day26【反射、注解】

今日内容

- 反射
- 注解

学习目标

- [ ] 能够通过反射技术获取Class字节码对象。
- [ ] 能够通过反射技术获取构造方法对象，并创建对象。
- [ ] 能够通过反射获取成员方法对象，并且调用方法
- [ ] 能够通过反射获取属性对象，并且能够给对象的属性赋值和取值
- [ ] 能够说出注解的作用
- [ ] 能够自定义注解和使用注解
- [ ] 能够说出常用的元注解及其作用
- [ ] 能够解析注解并获取注解中的数据
- [ ] 能够完成注解的MyTest案例

第一章 反射

1.1 类加载器

1.1.1 类的加载

- 当我们的程序在运行后，第一次使用某个类的时候，会将此类的class文件读取到内存，并将此类的所有信息存储到一个Class对象中。

说明：

> 1. 上图：Class对象是指：java.lang.Class类的对象，此类由Java类库提供，专门用于存储类的信息。
> 2. 我们程序中可以通过："类名.class"，或者"对象.getClass()"方法获取这个Class对象。

1.1.2 类的加载时机

1. 创建类的实例。
2. 调用类的静态变量，或者为静态变量赋值。
3. 调用类的静态方法。
4. 使用反射方式来强制创建某个类或接口对应的java.lang.Class对象。
5. 初始化某个类的子类。
6. 直接使用java.exe命令来运行某个主类。

   以上六种情况的任何一种，都可以导致JVM将一个类加载到方法区。

1.1.3 类加载器

类加载器：是负责将磁盘上的某个class文件读取到内存并生成Class的对象。

- Java中有三种类加载器，它们分别用于加载不同种类的class：
  - 启动类加载器(Bootstrap ClassLoader)：用于加载系统类库<JAVA_HOME>\bin目录下的class，例如：rt.jar。
  - 扩展类加载器(Extension ClassLoader)：用于加载扩展类库<JAVA_HOME>\lib\ext目录下的class。
  - 应用程序类加载器(Application ClassLoader)：用于加载我们自定义类的加载器。

```
public class Test{
    public static void main(String[] args){

System.out.println(Test.class.getClassLoader());//sun.misc.Launcher$AppClassLoad
er
        System.out.println(String.class.getClassLoader());//null(API中说明：一些实现
可能使用null来表示引导类加载器。 如果此类由引导类加载器加载，则此方法将在此类实现中返回null。
)
    }
}
```

1.1.4 双亲委派机制

上图展示了"类加载器"的层次关系，这种关系称为类加载器的"双亲委派模型"：

- "双亲委派模型"中，除了顶层的启动类加载器外，其余的类加载器都应当有自己的"父级类加载器"。
- 这种关系不是通过"继承"实现的，通常是通过"组合"实现的。通过"组合"来表示父级类加载器。
- "双亲委派模型"的工作过程：
  - 某个"类加载器"收到类加载的请求，它首先不会尝试自己去加载这个类，而是把请求交给父级类加载器。
  - 因此，所有的类加载的请求最终都会传送到顶层的"启动类加载器"中。
  - 如果"父级类加载器"无法加载这个类，然后子级类加载器再去加载。

1.1.5 双亲委派机制的好处

双亲委派机制的一个显而易见的好处是：Java的类随着它的类加载器一起具备了一种带有优先级的层次关系。例如：java.lang.Object。它存放在rt.jar中。无论哪一个类加载器要加载这个类，最终都是委派给处于顶端的"启动类加载器"进行加载，因此java.lang.Object类在程序的各种类加载器环境中都是同一个类。

相反，如果没有"双亲委派机制"，如果用户自己编写了一个java.lang.Object，那么当我们编写其它类时，这种隐式的继承使用的将会是用户自己编写的java.lang.Object类，那将变得一片混乱。

1.2 反射的概述

1.2.1 反射的引入

- 问题：IDEA中的对象是怎么知道类有哪些属性，哪些方法的呢?

