Java 注释类型和 APT

Annotation and APT in Java

凌　晨　(总后科研所信息系统室　100071)
陈芳莉　(解放军国防大学研究生 17 队　100091)

摘要:自 J2SE5.0 开始,元数据已经正式成为 Java 语言的一部分。本文介绍了 J2SE5.0 中提供的元数据类型——注释——的基本使用规则,以及它给程序开发带来的巨大便利。同时,结合 APT(Annotations Processing Tool)介绍了注释的高级应用——样本代码生成。

关键词:APT　注释　元注释　注释处理机　包　导入　工厂类

在 Java 正式推出元数据之前,开发者仅能够使用像 Javadoc 和 XDoclet 这类工具把元数据添加到代码中。然而,在 J2SE5.0 中引入了"注释(annotations)"的语法,他整合了元数据技术并直接嵌入 Java 中,提供了一个给程序元素赋予属性的标准。为了更好的应用注释类型,Sun 公司发布了 APT——Annotations Processing Tool,它能够让你写出你自己的注释处理机(Annotations processor)。这样你就可以遵照注释来分析 Java 的源文件并自动地产生样本代码。这种功能是由 mirror API 提供的,mirror API 的功能与 java 标准库中的 reflection API 极为相似。本文介绍使用注释的方法并讲解了如何使用 APT 来分析 Java 的源文件及如何产生样本代码。

使用注释类型和使用其他如 final,static,transient 等关键字一样,所不同的是在其前面放一个"@"。每种注释类型都针对一系列特定的程序元素。也就是说,每个注释类型总是用于修饰一个包、类型(包括 class,enum,和 interface)、域(包括一个 enum 常量)、构造器、方法、参数、本地变量声明、或者是这些的组合。目前,根据使用的参数对注释类型进行分类,J2SE5.0 提供了三种类型的注释,他们之间存在着细微的语法差别。如表 1 所示。

1　Java.lang 中的注释类型

J2SE5.0 直接在 API 中融入了对注释的支持。有三种注释类型不但被直接声明在 java.lang 中,而且可以被 java 编译器直接使用,所以使用他们时不需要任何的导入(Import)声明。本文通过具体的实例来帮助读者了解其语法。

表 1

annotation	描述	语法例子
Marker	不带参数	@Marker Annotation
Single valued	带一个参数	@Single Value annotation("VALUE")
Regular	带多个参数	@Regular annotation(name = "John Q.",bugValue = 1)

(1) @Deprecated。任何已经废弃或者是由于某些原因不应该再使用的声明,无论他是否已经被替换,都需要将其声明为废弃(deprecated),这是 java 的一个标准。在引入注释类型前,标识一个已经被废弃的声明的唯一方式是使用 javadoc。如今,在新的 J2SE5.0 中,当你使用一个废弃的声明时,该注释类型将通过编译器或 IDEs(如 Eclipse)通知你。这就是一个 marker annotation (不需要参数的注释类型)。你可以用它来注释任何程序元素,如下所示:

```
/*使用 annotation 的一个标识废弃的(deprecated)方法*/
Public class DeprecateExample{
//如果你使用这个方法,将得到一个警告
@Deprecated
Public void hasDeprecated(){
  System.out.prinLn("I am deprecated!");
  }
```

```
}
```

(2) @ Suppress Warnings。在编译时,如果代码符合语法但却存在潜在问题编译器会输出"警告(warnings)"。Sun 公司的 javac 编译器通过使用 -Xlint 选项开启警告, -Xlint 选项和@ Suppress Warnings 这个注释类型关系十分紧密。需要注意的是不同的厂商推出的 java 编译器还是定义了各自专用的警告信息。本文中所有的代码仅针对 SUN 的 J2SE5. 0。看一个例子:

```
import java. util. ArrayList;
Public class UncheckedExample{
  Public void uncheckedGenerics( ){
    ArrayList blahblah = new ArrayList( );
    //警告!
    blahblah. add(2);
  }
}
```

