day13【JUnit单元测试、反射、注解、动态代理、JDK8新特性】

今日内容

- JUnit单元测试------->必须掌握
  - 使用步骤
  - 会使用JUnit里面的常用注解
- 反射------->必须掌握------>相对难点
  - 获取类的Class对象------->重要
  - 反射操作构造方法------->重要
  - 反射操作成员方法------->重要
  - 反射操作成员变量
- 注解------->必须掌握
  - 使用注解------->重要
  - 自定义注解以及注解解析
- 动态代理------->必须掌握------->难点
  - 代理模式------->重要
  - 如何动态生成代理对象------->重要
- JDK8新特性
  - 方法引用----->理解
  - Base64编码和解码------->必须掌握

第一章 Junit单元测试

1.1 Junit单元测试

- 概述: Junit是Java语言编写的第三方单元测试框架(工具类)
- 作用: 用来做"单元测试"——针对某个普通方法，可以像main()方法一样独立运行，它专门用于测试某个方法。
- 使用步骤:
  - 导入Junit的jar包
  - 把jar包添加到classpath路径中
  - 书写测试方法
  - 在测试方法的上面写上@Test注解
  - 执行测试方法
- 执行:
  - 选中方法名---->右键--->选中执行　　执行当前选中的测试方法
  - 选中类名------->右键--->选中执行　　执行当前类中所有的测试方法
  - 选中模块------->右键--->选中执行,选择all Tests　执行当前模块中所有的测试方法
  - 执行结果查看:
    - 绿色: 表示执行成功
    - 红色: 表示执行失败
- 案例:

```
public class Demo1 {

    @Test
    public void test1(){
        System.out.println("test1...");
    }

    @Test
    public void test2(){
        System.out.println("test2...");
    }

}
```

1.2 Junit单元测试的注意事项

- Junit的测试方法必须没有参数
- Junit的测试方法必须没有返回值
- Junit的测试方法必须使用public修饰
- Junit的测试方法必须使用@Test注解修饰

1.3 Junit其他注解

- @Before：用来修饰方法，该方法会在每一个测试方法执行之前执行一次。
- @After：用来修饰方法，该方法会在每一个测试方法执行之后执行一次。
- @BeforeClass：用来修饰静态方法，该方法会在所有测试方法之前执行一次，而且只执行一次。
- @AfterClass：用来修饰静态方法，该方法会在所有测试方法之后执行一次，而且只执行一次。

```
public class Demo {
    /*
        - @Before：用来修饰方法，该方法会在每一个测试方法执行之前执行一次。
        - @After：用来修饰方法，该方法会在每一个测试方法执行之后执行一次。
        - @BeforeClass：用来修饰静态方法，该方法会在所有测试方法之前执行一次，而且只执行一
次。
        - @AfterClass：用来修饰静态方法，该方法会在所有测试方法之后执行一次，而且只执行一次
     */
    @BeforeClass
    public static void bc1(){
        System.out.println("bc1...");
    }


    @Before
    public void b1(){
        System.out.println("b1...");
    }

    @Test
    public void test1(){
        System.out.println("test1...");
    }
```

```
    @Test
    public void test2(){
        System.out.println("test2...");
    }

    @After
    public void a1(){
        System.out.println("a1...");
    }

    @AfterClass
    public static void ac1(){
        System.out.println("ac1...");
    }

}
```

1.4 Junit断言

- 断言：预先判断某个条件一定成立，如果条件不成立，则直接报错。 使用Assert类中的assertEquals()方法
- 案例:

```
public class Demo {
    @Test
    public void test1() {
        int sum = getSum(10, 20);

        // 断言：预先判断某个条件一定成立，如果条件不成立，则直接报错。 使用Assert类中的assertEquals()方法
        // 参数1: 期望的值
        // 参数2: 实际的值
        // 如果时间的值和期望的值相等,就不报错,否则就直接报错
        Assert.assertEquals(30,sum);

        sum += 10;
        System.out.println("sum:" + sum);// sum: 40
    }


    public int getSum(int num1, int num2) {
        return num1 * num2;
    }

}
```

第二章 反射

2.1 类加载器

- 作用：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
- 类加载时机：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
- 获取类加载器：
  - 通过类的Class对象调用getClassLoader()方法获取类加载器：类名.class.getClassLoader()

```
public class Test {
    public static void main(String[] args) {
        System.out.println("Person类的类加载器:
"+Person.class.getClassLoader());
    }
}
```

2.2 扩展类加载器结合Properties的使用

```
public class Utils {

    public static String driverClass;
    public static String jdbcUrl;
    public static String username;
    public static String password;

    static {
        try {
            // 1.创建Properties对象
            Properties pro = new Properties();

            // 2.调用load方法,加载配置文件---一定要在src路径下
            //FileInputStream is = new
FileInputStream("day13\\src\\db.properties");
            // 返回的流默认就已经到达了src路径
            InputStream is =
Utils.class.getClassLoader().getResourceAsStream("db.properties");
            pro.load(is);

            // 3.取出数据
            driverClass = pro.getProperty("driverClass");
            jdbcUrl = pro.getProperty("jdbcUrl");
            username = pro.getProperty("username");
            password = pro.getProperty("password");

        } catch (IOException e) {

        }
    }
```

```
}
```

2.3 反射的概述

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)

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

反射在实际开发中的应用

* 开发IDE(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....