通过反射技术对象类进行了解剖得到了类的所有成员。

1.2.2 反射的概念

反射是一种机制，利用该机制可以在程序运行过程中对类进行解剖并操作类中的所有成员(成员变量，成员方法，构造方法)。

1.2.3 使用反射操作类成员的前提

要获得该类字节码文件对象，就是Class对象。

1.2.4 反射在实际开发中的应用

```
* 开发IDEA(集成开发环境)，比如IDEA,Eclipse
* 各种框架的设计和学习 比如Spring，Hibernate，Struct，Mybaits....
```

1.3 Class对象的获取方式

1.3.1 三种获取方法

```
* 方式1：通过类名.class获得
* 方式2：通过对象名.getClass()方法获得
* 方式3：通过Class类的静态方法获得： static Class forName("类全名")
    * 每一个类的Class对象都只有一个。
```

- 示例代码

```
package com.itheima._03反射;
public class Student{

}
```

```
public class ReflectDemo01 {
    public static void main(String[] args) throws ClassNotFoundException {
        // 获得Student类对应的Class对象
        Class c1 = Student.class;

        // 创建学生对象
        Student stu = new Student();
        // 通过getClass方法
        Class c2 = stu.getClass();
        System.out.println(c1 == c2);

        // 通过Class类的静态方法获得： static Class forName("类全名")
        Class c3 = Class.forName("com.itheima._03反射.Student");
        System.out.println(c1 == c3);
        System.out.println(c2 == c3);
```

```
    }
}
```

1.3.2 Class类常用方法

```
String getSimpleName(); 获得类名字符串：类名
String getName();  获得类全名：包名+类名
T newInstance() ;  创建Class对象关联类的对象
```

- 示例代码

```
public class ReflectDemo02 {
    public static void main(String[] args) throws Exception {
        // 获得Class对象
        Class c = Student.class;
        // 获得类名字符串：类名
        System.out.println(c.getSimpleName());
        // 获得类全名：包名+类名
        System.out.println(c.getName());
        // 创建对象
        Student stu = (Student) c.newInstance();
        System.out.println(stu);
    }
}
```

1.4 反射之操作构造方法

1.4.1 Constructor类概述

```
反射之操作构造方法的目的
    * 获得Constructor对象来创建类的对象。

Constructor类概述
    * 类中的每一个构造方法都是一个Constructor类的对象
```

1.4.2 Class类中与Constructor相关的方法

```
1. Constructor getConstructor(Class... parameterTypes)
        * 根据参数类型获得对应的Constructor对象。
        * 只能获得public修饰的构造方法
 2. Constructor getDeclaredConstructor(Class... parameterTypes)
        * 根据参数类型获得对应的Constructor对象
        * 可以是public、protected、(默认)、private修饰符的构造方法。
 3. Constructor[] getConstructors()
        * 获得类中的所有构造方法对象，只能获得public的
 4. Constructor[] getDeclaredConstructors()
        * 获得类中的所有构造方法对象
        * 可以是public、protected、(默认)、private修饰符的构造方法。
```

1.4.3 Constructor对象常用方法

1. T newInstance(Object... initargs)"--
   根据指定的参数创建对象
2. void setAccessible(true)
   设置"暴力反射"--是否取消权限检查，true取消权限检查，false表示不取消