为了编译上面的代码,你在命令行中输入:

```
javac -Xlint:unchecked UncheckedExample. java
```

这样会出现一个警告信息。但是,当你加入:

```
@ Suppress Warnings( value = { "unchecked" } )
```

你就不会再看到这个警告。如下面的代码所示:

```
import java. util. ArrayList;
@ SuppressWarnings( value = { "unchecked" } )
Public class UncheckedExample{
  Public void uncheckedGenerics( ){
    ArrayList blahblah = new ArrayList( );
    //没有警告了!
    blahblah. add(2);
  }
}
```

通过 value 数组可以同时取消多个警告;如:

```
@ Suppress Warnings( value = { "unchecked", "finally" } )
```

下表中列出了目前已经提供的可选项:

如果不给 -Xlint 任何的选项,就会激活所有的警告。通过设置 -Xlint: -<option>也可以取消警告,方法是增加一个额外的减号("-")。

(3) @ Override。在覆盖(override)一个方法时,如果使用@ Override 这个注释类型,它就能让编译器捕获那些你认为已经被覆盖而实际上并没有被覆盖的方法。这对保持类的层次性尤其有用。举例来说,当我们设法实现 java. util. Collection 这个接口时,想要覆盖 Object get(int index)这个方法,此时会得到一个编译错误。如果没有使用@ Override 注释 get( )这个方法,编译器会认为此时创建了一个新的方法。如下例所示:

表 2

选 项	描 述
none	没有警告
unchecked	给出类型的 unchecked conversions 的详细内容
path	检查在环境变量(比如类路径)中不存在的路径
serial	检查在 serializable classes 中提供的 serialVersionUID
finally	检查 finally classes 能够被完全的 normally
fallthrough	确保在每个 case 语句后你用了 break,否则在 switch 语句中执行下一个 case 语句
depreciation	检查已经被废除(depreciated)的程序元素的使用

```
//使用@ Override annotation
Package net. ling;
Import java. util. *;
/* 使用@ Override 这个注释类型 */
Public class OverrideExample implements Collection{
//覆盖 get( )
@ Override
Public Object get(int index){
  Object value = null;//code to get stuff
Return value;
  }
}
```

上面的代码无法编译,因为是 List interface 而不是 Collection interface 定义 get( )这个方法。List interface 派生于 Collection interface,这对那些习惯于用 List interface 的使用者来说确实容易产生误导。

2　元注释类型

一个注释类型对于注释的意义,就如同一个接口

对于对象。事实上，注释类型就像接口一样存在并且在某些行为上也很相似。针对注释的注释类型被称为元注释类型（Meta annotation），他们都派生于 java.lang.annotation.Annotation 接口并且都拥有带返回类型的方法。这些方法只能返回简单的类型，比如字符串、枚举、或者其他的注释类型，或者是含有这些类型的数组。

除了上面三种定义在 java.lang 包（package）里的注释类型以外，在 java.lang.annotation 中还定义了四种元注释类型。

（1）@Retention。在 J2SE5.0 中所提供的注释类型都会用到 retention 提供的三种保留策略之中的一种。第一种，当使用源码（source）级别的保留策略（Retention Policy.SOURCE）时，该注释类型只存在于源代码中，并且不能通过编译成为字节码。第二种，当使用类（class）级别的保留策略（Retention Policy.CLASS）时，该注释类型只能在编译后运行前的字节码层次中访问，一般在默认的情况下都是使用该策略。第三种，当使用运行时级别的保留策略（RetentionPolicy.RUNTIME）时，表明该注释类型能够在运行时通反射（reflection）被存取。因为 APT 是一个源码级别（source-level）的处理工具，所以本文关注的是拥有源码（source）级别的保留策略的注释。

（2）@Target。用于指示注释类型所适用的程序元素的种类。如果注释类型声明中不存在 Target 元注释，则声明的注释类型可以用在任意程序元素上。如果存在这样的元注释，则编译器会强制执行指定的使用限制。在后面的 APT 示例中将对其使用做详细介绍。

