注解

天津卓讯科技有限公司

13.1 注解

- 13.1.1 注解概述
- 13.1.2 使用Java SE API 内置注解类型
- 13.1.3 自定义注解类型
- 13.1.4 元注解

元数据

- 元数据(metadata)：就是关于数据的数据。
  - 示例：
    - 表格中呈现的是数据，而表格还会有额外的数据来说明表格的作用，这个就是表格的元数据。
    - 数据库中的表存放了数据，但表还需要有表的定义、字段的定义等，这个就是数据库表的元数据。
    - XML文件可以存放数据。但xml文件的每个标签还需要有相应的描述，这些描述就是XML标签的元数据。
- 元数据可以用于创建文档，跟踪代码中的依赖性，甚至执行基本编译时检查

什么是Annotation

- JDK5.0通过名为Annotation(注解)的新功能将一个更通用的元数据工具合并到核心Java语言中。
- 注解是可以添加到代码中的修饰符，对程序代码做出一些说明和解释。可以用于包声明、类声明、构造方法、方法、字段、参数和变量。
  - 这样就将程序的元素和元数据联系起来。编译器就可以将元数据存储在Class文件中。之后虚拟机和其它对象可以根据这些元数据来决定如何使用这些程序元素或改变它们的行为。
- JDK5.0包含了内置注解，还支持编写定制注解

注解基本知识

- 注解采用“@”标记形式 ，后面跟注解类型名称。通过(name=value)向注解提供参数数据。每次使用这类表示法时，就是在生成注解。
  - 注解类型和注解的区别：注解类型类似于类，注解类似于该类的实例

13.1.2内置注解类型—Override

- **Override** 指明被注解的方法必须是重写超类中的方法。仅能用于方法之上。
  - 编译器在编译源代码时会检查用@Override标注的方法是否有重写父类的方法

```
public class InternalAnnotationTest {
    @Override
    public String toString() {
        return super.toString() + " [Override toString]";
    }

    public static void main(String[] args) {
        TestAnnotation test = new TestAnnotation();
        System.out.println(test);
    }
}
```

13.1.2内置注解类型—Deprecated

- Deprecated 指明被注解的方法为过时的方法，不建议使用了。能用于方法之上。
  - 当编译调用到被标注为Deprecated的方法的类时，编译器会产生警告。

```
public class InternalAnnotationTest {
    …
    @Deprecated
    public void test(){
        System.out.println("[Deprecated Annotation]");
    }
}
```

内置注解类型—SuppressWarnings

- SuppressWarnings 指明被注解的方法在编译时如果有警告信息，就阻止警告。可放置任何位置。
- 它有一个必需属性：value
  - 是String[]类型的，指定取消显示的警告集。警告类型有
    - unused　　未被使用的警告
    - deprecation　　使用了不赞成使用的类或方法时的警告
    - unchecked　　执行了未检查的转换时的警告
    - rawtypes　　没有用泛型 (Generics) 的警告
    - fallthrough　　当 Switch 程序块直接通往下一种情况而没有 Break 时的警告。
    - path　　在类路径、源文件路径等中有不存在的路径时的警告。
    - serial　　当在可序列化的类上缺少 serialVersionUID 定义时的警告。
    - finally　　任何 finally 子句不能正常完成时的警告。
    - all　　关于以上所有情况的警告。

示例

```
public class InternalAnnotationTest {
@SuppressWarnings(value={"unchecked", "deprecation"})
    public void test2() {
        Map map = new HashMap();
        map.put("a", "bbb");
        System.out.println(map);
    }
}
```

- 注解如果仅接收一个参数且名称是“value”，则可省略“value=“

```
public class InternalAnnotationTest {
    @SuppressWarnings({"unchecked", "deprecation"})
    public void test2() {
        Map map = new HashMap();
        map.put("a", "bbb");
        System.out.println(map);
    }
}
```

13.1.3 自定义注解类型

- 语法：

```
[访问修饰符] @interface 注解类型名 {
    数据类型 属性名() [default 默认值];      //定义属性
}
```

- 示例

```
public @interface MyDebug{
}

public @interface MyAnnotation{
    String value();
}

public @interfacle MyType{
    int age() default 18;
}
```

