面向剖面编程与 JBoss

作者：比尔 柏克，阿德莱恩 布鲁克
2003 年 5 月 28 日

翻译：JR Java技术文献翻译小组 coolients
网址：www.javaresearch.org
日期：2004-06-12
版本：1.0.0.0

概述

面向剖面编程（Aspect-Oriented Programming, AOP）是一个令人振奋的新领域，它将带来的影响，应该不亚于面向对象编程(OOP)15-20 年前所产生过的影响。不过 AOP 与 OOP 并不是相互竞争的两种技术，两者实际上互补得很好。OOP 擅长于用模块概念来描述对象层次关系(hierarchy)中的共同行为。但它也有局限性。如果一组互不关联的对象模块之间有一些共同行为动作需要处理，OOP 就有点力所不逮。而这就是 AOP 登台亮相的时候了。当这些互相独立的不同对象模块有一些具有横切性（crosscutting）的事务行为需要定义时，AOP 为你提供了方便，因为它能将有关功能行为表述为独立的层（layer），而不是将这些功能嵌入(embed)到当前模块中。这种程序描述方式改善了可读性，而且能使软件维护更为便利。OOP 软件开发的构思是一个从上到下的纵式发展过程，而 AOP 却是个从左到右的横向过程。两者是垂直交切的关系，但又有很好的互补性。

我们知道 OOP 的装备是继承，封装和多态，而 AOP 的组件则是指示（advice）/拦截器（interceptor），引导（introduction），元数据（metadata）和切分点（pointcut）。下面我们介绍一下这些概念。

指示/拦截器

一个指示就是一段由特定事件触发的程序逻辑，是能够被插入在调用者（激活其它方法的主体）和被调用者（被激活的方法）之间的行为。指示其实就是 AOP 核心概念。指示使你能够将诸如日志和观测之类的功能运用在现有的对象模型上而不必过问实现细节。

在 JBoss AOP 中，我们是用拦截器来实现指示。你可以定义拦截器，让它拦截方法调用、构造器调用和域访问。我们以后将讨论怎样将这些拦截器运用到现有的对象模块之上。

引导 (introduction)

借助引导，我们可以将方法或域增加到一个现有的类中。它们甚至允许你修改某个现有的类目前所实现的接口，并且引入一个实现了那些新接口的混合类。引导使你能够将多继承特性注入普通的 Java 类。一个极好的引导用例是你想让你手头处理的一个剖面有一个运行期接口。你希望这个剖面能跨越对象层次，在不同的对象类中能广泛地应用。但你同时也希望应用程序开发者能够调用专为剖面制定的 API.

```
Apple apple = new Apple();
LoggingAPI logging = (LoggingAPI)apple;
Apple.setLoggingLevel(VERBOSE);
```

引导可以用来将一个新的 API 追加到一个现有的对象模块上。

元数据

元数据是另一种能够追加到现有类之中的信息。它可以在在静态状态下或者在运行期追加。如果能够将元数据动态追加到一个给定的对象实例中，其意义就更大了。元数据最有用的一个情况是，你想写一个可运用于任何对象的全面通用的剖面，而程序逻辑却又需要知道各个类的具体信息。对于元数据的应用，有一个很好的类比，这就是EJB规范。在EJB的XML部署描述符中，你会针对每一个方法分别定义事务属性。应用程序服务器于是知道应该在什么时候和什么地方开始、挂起或者提交一个事务，因为你已经在EJB类和事务管理控件之间建立了元数据绑定关系（记录在bean的XML配置文件中），这些数据绑定关系定义了Required，RequiresNew，Support等等方法。

C#直接把元数据作为内置成分。XDoclet是另一个元数据实际运用的好例子。如果你曾经用Xdoclet来生成EJB文件和部署描述符，你就会知道元数据的能量。Java社区行动（Java Community Process, JCP）也认为，在JDK1.5 中，Java语言应该加入元数据。（见JSR175）。不过，在JSR175 实现之前，一个好的AOP框架也应该提供一种机制，用来声明在运行期有效的类级元数据。

切分点

如果说拦截器，引导和元数据是AOP的特征成分，那么切分点就是粘合剂。切分点告诉AOP框架，哪些拦截器绑定到哪些类， 哪些原数据将应用于哪些类，或者哪一个引导将被引入哪些类。切分点决定了各种AOP特征将怎样被运用于你的应用程序中的类。