2.4 获取Class对象

- 通过类名.class获得
- 通过对象名.getClass()方法获得
- 通过Class类的静态方法获得： static Class forName("类全名")

- 示例代码

```
public class Test {
    public static void main(String[] args)throws Exception {
        // 1.方式一: 类名.class
        Class<Person> c1 = Person.class;

        // 2.方式二: 对象名.getClass();
        Person p = new Person();
        Class<? extends Person> c2 = p.getClass();

        // 3.方式三: Class.forName("类的全名(包名+类名)")
        Class<?> c3 = Class.forName("com.itheima.demo6_反射获取字节码对
象.Person");

        // 注意: 类的字节码对象在内存中只有一份，多次获取，拿到的都是同一个字节码对象
        System.out.println(c1);
        System.out.println(c2);
        System.out.println(c3);
        System.out.println(c1==c2);// true
        System.out.println(c2==c3);// true


    }
```

```
}
```

2.5 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---相当于调用该类的空参构造方法
```

- 示例代码

```
public class Test {
    public static void main(String[] args) throws Exception{
        // 获取Person类字节码对象
        Class<Person> c = Person.class;

        //String getSimpleName(); 获得类名字符串：类名
        System.out.println("类名: "+c.getSimpleName());

        //String getName();  获得类全名：包名+类名
        System.out.println("类的全名: "+c.getName());

        //T newInstance() ;  创建Class对象关联类的对象---相当于调用该类的空参构造方
法
        Person p = c.newInstance();// 相当于执行了new Person();

    }
}
```

2.6 反射之操作构造方法

Constructor类概述

```
Constructor类概述
    * 在反射中,类中的每一个构造方法都会封装成一个Constructor类的对象
```

通过反射获取类的构造方法

使用Class类的方法：

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)---推荐
   * 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()---推荐
   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。

```
public class Test1_通过反射获取类的构造方法 {

    public static void main(String[] args) throws Exception {
        // 获取类的Class对象
        Class<Person> c = Person.class;

        // 需求1:通过反射获取第1个构造方法
        Constructor<Person> con1 = c.getDeclaredConstructor();
        System.out.println("con1:" + con1);

        // 需求2:通过反射获取第2个构造方法
        Constructor<Person> con2 = c.getDeclaredConstructor(String.class,
int.class);
        System.out.println("con2:" + con2);

        // 需求3:通过反射获取第3个构造方法
        Constructor<Person> con3 = c.getDeclaredConstructor(int.class);
        System.out.println("con3:" + con3);

        // 需求4:通过反射获取第4个构造方法
        Constructor<Person> con4 = c.getDeclaredConstructor(String.class);
        System.out.println("con4:" + con4);

        // 需求5:通过反射获取所有构造方法
        Constructor<?>[] arr = c.getDeclaredConstructors();
        for (Constructor<?> con : arr) {
            System.out.println("con:" + con);
        }

    }

}
```

通过反射执行构造方法

Constructor类的常用方法
1. T newInstance(Object... initargs)
    根据指定的参数创建对象
    参数:被执行的构造方法需要的实际参数
2. void setAccessible(boolean b)
   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

```
public class Test2_通过反射执行类的构造方法 {

    public static void main(String[] args) throws Exception {
        // 获取类的Class对象
        Class<Person> c = Person.class;

        // 需求1:通过反射获取第1个构造方法
        Constructor<Person> con1 = c.getDeclaredConstructor();
        System.out.println("con1:" + con1);

        // 需求2:通过反射获取第2个构造方法
        Constructor<Person> con2 = c.getDeclaredConstructor(String.class,
int.class);
        System.out.println("con2:" + con2);

        // 需求3:通过反射获取第3个构造方法
        Constructor<Person> con3 = c.getDeclaredConstructor(int.class);
        System.out.println("con3:" + con3);

        // 需求4:通过反射获取第4个构造方法
        Constructor<Person> con4 = c.getDeclaredConstructor(String.class);
        System.out.println("con4:" + con4);

        System.out.println("--------------------");

        // 需求: 通过反射执行con1表示的构造方法来创建Person对象
        Person p1 = con1.newInstance();
        System.out.println("p1:" + p1);

        // 需求: 通过反射执行con2表示的构造方法来创建Person对象
        Person p2 = con2.newInstance("张三", 18);
        System.out.println("p2:" + p2);

        // 需求: 通过反射执行con3表示的构造方法来创建Person对象
        Person p3 = con3.newInstance(19);
        System.out.println("p3:" + p3);

        // 需求: 通过反射执行con4表示的构造方法来创建Person对象
        // 暴力反射:取消权限检查
        con4.setAccessible(true);
        Person p4 = con4.newInstance("王五");
        System.out.println("p4:" + p4);

    }

}
```

2.7 反射之操作成员方法

Method类概述

```
Method类概述
    * 在反射里面,类的每一个成员方法都会封装成一个Method类的对象。
```

通过反射获取类的成员方法

```
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成员方法对象，只能获得public的

* Method getDeclaredMethod(String name,Class... args);----->推荐
    * 根据方法名和参数类型获得对应的成员方法对象，包括public、protected、(默认)、private的
      参数1:要获取的方法的方法名
      参数2:要获取的方法的形参类型的Class对象

* 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();----->推荐
    * 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
```

```java
public class Test1_通过反射获取类的成员方法 {
    public static void main(String[] args) throws Exception {
        // 获取Person类的Class对象
        Class<Person> c = Person.class;

        // 需求: 通过反射获取第1个成员方法
        Method show1M = c.getDeclaredMethod("show1");
        System.out.println("show1M:" + show1M);

        // 需求: 通过反射获取第2个成员方法
        Method show2M = c.getDeclaredMethod("show2", int.class, String.class);
        System.out.println("show2M:" + show2M);

        // 需求: 通过反射获取第3个成员方法
        Method show3M = c.getDeclaredMethod("show3", int.class);
        System.out.println("show3M:" + show3M);

