

Junit、反射、注解

学习目标

1. 能够使用Junit进行单元测试
2. 能够通过反射技术获取Class字节码对象
3. 能够通过反射技术获取构造方法对象，并创建对象。
4. 能够通过反射获取成员方法对象，并且调用方法。
5. 能够说出注解的作用
6. 能够自定义注解和使用注解
7. 能够说出常用的元注解及其作用
8. 能够解析注解并获取注解中的数据
9. 能够完成注解的MyTest案例

第1章 Junit单元测试

1.1 Junit的概述

Junit是一个Java语言的单元测试框架，简单理解为可以用于取代java的main方法。Junit属于第三方工具，一般情况下需要导入jar包。不过，多数Java开发环境已经集成了Junit作为单元测试工具。

在Java中，一个类就是一个单元。

单元测试是开发者编写的一小段代码，用于检验某个类某个方法的功能或某个业务逻辑是否正确。

1.2 Junit的使用

1.2.1 下载

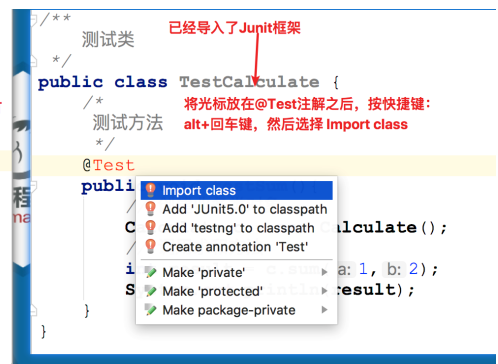
一般IDE都内置了junit,若需要自行下载jar包,可以访问官网,官网地址如下：<http://www.junit.org>

1.2.2 步骤

1. 编写业务类
2. 创建测试类
 - 编写测试方法

1.2.3 测试方法

1. 特点
 - 方法命名规则：以test开头，使用驼峰命名法。
 - 方法声明上：必须使用注解：@Test，必须使用public修饰符，没有返回值，方法没有参数。



2. 运行测试方法

- 选中方法名：右键 --> Run 测试方法名，则运行选中的测试方法

比如测试方法名为 testSum，则右键 --> Run testSum

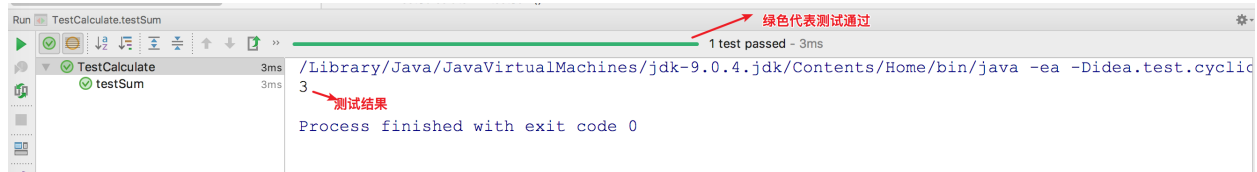
- 选中类名：右键 --> Run 类名，则运行该类的所有测试方法

比如类名为 TestCalculate，则右键 --> Run TestCalculate

- 选中模块名或项目名：右键 --> Run 'All Tests'，则运行整个模块中所有类的所有测试方法。

3. 查看测试结果

- 绿色：表示测试通过，如下图



- 红色：表示失败或出现错误，如下图



1.2.4 常用注解

- @Before：在每个测试方法之前都会运行一次
- @After：在每个测试方法运行以后运行的方法
- @BeforeClass：在所有的测试方法运行之前，只运行一次，而且必须用在静态方法上面。
- @AfterClass：所有的测试方法运行以后，只运行一次，必须用在静态方法上面。

1.2.5 示例代码

- 业务类

```
/**
 * 业务类
 */
public class Calculate {

    /**
```



```
        求a和b之和
    */
    public int sum(int a,int b){
        return a + b;
    }

    /**
        求a和b之差
    */
    public int sub(int a,int b){
        return a - b;
    }
}
```

