使用注解生成代码

一、概述

生成源代码很容易，但是生成正确的源代码就不容易了。要使用优雅高效的方法来生成源代码将是一个繁重的任务。

幸运的是从去年开始，MDE[1]（Model-Driven Engineering，也就是模型驱动工程设计，有时候也称为模型驱动开发或者模型驱动架构）已经有助于实践这个目标。这种设计更多的是倾向于艺术层面而不是科学——它针对的是经验丰富的程序员（译注：原文此句为 task for ninja coders，国外有把经验丰富的程序员比作忍者的习惯）——是基于经过验证的流程和工具所提取出来的成熟方法论。

尽管我们可以认为生成源代码是 MDE 方法论的一个天然切入点，但是 MDE 涵盖的范围远远不止这些。

注解处理器只是众多我们用来生成源代码工具中的其中一种而已。

二、MDE 中的 Model 和 Meta-model

在开始讲解如何使用注解处理器生成源代码相关细节之前，这里有几个我们需要先了解的概念，因为在接下来的章节中我们将会使用这些概念：model（模型）和 meta-model（元模型）。

MDE 的一个重要支柱就是它抽象的结构。我们将想要创建的软件系统在不同的细节层面使用不同的方法进行建模。当对一个抽象层建模之后，我们就可以对下一个和再下一个层面继续建模，直到一个可部署的产品被完整地建立起来。

从这个角度来看，无论我们使用的是哪一个细节层面，一个模型（覆盖的范围）都不会超过对应用来代表系统的抽象层。

元模型（meta-model），就是我们用来定义模型的规则。你可以认为它是模型的 schema 或者语法。

三、过注解处理器生成源代码

从一开始讨论到现在，注解处理器无疑是一种定义元模型和创建模型的优秀方法。注解类型扮演的是元模型角色，而一段代码块中所有注解的集合扮演的则是一个模型的角色。

我们可以利用模型来生成配置文件或者从一个已存在的源文件中派生出一个新的源文件。例如，创建一个远程代理，或者为被注解的 bean 创建一个可访问内部数据的入口对象。

这种方法的核心在于注解处理器。一个处理器能够读取源代码中的所有注解——也就是提取模型，并且通过它能够做任何我们想要做的事情——打开文件并添加内容等。Java 编译器会处理好模型验证的问题（注解必须匹配在注解处理器中注册的类型）。

四、Filer

在本系列的第二部分中曾经提到，每一个处理器都能够获取到一个 processing environment 对象，通过它能够获取到一些有趣的工具类对象。其中一个就是 Filer。

在 javax.annotation.processing.Filer[2]接口中定义了一些创建源文件、class 文件或者生成资源的方法。通过使用 Filer，我们可以确保使用了正确的文件目录，以避免丢失文件系统中生成的一些重要数据。

（另外，）我们需要关注的重点：一方面是考虑是否要写一个在 javac 上附加-d 或者-s 选项的生成器，另一方面就是 Maven POM 中定义文件夹。

下面是一个如何在注解处理器中创建 Java 源文件的例子。正如我们创建了一个 Bean 信息类，生成的类名与被注解的类名相同，只是在它后面添加上了“BeanInfo”后缀：

```
if (e.getKind() == ElementKind.CLASS) {
      TypeElement classElement = (TypeElement) e;
      PackageElement packageElement =
            (PackageElement) classElement.getEnclosingElement();

      JavaFileObject jfo = processingEnv.getFiler().createSourceFile(
            classElement.getQualifiedName() + "BeanInfo");

      BufferedWriter bw = new BufferedWriter(jfo.openWriter());
      bw.append("package ");
      bw.append(packageElement.getQualifiedName());
      bw.append(";");
      bw.newLine();
      bw.newLine();
      // rest of generated class contents
   }
```

上面的例子非常简单有趣，但是很糟糕。

我们将从注解中获取所需信息（代表模型）的逻辑与生成文件（代表视图，译注：指的是 MVC 设计模式中的 V 层）的逻辑混合在一起。

使用这种方法很难写出一个像样的生成器。如果我们需要在这个过程加入更复杂的东西，那么这个过程就会变得非常繁杂，并且容易出错，也很难维护。

因此，我们需要一种更加优雅的方式：

将模型从视图中清晰的分离出来。
使用模板来减轻生成文件的任务压力。
让我们来看一个使用这种方式的例子：如何利用 Apache Velocity 来生成我们想要的生成器。