        // 需求: 通过反射获取第4个成员方法
        Method show4M = c.getDeclaredMethod("show4");
        System.out.println("show4M:" + show4M);

        // 需求: 通过反射获取第5个成员方法
        Method show5M = c.getDeclaredMethod("show1", int.class, String.class);
        System.out.println("show5M:" + show5M);

        System.out.println("-------------");
        // 需求: 通过反射获取所有成员方法
        Method[] arr = c.getDeclaredMethods();
        for (Method m : arr) {
            System.out.println("m:" + m);
```

```
        }

    }
}
```

通过反射执行成员方法

```
Method对象常用方法
*  Object invoke(Object obj, Object... args) 执行Method对象封装的成员方法
    * 参数1:调用该方法的对象
    * 参数2:调用该法时传递的实际参数
      返回值:该方法执行完毕后的返回值

*  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
```

```
public class Test2_通过反射执行类的成员方法 {
    public static void main(String[] args) throws Exception {
        // 获取Person类的Class对象
        Class<Person> c = Person.class;

        // 需求: 通过反射获取第1个成员方法
        Method show1M = c.getDeclaredMethod("show1");
        System.out.println("show1M:" + show1M);

        // 需求: 通过反射获取第2个成员方法
        Method show2M = c.getDeclaredMethod("show2", int.class, String.class);
        System.out.println("show2M:" + show2M);

        // 需求: 通过反射获取第3个成员方法
        Method show3M = c.getDeclaredMethod("show3", int.class);
        System.out.println("show3M:" + show3M);

        // 需求: 通过反射获取第4个成员方法
        Method show4M = c.getDeclaredMethod("show4");
        System.out.println("show4M:" + show4M);

        // 需求: 通过反射获取第5个成员方法
        Method show5M = c.getDeclaredMethod("show1", int.class, String.class);
        System.out.println("show5M:" + show5M);

        System.out.println("-------------");

        // 通过反射获取Person类的对象
        Person p = c.getDeclaredConstructor().newInstance();

        // 需求: 通过反射执行show1M封装的成员方法
        show1M.invoke(p);// 相当于执行 p.show1();

        // 需求: 通过反射执行show2M封装的成员方法
        Object res1 = show2M.invoke(p, 100, "java");// 相当于执行  res1 =
p.show2(100, "java");
        System.out.println("res1:"+res1);
```

```
        // 需求: 通过反射执行show3M封装的成员方法
        show3M.invoke(p,200);// 相当于执行 p.show3(200);

        // 需求: 通过反射执行show4M封装的成员方法
        Object res2 = show4M.invoke(p);// 相当于执行 res2 = p.show4()
        System.out.println("res2:"+res2);

        // 需求: 通过反射执行show5M封装的成员方法
        show5M.setAccessible(true);
        Object res3 = show5M.invoke(p, 300, "php");// 相当于执行  res3 =
p.show1(300, "php")
        System.out.println("res3:"+res3);

    }
}
```

2.8 反射之操作成员变量【自学】

Field类概述

```
Field类概述
    * 在反射里面,类中的每一个成员变量都会封装成一个Field类的对象。
```

通过反射获取类的成员变量

```
Class类中的方法
* 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
       参数:属性名
* Field getDeclaredField(String name);----->推荐
    *  根据成员变量名获得对应Field对象，包括public、protected、(默认)、private的
       参数:属性名

* 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();---->推荐
    * 获得所有的成员变量对应的Field对象，包括public、protected、(默认)、private的
```

```
public class Test1_通过反射获取类的成员变量 {
    public static void main(String[] args) throws Exception {
        // 获取Person类的Class对象
        Class<Person> c = Person.class;

        // 需求: 通过反射获取第1个成员变量
        Field f1 = c.getDeclaredField("name");
        System.out.println("f1:" + f1);

        // 需求: 通过反射获取第2个成员变量
        Field f2 = c.getDeclaredField("age");
        System.out.println("f2:" + f2);
```

```
        // 需求: 通过反射获取第3个成员变量
        Field f3 = c.getDeclaredField("sex");
        System.out.println("f3:" + f3);

        System.out.println("-----");

        // 需求: 通过反射获取所有成员变量
        Field[] arr = c.getDeclaredFields();
        for (Field field : arr) {
            System.out.println("field:" + field);
        }
    }
}
```

通过反射访问成员变量

```
Field对象常用方法
    给对象的属性赋值的方法
void  set(Object obj, Object value) ----->推荐
    参数1: 给哪个对象的属性赋值---该类的对象
    参数2: 给属性赋的值

   获取对象属性的值的方法
Object get(Object obj)  ----->推荐

void setAccessible(true);暴力反射，设置为可以直接访问私有类型的属性。 ----->推荐
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。

```
public class Test2_通过反射操作类的成员变量 {
    public static void main(String[] args) throws Exception {
        // 获取Person类的Class对象
        Class<Person> c = Person.class;

        // 需求: 通过反射获取第1个成员变量
        Field f1 = c.getDeclaredField("name");
        System.out.println("f1:" + f1);

        // 需求: 通过反射获取第2个成员变量
        Field f2 = c.getDeclaredField("age");
        System.out.println("f2:" + f2);

        // 需求: 通过反射获取第3个成员变量
        Field f3 = c.getDeclaredField("sex");
        System.out.println("f3:" + f3);

        System.out.println("-----");

        // 通过反射创建Person类的对象
        Person p = c.getDeclaredConstructor().newInstance();
```

```
        // 需求: 通过反射给name属性赋值并取值
        f1.set(p, "张三");
        System.out.println("p对象的name属性值:" + f1.get(p) + ",name属性的类型:" +
f1.getType());

        // 需求: 通过反射给age属性赋值并取值
        f2.set(p,18);
        System.out.println("p对象的age属性值:" + f2.get(p) + ",age属性的类型:" +
f2.getType());

        // 需求: 通过反射给sex属性赋值并取值
        f3.setAccessible(true);// 取消权限检查
        f3.set(p,"男");
        System.out.println("p对象的sex属性值:" + f3.get(p) + ",sex属性的类型:" +
f3.getType());
    }
}
```

