Annotation（注解）

北京理工大学计算机学院
金旭亮

主要内容

- Annotation概述
- 自定义Annotation
- Annotation实际应用示例

1 Annotation概述

什么是注解（Annotation）

- 注解相当于一种标记，加了注解就等于给代码打上了某个标记。javac编译器、Eclipse之类IDE以及其他一些注解处理工具（Annotation Processing Tool，APT）就可以利用它来完成一些特殊的工作。
- 注解可以附加在包、类、字段、方法、方法参数以及局部变量上。
- 在各种软件开发工具和框架中，注解被广泛使用，比如JUnit工具会自动运行标记有@Test的单元测试方法，Spring MVC框架会将标有@Controller的类识别为控制器等等。

关于注解

- 带有注解的代码被编译时，如果程序员明确指明需要的话，javac编译器可以将相关信息保存到class文件中，当JVM装载.class文件时，这些注解信息一并被装入内存。
- 你可以自己编程写代码，在程序运行时使用反射技术读取这些注解信息，干一些特定的活。
- JDK中内置了一些“直接可用”的注解，主要集中于java.lang、java.lang.annotation和javax.annotation中。

JDK中的预定义注解-1

@Deprecated：标记类的成员已过时

示例：AnnotationTest.java

@SuppressWarnings(“deprecation”)：忽略编译时的警告信息

JDK中的预定义注解-2

@Override：要求子类必须覆盖基类的方法

```
class MyClass {
    //@Override要求子类必须覆盖基类的方法，因此，如果参数类型为MyClass，
    //将不能通过编译
    //因为基类Object的equals方法，其参数类型是Object

    //@Override
    public boolean equals(MyClass obj) {
            return super.equals(obj);
    }
}
```

JDK中的预定义注解-3

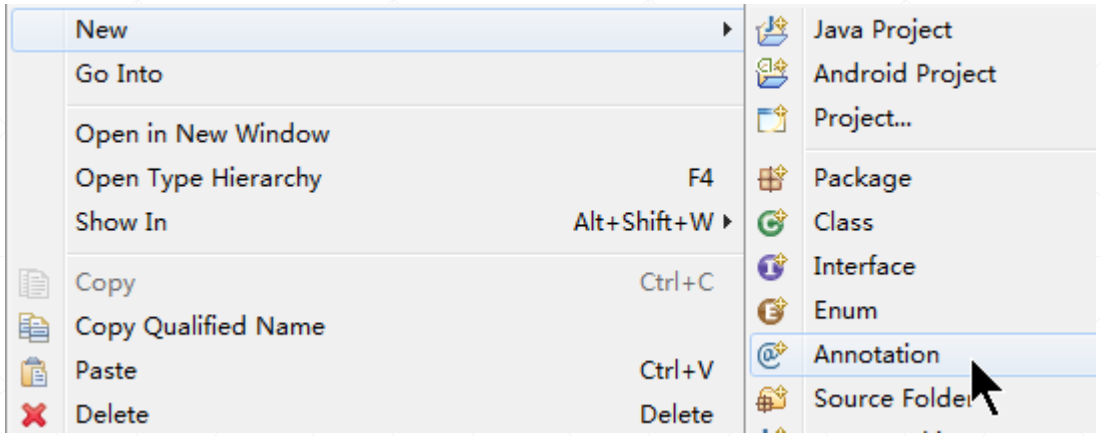
@Serializable：指定某个类是可以序列化的

@FunctionalInterface：指定某个方法重写了基类的同名方法

2 自定义Annotation

自定义注解

注解，其实就是一个使用@interface定义的接口。




```
public @interface MyTestAnnotation
{
}
```

使用注解

- 注解定义好之后，就可以使用于源代码中

```
@MyTestAnnotation
public class UserDefineAnnotationTest {
}
```

访问注解

- 当其它类使用附加了注解的类时，可以通过反射来动态地查询注解

```
if(AnnotationTest.class.isAnnotationPresent( MyTestAnnotation.class))
{
      MyTestAnnotation annotation=
            (MyTestAnnotation) UserDefineAnnotationTest.class.getAnnotation(
            MyTestAnnotation.class);
       System.out.println(annotation);
}
```

JDK中还提供了getAnnotations()/getDeclaredAnnotations()、getAnnotationsByType()/getDeclaredAnnotationsByType()方法获取特定类型上的注解集合。

元注解

- 定义注解的类型也可以添加“注解”，这种针对“注解”的“注解”，称为“**元注解（meta-annotation）**”。

```
@Retention(RetentionPolicy.RUNTIME)  //指明注解生存的时间
@Target({ElementType.METHOD,ElementType.TYPE})  //指明注解适用的场合
public @interface MyTestAnnotation {

}
```

可以通过查询文档或源码了解特定注解的含义与功能。

RetentionPolicy说明

前页PPT中使用JDK所提供的元注解@Retention(RetentionPolicy.RUNTIME)来定义用户自定义的注解，其含义如下所示：

RetentionPolicy值	说明
RetentionPolicy.RUNTIME	编译器将把Annotation记录在class文件中，当运行Java程序时，JVM将装载Annotation，程序可以通过反射读取它们。
RetentionPolicy.CLASS	编译器将把Annotation记录在class文件中，当运行Java程序时，JVM不能获取Annotation值。
RetentionPolicy.SOURCE	Annotation只保留在源代码中（供特定的工具处理），编译时直接丢弃。

可以为注解添加各种属性

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD,ElementType.TYPE})
public @interface MyTestAnnotation {
        String color();
}
```

- 使用：

