基础加强

今日内容介绍

- 使用自定义注解模仿@Test
- 使用动态代理解决网站字符集编码

今日内容学习目标

- 知道 JDK 提供的三个注解作用
- 学会注解的使用
- 了解注解的定义与解析
- 可以使用 Proxy 编写动态代理类
- 知道类加载器的作用，以及应用场景

第1章 案例：自定义注解模拟@Test

1.1 案例介绍

使用 Junit 是单元测试的工具.在一个类中使用 @Test 对程序中的方法进行测试.
自定义一个注解@MyTest 也将这个注解加在类的方法上. 使这个方法得到执行.

1.2 案例相关知识：注解

1.2.1 概述

- 什么是注解：Annotation 注解，是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
  - 对比注释：注释是给开发人员阅读的，注解是给计算机提供相应信息的。
- 注解的作用：
  1. 编译检查：通过代码里标识注解，让编译器能够实现基本的编译检查。例如：@Override
  2. 代码分析：通过代码里标识注解，对代码进行分析，从而达到取代 xml 目的。
  3. 编写文档：通过代码里标识注解，辅助生成帮助文档对应的内容
-

1.2.2 JDK 提供的注解

1. @Deprecated 表示被修饰的方法已经过时。过时的方法不建议使用，但仍可以使用。
   - 一般被标记位过时的方法都存在不同的缺陷：1 安全问题；2 新的 API 取代
2. @Override JDK5.0 表示复写父类的方法；jdk6.0 还可以表示实现接口的方法
3. @SuppressWarnings 表示抑制警告，被修饰的类或方法如果存在编译警告，将被编译器忽略

   deprecation ，或略过时

   rawtypes ，忽略类型安全

   unused ， 忽略不使用

   unchecked ，忽略安全检查

   null，忽略空指针

   all，忽略所有

- @Deprecated

```
//#1 方法过期
class Parent1_1{
    @Deprecated
    public void init(){

    }
}
```

```
 7 //#1 方法过期
 8 class Parent1_1{
 9     @Deprecated
10     public void init(){
11
12     }
13 }
```

- @Override 复写父类方法

```
//#2.1 JDK5.0 复写父类方法
class Parent1_2{
    public void init(){

    }
}
class Son1_2 extends Parent1_2{
    @Override
    public void init() {
    }
}
```

- @Override 实现接口方法

```
//#2.2 JDK6.0 实现父接口方法
interface Parent1_3{
    public void init();
}
class Son1_3 implements Parent1_3{
    @Override
    public void init() {

    }
}
```

- @SupperssWarings

```
//#3 抑制警告
// serial : 实现序列号接口，但没有生产序列号
@SuppressWarnings("serial")
class Parent1_4 implements java.io.Serializable{

    //null : 空指针
    @SuppressWarnings("null")
    public void init(){

        //rawtypes : 类型安全，没有使用泛型
        //unused : 不使用
        @SuppressWarnings({ "rawtypes", "unused" })
        List list = new ArrayList();


        String str = null;

        str.toString();
```

```
    }


}
```

```
63 class Parent1_5 implements java.io.Serializable{
64
65     public void init(){      没有抑制警告
66
67         List list = new ArrayList();
68
69         String str = null;
70         str.toString();
71     }
72 }
```

1.2.3 自定义注解：定义—基本语法

- 定义注解使用关键字： **@interface**
  1. 定义类： **class**
  2. 定义接口：**interface**
  3. 定义枚举：**enum**

```
// #1 定义注解
@interface MyAnno1{


}
```

- 定义带有属性的注解

```
//#2 定义含有属性的注解
@interface MyAnno2{
    public String username() default "jack";
}
```

- 属性格式：修饰符　返回值类型　属性名()　[**default** 默认值]
  1. 修饰符：默认值 **public abstract** ，且只能是 **public abstract**。

     Illegal modifier for the annotation attribute Anno2_2.username; only public & abstract are permitted

  2. 返回值类型：基本类型、字符串 **String**、**Class**、注解、枚举，以及以上类型的一维数组

     Invalid type Date for the annotation attribute Anno2_2.username; only primitive type, String, Class, annotation, enumeration are permitted or 1-dimensional arrays thereof

  3. 属性名：自定义
  4. **default** 默认值：可以省略
- 完整案例

```
//#3 完整含属性注解
@interface MyAnno3{
    int age() default 1;
```

```
    String password();
    Class clazz();
    MyAnno2 myAnno(); // 注解
    Color color(); // 枚举
    String[] arrs();
}
enum Color{
    BLUE,RED;
}
```

1.2.4 自定义注解：使用

- 使用格式：@注解类名(属性名= 值 , 属性名 = 值 ,.....)