（3）@Documented. 指示某一类型的注释将通过 javadoc 或类似的默认工具进行文档化。如果类型声明是用@Documented 来注释的，则其注释将成为注释元素的公共 API 的一部分。

（4）@Inherited. 如果一个注释类型被这个元注释类型所修饰，则任何应用了该注释的程序元素——类（classes）、接口（interface）和枚举（enums），其派生元素都会通过"继承"获得该注释。这是一个"当且仅当"条件——如果你不想子类继承这个 annotation，那么就别用它。

3 样本代码和 APT

尽管注释类型被正式推出的时间并不长，但他已经在"样本代码生成"上家喻户晓了，而这一切都得益于 APT（annotation processing tool）。根据 sun 官方的解释，APT 是一个命令行工具，它对源代码进行检测找出其中的注释类型，并使用注释处理机（annotation processors）来处理注释类型。而注释处理机使用了一套 mirror API 并具备对 JSR175 规范的支持。

使用 APT 的主要目的是简化开发者的工作量，因为 APT 可以在编译程序源代码的同时，生成一些附属文件（比如源文件、类文件、程序发布描述文字等），这些附属文件的内容也都是与源代码相关的。换句话说，使用 APT 就是代替了传统的对代码信息和附属文件的维护工作。

本文的示例就是根据程序元素上的注释，APT 的注释处理机产生新的异常类（exception class）。根据传递给注释不同的值，生成的异常类会有细微的不同。下面这段代码包括这个例子需要的两个注释类中的一个。

```
package net.ling.annotations; //ApplicationException.java
import java.lang.annotation.*;
/** APT 使用该注释类型产生新的异常类 */
@Retention(RetentionPolicy.SOURCE)
// 此处"@Target" 接受的参数为一个空数组
// 这意味着该注释类型(ApplicationException)只能作
// 为其他注释类型中的一部分，而不能直接使用
@Target({})
public @interface ApplicationException {
 /** 这个新异常类的名字 */
 String exceptionName();
 String addedInforType() default "";
 String addedInfoVarName() default "";
}
```

注释类型给"default"关键字赋予了第二个含义（第一个是指在 switch 语句中的"default case"）。在注释类定义的某个方法中，使用 default 来指定默认的返回值。

下面的代码定义了一个名为 ApplicationExceptions 新的注释类型，其中包含一个类型为 ApplicationException 的数组。

```
package net. ling. annotations; // ApplicationExceptions. java
import java. lang. annotation. *;
@ Target( ElementType. TYPE)
@ Retention( RetentionPolicy. SOURCE)
@ Documented
public @ interface ApplicationExceptions {
    ApplicationException [ ] applicationExceptions();
}
```

APT 真正的威力来自他对 Java 源文件中注释类型的处理方式。它使用注释类型来做：

(1) 指定要处理的源文件

(2) 适当地分析源文件，产生样本代码或者外部配置文件

基本工作流程是：首先，APT 运行注释处理机，根据提供的源文件中的注释生成源代码文件和其它的文件(文件具体内容由注释处理机的编写者决定)，接着 APT 将生成的源代码文件和提供的源文件进行编译生成类文件。在本文的例子中 ApplicationExceptionsApf. java 定义了基于 ApplicationExceptions 注释类型创建样本代码的注释处理机工厂和类。APT 首先检测在源代码文件中存在哪些注释，然后将查找我们编写的注释处理机中的工厂类(ApplicationExceptionsApf)，并且要求该工厂类提供处理源文件中所涉及的注释类型的注释处理机(annotation processor)。接下来，一个对应的注释处理机将被执行，如果在注释处理机生成源代码文件时，生成的文件中含有注释，则 APT 将重复上面的过程直到没有新文件生成。

```
package net. ling. apt. processors; // ApplicationExceptionsApf. java
import net. ling. annotations. ApplicationException;
import net. ling. annotations. ApplicationExceptions;
...
public class ApplicationExceptionsApf implements
    AnnotationProcessorFactory {
  private static final Collection < String > supportedAnnotations =
    unmodifiableCollection(
      Arrays. asList( " net. ling. annotations. ApplicationExceptions" ) );
  ......
  public AnnotationProcessor getProcessorFor(
    Set < AnnotationTypeDeclaration > atds,
    AnnotationProcessorEnvironment env) {
    ... ...
    return AnnotationProcessors. NO_OP;
  }
  /* 这是一个简单的处理机，他会根据相应的注释生成样本代码 */
  private static class ApplicationExceptionsAp
    implements AnnotationProcessor {
    private final AnnotationProcessorEnvironment env;
    ApplicationExceptionsAp ( AnnotationProcessorEnvironment env) {
      this. env = env;
      ... ...
    }
    public void process( ) {
    Messager msg = env. getMessager( );
    ......
    }
    ...
  }
}
```