AOP的实际运用

例1．使用拦截器

JBoss 4.0 带了一个 AOP 框架。这个框架和 JBoss 应用服务器紧密地结合，但也可以在你自己的应用程序中独立运行。要理解一个概念，最好的办法莫过于看看它的实际应用，所以让我们用一个 JBoss AOP 里面的例子，来说明所有这些组分是如何协同工作的。在本文的后半部分，我们要用 AOP 来建立一个简单的追踪框架。

定义一个拦截器

为了实现我们的这个小型追踪框架，我们必须作的第一件事是定义一个拦截器，让它来实行具体操作。在JBOSS AOP中，所有的拦截器必须实现org.jboss.aop.Interceptor接口。

```
public interface Interceptor
{
  public String getName();
  public InvocationResponse invoke(Invocation invocation) throws Throwable;
}
```

在 JBoss AOP 中，被拦截的域、构造器和方法一律被转成通用的 invoke 方法调用。方法的参数被填入一个 Invocation 对象，而方法的返回值、域的存取以及构造器则被填入一个 InvocationResponse 对象。Invocation 对象同时还驱动拦截链。为了清楚地说明这个，我们来看一下在下面这个例子中，各个对象是如何相互配合的。

```
import org.jboss.aop.*;
import java.lang.reflect.*;

public class TracingInterceptor implements Interceptor
{
  public String getName() { return TracingInterceptor; }
  public InvocationResponse invoke(Invocation invocation)
     throws Throwable
  {
    String message = null;

    if (invocation.getType() == InvocationType.METHOD)
    {
      Method method = MethodInvocation.getMethod(invocation);
      message       = method:  + method.getName();
    }
    else if (invocation.getType() == InvocationType.CONSTRUCTOR)
    {
```

```
        Constructor c = ConstructorInvocation.getConstructor(invocation);
        message       = constructor:  + c.toString();
      }
      else
      {
        // Do nothing for fields.  Just too verbose（对于域什么也不做。太琐碎).
        return invocation.invokeNext();
      }

      System.out.println(Entering  + message);

      // Continue on.  Invoke the real method or constructor (继续。调用真正的方法或者构造
器).
      InvocationResponse rsp = invocation.invokeNext();
      System.out.println(Leaving  + message);
      return rsp;
    }
  }
```

上面的拦截器将拦截所有的对域，构造器或方法的调用。如果调用的类型是一个方法或者构造器，一个带有方法或构造器签名的踪迹信息将输出到控制台。

挂接拦截器

好了，这样我们就定义了拦截器。但是怎么将这个拦截器挂接到具体的类？办法就是定义一个切分点。对于JBoss AOP，切分点是在一个XML文件中定义的。让我们看一下具体例子。

```
<?xml version="1.0" encoding="UTF-8">
<aop>
  <interceptor-pointcut class="POJO">
    <interceptors>
      <interceptor class="TracingInterceptor" />
    </interceptors>
  </interceptor-pointcut>
</aop>
```

上面的切分点将 TracingInterceptor 挂接到一个叫做 POJO 的类。这似乎有点笨拙；难道我们要为每一个想追踪的类都创建一个切分点吗？幸亏没那么糟糕：在 interceptor-pointcut 类属性里可以用任何的正则表达式。所以如果你想追踪 JVM 载入的所有的类，就把属性中的类表达式改成 .*。如果你仅仅想追踪一个特定的包，那么表达式可以是 com.acme.mypackge.*。

当 JBoss AOP 独立运行时，任何符合 META-INF/jboss-aop.xml 模式的 XML 文件将被 JBoss AOP 运行期程序所载入。如果相关的路径被包含在任何 JAR 文件或你的 CLASSPATH 的目录中，该 XML 文件将在启动之时由 JBoss AOP 运行期程序载入。

运行这个例子

我们将用上面定义的切分点去运行本例。POJO 类内容如下：

```
public class POJO
{
  public POJO() {}
  public void helloWorld() { System.out.println(Hello World!); }
  public static void main(String[] args)
  {
    POJO pojo = new POJO();
    pojo.helloWorld();
  }
}
```

TracingInterceptor 将拦截对 main()， POJO()和 helloWorld()的调用。输出结果应该是这样:

```
Entering method: main
Entering constructor: public POJO()
Leaving constructor: public POJO()
Entering method: helloWorld
Hello World!
Leaving method: helloWorld
Leaving method: main
```

