoc style comments a complete list of which is available

here. Using annotations, our aspect class would look as follows:

```
package testAOP;

import
org.codehaus.aspectwerkz.joinpoint.JoinPoint
;

public class MyAspectWithAnnotations {

   /**
    * @Before execution(*
testAOP.HelloWorld.greet(..))
    */
    public void beforeGreeting(JoinPoint
joinPoint) {
        System.out.println("before
greeting...");
    }

   /**
    * @After execution(*
testAOP.HelloWorld.greet(..))
    */
    public void afterGreeting(JoinPoint
joinPoint) {
        System.out.println("after
greeting...");
    }
}
```

After adding annotations you need to run a special AspectWerkz tool. This is done after compiling your aspect class files, (i.e. after running `javac`). The tool, known as the `AnnotationC` compiler, can be invoked as follows, passing in the source directory (`.`), and the class directory (`target`):

```
java -cp
$ASPECTWERKZ_HOME/lib/aspectwerkz-2.0.RC1.ja
r
org.codehaus.aspectwerkz.annotation.Annotati
onC . target
```

For AspectWerkz 1.0 final:

```
java -cp
$ASPECTWERKZ_HOME/lib/aspectwerkz-1.0.jar
org.codehaus.aspectwerkz.annotation.Annotati
onC . target
```

More information on the `AnnotationC` compiler can be found here.

Although using annotations means you don't have to write all aspect details in XML, you do still have to create a tiny XML 'stub' which tells the AspectWerkz runtime system which Java classes it should load and treat as aspects. An example of this is show below:

```
<aspectwerkz>
    <system id="AspectWerkzExample">
        <aspect
class="testAOP.MyAspectWithAnnotations "/>
    </system>
</aspectwerkz>
```

Weaving in and running the Aspect

There are basically two ways to actually weave the code together, one called *online weaving* performs the weaving as the classes are loaded into the JVM. The other is *offline weaving*, and is done before the code is actually run.

Using online weaving

When using *online weaving* you need to decide which JVM your going to use. This is because the *hook* which allows AspectWerkz to weave the classes together on the fly, is different in *Sun HotSpot* (where JDI/HotSwap is used), as apposed to *BEA JRockit* (where a PreProcessor is used). The default is setup to use Sun JDK 1.4.2, however if you want to use JRockit, simply edit the `bin/aspectwerkz` file (`aspectwerkz.bat` on windows), locate the "Sample Usage for JRockit" line and uncomment the command. Of course, remember to comment out the existing Java command above it.

Using JRockit is the preferred choice since it will not only perform much better (no need to run in debug mode, which using HotSwap, e.g. Sun and IBM, requires) and be more stable, but will also work on JDK 1.3, 1.4 and 1.5.

Performing the weaving is then just a matter of using the `aspectwerkz` command line tool to run `java` with the relevant classes, pointing it to the definition file, (even if using annotations you still need the 'stub' definition file), i.e.

```
$ASPECTWERKZ_HOME/bin/aspectwerkz
-Daspectwerkz.definition.file=aop.xml -cp
target testAOP.HelloWorld
```

This produces the expected output:

```
before greeting...
Hello World!
after greeting...
```

Using offline weaving

With *offline weaving*, the test applications classes are modified on the disk with the aspect calls. That is to say *offline weaving* amends your actual class definition, (as opposed to *online weaving* which doesn't modify any classes). To perform *offline weaving*, you use the `aspectwerkz` command line tool with the `-offline` option, as follows:

```
$ASPECTWERKZ_HOME/bin/aspectwerkz -offline
aop.xml -cp target target
```

The last option on the command (`target`) tells AspectWerkz where your classfiles are and is very important that you type in correctly, else nothing will get weaved into your target classes and you will wonder why nothing is happening.

Running the aspect is then just a matter of invoking your main class, although you still need some of the AspectWerkz jar's on your classpath, and you still need to provide an XML definition file:

```
java -cp
$ASPECTWERKZ_HOME/lib/aspectwerkz-2.0.RC1.ja
r:target
     -Daspectwerkz.definition.file=aop.xml
testAOP.HelloWorld
```

For AspectWerkz 1.0 final:

```
java -cp
$ASPECTWERKZ_HOME/lib/aspectwerkz-1.0.jar:ta
rget
     -Daspectwerkz.definition.file=aop.xml
testAOP.HelloWorld
```

Note: Windows users need to replace the ":" path separator by a ";"

This produces the expected output:

```
before greeting...
Hello World!
after greeting...
```

Conclusion

Now we have learned how to:

1. write a simple aspect in pure Java
2. define the aspect using annotations and XML
3. weave the aspect into your application using online and offline mode
4. run it

Want more?

Then read the next tutorial Hijacking Hello World or the online documentation

Want to use AOP in your appplication server?

Then start by reading this dev2dev article on how to enable AOP in WebLogic Server (the concepts are generic an works for any application server).

Credits

This tutorial is based on a tutorial written by David Maddison (with modifications and enhancements by jboner jboner)