第三章 注解

3.1 注解概述

注解概述

- 注解(annotation),是一种代码级别的说明,和类 接口平级关系.
  - 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标记，看你的程序有什么标记，就去干相应的事
  - 我们之前使用过的注解：

    1).@Override：子类重写方法时——编译时起作用

    2).@FunctionalInterface：函数式接口——编译时起作用

    3).@Test：JUnit的测试注解——运行时起作用

注解的作用

- 生成帮助文档：@author和@version
- 执行编译期的检查 例如:@Override
- 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。

3.2 JDK提供的三个基本的注解

@Override:描述方法的重写.

@SuppressWarnings:压制\忽略警告.

@Deprecated:标记过时

```java
@SuppressWarnings("all")
public class Test {
    /*
        @Override:描述方法的重写.
        @SuppressWarnings:压制\忽略警告.
        @Deprecated:标记过时
     */
    @Override
    public String toString() {
        return super.toString();
    }

    public static void main(String[] args) {
        int num;
    }

    @Deprecated
    public static void show1(){
        System.out.println("过时的方法...");
    }

    public static void show2(){
        System.out.println("更新的方法...");
    }

}
```

3.3 自定义注解

- 格式:

```java
public @interface 注解名{
    // 定义属性
}
```

- 属性: `数据类型 属性名();`
    - 基本类型
    - String类型
    - Class类型
    - 注解类型
    - 枚举类型
    - 以及以上五种类型的一维数组类型
- 案例:
    - 定义一个不带属性的注解

```java
// 不带属性的注解
public @interface MyAnnotation01 {
}
```

- 定义一个带属性的注解

```
// 带属性的注解
public @interface MyAnnotation02 {
    // 属性: 数据类型 属性名();
    // - 基本类型
    int num();
    double numD();

    // - String类型
    String str();

    // - Class类型
    Class c();

    // - 注解类型
    MyAnnotation01 ma();

    // - 枚举类型
    Gender g();

    // - 以及以上五种类型的一维数组类型
    int[] arr1();
    String[] arr2();
    Class[] arr3();
    MyAnnotation01[] arr4();
    Gender[] arr5();

}
```

3.4 使用注解并给注解属性赋值

- 使用注解格式:

```
不带属性的注解: @注解名  或者@注解名()
带属性的注解:  @注解名(属性名=属性值,属性名=属性值,...)
注意:
    1.使用带属性的注解必须给注解中所有的属性赋值
    2.某个位置使用同一个注解只能使用一次,不能多次使用
```

- 案例演示

```
@MyAnnotation01
@MyAnnotation02(name="张三",age=18,arr={3.14,4.2,5.6})
public class Test {

    @MyAnnotation01
    @MyAnnotation02(name="张三",age=18,arr={3.14,4.2,5.6})
    int num;

    @MyAnnotation01
```

```
    @MyAnnotation02(name="张三",age=18,arr={3.14,4.2,5.6})
    public static void main(String[] args) {
        @MyAnnotation01
        @MyAnnotation02(name="张三",age=18,arr={3.14,4.2,5.6})
        int num;
    }
}
```

3.5 使用注解的注意事项

- 一旦注解有属性了,使用注解的时候,属性必须有值
- 若属性类型是一维数组的时候,当数组的值只有一个的时候可以省略{}
- 如果注解中只有一个属性,并且属性名为value,那么使用注解给注解属性赋值的时候,注解属性名value可以省略
- 注解属性可以有默认值 格式:属性类型 属性名() default 默认值;

```
public @interface MyAnnotation01 {
    String name();
    int age();
}

public @interface MyAnnotation02 {
    String[] names();
    int age();
}

public @interface MyAnnotation03 {
    String value();
}

public @interface MyAnnotation033 {
    String[] value();
}

public @interface MyAnnotation04 {
    String name() default "张三";
    int age() default 18;
}
```

```
public class Test {
    //  - 一旦注解有属性了,使用注解的时候,属性必须有值
    @MyAnnotation01(name = "zs", age = 18)
    public void method1() {
    }

    //  - 若属性类型是一维数组的时候,当数组的值只有一个的时候可以省略{}
    //@MyAnnotation02(names={"itheima"},age=15) // 标准
    @MyAnnotation02(names="itheima",age=15) // 省略
    public void method2(){

    }
```

```
    //  - 如果注解中只有一个属性,并且属性名为value,那么使用注解给注解属性赋值的时候,注解属性名value可以省略
    //@MyAnnotation03(value="itcast")// 标准
    @MyAnnotation03("itcast")// 标准
    public void method3(){

    }

    //  - 若属性类型是一维数组的时候,当数组的值只有一个的时候可以省略{}
    //  - 如果注解中只有一个属性,并且属性名为value,那么使用注解给注解属性赋值的时候,注解属性名value可以省略
    //@MyAnnotation033(value={"java"}) // 标准
    //@MyAnnotation033(value="java") // 省略
    @MyAnnotation033("java") // 省略
    public void method33(){

    }

    //  - 注解属性可以有默认值  格式:属性类型 属性名() default 默认值;
    //@MyAnnotation04     // 可以不赋值,使用默认值
    @MyAnnotation04(name="小三",age=19)    // 可以赋值,其实就是修改值
    public void method04(){

    }

}
```