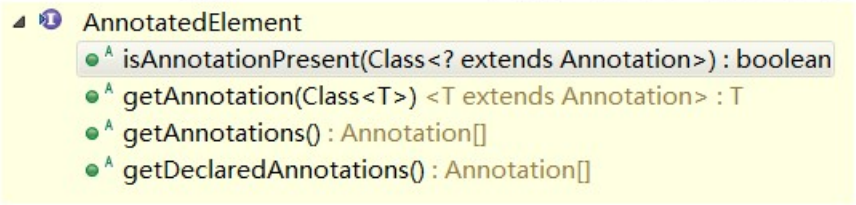
```
@MyAnno1
@MyAnno2(username="rose")
@MyAnno3(
        age=18 ,
        password="1234" ,
        clazz=String.class ,
        myAnno=@MyAnno2 ,
        color = Color.RED ,
        arrs = {"itcast","itheima"}
)
public class TestAnno2 {
```

- 注解使用的注意事项：
  - 注解可以没有属性，如果有属性需要使用小括号括住。例如：@MyAnno1 或 @MyAnno1()
  - 属性格式：属性名=属性值，多个属性使用逗号分隔。例如：@MyAnno2(username="rose")
  - 如果属性名为 value，且当前只有一个属性，value 可以省略。
  - 如果使用多个属性时，k 的名称为 value 不能省略
  - 如果属性类型为数组，设置内容格式为：{1,2,3 }。例如：arrs = {"itcast","itheima"}
  - 如果属性类型为数组，值只有一个{} 可以省略的。例如：arrs = "itcast"
  - 一个对象上，注解只能使用一次，不能重复使用。

1.2.5 自定义注解：解析

- 如果给类、方法等添加注解，如果需要获得注解上设置的数据，那么我们就必须对注解进行解析，JDK 提供 java.lang.reflect.AnnotatedElement 接口允许在运行时通过反射获得注解。

- 常用方法：
  - boolean isAnnotationPresent(Class annotationClass) 当前对象是否有注解
  - T getAnnotation(Class<T> annotationClass) 获得当前对象上指定的注解
  - Annotation[] getAnnotations() 获得当前对象及其从父类上继承的，所有的注解
  - Annotation[] getDeclaredAnnotations() 获得当前对象上所有的注解



- 测试

```
@MyAnno1
public class TestAnno2 {
    public static void main(String[] args) {
        boolean b = TestAnno2.class.isAnnotationPresent(MyAnno1.class);
        System.out.println(b); //false
    }
}
```

当运行上面程序后，我们希望输出结果是 true，但实际是 false。TestAnno2 类上有@MyAnno1 注解，但运行后不能获得，因为每一个自定义注解，需要使用 JDK 提供的元注解进行修饰才可以真正的使用。

1.2.6 自定义注解：定义—元注解

- 元注解：用于修饰注解的注解。（用于修饰自定义注解的 JDK 提供的注解）
- JDK 提供 4 种元注解：
  - @Retention 用于确定被修饰的自定义注解生命周期
    - RetentionPolicy.SOURCE 被修饰的注解只能存在源码中，字节码 class 没有。用途：提供给编译器使用。
    - RetentionPolicy.CLASS 被修饰的注解只能存在源码和字节码中，运行时内存中没有。用途：JVM java 虚拟机使用
    - RetentionPolicy.RUNTIME 被修饰的注解存在源码、字节码、内存（运行时）。用途：取代 xml 配置
  - @Target 用于确定被修饰的自定义注解 使用位置
    - ElementType.TYPE 修饰 类、接口
    - ElementType.CONSTRUCTOR 修饰构造

- ElementType.METHOD 修饰方法
- ElementType.FIELD 修饰字段

- @Documented 使用 javaDoc 生成 api 文档时，是否包含此注解 (了解)
- @Inherited 如果父类使用被修饰的注解，子类是否继承。（了解）

注释类型摘要	
Documented	指示某一类型的注释将通过 javadoc 和类似的默认工具进行文档化。
Inherited	指示注释类型被自动继承。
Retention	指示注释类型的注释要保留多久。
Target	指示注释类型所适用的程序元素的种类。

枚举摘要	
ElementType	程序元素类型。
RetentionPolicy	注释保留策略。

- 修改注解类，在运行测试实例，输出结果为：true。

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno1{


}
```

1.3 案例分析

- 模拟 Junit 测试，首先需要编写自定义注解@MyTest，并添加元注解，保证自定义注解只能修改方法，且在运行时可以获得。
- 其次编写目标类（测试类），然后给目标方法（测试方法）使用@MyTest 注解
- 最后编写测试类，使用 main 方法模拟 Junit 的右键运行。

1.4 案例实现

- 步骤 1：编写自定义注解类@MyTest

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface MyTest {


}
```