五、Velocity 的历史简介

Velocity，Apache 软件基金会的一个项目，是一个用 Java 写的模板引擎，用于将模板和从 Java 对象中获取的数据进行混合生成各种文字类型的文件。

Velocity 经常在当下流行的 MVC 模式中被用来渲染视图，或者在 XML 文件中作为 XSLT 的替代品进行数据转换。

Velocity 拥有自己的语言，也就是 Velocity Template Language（VLT），它是生成简单易读模板的关键。使用 VLT，我们可以简单且直观地定义变量，控制流程和迭代，以及访问 Java 对象中包含的信息。

下面是一段 Velocity 模板片段：

```
#foreach($field in $fields)
    /**
     * Returns the ${field.simpleName} property descriptor.
     *
     * @return the property descriptor
     */
    public PropertyDescriptor ${field.simpleName}PropertyDescriptor() {
        PropertyDescriptor theDescriptor = null;
        return theDescriptor;
    }
#end
#foreach($method in $methods)
    /**
     * Returns the ${method.simpleName}() method descriptor.
     *
     * @return the method descriptor
     */
    public MethodDescriptor ${method.simpleName}MethodDescriptor() {
```

```
            MethodDescriptor descriptor = null;
            return descriptor;
        }
#end
```

六、Velocity 生成器使用方法

现在我们决定使用 Velocity 来升级我们的生成器，我们需要按照下面的步骤进行重新设计：

写一个用来生成代码的模板。
注解处理器从每一轮的 environment 中读取被注解的元素并将它们保存到容易访问的 Java 对象中——包括一个保存 field 的 map 对象，一个保存 method 的 map 对象，类名和包名等等。
注解处理器实例化 Velocity 的 context。
注解处理器加载 Velocity 的模板。
注解处理器创建源文件（通过使用 Filer），并且连同 Velocity Context 将一个写入器（writer）传递给 Velocity 的模板。
Velocity 引擎生成源代码。
通过使用这种方法，你会发现处理器/生成器的代码非常清晰，结构良好，并且易于理解和维护。

下面让我们一步一步来实现：

步骤 1：写模板

为了简单起见，我们不会列出完整的 BeanInfo 生成器代码，只是列出部分与注解处理器一块编译时需要的 field（成员变量）和 method（方法）。

接下来让我们创建一个名为 beaninfo.vm 的（模板）文件，并把它放到包含注解处理器的 Maven artifact 项目的 src/main/resources 目录下。模板内容的示例如下：

```
package ${packageName};

import java.beans.MethodDescriptor;
import java.beans.ParameterDescriptor;
import java.beans.PropertyDescriptor;
import java.lang.reflect.Method;

public class ${className}BeanInfo
        extends java.beans.SimpleBeanInfo {

        /**
         * Gets the bean class object.
```

```
     *
     * @return the bean class
     */
    public static Class getBeanClass() {

        return ${packageName}.${className}.class;
    }

    /**
     * Gets the bean class name.
     *
     * @return the bean class name
     */
    public static String getBeanClassName() {

        return "${packageName}.${className}";
    }

    /**
     * Finds the right method by comparing name & number of parameters in the class
     * method list.
     *
     * @param classObject the class object
     * @param methodName the method name
     * @param parameterCount the number of parameters
     *
     * @return the method if found, <code>null</code> otherwise
     */
    public static Method findMethod(Class classObject, String methodName, int parameterCount)
{

        try {
            // since this method attempts to find a method by getting all
            // methods from the class, this method should only be called if
            // getMethod cannot find the method
            Method[] methods = classObject.getMethods();
            for (Method method : methods) {
                if (method.getParameterTypes().length == parameterCount
                        && method.getName().equals(methodName)) {
                    return method;
                }
            }
```

```
            } catch (Throwable t) {
                return null;
            }
            return null;
        }
#foreach($field in $fields)

        /**
         * Returns the ${field.simpleName} property descriptor.
         *
         * @return the property descriptor
         */
        public PropertyDescriptor ${field.simpleName}PropertyDescriptor() {

            PropertyDescriptor theDescriptor = null;
            return theDescriptor;
        }
#end
#foreach($method in $methods)

        /**
         * Returns the ${method.simpleName}() method descriptor.
         *
         * @return the method descriptor
         */
        public MethodDescriptor ${method.simpleName}MethodDescriptor() {

            MethodDescriptor descriptor = null;

            Method method = null;
            try {
                // finds the method using getMethod with parameter types
                // TODO parameterize parameter types
                Class[] parameterTypes = {java.beans.PropertyChangeListener.class};
                method = getBeanClass().getMethod("${method.simpleName}", parameterTypes);

            } catch (Throwable t) {
                // alternative: use findMethod
                // TODO parameterize number of parameters
                method = findMethod(getBeanClass(), "${method.simpleName}", 1);
            }
```

```
            try {
                // creates the method descriptor with parameter descriptors
                // TODO parameterize parameter descriptors
                ParameterDescriptor parameterDescriptor1 = new ParameterDescriptor();
                parameterDescriptor1.setName("listener");
                parameterDescriptor1.setDisplayName("listener");
                ParameterDescriptor[] parameterDescriptors = {parameterDescriptor1};
                descriptor = new MethodDescriptor(method, parameterDescriptors);

            } catch (Throwable t) {
                // alternative: create a plain method descriptor
                descriptor = new MethodDescriptor(method);
            }

            // TODO parameterize descriptor properties

descriptor.setDisplayName("${method.simpleName}(java.beans.PropertyChangeListener)");
            descriptor.setShortDescription("Adds a property change listener.");
            descriptor.setExpert(false);
            descriptor.setHidden(false);
            descriptor.setValue("preferred", false);

            return descriptor;
        }
#end
}
```