13.1.4元注解

- 元注解：对注解的注解
  - 为注解类型提供某种元数据，以便编译器保证按照预期的目的使用注解
- 使用系统预定义的元注解可以对你的注解进行注解
  - @Target　指定此注解的适用时机
  - @Retention　告知编译器如何处理此注解
  - @Documented　要求此注解成为API文件的一部分
  - @Inherited　指定子类是否继承父类的注解

@Target

- 在定义注解类型时，使用java.lang.annotation.Target可以定义其适用的时机。
  - 在定义时要指定为java.lang.annation.ElementType的枚举值之一：

```
package java.lang.annotation;

public enum ElementType{
    TYPE,                   //适用于 类,接口,枚举
    FIELD,                  //适用于 成员字段
    METHOD,                 //适用于 方法
    PARAMETER,              //适用于 方法的参数
    CONSTRUCTOR,            //适用于 构造方法
    LOCAL_VARIABLE,         //适用于 局部变量
    ANNOTATION_TYPE,        //适用于 注解类型
    PACKAGE                 //适用于 包
}
```

示例

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Target;

@Target({ElementType.CONSTRUCTOR, ElementType.METHOD})
public @interface MethodAnnotation {
}

@MethodAnnotation   //error
class TestMethodAnnotation{
    @MethodAnnotation
    public void test(){
    }
}
```

@Retention

- 使用java.lang.annotation.Retention用来告诉编译器如何处理当前注解。
  - 在使用Retention类型时，需要提供java.lang.annotation.RetentionPolicy的枚举类型，它的定义如下：

```
package java.lang.annotation;

public enum RetentionPolicy {
    SOURCE,      //编译器处理完后，并不将它保留到编译后的类文件中
    CLASS,       //编译器将注解保留在编译后的类文件中，但是在运行时忽略它
    RUNTIME      //编译器将注解保留在编译后的类文件中，并在第一次加载类时读取它
}
```

  - 内置注解中的Override、SuppressWarnings的retentionPolicy为SOURCE，而Deprecated为RUNTIME

@Documented

- 在默认情况下，注解不包括在Javadoc 中，用java.lang.annatation.Documented可以使此注解加入到Javadoc中。
- 定义为Documented的注解必须要设置Retention的值为RetentionPolicy.RUNTIME。

```
import java.lang.annotation.Documented;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;

@Documented
@Retention(RetentionPolicy.RUNTIME)
public @interface DocAnnotation {
}
```

@Inherited

- 定义的注解类型使用于程序代码上后，默认父类中的注解并不会继承至子类中。如果想让父类中的注解被继承到子类中，可以在定义注解类型时加上java.lang.annotation.Inherited类型的注解

```
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
public @interface InheritedAnno {
    String value();
    String name();
}

@InheritedAnno(value="abc", name="bcd")
public class Parent{}

class SubClass extends Parent{}
```

反射与注解

- 要让自定义注解对程序起作用，必须依赖反射来完成

13.1 反射

- Java反射API
- Class类
- 获取类信息
- 生成对象
- 调用方法
- 访问成员变量的值
- 操作数组
- 获取泛型信息
- 获取注解信息

反射概述

- 反射是Java语言的特征之一。它允许在运行时动态加载类、获取类信息、生成对象、操作对象的属性或方法等。主要提供了以下功能：
  - 在运行时判断任意一个对象所属的类；
  - 在运行时构造任意一个类的对象；
  - 在运行时判断任意一个类所具有的成员变量和方法；
  - 在运行时调用任意一个对象的方法。甚至是private方法。
  - 生成动态代理。

反射API

- java.lang.Class类：Class类的实例表示正在运行的Java应用程序中的类和接口。
- java.lang.reflect.Field类：提供有关类或接口的属性的信息，以及对它的动态访问权限。
- java.lang.reflect.Constructor类：提供关于类的单个构造方法的信息以及对它的访问权限。
- java.lang.reflect.Method类：提供关于类或接口上单独某个方法的信息。所反映的方法可能是类方法或实例方法（包括抽象方法）。
- java.lang.reflect.Array类：提供了动态创建数组和访问数组的静态方法。该类中的所有方法都是静态方法。

Class类

- Class类是Java反射的起源。
- 运行中的类或接口在JVM中都会有一个对应的Class对象存在，它保存了对应类或接口的类型信息。
  - 包括：基本数据类型、void、数组、enum、annotation
- JVM为每种类型管理着一个独一无二的Class对象

获取class对象

- 有三种方式可以获取每一个对象对应的Class对象：
  - 使用Object类中的getClass()方法。
  - 使用Class类的静态方法forName(String className);
  - 使用class常量。格式：类名.class;