- 测试类

```
import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;

public class TestCalculate {

    @BeforeClass
    public static void init() {
        System.out.println("类加载时，只运行一次");
    }

    @Before
    public void myBefore(){
        System.out.println("方法前");
    }

    /**
        测试方法
    */
    @Test
    public void testSum(){
        // 创建业务类对象
        Calculate c = new Calculate();
        // 调用求和方法
        int result = c.sum(1,2);
        System.out.println(result);
    }

    /**
        测试方法
    */
    @Test
    public void testSub(){
        // 创建业务类对象
        Calculate c = new Calculate();
```

```
// 调用求和方法
int result = c.sub(1,2);
System.out.println(result);
}

@After
public void myAfter(){
    System.out.println("方法后");
}

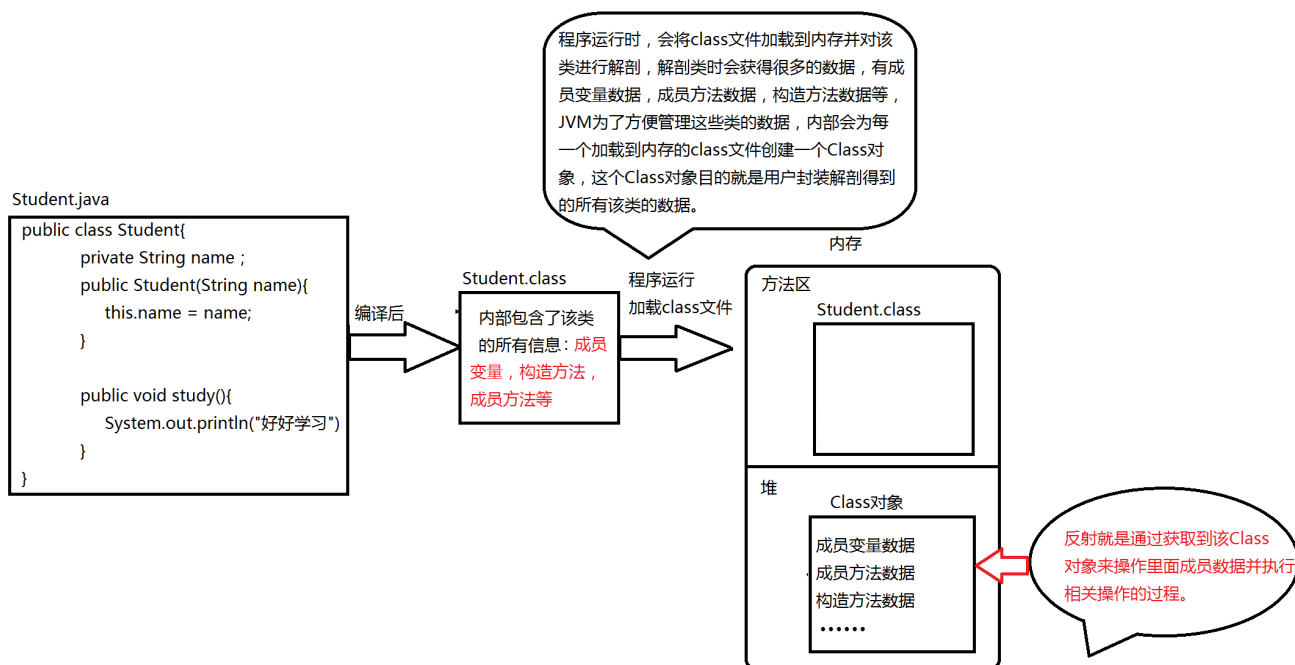
@AfterClass
public static void destory() {
    System.out.println("类结束前，只运行一次");
}
}
```