3.6 元注解

什么是元注解

定义在注解上的注解(修饰注解的注解)

常见的元注解

@Target:表示该注解作用在什么上面(位置),**默认注解可以在任何位置**. 值为:ElementType的枚举值

METHOD:方法

TYPE:类 接口

FIELD:字段

CONSTRUCTOR:构造方法声明

LOCAL_VARIABLE:局部变量

....

```
// 不带属性注解
//@Target(value={ElementType.METHOD,ElementType.FIELD}) // 当前注解只能使用在方法或者成员变量上
//@Target(value={ElementType.TYPE})
//@Target(ElementType.TYPE)// 当前注解只能使用在类上
//如果没有@Target注解修饰,默认任何位置都可以使用当前注解
public @interface MyAnnotation01 {
}
```

@Retention:定义该注解保留到那个代码阶段, 值为:RetentionPolicy类型,==**默认只在源码阶段保留**==

SOURCE:只在源码上保留(默认)

CLASS:在源码和字节码上保留

RUNTIME:在所有的阶段都保留

.java (源码阶段) ----编译---> **.class(字节码阶段)** ----加载内存--> **运行(RUNTIME)**

3.7 注解解析

java.lang.reflect.AnnotatedElement接口: Class、Method、Field、Constructor等实现了AnnotatedElement

- **T getAnnotation(Class annotationType):得到指定类型的注解对象。没有返回null。**
- **boolean isAnnotationPresent(Class<?extends Annotation> annotationType)**：判断指定的注解有没有。

```
public class Test {

    public static void main(String[] args) throws Exception {
        /*
            java.lang.reflect.AnnotatedElement接口: Class、Method、Field、Constructor等实现了AnnotatedElement
            - T getAnnotation(Class<T>   annotationType):得到指定类型的注解对象。没有返回null。
            - boolean isAnnotationPresent(Class<?extends Annotation> annotationType)：判断指定的注解有没有。
         */
        // 需求1:获取show1方法上MyAnnotation01注解属性的值
        // 1.获取Test类的字节码对象
        Class<Test> c = Test.class;

        // 2.获取show1方法对应的Method对象
        Method show1M = c.getDeclaredMethod("show1");

        // 3.使用Method对象调用getAnnotation方法获得show1方法上的MyAnnotation01注解对象
        MyAnnotation01 annotation = show1M.getAnnotation(MyAnnotation01.class);

        // 4.使用MyAnnotation01注解对象获取name和age属性的值
        System.out.println(annotation.name() + "," + annotation.age());
```

```
        // 需求2: 判断show1和show2方法上是否有MyAnnotation01注解
        boolean res1 = show1M.isAnnotationPresent(MyAnnotation01.class);
        System.out.println("res1:"+res1);// res1: true

        Method show2M = c.getDeclaredMethod("show2");
        boolean res2 = show2M.isAnnotationPresent(MyAnnotation01.class);
        System.out.println("res2:"+res2);// res2: false
    }

    @MyAnnotation01(name = "张三", age = 18)
    public void show1() {

    }

    public void show2() {

    }
}
```

3.8 完成注解的MyTest案例

需求

在一个类(测试类,TestDemo)中有三个方法,其中两个方法上有@MyTest,另一个没有.还有一个主测试类(MainTest)中有一个main方法. 在main方法中,让TestDemo类中含有@MyTest方法执行. 自定义@MyTest, 模拟单元测试.

分析

代码实现

- MyTest.java

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

- TestDemo.java

```
public class TestDemo {

    @MyTest
    public void test1(){
        System.out.println("test1...");
    }

    @MyTest
    public void test2(){
        System.out.println("test2...");
```

```
    }

    public void test3(){
        System.out.println("test3...");
    }

}
```

- MainTest.java

```
public class MainTest {
    public static void main(String[] args) throws Exception {
        // 1.获取TestDemo类的字节码对象
        Class<TestDemo> c = TestDemo.class;

        // 2.获取该类中所有的方法
        Method[] arr = c.getDeclaredMethods();

        // 通过反射创建TestDemo对象
        TestDemo td = c.getDeclaredConstructor().newInstance();

        // 3.循环遍历所有的方法
        for (Method m : arr) {
            // 4.在循环中,判断遍历出来的方法上是否有MyTest注解
            boolean res = m.isAnnotationPresent(MyTest.class);
            // 5.如果有,就执行当前方法
            if (res) {
                m.invoke(td);
            }
        }

    }
}
```

第四章 动态代理

代理模式概述

为什么要有“代理”？生活中就有很多代理的例子，例如，我现在需要出国，但是我不愿意自己去办签证、预定机票和酒店（觉得麻烦，那么就可以找旅行社去帮我办，这时候旅行社就是代理，而我自己就是被代理了。

代理模式的定义：被代理者没有能力或者不愿意去完成某件事情，那么就需要找个人代替自己去完成这件事,这个人就是代理者, 所以代理模式包含了3个角色: 被代理角色　代理角色　抽象角色(协议)