```
public class ClassDemo {
    public static void main(String[] args) {
        String str = "java技术";
        Class stringClass = str.getClass();
        //Class stringClass = Class.forName("java.lang.String");
        //Class stringClass = String.class;
        System.out.println("类名：" + stringClass.getName());
        System.out.println("是否为接口：" + stringClass.isInterface());
        System.out.println("是否为基本类型：" + stringClass.isPrimitive());
        System.out.println("父类名称：" +  stringClass.getSuperclass().getName());
    }
}
```

通过Class获取类信息

- 获取类的包名： Package getPackage()
- 获取类的修饰符： int getModifiers()
  - 参见Modifier类
- 获取类的全限定名：String getName()
- 获取类的父类：Class<? super T> getSuperclass()
- 获取类实现的接口：Class<?>[] getInterfaces()

通过Class获取类信息

- 获取类的成员变量
  - Field[] getFields()　//仅公有的
  - Field[] getDeclaredFields()　//所有的
  - Field getField(String name)

通过Class获取类信息

- 获取类的构造方法
  - Constructor<?>[] getConstructors()
  - Constructor<?>[] getDeclaredConstructors()
  - Constructor<T> getConstructor(Class<?>... parameterTypes)
  - Constructor<T> getDeclaredConstructor(Class<?>... parameterTypes)

通过Class获取类信息

- 获取类的成员方法
  - Method[] getMethods()
  - Method[] getDeclaredMethods()
  - Method getMethod(String name, Class<?>... parameterTypes)
  - Method getDeclaredMethod(String name, Class<?>... parameterTypes)

示例

- 使用反射分析类的所有成员变量、构造方法和成员方法
  - ReflectionTest.java

使用反射生成对象

- 调用无参构造方法

```
Class c = Class.forName("java.util.ArrayList");
List list = (List) c.newInstance();
```

- 调用带参构造方法

```
//第1步：加载指定名称的类,获取对应的Class对象,
Class clazz = Class.forName("java.util.Date");
//第2步：获取具有指定参数类型的构造方法
Constructor constructor = clazz.getConstructor(long.class);
//第3步：给指定的构造方法传入参数值,创建出一个对象
Date date = (Date)constructor.newInstance(123456789000L);
```

操作成员变量值

- 使用反射取得成员变量的对象代表Field类的实例，再通过它的getXxx()方法来获取值；set()方法来修改值。

```
Class c = Class.forName("reflect.Student");
Object obj = c.newInstance();
Field sidField = c.getField("sid");
sidField.setInt(obj, 100);

//私有属性
Field nameField = c.getDeclaredField("name");
nameField.setAccessible(true);
nameField.set(obj, "张三");
System.out.println(obj);
```

调用方法

- 使用反射取得方法的对象代表Method类的实例，通过它的invoke()方法来动态调用这个方法。
  - 调用私有方法时，先通过调用setAccessible(true)取消 Java语言对本方法的访问检查。

```
Class c = Class.forName("entity.Student");
Object obj = c.newInstance(); //使用无参构造方法创建对象
Class[] param1 ={String.class}; //设定参数类型
//根据参数类型获得方法对象
Method addMethod = c.getMethod("setName", param1);
//给定参数调用指定对象的此 Method 对象表示的底层方法
addMethod.invoke(obj, "张三");

Method privateMethod = c.getDeclaredMethod("testPrivateMethod");  //私有方法
privateMethod.setAccessible(true);
privateMethod.invoke(obj);
System.out.println(obj);
```

获取泛型信息

- ParameterizedType这个接口，它提供的getActualTypeArguments()方法用来返回表示此类型实际类型参数的Type对象的数组

获取注解的相关信息

- public Annotation[] getAnnotations() //返回此元素上存在的所有注释
- public Annotation[] getDeclaredAnnotations() //返回直接存在于此元素上的所有注释。
- public <A extends Annotation> A getAnnotation(Class<A> annotationClass) //如果存在该元素的指定类型的注释，则返回这些注释，否则返回 null
- public boolean isAnnotationPresent(Class<? extends Annotation> annotationClass) //如果指定类型的注释存在于此元素上，则返回 true

总结