1.4.4 示例代码

```
public class Student{
    private String name;
    private String sex;
    private int age;

    //公有构造方法
    public Student(String name,String sex,int age){
        this.name = name;
        this.sex = sex;
        this.age = age;
    }
    //私有构造方法
    private Student(String name,String sex){
        this.name = name;
        this.sex = sex;
    }
}
```

```
public class ReflectDemo03 {

    /*
      Constructor[] getConstructors()
           获得类中的所有构造方法对象，只能获得public的
      Constructor[] getDeclaredConstructors()
            获得类中的所有构造方法对象，包括private修饰的
    */
    @Test
    public void test03() throws Exception {
        // 获得Class对象
        Class c = Student.class;
        //  获得类中的所有构造方法对象，只能获得public的
        // Constructor[] cons = c.getConstructors();
        // 获取类中所有的构造方法，包括public、protected、(默认)、private的
        Constructor[] cons = c.getDeclaredConstructors();
        for (Constructor con:cons) {
            System.out.println(con);
        }
    }

    /*
       Constructor getDeclaredConstructor(Class... parameterTypes)
           根据参数类型获得对应的Constructor对象
    */
    @Test
    public void test02() throws Exception {
        // 获得Class对象
```

```
        Class c = Student.class;
        // 获得两个参数构造方法对象
        Constructor con = c.getDeclaredConstructor(String.class,String.class);
        // 取消权限检查(暴力反射)
        con.setAccessible(true);
        // 根据构造方法创建对象
        Object obj = con.newInstance("rose","女");
        System.out.println(obj);
    }

    /*
        Constructor getConstructor(Class... parameterTypes)
            根据参数类型获得对应的Constructor对象
     */
    @Test
    public void test01() throws Exception {
        // 获得Class对象
        Class c = Student.class;
        // 获得无参数构造方法对象
        Constructor con = c.getConstructor();
        // 根据构造方法创建对象
        Object obj = con.newInstance();
        System.out.println(obj);

        // 获得有参数的构造方法对象
        Constructor con2 = c.getConstructor(String.class,
String.class,int.class);
        // 创建对象
        Object obj2 = con2.newInstance("jack", "男",18);
        System.out.println(obj2);
    }
}
```

1.5 反射之操作成员方法

1.5.1 Method类概述

反射之操作成员方法的目的

* 操作Method对象来调用成员方法

Method类概述

* 每一个成员方法都是一个Method类的对象。

1.5.2 Class类中与Method相关的方法

```
* Method getMethod(String name,Class...args);
    * 根据方法名和参数类型获得对应的构造方法对象，只能获得public的

* Method getDeclaredMethod(String name,Class...args);
    * 根据方法名和参数类型获得对应的构造方法对象，包括public、protected、(默认)、private的

* Method[] getMethods();
    * 获得类中的所有成员方法对象，返回数组，只能获得public修饰的且包含父类的

* Method[] getDeclaredMethods();
    * 获得类中的所有成员方法对象，返回数组,只获得本类的，包括public、protected、(默认)、
private的
```

1.5.3 Method对象常用方法

```
*  Object invoke(Object obj, Object... args)
    * 调用指定对象obj的该方法
    * args：调用方法时传递的参数
*  void setAccessible(true)
    * 设置"暴力访问"--是否取消权限检查，true取消权限检查，false表示不取消
```