你可以在这里下载JBoss AOP和示范代码。编译和执行方法:

```
$ cd oreilly-aop/example1
$ export CLASSPATH=.;jboss-common.jar;jboss-aop.jar;javassist.jar
$ javac *.java
$ java -Djava.system.class.loader=org.jboss.aop.standalone.SystemClassLoader POJO
```

JBoss AOP 对挂接的拦截器做字节码操作。因为没有编译步骤，AOP 运行期程序必须对 ClassLoader 有完全控制权。如果你的程序是在非 JBoss 应用服务器之外运行，你必须用专为 JBoss 制定的类载入器来覆盖系统的类载入器。

例 2：使用元数据

TraceingInterceptor 不追踪域访问是因为那有点过于琐碎。软件开发人员为了将域访问封装起来，通常会具体实现 get()和 set()方法。这种情况下，如果 TracingInterceptor 能够自动将这些方法过滤出来不去追踪，那是最好不过的。本例旨在演示如何用 JBoss AOP 元数据针对单个方法来实现这一过滤功能。元数据通常是用在更复杂的事务上，诸如定义事务属性，定义各个方法的安全角色，或是定义稳固性映射。但是本例应该足可以演示元数据在启用了 AOP 的应用程序中如何使用。

定义类的元数据

为了增加这一过滤功能，我们将提供一个旗标，用它来关闭追踪功能。让我们回到我们的AOP的XML文件修改定义标签，目的是去掉对get()和set()方法的追踪。事实上，对于main()函数的追踪毫无意义，所以我们也将它滤除。

```
<?xml version="1.0" encoding="UTF-8">
<aop>
  <class-metadata group="tracing" class="POJO">
    <method name="(get.*)|(set.*)">
      <filter>true</filter>
    </method>
    <method name="main">
      <filter>true</filter>
    </method>
  </class-metadata>
</aop>
```

上面的 XML 定义了一组叫做 tracing 的属性。这个过滤属性将挂接到每一个名字以 get 或者 set 开头的方法上。正则表达式格式用 JDK-1.4 定义的表达式。此元数据可通过 Invocation 对象在 TracingInterceptor 内部访问到。

*访问**Metadata***

此元数据在运行期必须是可以访问到的，不然它就没有实用意义了。类的元数据是通过Invocation对象访问的。为了在本例里使用它，TracingInterceptor必须要做一点小小的修改。

```
public class TracingInterceptor implements Interceptor
{
  public String getName() { return TracingInterceptor; }
  public InvocationResponse invoke(Invocation invocation)
    throws Throwable
  {
    String filter=(String)invocation.getMetaData(tracing, filter);
    if (filter != null && filter.equals(true))
      return invocation.invokeNext();

    String message = null;

    if (invocation.getType() == InvocationType.METHOD)
    {
      Method method = MethodInvocation.getMethod(invocation);
      message       = method:  + method.getName();
    }
    else if (invocation.getType() == InvocationType.CONSTRUCTOR)
    {
      Constructor c = ConstructorInvocation.getConstructor(invocation);
      message       = constructor:  + c.toString();
    }
    else
```

```
    {
      // Do nothing for fields.  Just too verbose.
      return invocation.invokeNext();
    }

    System.out.println(Entering  + message);

    // Continue on.  Invoke the real method or constructor.
    InvocationResponse rsp = invocation.invokeNext();
    System.out.println(Leaving  + message);
    return rsp;
  }
}
```

运行例子 2:

POJO 类将扩展一点：增加 get()和 set()方法。

```
public class POJO
{
  public POJO() {}
  public void helloWorld() { System.out.println(Hello World!); }

  private int counter = 0;

  public int getCounter() { return counter; }
  public void setCounter(int val) { counter = val; }
  public static void main(String[] args)
  {
    POJO pojo = new POJO();
    pojo.helloWorld();
    pojo.setCounter(32);
    System.out.println(counter is:  + pojo.getCounter());
  }
}
```

TracingInterceptor 将拦截对 main()(原文如此。但前边曾提到对不 main()做追踪。此例的示范输出结果也未出现对 main()的追踪)，POJO()和 helloWorld()调用。输出结果应该是这样：

```
Entering constructor: public POJO()
Leaving constructor: public POJO()
Entering method: helloWorld
Hello World!
Leaving method: helloWorld
```

你可以在这里下载JBoss AOP和示范代码。编译和执行方法:

```
$ cd oreilly-aop/example2
$ export CLASSPATH=.;jboss-common.jar;jboss-aop.jar;javassist.jar
$ javac *.java
```

```
$ java -Djava.system.class.loader=org.jboss.aop.standalone.SystemClassLoader POJO
```