从代码中可以看出每个处理机实现了在 com. sun. mirror. apt 包中的 AnnotationProcessor 接口，这个接口有一个名为"process"的方法，该方法将在 APT 调用 processor 时被用到。一个处理机可以处理一种或者多种 annotation 类型。一个处理机实例被其相应的工厂返回，此工厂实现了 AnnotationProcessorFactory 接口。APT 将调用工厂类的 getProcessorFor 方法来获得处理机。在调用过程中，APT 将提供给工厂类一个 AnnotationProcessorEnvironment 类型的处理机环境对象，在这个对象中，处理机将找到其执行所需要的每件东西，包括对所操作的程序结构的参考，与 APT 通讯并

合作完成新文件的建立和警告/错误信息的传输。

从以上例子中可以看出处理机工作很简单:他首先检查提供给 APT 环境的每一类型的声明,然后得到 ApplicationExceptions 注释并在内部处理每一个 ApplicationExceptions 值,产生一个包括合适的异常定义的 a. java 文件。

如何使用这个注释处理机? APT 提供了两种方式:通过 APT 的" -factory"命令行参数提供,或者让工厂类在 APT 的发现过程中被自动定位。前者对于一个已知的 factory 来讲是一种主动而简单的方式;而后者则是需要在 jar 文件的 META - INF/services 目录中提供一个特定的发现路径。在包含 factory 类的 jar 文件中做以下的操作:在 META - INF/services 目录中建立一个名为 com. sun. mirror. apt. AnnotationProcessorFactory 的 UTF - 8 编码文件,并写入所有要使用到的 factory 类全名,每个类为一个单独行。

下面的代码是一个使用 ApplicationExceptions 类的例子。

```
package net. ling; // ExceptionAnnotationTest. java
import net. ling. annotations. *;
/** 测试上面的新注释类型*/
@ApplicationExceptions(
  applicationExceptions = {@ ApplicationException
(exceptionName = "Test",
    addedInformationType = "int",
    addedInformationVariableName = "status"),
  @ ApplicationException (exceptionName = "
App")})
public class ExceptionAnnotationTest {
}
```

本文使用命令行选项来产生和执行注释处理机。在执行下面的命令前,确保 tools. jar 文件已经在你的类路径里面了并且你已经编译过注释处理机:

Apt -factory net. ling. apt. processors. ApplicationExceptionsApf net\ling*. java

这样就应该已经产生两个文件:TestException. java 和 AppException. java

4 结论

本文介绍了 J2SE5.0 提供的注释类型并且介绍了如何使用 APT 来产生样本文件异常类代码。APT 就像一个在编译时处理注释类型的 javac,希望大家能重视这个新特性。从 SUN 的官方网站上了解到在代号为"野马"的 J2SE6.0 的 beta 版中已将 APT 的功能写入到了 javac 中,这样只要执行带有特定参数的 javac 就能达到 APT 的功能。

参考文献

1 Java API 文档中文版, http://gceclub. sun. com. cn/chinese_java_docs. html,2003.3。

2 《Java 2 全方位学习》, 朱仲杰, 机械工业出版社, 2003.3。

3 《Head First Java, Second Edition》, Kathy Sierra, Bert Bates O'Reilly,2005.2。