第2章 反射

2.1 反射的基本概念

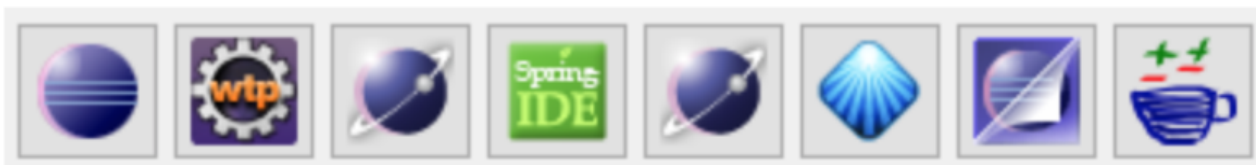
2.1.1 什么是反射

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方法，属性，构造方法等成员。

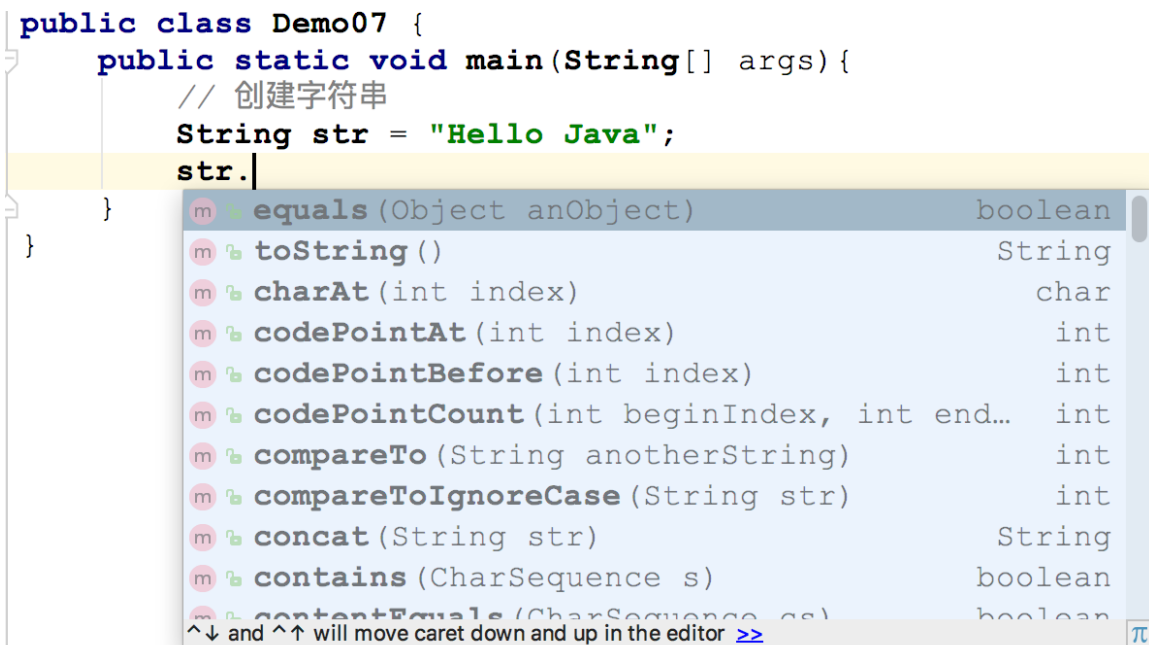


2.1.2 反射在实际开发中的应用

1. 开发IDE(集成开发环境)



- 以上的IDE内部都大量使用了反射机制，我们在使用这些IDE写代码也无时无刻的使用着反射机制，一个常用反射机制的地方就是当我们通过对象调用方法或访问属性时，开发工具都会以列表的形式显示出该对象所有的方法或属性，以供方便我们选择使用，如下图：



- 这些开发工具之所以能够把该对象的方法和属性展示出来就使用利用了反射机制对该对象所有类进行了解剖获取到了类中的所有方法和属性信息，这是反射在IDE中的一个使用场景。