注意在这个模板运作之前，我们需要将以下信息传递给 Velocity：

packageName：生成类的完整包名。
className：生成类的类名。
fields：源类中包含的 filed 的集合。我们需要从每个 field 中获取以下信息：
simpleName：filed 的变量名。
type：filed 的类型。
description：filed 的自我描述（在本例中没有使用）
……
methods：源类中包含的 method 的集合。我们需要从每个 method 中获取以下信息：
simpleName：method 的方法名。
arguments：method 的参数（在本例中没有使用）
returnType：method 的返回类型（在本例中没有使用）
description：method 的自我描述（在本例中没有使用）
……

所有的这些信息（也就是模型）都需要从源类里面匹配的注解中提取，并保存到 JavaBean 后传递给 Velocity。

步骤 2：注解处理器读取 Model

下面让我们创建一个注解处理器。正如本系列第二部分中所提到的，不要忘记给处理器添加注解，好让它能够处理 BeanInfo 注解类型：

```
@SupportedAnnotationTypes("example.annotations.beaninfo.BeanInfo")
@SupportedSourceVersion(SourceVersion.RELEASE_6)
public class BeanInfoProcessor
    extends AbstractProcessor {
      ...
}
```

注解处理器的方法需要从注解和源类本身中提取构建模型所需要的信息。你可以将全部需要的信息都保存到 JavaBean 里面，不过在这个示例中我们使用的是 javax.lang.model.element 类型，因为我们不打算传递太多细节给 Velocity（当然，需要的数据还是会传递过去的，在这个例子中我们要创建的是一个完整的 BeanInfo 生成器）：

```
String fqClassName = null;
String className = null;
String packageName = null;
Map<String, VariableElement> fields = new HashMap<String, VariableElement>();
Map<String, ExecutableElement> methods = new HashMap<String, ExecutableElement>();

for (Element e : roundEnv.getElementsAnnotatedWith(BeanInfo.class)) {

    if (e.getKind() == ElementKind.CLASS) {

        TypeElement classElement = (TypeElement) e;
        PackageElement packageElement = (PackageElement)
classElement.getEnclosingElement();

        processingEnv.getMessager().printMessage(
            Diagnostic.Kind.NOTE,
            "annotated class: " + classElement.getQualifiedName(), e);

        fqClassName = classElement.getQualifiedName().toString();
        className = classElement.getSimpleName().toString();
        packageName = packageElement.getQualifiedName().toString();
```

```
    } else if (e.getKind() == ElementKind.FIELD) {

        VariableElement varElement = (VariableElement) e;

        processingEnv.getMessager().printMessage(
            Diagnostic.Kind.NOTE,
            "annotated field: " + varElement.getSimpleName(), e);

        fields.put(varElement.getSimpleName().toString(), varElement);

    } else if (e.getKind() == ElementKind.METHOD) {

        ExecutableElement exeElement = (ExecutableElement) e;

        processingEnv.getMessager().printMessage(
            Diagnostic.Kind.NOTE,
            "annotated method: " + exeElement.getSimpleName(), e);

        methods.put(exeElement.getSimpleName().toString(), exeElement);
    }
}
```

