1. 注解
2. 概述

也叫元数据。一种代码级别的说明。它是JDK1.5及以后版本引入的一个特性，与类、接口、枚举是在同一个层次。它可以声明在包、类、字段、方法、局部变量、方法参数等的前面，用来对这些元素进行说明，注释。

1. 使用方法

@注解名称

1. 注解的作用
2. 编写文档

通过代码里标识的注解生成文档。

1. 代码分析

通过代码里标识的注解对代码进行分析

1. 编译检查

通过代码里标识的注解让编译器能够实现基本的编译检查（如@Override）

二．JDK内置注解

1. @Override

检测该注解标注的方法是否是继承自父类的。

1. @Deprecated

表示该注解标注的内容已经过时了，可以用但是不建议使用。

3，@SuppressWarnings

抑制警告，抑制已经显示的警告。一般传递参数all

@SuppressWarnings(“all”)

三 自定义注解

1. 格式

元注解

public @interface 注解名称{}

1. 注解的本质

注解本质上是一个接口，改接口默认继承Annotation接口

1. 注解的属性

注解中的属性即接口中的抽象方法，属性的返回值类型只能用以下的类型：

1. 基本数据类型
2. String
3. 枚举
4. 注解
5. 以上类型的数组

定义了属性后，在使用注解时需要给属性赋值。即给抽象方法的返回值赋值。

注意：

如果使用default关键字给属性默认初始化值，则使用注解时可以不赋值属性。

如果只有一个属性要赋值，该属性名称为value，则value可省略直接定义值。

四．元注解

（一）概述

预定义的注解，用于描述注解的注解。

（二）举例

1. @Target

用于描述注解能够作用的位置

TYPE:作用于类上

METHOD:作用于方法上

FIELD:作用于成员变量上

2．@Retention

描述注解被保留的阶段，三个阶段：源代码夹断，Class类阶段，运行阶段。

@Retention(RetentionPolicy.SOURCE)

@Retention(RetentionPolicy.CLASS)

@Retention(RetentionPolicy.RUNTIME)

1. @Documented

描述注解是否被抽取到api文档中显示。

1. @Inherited

描述注解是否被子类继承

五．注解的使用

注解通常用于取代配置文件，通过解析注解即可获取注解中定义的属性。

1. 定义注解：

@Target({Element.Type.TYPE})

@Retention(RetentionPolicy.RUNTIME)

Public @interface Pro{

String className();

String methodName();

}

1. 定义类

@Pro(className = “xx.xx”, methodName = “xx”)

Public static class ReTest{}

我们要在ReTest类中利用注解取得相应的className和methodName，需要通过以下的步骤来获取。

1. 解析注解

获取该类的字节码文件对象

Class<ReTest> ret = ReTest.class;

1. 获取注解定义位置的对象

获取注解对象就是在内存中生成了该注解接口的子类实现对象。

Pro an = ret.getAnnotation(Pro.class)

1. 调用注解对象中定义的抽象方法，获取返回值

String className = an.className();

String methodName = an.methodName();

六．注解用于搭建简单的测试框架

（一）需求

将注解加于方法之上，即可测试该方法是否会出现异常，并将异常记录在文件中。

（二）代码

测试计算器类Calculator中的方法是有会出现异常，将@Check注解放置于方法上，有注解的方法被测试类TestCheck测试，测试结果放在bug.txt文本文件中。

每个文件的代码如下：

1. 计算器类Calculator

package 注解实现测试框架;  
  
public class Calculator {  
 //加法  
 @Check  
 public void add(){  
 System.*out*.println("1 + 0 =" + (1 + 0));  
 }  
  
 //减法  
 @Check  
 public void sub(){  
 System.*out*.println("1- 0 = " + (1-0));  
 }  
  
 //除法  
 @Check  
 public void div(){  
 System.*out*.println("1 / 0 = " + (1 / 0));  
 }  
}

2.自定义注解@Check

package 注解实现测试框架;  
  
import java.lang.annotation.ElementType;  
import java.lang.annotation.Retention;  
import java.lang.annotation.RetentionPolicy;  
import java.lang.annotation.Target;  
  
@Retention(RetentionPolicy.*RUNTIME*)  
@Target(ElementType.*METHOD*)  
public @interface Check {  
  
}

3.测试类

package 注解实现测试框架;  
  
import java.io.BufferedWriter;  
import java.io.FileWriter;  
import java.io.IOException;  
import java.lang.reflect.InvocationTargetException;  
import java.lang.reflect.Method;  
  
public class TestCheck {  
 public static void main(String[] args) throws IOException {  
 Calculator c = new Calculator();  
 //获取该类的字节码文件对象  
 Class<? extends Calculator> cls = c.getClass();  
 //获取所有方法  
 Method[] methods = cls.getMethods();  
  
 int num = 0;  
 BufferedWriter bw = new BufferedWriter(new FileWriter("bug.txt"));  
  
 for(Method method : methods){  
 //判断方法上是否有Check注解  
 if(method.isAnnotationPresent(Check.class)){  
 //有Check注解，执行  
 try{  
 method.invoke(c);  
 } catch (Exception e) {  
 //捕获异常，记录到文件中  
  
 num++;  
 bw.write(method.getName() + "方法出现异常");  
 bw.newLine();  
 bw.write("异常的名称：" + e.getCause());  
 bw.newLine();  
 bw.write("异常的原因：" + e.getMessage());  
 bw.newLine();  
 bw.write("----------------");  
 bw.newLine();  
 }  
 }  
 }  
  
 bw.write("本次测试一共出现" + num + "次异常");  
  
 bw.flush();  
 bw.close();  
  
 }  
}