1.5.4 示例代码

```
public class Student{
    private void eat(String str){
        System.out.println("我吃：" + str);
    }

    private void sleep(){
        System.out.println("我睡觉...");
    }

    public void study(int a){
        System.out.println("我学习Java，参数a = " + a);
    }
}
```

```
public class ReflectDemo04 {

    // 反射操作静态方法
    @Test
    public void test04() throws Exception {
        // 获得Class对象
        Class c = Student.class;
        // 根据方法名获得对应的公有成员方法对象
        Method method = c.getDeclaredMethod("eat",String.class);
        // 通过method执行对应的方法
        method.invoke(null,"蛋炒饭");
    }

    /*
     * Method[] getMethods();
```

```
     * 获得类中的所有成员方法对象，返回数组，只能获得public修饰的且包含父类的
   * Method[] getDeclaredMethods();
     * 获得类中的所有成员方法对象，返回数组,只获得本类的，包含private修饰的
 */
 @Test
 public void test03() throws Exception {
     // 获得Class对象
     Class c = Student.class;
     // 获得类中的所有成员方法对象，返回数组，只能获得public修饰的且包含父类的
     // Method[] methods = c.getMethods();
     // 获得类中的所有成员方法对象，返回数组,只获得本类的，包含private修饰的
     Method[] methods = c.getDeclaredMethods();
     for (Method m: methods) {
         System.out.println(m);
     }

 }

 /*
    Method getDeclaredMethod(String name,Class...args);
        * 根据方法名和参数类型获得对应的构造方法对象，
 */
 @Test
 public void test02() throws Exception {
     // 获得Class对象
     Class c = Student.class;

     // 根据Class对象创建学生对象
     Student stu = (Student) c.newInstance();
     // 获得sleep方法对应的Method对象
     Method m =  c.getDeclaredMethod("sleep");
     // 暴力反射
     m.setAccessible(true);

     // 通过m对象执行stuy方法
     m.invoke(stu);
 }

 /*
     Method getMethod(String name,Class...args);
         * 根据方法名和参数类型获得对应的构造方法对象，
  */
 @Test
 public void test01() throws Exception {
     // 获得Class对象
     Class c = Student.class;

     // 根据Class对象创建学生对象
     Student stu = (Student) c.newInstance();
     // 获得study方法对应的Method对象
     Method m =  c.getMethod("study");
     // 通过m对象执行stuy方法
     m.invoke(stu);


     /// 获得study方法对应的Method对象
     Method m2  = c.getMethod("study", int.class);
     // 通过m2对象执行stuy方法
```

```
        m2.invoke(stu,8);
    }
}
```

1.6 反射之操作成员变量【自学】

1.6.1 Field类概述

```
反射之操作成员变量的目的
    * 通过Field对象给对应的成员变量赋值和取值

Field类概述
    * 每一个成员变量都是一个Field类的对象。
```

1.6.2 Class类中与Field相关的方法

```
* Field getField(String name);
    *  根据成员变量名获得对应Field对象，只能获得public修饰
* Field getDeclaredField(String name);
    *  根据成员变量名获得对应Field对象，包括public、protected、(默认)、private的
* Field[] getFields();
    * 获得所有的成员变量对应的Field对象，只能获得public的
* Field[] getDeclaredFields();
    * 获得所有的成员变量对应的Field对象，包括public、protected、(默认)、private的
```

1.6.3 Field对象常用方法

```
void  set(Object obj, Object value)
void setInt(Object obj, int i)
void setLong(Object obj, long l)
void setBoolean(Object obj, boolean z)
void setDouble(Object obj, double d)

Object get(Object obj)
int getInt(Object obj)
long getLong(Object obj)
boolean getBoolean(Object ob)
double getDouble(Object obj)

void setAccessible(true);// 暴力反射，设置为可以直接访问私有类型的属性。
Class getType(); // 获取属性的类型，返回Class对象。
```

> setXxx方法都是给对象obj的属性设置使用，针对不同的类型选取不同的方法。
>
> getXxx方法是获取对象obj对应的属性值的，针对不同的类型选取不同的方法。

1.6.4 示例代码

```java
public class Student{
    public String name;
    private String gender;

    public String toString(){
        return "Student [name = " + name + " , gender = " + gender + "]";
    }
}
```

```java
public class ReflectDemo05 {
    /*
        Field[] getFields();
            * 获得所有的成员变量对应的Field对象，只能获得public的
        Field[] getDeclaredFields();
            * 获得所有的成员变量对应的Field对象，包含private的
     */
    @Test
    public void test02() throws Exception {
        // 获得Class对象
        Class c  = Student.class;
        // 获得所有的成员变量对应的Field对象
        // Field[] fields = c.getFields();
        // 获得所有的成员变量对应的Field对象，包括private
        Field[] fields = c.getDeclaredFields();
        for (Field f: fields) {
            System.out.println(f);
        }
    }

    /*
        Field getField(String name);
            根据成员变量名获得对应Field对象，只能获得public修饰
        Field getDeclaredField(String name);
            *  根据成员变量名获得对应Field对象，包含private修饰的
     */
    @Test
    public void test01() throws Exception {
        // 获得Class对象
        Class c  = Student.class;
        // 创建对象
        Object obj = c.newInstance();
        // 获得成员变量name对应的Field对象
        Field f = c.getField("name");
        // 给成员变量name赋值
        // 给指定对象obj的name属性赋值为jack
        f.set(obj,"jack");

        // 获得指定对象obj成员变量name的值
        System.out.println(f.get(obj)); // jack
        // 获得成员变量的名字
        System.out.println(f.getName()); // name


        // 给成员变量gender赋值
```

```
        // 获得成员变量gender对应的Field对象
        Field f1 = c.getDeclaredField("gender");
        // 暴力反射
        f1.setAccessible(true);
        // 给指定对象obj的gender属性赋值为男
        f1.set(obj,"男");

        System.out.println(obj);

    }
}
```