静态代理:

```
// 协议
public interface FindHappy {
    void happy();
}
// 被代理者
public class JinLian implements FindHappy {
```

```
    @Override
    public void happy() {
        System.out.println("金莲正在happy...");
    }
}

// 被代理者
public class YanPoXi implements FindHappy {
    @Override
    public void happy() {
        System.out.println("阎婆惜正在happy...");
    }
}


// 代理者
public class WangPo implements FindHappy {

    FindHappy fh;

    public WangPo(FindHappy fh) {
        this.fh = fh;
    }

    @Override
    public void happy() {
        System.out.println("王婆开好房间,把2人约到房间...");
        // 被代理者去happy
        fh.happy();

        System.out.println("王婆打扫战场....");
    }
}

public class Test {
    public static void main(String[] args) {
        // 创建JinLian对象
        JinLian jl = new JinLian();
        //fh.happy();

        // 创建YanPoXi对象
        YanPoXi ypx = new YanPoXi();

        // 创建代理对象WangPo
        WangPo wp = new WangPo(ypx);
        //WangPo wp = new WangPo(jl);
        wp.happy();
    }
}
```

动态代理和相关api介绍

- 概述 : 动态代理就是在程序运行期间,直接通过反射生成一个代理对象,代理对象所属的类是不需要存在的
- 动态代理的获取:

  jdk提供一个Proxy类可以直接给实现接口类的对象直接生成代理对象
- Java.lang.reflect.Proxy类可以直接生成一个代理对象
- public static Object newProxyInstance(ClassLoader loader, Class<?>[] interfaces, InvocationHandler h)生成一个代理对象
  - 参数1:ClassLoader loader 被代理对象的类加载器
  - 参数2:Class<?>[] interfaces 被代理对象的要实现的接口
  - 参数3:InvocationHandler h (接口)执行处理类
  - 返回值: 代理对象
  - 前2个参数是为了帮助在jvm内部生成被代理对象的代理对象,第3个参数,用来监听代理对象调用方法,帮助我们调用方法
- InvocationHandler中的Object invoke(Object proxy, Method method, Object[] args)方法：调用代理类的任何方法，此方法都会执行
  - 参数1:代理对象(慎用)
  - 参数2:当前代理对象调用的方法
  - 参数3:当前代理对象调用的方法运行时传递过来的参数
  - 返回值:当前方法执行的返回值

案例演示

```
public class Test {
    public static void main(String[] args) {
        // 创建JinLian对象
        JinLian jl = new JinLian();

        // 获取被代理类的类加载器
        ClassLoader classLoader = JinLian.class.getClassLoader();

        // 获取被代理类实现的所有接口的字节码对象
        Class<?>[] interfaces = JinLian.class.getInterfaces();

        // 动态生成一个代理对象
        FindHappy proxy = (FindHappy) Proxy.newProxyInstance(classLoader,
interfaces, new InvocationHandler() {
            @Override
            public Object invoke(Object proxy, Method method, Object[] args)
throws Throwable {
                // 只要代理对象调用方法就会来到这里
                // 参数1: 代理对象
                // 参数2: 代理对象调用的方法
                // 参数3: 代理对象调用的方法传入的实际参数
                System.out.println("代理对象开好房间,把2人约到房间...");
                // 被代理者去happy
                method.invoke(jl,args);
                System.out.println("代理对象打扫战场....");
                return null;
            }
        });
```

```
        // 使用代理对象调用方法
        proxy.happy();

    }
}
```

第五章JDK8新特性

5.1 方法引用

- 概述:方法引用使用一对冒号 :: , 方法引用就是用来在一定的情况下,替换Lambda表达式
- 使用场景:
  - -如果一个Lambda表达式大括号中的代码和另一个方法中的代码一模一样,那么就可以使用方法引用把该方法引过来,从而替换Lambda表达式
  - 如果一个Lambda表达式大括号中的代码就是调用另一方法,那么就可以使用方法引用把该方法引过来,从而替换Lambda表达式

```
public class Test {

    public static void printStr(){
        System.out.println("哈哈哈哈,快要下课啦....");
        System.out.println("哈哈哈哈,快要下课啦....");
        System.out.println("哈哈哈哈,快要下课啦....");
    }

    public static void main(String[] args) {
        /*
            - 概述:方法引用使用一对冒号 :: , 方法引用就是用来在一定的情况下,替换Lambda表达式
            - 使用场景:
              - 如果一个Lambda表达式大括号中的代码和另一个方法中的代码一模一样,那么就可以使用方法引用把该方法引过来,从而替换Lambda表达式
              - 如果一个Lambda表达式大括号中的代码就是调用另一方法,那么就可以使用方法引用把该方法引过来,从而替换Lambda表达式

            Lambda表达式使用套路:
                1.分析能不能使用Lambda表达式--->判断是否是函数接口
                2.如果可以使用Lambda表达式,就直接写上()->{}
                3.填充小括号里面的内容---->小括号中的内容和函数式接口中抽象方法的形参列表一致
                4.填充大括号里面的内容---->大括号中的内容和实现函数式接口中抽象方法的方法体一致

            Lambda表达式省略规则:
                1.小括号中的参数类型可以省略
                2.小括号中如果只有一个参数,那么小括号也可以省略
                3.大括号中如果只有一条语句,那么大括号,分号,return可以省略(必须一起省略)
         */
        // 如果一个Lambda表达式大括号中的代码和另一个方法中的代码一模一样
```

```
        /*new Thread(()->{
            System.out.println("哈哈哈哈,快要下课啦....");
            System.out.println("哈哈哈哈,快要下课啦....");
            System.out.println("哈哈哈哈,快要下课啦....");
        }).start();*/

        //如果一个Lambda表达式大括号中的代码就是调用另一方法
        /*new Thread(()->{
            Test.printStr();
        }).start();*/

        new Thread(Test::printStr).start();


    }
}
```