- 步骤 2：编写目标类 AnnotationDemo

```
public class AnnotationDemo {
```

```
    @MyTest
    public void demo1(){
        System.out.println("demo1 执行了...");
    }
    @MyTest
    public void demo2(){
        System.out.println("demo2 执行了...");
    }
    public void demo3(){
        System.out.println("demo3 执行了...");
    }
}
```

- 步骤 3：编写测试方法

```
public class App {
    public static void main(String[] args) {
        try {
            //1.1 反射：获得类的字节码对象.Class
            Class clazz = AnnotationDemo.class;
            //1.2 获得实例对象
            Object obj = clazz.newInstance();

            //2 获得目标类所有的方法
            Method[] allMethod = clazz.getMethods();
            //3 遍历所有的方法
            for (Method method : allMethod) {
                //3.1 判断方法是否有 MyTest 注解
                boolean flag = method.isAnnotationPresent(MyTest.class);
                if (flag) {
                    //4 如果有注解运行指定的类
                    method.invoke(obj, args);
                }
            }
        } catch (Exception e) {
            e.printStackTrace();
        }

        /* 输出结果：
         * demo1 执行了...
         * demo2 执行了...
         */
    }
}
```

第2章 使用动态代理解决网站的字符集编码

2.1 介绍

学习过滤器时，我们使用“装饰者”对 request 进行增强，从而使 get 和 post 使用 request.getParameter() 获得的数据都没有乱码。本案例我们将使用一个全新的技术—动态代理，对“统一 GET 和 POST 乱码”案例进行重写。

2.2 相关知识点：Proxy

- Proxy.newProxyInstance
  - 参数 1：loader ，类加载器，动态代理类 运行时创建，任何类都需要类加载器将其加载到内存。
    - 一般情况：当前类.class.getClassLoader();
  - 参数 2：Class[] interfaces 代理类需要实现的所有接口
    - 方式 1：目标类实例.getClass().getInterfaces();
      注意：只能获得自己接口，不能获得父元素接口
    - 方式 2：new Class[]{UserService.class}
      例如：jdbc 驱动 --> DriverManager 获得接口 Connection
  - 参数 3：InvocationHandler 处理类，接口，必须进行实现类，一般采用匿名内部
    - 提供 invoke 方法，代理类的每一个方法执行时，都将调用一次 invoke
      - 参数 31：Object proxy ：代理对象
      - 参数 32：Method method：代理对象当前执行的方法的描述对象（反射）
        执行方法名：method.getName()
        执行方法：method.invoke(对象，实际参数)
      - 参数 33：Object[] args :方法实际参数

2.3 案例实现

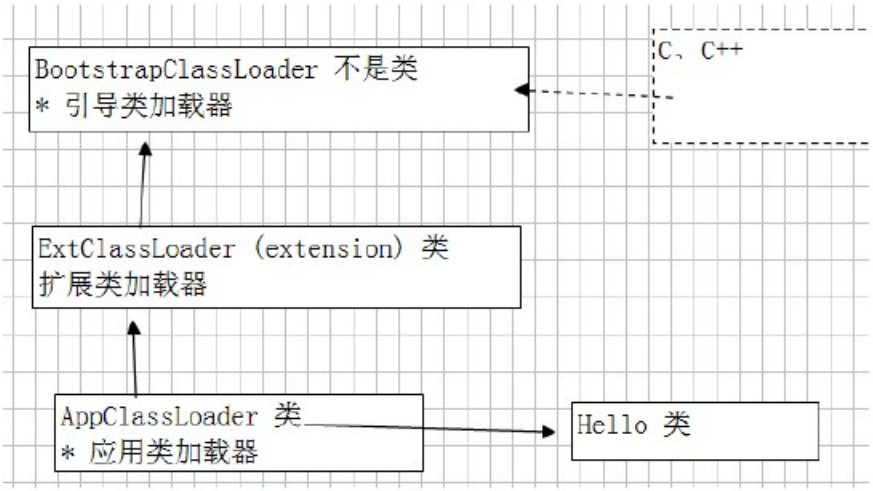
```
@WebFilter("/*")
public class EncodingFilter implements Filter {

    @Override
    public void init(FilterConfig filterConfig) throws ServletException {

    }

    @Override
```

```
        public void doFilter(ServletRequest req, ServletResponse response, FilterChain
chain)
                throws IOException, ServletException {

            final HttpServletRequest request = (HttpServletRequest) req;

            HttpServletRequest                    requestProxy                    =
(HttpServletRequest)Proxy.newProxyInstance(
                    EncodingFilter.class.getClassLoader(),
                    new Class[]{HttpServletRequest.class},
                    new InvocationHandler() {
                    @Override
                    public Object invoke(Object proxy, Method method, Object[] args)
throws Throwable {
                        if("get".equals(request.getMethod()))  {
                            //对指定方法进行增强
                            if("getParameter".equals(method.getName())){
                                // 执行方法获得返回值
                                String value = (String) method.invoke(request, args);
                                return new String(value.getBytes("UTF-8") , "UTF-8");
                            }
                        }
                        //放行
                        return method.invoke(request, args);
                    }
            });
            //放行
            chain.doFilter(requestProxy, response);

        }

        @Override
        public void destroy() {

        }

    }
```