第二章 注解

2.1 注解的概述

2.1.1 注解的概念

- 注解是JDK1.5的新特性。
- 注解相当一种标记，是类的组成部分，可以给类携带一些额外的信息。
- 标记(注解)可以加在包，类，字段，方法，方法参数以及局部变量上。
- 注解是给编译器或JVM看的，编译器或JVM可以根据注解来完成对应的功能。

> ***注解(Annotation)相当于一种标记，在程序中加入注解就等于为程序打上某种标记，以后，javac编译器、开发工具和其他程序可以通过反射来了解你的类及各种元素上有无何种 标记，看你的程序有什么标记，就去干相应的事，标记可以加在包、类，属性、方法，方法的参数以及局部变量上。***

2.1.2 注解的作用

注解的作用就是给程序带入参数。

以下几个常用操作中都使用到了注解：

1. **生成帮助文档**：**@author和@version**
    - **@author**：用来标识作者姓名。
    - **@version**：用于标识对象的版本号，适用范围：文件、类、方法。
        - 使用**@author和@version**注解就是告诉**Javadoc工具**在生成帮助文档时把作者姓名和版本号也标记在文档中。如下图：

          
2. **编译检查**：**@Override**
    - **@Override**：用来修饰方法声明。
        - 用来告诉**编译器**该方法是重写父类中的方法，如果父类不存在该方法，则编译失败。如下图

      
3. **框架的配置(框架=代码+配置)**
    - 具体使用请关注框架课程的内容的学习。

2.1.3 常见注解

1. **@author**：用来标识作者名，eclipse开发工具默认的是系统用户名。
2. **@version**：用于标识对象的版本号，适用范围：文件、类、方法。
3. **@Override** ：用来修饰方法声明，告诉编译器该方法是重写父类中的方法，如果父类不存在该方法，则编译失败。

2.2 自定义注解

2.2.1 定义格式

```
public @interface 注解名{

}
如：定义一个名为Student的注解
public @interface Student {

}
```

2.2.2 注解的属性

1. **属性的格式**
   - **格式1：数据类型 属性名();**
   - **格式2：数据类型 属性名() default 默认值;**
2. **属性定义示例**

```
// 姓名
String name();
// 年龄
int age() default 18;
// 爱好
String[] hobby();
```

3. **属性适用的数据类型**

```
* 八种数据数据类型(int，short，long，double，byte，char，boolean，float)
* String，Class，注解类型，枚举类
* 以上类型的数组形式
```

2.3 使用自定义注解

2.3.1 定义和注解

1. **定义一个注解：Book**
   - 包含属性：String value() 书名
   - 包含属性：double price() 价格，默认值为 100
   - 包含属性：String[] authors() 多位作者
2. 代码实现

```
@Target({ElementType.METHOD,ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
public @interface Book {
    String value();
    double price() default 100;
    String[] authros();
}
```

2.3.2 使用注解

1. **定义类在成员方法上使用Book注解**

```
public class AnnotationDemo01 {
    @Book(value = "JavaEE开发详解",authros = {"黑马程序员","传智学院"})
    public void show(){

    }
}
```