5.2 方法引用的分类

构造方法引用

```
public class Test1_构造方法引用 {

    public static void main(String[] args) {
        /*
            构造方法引用:   类名::new
         */
        // 获取一个流
        Stream<String> stream = Stream.of("迪丽热巴", "马尔扎哈", "古力娜扎");

        // 需求: 把stream流中的元素转换为Person对象,打印输出
        //stream.map((String t)->{return new Person(t);}).forEach((Person p)->
{System.out.println(p);});

        // 发现: map方法传入的Lambda表达式大括号中就是调用Person类的构造方法,符合方法引用的
场景
        stream.map(Person::new).forEach((Person p)->{System.out.println(p);});


    }
}
```

静态方法引用

```
public class Test2_静态方法引用 {
    public static void main(String[] args) {
        /*
            静态方法引用: 类名::方法名
         */
        // 获取一个流
        Stream<String> stream = Stream.of("110", "120", "114");
```

```
        // 需求: 把stream流中的元素转换为Integer对象,打印输出
        // stream.map((String t)->{return
Integer.parseInt(t);}).forEach((Integer i)->{System.out.println(i+1);});

        // 发现: map方法传入的Lambda表达式大括号中就是调用Interger类的parseInt静态方法,符
合方法引用的场景
        stream.map(Integer::parseInt).forEach((Integer i)->
{System.out.println(i+1);});

    }
}
```

有参数成员方法引用

- 成员方法有参数

```
public class Test3_成员方法引用_有参数 {
    public static void main(String[] args) {
        /*
            成员方法引用_有参数: 对象名::方法名
         */
        // 获取一个流
        Stream<String> stream = Stream.of("110", "120", "114");

        // 需求: 把stream流中的元素打印输出
        /*stream.forEach((String t)->{
            System.out.println(t);
        });*/

        // 发现: forEach方法传入的Lambda表达式大括号中其实就是调用System.out对象的
println方法,符合
        stream.forEach(System.out::println);
    }

}
```

没有参数成员方法引用

- 成员方法没有参数

```
public class Test4_成员方法引用_没有参数 {
    public static void main(String[] args) {
         /*
            成员方法引用_没有参数: 类名::方法名
         */
        // 获取一个流
        Stream<String> stream = Stream.of("迪丽热巴", "杨颖","马尔扎哈", "古力娜
扎");

        // 需求: 把stream流中的元素转换为这些元素的字符串长度,打印输出
        //stream.map((String t)->{return
t.length();}).forEach(System.out::println);
```

```
        // 发现: map方法传入的Lambda表达式大括号中就是调用String类的length方法,符合方法引用的场景
        stream.map(String::length).forEach(System.out::println);

    }

}
```

总结方法引用使用套路

```
1.分析判断是否能使用方法引用替换Lambda表达式----->根据方法引用的使用场景
2.如果可以,那就确定要引用的方法类型
3.根据方法引用的格式,引用方法即可
    构造方法引用:    类名::new
    静态方法引用: 类名::方法名
    成员方法引用_有参数: 对象名::方法名
    成员方法引用_没有参数: 类名::方法名
```

5.3 Base64

Base64概述

- Base64是jdk8提出的一个新特性,可以用来进行按照一定规则编码和解码

Base64编码和解码的相关方法

- 编码的步骤:
  - 获取编码器
  - 调用方法进行编码
- 解码步骤:
  - 获取解码器
  - 调用方法进行解码
- Base64工具类提供了一套静态方法获取下面三种BASE64编解码器:
  - **基本**：输出被映射到一组字符A-Za-z0-9+/，编码不添加任何行标，输出的解码仅支持A-Za-z0-9+/。
  - **URL**：输出映射到一组字符A-Za-z0-9+_，输出是URL和文件。
  - **MIME**：输出隐射到MIME友好格式。输出每行不超过76字符，并且使用'\r'并跟随'\n'作为分割。编码输出最后没有行分割。
- 获取编码器和解码器的方法

```
static Base64.Decoder getDecoder() 基本型 base64 解码器。
static Base64.Encoder getEncoder() 基本型 base64 编码器。

static Base64.Decoder getMimeDecoder() Mime型 base64 解码器。
static Base64.Encoder getMimeEncoder() Mime型 base64 编码器。

static Base64.Decoder getUrlDecoder() Url型 base64 解码器。
static Base64.Encoder getUrlEncoder() Url型 base64 编码器。
```

- 编码和解码的方法:

```
Encoder编码器:  encodeToString(byte[] bys)编码
Decoder解码器:  decode(String str) 解码
```

案例演示

- 基本

```
public class Test1_基本型 {
    public static void main(String[] args) {

        // 1.获取编码器
        Base64.Encoder encoder = Base64.getEncoder();

        String str = "name=中国?password=123456";

        // 2.对str进行编码
        String s = encoder.encodeToString(str.getBytes());
        System.out.println("编码后:" + s);


        // 3.获取解码器
        Base64.Decoder decoder = Base64.getDecoder();

        // 4.对s进行解码
        byte[] bys = decoder.decode(s);
        System.out.println("解码后:" + new String(bys));
    }
}
```

- URL

```
public class Test2_URL型 {
    public static void main(String[] args) {

        // 1.获取编码器
        Base64.Encoder encoder = Base64.getUrlEncoder();

        String str = "name=中国?password=123456";

        // 2.对str进行编码
        String s = encoder.encodeToString(str.getBytes());
        System.out.println("编码后:" + s);


        // 3.获取解码器
        Base64.Decoder decoder = Base64.getUrlDecoder();

        // 4.对s进行解码
        byte[] bys = decoder.decode(s);
        System.out.println("解码后:" + new String(bys));
    }
}
```