步骤 3：初始化 Velocity Context 并加载模板

下面的代码片段展示了如何初始化 Velocity Context 并加载模板：

```
if (fqClassName != null) {

    Properties props = new Properties();
    URL url = this.getClass().getClassLoader().getResource("velocity.properties");
    props.load(url.openStream());

    VelocityEngine ve = new VelocityEngine(props);
    ve.init();

    VelocityContext vc = new VelocityContext();

    vc.put("className", className);
    vc.put("packageName", packageName);
    vc.put("fields", fields);
    vc.put("methods", methods);
```

```
        Template vt = ve.getTemplate("beaninfo.vm");
        ...
}
```

Velocity 配置文件，在本示例中名为 velocity.properties，它应该被放置到 src/main/resources 目录下。下面是它的内容示例：

```
runtime.log.logsystem.class = org.apache.velocity.runtime.log.SystemLogChute

resource.loader = classpath
classpath.resource.loader.class =
org.apache.velocity.runtime.resource.loader.ClasspathResourceLoader
```

这组属性配置了 Velocity 的日志，以及一个用于查找模板的基本资源加载路径。

步骤 4：创建新的源文件并生成源代码

紧接着，让我们创建新的源文件，并将这个新文件作为模板的目标来运行模板。下面的代码片段展示了如何操作：

```
JavaFileObject jfo = processingEnv.getFiler().createSourceFile(
        fqClassName + "BeanInfo");

processingEnv.getMessager().printMessage(
        Diagnostic.Kind.NOTE,
        "creating source file: " + jfo.toUri());

Writer writer = jfo.openWriter();

processingEnv.getMessager().printMessage(
        Diagnostic.Kind.NOTE,
        "applying velocity template: " + vt.getName());

vt.merge(vc, writer);

writer.close();
```

步骤 5：打包并运行

最后，注册注解处理器（记得添加本系列第二部分中提到过的 service 配置文件），然后打包。再通过终端命令行，Eclipse 或者 Maven 工具在 client（客户）项目中进行调用和编译。

假设 client 项目中的 client 类如下：

```
package example.velocity.client;
import example.annotations.beaninfo.BeanInfo;
@BeanInfo public class Article {
	@BeanInfo private String id;
	@BeanInfo private int department;
	@BeanInfo private String status;
	public Article() {
		super();
	}
	public String getId() {
		return id;
	}
	public void setId(String id) {
		this.id = id;
	}
	public int getDepartment() {
		return department;
	}
	public void setDepartment(int department) {
		this.department = department;
	}
	public String getStatus() {
		return status;
	}
	public void setStatus(String status) {
		this.status = status;
	}
	@BeanInfo public void activate() {
		setStatus("active");
	}
	@BeanInfo public void deactivate() {
		setStatus("inactive");
	}
}
```

当我们在终端上执行 javac 命令后，我们可以在控制台上看到找到的被注解元素，以及生成的 BeanInfo 类：

```
Article.java:6: Note: annotated class: example.annotations.velocity.client.Article
public class Article {
```

```
        ^
Article.java:9: Note: annotated field: id
    private String id;
                   ^
Article.java:12: Note: annotated field: department
    private int department;
                ^
Article.java:15: Note: annotated field: status
    private String status;
                   ^
Article.java:53: Note: annotated method: activate
    public void activate() {
                ^
Article.java:59: Note: annotated method: deactivate
    public void deactivate() {
                ^
Note: creating source file: file:/c:/projects/example.annotations.velocity.client/src/main/java/example/annotations/velocity/client/ArticleBeanInfo.java
Note: applying velocity template: beaninfo.vm
Note: example\annotations\velocity\client\ArticleBeanInfo.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

如果我们检查下源代码目录，我们将会找到我们生成的 BeanInfo 类。任务完成！