2. 使用注意事项
    - 如果属性有默认值，则使用注解的时候，这个属性可以不用赋值。
    - 如果属性没有默认值，那么在使用注解时一定要给属性赋值。

2.3.3 特殊属性value

```
/**
    特殊属性value
        * 如果注解中只有一个属性且名字叫value，则在使用该注解时可以直接给该属性赋值，而不需要给出属性名。
        * 如果注解中除了value属性之外还有其他属性且只要有一个属性没有默认值，则在给属性赋值时value属性名也不能省略了。
    小结：如果注解中只有一个属性时，一般都会将该属性名命名为value
 */
@interface TestA{
   String[] value();
   int age() default 100;
   String name();
}

@interface TestB{
    String name();
}

@TestB(name = "zzz")
@TestA(name = "yyy",value = {"xxx","xxx"})
public class AnnotationDemo02 {

}
```

2.4 注解之元注解

元注解概述

* Java官方提供的注解
* 用来定义注解的注解
* 任何官方提供的非元注解的定义都使用到了元注解。

常用的元注解

* @Target
  * 作用：用来标识注解使用的位置，如果没有使用该注解标识，则自定义的注解可以使用在任意位置。
  * 可使用的值定义在ElementType枚举类中，常用值如下
    TYPE，类，接口
    FIELD，成员变量
    METHOD，成员方法
    PARAMETER，方法参数
    CONSTRUCTOR，构造方法
    LOCAL_VARIABLE，局部变量

* @Retention
  * 作用：用来标识注解的生命周期(有效范围)
  * 可使用的值定义在RetentionPolicy枚举类中，常用值如下
    * SOURCE：注解只作用在源码阶段，生成的字节码文件中不存在
    * CLASS：注解作用在源码阶段，字节码文件阶段，运行阶段不存在，默认值
    * RUNTIME：注解作用在源码阶段，字节码文件阶段，运行阶段

2.5 注解解析

什么是注解解析

* 使用Java技术获得注解上数据的过程则称为注解解析。

与注解解析相关的接口

* Annotation：注解类，该类是所有注解的父类。
* AnnotatedElement:该接口定义了与注解解析相关的方法
  T getAnnotation(Class<T> annotationClass) 根据注解类型获得对应注解对象
  Annotation[] getAnnotations()
    * 获得当前对象上使用的所有注解，返回注解数组，包含父类继承的
  Annotation[] getDeclaredAnnotations()
    * 获得当前对象上使用的所有注解，返回注解数组,只包含本类的
  boolean isAnnotationPresent(Class<Annotation> annotationClass)
    * 判断当前对象是否使用了指定的注解，如果使用了则返回true，否则false

获取注解数据的原理

* 注解作用在哪个成员上就会得该成员对应的对象来获得注解
  * 比如注解作用成员方法，则要获得该成员方法对应的Method对象
  * 比如注解作用在类上，则要该类的Class对象
  * 比如注解作用在成员变量上，则要获得该成员变量对应的Field对象。
* Field,Method,Constructor，Class等类都是实现了AnnotatedElement接口

2.5.1 需求说明

1. 定义注解Book，要求如下：
   - 包含属性：String value() 书名
   - 包含属性：double price() 价格，默认值为 100
   - 包含属性：String[] authors() 多位作者
   - 限制注解使用的位置：类和成员方法上

- 指定注解的有效范围：RUNTIME

2. 定义BookStore类，在类和成员方法上使用Book注解
3. 定义TestAnnotation测试类获取Book注解上的数据

2.5.2 代码实现

1. **注解Book**

```
@Target({ElementType.METHOD,ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
public @interface Book {
    String value();
    double price() default 100;
    String[] authros();
}
```

1. BookShelf类

```
@Book(value = "红楼梦",authros = {"曹雪芹"})
public class BookShelf  {

    @Book(value = "西游记",authros = {"吴承恩","白求恩"},price = 200)
    public void showBook(){

    }
}
```