- MIME

```
public class Test3_MIME型 {
    public static void main(String[] args) {

        // 1.获取编码器
        Base64.Encoder encoder = Base64.getMimeEncoder();

        String str = "";
        for (int i = 0; i < 100; i++) {
            str += i;
        }

        // 2.对str进行编码
        String s = encoder.encodeToString(str.getBytes());
        System.out.println("编码后:" + s);


        // 3.获取解码器
        Base64.Decoder decoder = Base64.getMimeDecoder();

        // 4.对s进行解码
        byte[] bys = decoder.decode(s);
        System.out.println("解码后:" + new String(bys));
    }
}
```

总结

```
必须练习:
    1.Junit单元测试使用
    2.获取类的字节码对象
    3.反射操作成员方法,构造方法
    4.使用注解---->3.4,3.5的案例
    5.动态代理---->金莲代理对象
    6.Base64的编码和解码


- 能够使用Junit进行单元测试
    1.下载Junit的jar包
    2.把Junit的jar包拷贝到模块下的lib文件夹中,并添加到classpath路径中
    3.编写测试方法
    4.在测试方法上书写@Test注解
    5.运行测试

- 能够通过反射技术获取Class字节码对象
      1.1 类名.class
      1.2 对象名.getClass();
      1.3 Class.forName("类的全路径");

- 能够通过反射技术获取构造方法对象，并创建对象。
    Constructor getDeclaredConstructor(Class... parameterTypes)
    Constructor[] getDeclaredConstructors()
```

```
    T newInstance(Object... initargs)
    void setAccessible(true);暴力反射

- 能够通过反射获取成员方法对象，并且调用方法。--------特别
         Method getDeclaredMethod(String name,Class...args);
         Method[] getDeclaredMethods();
         Object invoke(Object obj, Object... args)
         void setAccessible(true);暴力反射

- 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。
      Field getDeclaredField(String name);
        Field[] getDeclaredFields();
        void setAccessible(true);暴力反射
        void  set(Object obj, Object value)
        Object get(Object obj)

- 能够说出注解的作用
   作为配置\编译检查

- 能够自定义注解和使用注解
    格式:
        public @interface 注解名{
            属性
        }
    属性格式:
        数据类型 属性名();

    数据类型:
        1.基本类型
        2.String类型
        3.Class类型
        4.枚举类型
        5.注解类型
        6.以上类型的一维数组类型

    使用注解:
        无属性的注解: @注解名
        有属性的注解: @注解名(属性名=属性值,...)
    注意事项:
        1.如果注解属性是数组类型,并且数组的值只有一个,那么大括号可以省略
        2.如果注解只有一个属性,并且属性名为value,那么给属性赋值的时候value属性名可以省略
        3.注解属性有默认值,可以不给注解属性赋值   格式: 数据类型 注解名() default 属性值;

- 能够说出常用的元注解及其作用
      @Target:表示该注解作用在什么上面(位置),默认注解可以在任何位置
      @Retention:定义该注解保留到那个代码阶段， 值为:RetentionPolicy类型,==默认只在源码阶段保留==

- 能够解析注解并获取注解中的数据
java.lang.reflect.AnnotatedElement接口: Class、Method、Field、Constructor等实现了该接口
     - T getAnnotation(Class<T> annotationType):得到指定类型的注解引用。没有返回null。
     - boolean isAnnotationPresent(Class<?extends Annotation> annotationType):
                          判断指定的注解有没有。
    获取注解的属性值:  注解对象.属性名();

- 能够完成注解的MyTest案例
   1.获取类的字节码对象
```

2.获取该类的所有方法
3.循环遍历所有的方法
4.判断遍历出来的方法是否包含指定的注解,如果包含,就执行该方法

- 能够说出动态代理模式的作用
作用: 为了增强被代理类的方法

- 能够使用Proxy的方法生成代理对象
Java.lang.reflect.Proxy类可以直接生成一个代理对象
- public static Object newProxyInstance(ClassLoader loader, Class<?>[] interfaces, InvocationHandler h)生成一个代理对象
- 参数1:ClassLoader loader 被代理对象的类加载器
- 参数2:Class<?>[] interfaces 被代理对象的要实现的接口
- 参数3:InvocationHandler h (接口)执行处理类
- 返回值: 代理对象
- 前2个参数是为了帮助在jvm内部生成被代理对象的代理对象,第3个参数,用来监听代理对象调用方法,帮助我们调用方法

- InvocationHandler中的Object invoke(Object proxy, Method method, Object[] args)方法: 调用代理类的任何方法, 此方法都会执行
- 参数1:代理对象(慎用)
- 参数2:当前执行的方法
- 参数3:当前执行的方法运行时传递过来的参数
- 返回值:当前方法执行的返回值

- 能够使用四种方法的引用
总结:使用方法引用的步骤
1.分析要写的Lambda表达式的大括号中是否就是调用另一个方法
2.如果是,就可以使用方法引用替换,如果不是,就不能使用方法引用
3.确定引用的方法类型(构造方法,成员方法,静态方法,类的成员方法)
4.按照对应的格式去引用:
构造方法: 类名::new
成员方法(有参数): 对象名::方法名
静态方法: 类名::方法名
类的成员方法\成员方法(无参数): 类名::方法名

- 能够使用Base64对基本数据、URL和MIME类型进行编解码
static Base64.Decoder getDecoder() 基本型 base64 解码器。
static Base64.Encoder getEncoder() 基本型 base64 编码器。

static Base64.Decoder getUrlDecoder() Url型 base64 解码器。
static Base64.Encoder getUrlEncoder() Url型 base64 编码器。

static Base64.Decoder getMimeDecoder() Mime型 base64 解码器。
static Base64.Encoder getMimeEncoder() Mime型 base64 编码器。

Encoder编码器: encodeToString(byte[] bys)编码
Decoder解码器: decode(String str) 解码