例子 3 使用引导

如果我们能够根据具体的实例来决定关闭或打开追踪功能，那就真的叫酷了。JBoss AOP 已经有一个 API 能将元数据挂接到一个对象实例，但是我们姑且假装不知道，并且继续认为我们手头的这个追踪 API 是一个更好的方案。在本例中，我们将运用引导来改变 POJO 类的本身的定义。我们强迫 POJO 类去实现一个追踪接口，并且要提供一个混合类来处理新的追踪 API。这就是追踪接口:

```
public interface Tracing
{
  public void enableTracing();
  public void disableTracing();
}
```

定义一个混合的类

Tracing 接口将在混合类中实现。当一个 POJO 类被实例化时，一个混合类的实例将会被挂接到 POJO 类上。实现方法如下:

```
import org.jboss.aop.Advised;

public class TracingMixin implements Tracing
{
  Advised advised;

  Public TracingMixin(Object obj)
  {
    this.advised = (Advised)obj;
  }

  public void enableTracing()
  {
    advised._getInstanceAdvisor().getMetaData().addMetaData("tracing", "filter", true);
  }

  public void disableTracing()
  {
    advised._getInstanceAdvisor().getMetaData().addMetaData("tracing", "filter", false);
  }
}
```

enableTracing()方法将 filter 属性挂接到对象实例。在 disableTracing()方法作同样的事，但是将 filter 属性改为 false。这两个方法可作为一个例子，说明元数据不仅能够在类层次上实用，也能够在实例层次上应用。

挂接一个引导

好了，我们定义了追踪接口，也实现了混合类。下一步是将引导运用于 POJO 类。和运用拦截器类似，我们必须在 XML 中定义一个切分点。让我们看一下具体定义：

```
<?xml version="1.0" encoding="UTF-8">
<aop>
  <introduction-pointcut class="POJO">
    <mixin>
      <interfaces>Tracing</interfaces>
      <class>TracingMixin</class>
      <construction>new TracingMixin(this)</construction>
    </mixin>
  </introduction-pointcut>
</aop>
```

上面的切分点将强制 POJO 类实现 Tracing 接口。现在，当一个 POJO 实例被建立时，一个 TracingMixin 也将被实例化。TracingMixin 初始化的具体方法是在<construction>标签中定义。任何单行 Java 代码都可以放在<construction>标签中。

运行例子 3

本例中 POJO 类稍有扩展以演示如何访问 Tracing API。 TracingInterceptor 仍然和例子 2 一样。

```
public class POJO
{
  public POJO() {}
  public void helloWorld() { System.out.println(Hello World!); }

  public static void main(String[] args)
  {
    POJO pojo    = new POJO();
    Tracing trace = (Tracing)this;
    pojo.helloWorld();

    System.out.println("Turn off tracing.");

    trace.disableTracing();
    pojo.helloWorld();

    System.out.println("Turn on tracing.");

    trace.enableTracing();
    pojo.helloWorld();
  }
}
```

注意我们可以将 POJO 的类型转换为 Tracing 接口。输出结果应该是这样：

```
Entering constructor: POJO()
Leaving constructor: POJO()
Entering method: helloWorld
Hello World!
Leaving method: helloWorld
Turn off tracing.
Entering method: disableTracing
Leaving method: disableTracing
Hello World!
Turn on tracing.
Entering method: helloWorld
Hello World!
Leaving method: helloWorld
```

注意被增加到 TracingInterceptor 中的 interceptor-pointcut 也适用于那些通过 Tracing 引导导入的方法中。编译和运行本例的方法：

```
$ cd oreilly-aop/example3
$ export CLASSPATH=.;jboss-common.jar;jboss-aop.jar;javassist.jar
$ javac *.java
$ java -Djava.system.class.loader=org.jboss.aop.standalone.SystemClassLoader POJO
```

结论

面向剖面编程对于软件开发是一个强有力的新工具。有了JBoss4.0，你可以实现你自己的拦截器、元数据和引导，从而使你的软件开发过程更加灵活流畅。你可以访问我们的网站http://www.jboss.org以获取更详细的文档。你会有一些意外惊喜：我们在新框架的基础上，最近又实现了一个服务套件。来看一看吧。编程愉快。

比尔 柏克：JBossGroup 责任有限公司总设计师，JBoss4.0 领队
阿德莱恩 布鲁克：JBossGroup 责任有限公司支持部门主任