1. **TestAnnotation类**

```
/**
    什么是注解解析
        * 使用Java技术获得注解上数据的过程则称为注解解析。
    与注解解析相关的接口
        * Annotation: 注解类，该类是所有注解的父类。
        * AnnotatedElement:该接口定义了与注解解析相关的方法
            T getAnnotation(Class<T> annotationClass) 根据注解类型获得对应注解对象
            Annotation[]   getAnnotations()
               * 获得当前对象上使用的所有注解，返回注解数组，包含父类继承的
            Annotation[]   getDeclaredAnnotations()
               * 获得当前对象上使用的所有注解，返回注解数组,只包含本类的
            boolean    isAnnotationPresent(Class<Annotation> annotationClass)
               * 判断当前对象是否使用了指定的注解，如果使用了则返回true，否则false

        获取注解数据的原理
            * 注解作用在哪个成员上就会得该成员对应的对象来获得注解
               * 比如注解作用成员方法，则要获得该成员方法对应的Method对象
               * 比如注解作用在类上，则要该类的Class对象
               * 比如注解作用在成员变量上，则要获得该成员变量对应的Field对象。
            * Field,Method,Constructor，Class等类都是实现了AnnotatedElement接口
 */
public class AnnotationDemo04 {
    /*
        获得类上使用的注解数据
     */
    @Test
```

```
    public void test02() throws Exception {
        // 获得Class对象
        Class c = BookShelf.class;
        // 判断类上是否使用Book注解
        if(c.isAnnotationPresent(Book.class)){
            // 根据注解的Class对象获得对应的注解对象
            Book annotation = (Book) c.getAnnotation(Book.class);
            // 获得书名
            System.out.println(annotation.value());
            // 获得作者
            System.out.println(Arrays.toString(annotation.authros()));
            // 获得价格
            System.out.println(annotation.price());
        }
        // 获得当前对象上使用的所有注解，返回注解数组
        // Annotation[] annotations = c.getAnnotations();
        Annotation[] annotations = c.getDeclaredAnnotations();
        System.out.println(Arrays.toString(annotations));
    }

    /*
       获得成员方法上注解的数据
    */
    @Test
    public void test01() throws Exception {
        // 获得Class对象
        Class c = BookShelf.class;
        // 获得成员方法对应的Method对象
        Method m = c.getMethod("showBook");
        // 根据注解的Class对象获得对应的注解对象
        Book annotation = m.getAnnotation(Book.class);
        // 获得书名
        System.out.println(annotation.value());
        // 获得作者
        System.out.println(Arrays.toString(annotation.authros()));
        // 获得价格
        System.out.println(annotation.price());
    }
}
```

2.6 注解案例

2.6.1 案例说明

- 模拟Junit测试的@Test

2.6.2 案例分析

1. 模拟Junit测试的注释@Test，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饰方法，且在运行时可以获得。
2. 然后编写目标类（测试类），然后给目标方法（测试方法）使用 @MyTest注解，编写三个方法，其中两个加上@MyTest注解。
3. 最后编写调用类，使用main方法调用目标类，模拟Junit的运行，只要有@MyTest注释的方法都会运行。

2.6.3 案例代码

```
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 MyTest {

}

public class TestMyTest {

    @MyTest
    public void tests01(){
        System.out.println("test01");
    }


    public void tests02(){
        System.out.println("test02");
    }

    @MyTest
    public void tests03(){
        System.out.println("test03");
    }
}

/**
 * @author pkxing
 * @version 1.0
 * @Package com.itheima
 */
public class AnnotationDemo05 {
    public static void main(String[] args) throws Exception {
        // 获得Class对象
        Class c = TestMyTest.class;
        Object obj = c.newInstance();
        // 获得所有成员方法
        Method[] methods = c.getMethods();
        for(Method m:methods){
            // 判断m方法是否使用了MyTest注解
            if(m.isAnnotationPresent(MyTest.class)){
                // 调用方法
                m.invoke(obj);
            }
        }
    }
}
```