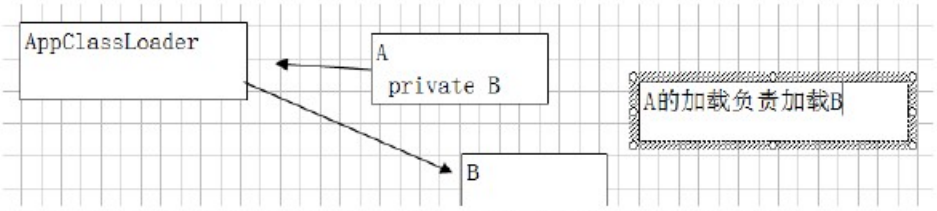
2.4总结

2.4.1 类加载器

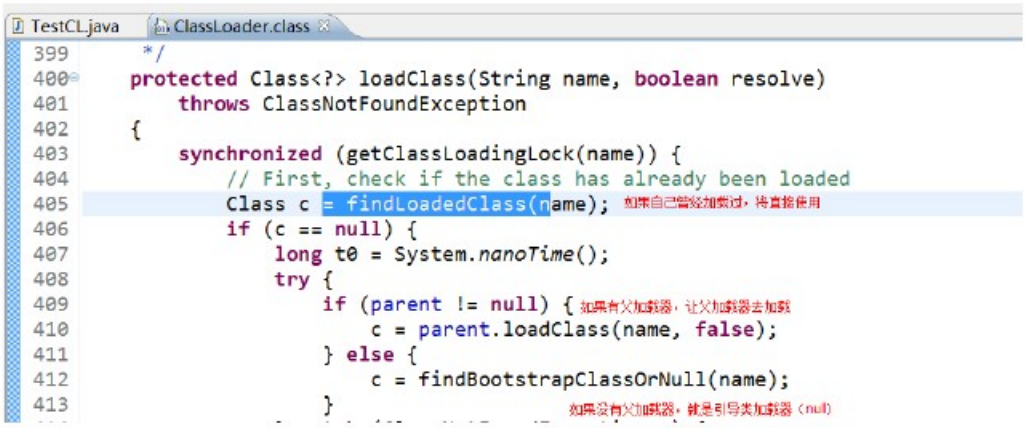
- 类加载器：类加载器是负责加载类的对象。将 class 文件（硬盘）加载到内存生成 Class 对象。
  所有的类加载器 都是 java.lang.ClassLoader 的子类



- 使用 类.class.getClassLoader() 获得加载自己的类加载器
- 类加载器加载机制：全盘负责委托机制
  全盘负责：A 类如果要使用 B 类（不存在），A 类加载器 C 必须负责加载 B 类。



  委托机制：A 类加载器如果要加载资源 B，必须询问父类加载是否加载。
  如果加载，将直接使用。
  如果没有机制，自己再加载。
- 采用 全盘负责委托机制 保证 一个 class 文件 只会被加载一次，形成一个 Class 对象。
- 注意：
  如果一个 class 文件，被两个类加载器加载，将是两个对象。
  提示 com.itheima.Hello 不能强制成 com.itheima.Hello
  h.getClass() -->A h.getClass() -->B
  自定义类加载，可以将一个 class 文件加载多次。


TestCL.java
ClassLoader.class
399 */
400 protected Class<?> loadClass(String name, boolean resolve)
401 throws ClassNotFoundException
402 {
403 synchronized (getClassLoadingLock(name)) {
404 // First, check if the class has already been loaded
405 Class c = findLoadedClass(name);
406 if (c == null) {
407 long t0 = System.nanoTime();
408 try {
409 if (parent != null) {
410 c = parent.loadClass(name, false);
411 } else {
412 c = findBootstrapClassOrNull(name);
413 }

第3章 总结