```
@MyTestAnnotation( color="red" )
public class AnnotationTest {
 }
```

特殊的value属性-1

- 如果注解中只有一个名为value的属性需要赋值，可以简化为：

```
@SuppressWarnings("deprecation")
```

- 以下是SuppressWarnings的定义：

```
public @interface SuppressWarnings {
        String[] value();
}
```

特殊的value属性-2

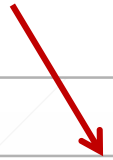
- 如果有多个属性值，只要为value属性值之外的属性指定默认值，则仍然可以使用简化的语法：

```
public @interface MyTestAnnotation {
    String color() default "blue";
    int[] arrayAttribute() default {3,4,5};
    String value();
}
```

注解类型的注解属性

- 注解的属性可以是各种类型，甚至是又一个注解：

```
public @interface MyTestAnnotation {
    MyMetaAnnotation annotationAttr() default @MyMetaAnnotation(“hello");
}
```



```
@Retention(RetentionPolicy.RUNTIME)
public @interface MyMetaAnnotation {
        String value();
}
```

UserDefineAnnotationTest.java

注解的“继承性”

```java
//定义一个具有“继承特性”的注解
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Inherited
public @interface Inheritable {

}
```

如果使用@Inherited，可以定义一个拥有“子类可以继承父类”特性的注解。

```java
@Inheritable
class Base{
}
//因为基类附加了有“继承特性”的注解，所以子类自动也拥有此注解
class Child extends Base{
}

public class InheritableTest {
    public static void main(String[] args) {
        //应该输出true
        System.out.println(Child.class.isAnnotationPresent(Inheritable.class));
    }
}
```

扩充了解：自定义注解时可以使用的类型

```
public @interface BugReport {
    //枚举类型
    enum Status { UNCONFIRMED, CONFIRMED, FIXED, NOTABUG };
    Status status() default Status.UNCONFIRMED;

    //原始数据类型
    boolean showStopper() default false;

    //字符串及字符串数组
    String assignedTo() default "[ none]";
    String[] reportedBy();

   //class类型
    Class <? > testCase() default Void.class;

    //另一个注解类型
    Reference ref() default @Reference(); // an annotation type
}
```

源代码中可以放置注解的地方

```
//附加在类上的注解
@Entity
public class User { . . . }

//附加在变量上的注解（仅在源代码级别有效，编译时会删除它们）
@SuppressWarnings("unchecked") List < User > users = . . .;
public User getUser(@Param("id") String userId)


//附加在泛型参数上的注解
public class Cache <@Immutable V > { . . . }

//附加在包上的注解
@GPL(version ="3")
package com.horstmann.corejava;
import org.gnu.GPL;
```

3 Annotation实际应用示例

实际开发中Annotation用在何处？

在实际开发中，你会发现有许多的开发框架，比如Spring，JUnit等都定义了自己的注解，并且基于这些注解完成特定的功能，比如Spring通过@Bean注解向IoC容器注册JavaBean，JUnit扫描@Test注解获取所有需要运行的单元测试方法。

如果你想设计自己的开发框架，那么，掌握注解的开发技巧是很必要的。通常你会这么干：

（1）定义若干个自己的注解
（2）将这些注解附加到相应的程序元素（比如类或方法）
（3）编写特定的工具（或代码）去提取这些注解，然后干特定的事情。

示例

```java
//自定义一个注解
@Retention(RetentionPolicy.RUNTIME)
public @interface MyFrameworkAnnotation {

}
```

```java
//此类有三个方法，两个附加了注解，另一个没有
//用于展示在运行时可以通过读取注解信息，确定是否应该调用特定的方法
public class MyFrameworkClass {
    @MyFrameworkAnnotation
    public void runTest() {
        System.out.println("MyFrameworkClass::runTest");
    }
    @Deprecated
    public void print() {
        System.out.println("MyFrameworkClass::print已被废弃");
    }
    public void info() {
        System.out.println("MyFrameworkClass::info");
    }

}
```

```java
public class MyFrameworkTest {

    public static void main(String[] args)
            throws IllegalAccessException, IllegalArgumentException,
            InvocationTargetException {

        MyFrameworkClass obj=new MyFrameworkClass();
        Annotation[] annotations=null;
        //提取所有的方法
        for (Method mtd : MyFrameworkClass.class.getMethods()) {
            annotations=mtd.getAnnotations();
            //检测是否方法附加有Annotation
            if(annotations.length>0) {
                System.out.println("方法"+mtd.getName()+"拥有的Annotation:");
                for (Annotation ann : annotations) {
                    System.out.println(ann.toString());
                }
                //检测附加的注解是不是@MyFrameworkAnnotation
                if(mtd.isAnnotationPresent(MyFrameworkAnnotation.class)) {
                    System.out.println("\n使用反射运行有@MyAnnotation注解的方法:");
                    mtd.invoke(obj);
                }
                System.out.println("----------------------");
            }
        }
    }
}
```

示例代码展示了注解典型的用法:
基于反射获取注解，然后再调用之。

小结

- 本讲介绍了注解（Annotation）的基础知识，但除非你要开发自己的框架，需要自定义注解的场景并不多。
- 不过有许多开发框架使用了注解，因此，了解注解的基础知识是非常必要的。
- 在学习各种开发框架时，关注它定义了哪些注解，这些注解又有哪些特定的作用，对于学好用好开发框架还是很重要的。
- 另外，我们还可以开发APT（Annotation Processing Tool），对源代码文件进行检测，找出源代码包容的注解信息，然后针对它们进行额外的处理（比如生成额外的文件），这块技术用得不算多，感兴趣的请自行上网搜索相关资源自学。