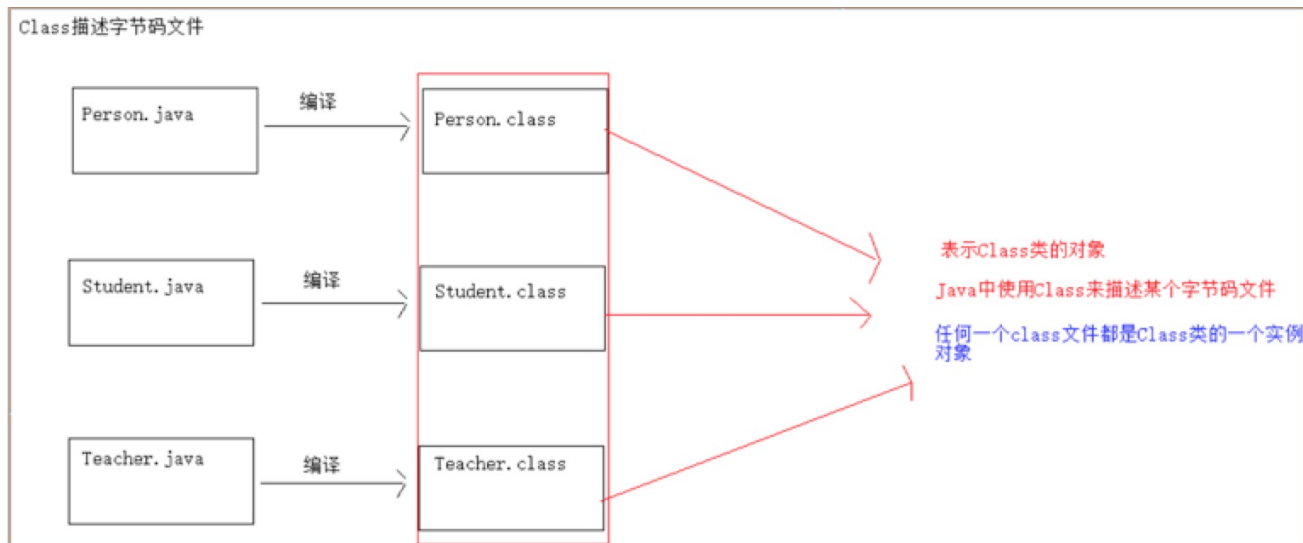
1. 各种框架的设计



- 以上三个图标上面的名字就是Java的三大框架，简称SSH.
- 这三大框架的内部实现也大量使用到了反射机制，所有要想学好这些框架，则必须要求对反射机制熟练了。

2.1.3 使用反射机制解剖类的前提

必须先要获取到该类的字节码文件对象，即**Class类型对象**。关于Class描述字节码文件如下图所示：



说明：

- 1) Java中使用Class类表示某个class文件.
- 2) 任何一个class文件都是Class这个类的一个实例对象.

2.2 获取Class对象的三种方式

- 创建测试类：Student

```
public class Student {
    static {
        System.out.println("静态代码块");
    }

    {
        System.out.println("构造代码块");
    }
}
```

2.2.1 方式1：通过类名.class获取

```
public class Demo01 {
    public static void main(String[] args) {
        // 获得Student的Class的对象
        Class c = Student.class;
        // 打印输出：class com.itheima.reflect.Student
        System.out.println(c);
    }
}
```

2.2.2 方式2：通过Object类的getClass()方法获取



```
public class Demo01 {  
    public static void main(String[] args) {  
        // 创建学生对象  
        Student stu = new Student();  
        // 获得学生类的Class对象  
        Class c = stu.getClass();  
        // 打印输出：class com.itheima.reflect.Student  
        System.out.println(c);  
    }  
}
```

2.2.3 方式3：通过Class.forName("全限定类名")方法获取

```
public class Demo01 {  
    public static void main(String[] args) throws Exception {  
        // 获得字符串的Class对象  
        Class c = Class.forName("java.lang.String");  
        // 打印输出：class java.lang.String  
        System.out.println(c);  
    }  
}
```

2.3 获取Class对象的信息

知道怎么获取Class对象之后，接下来就介绍几个Class类中常用的方法了。

2.3.1 Class对象相关方法

1. `String getSimpleName()`；获得简单类名，只是类名，没有包
`String getName()`；获取完整类名，包含包名+类名
`T newInstance()`；创建此 Class 对象所表示的类的一个新实例。要求：类必须有public的无参数构造方法

2.3.2 方法演示

```
public class Demo02 {  
    public static void main(String[] args) throws Exception {  
        // 获得字符串的Class对象  
        Class c = Class.forName("java.lang.String");  
        // 获得简单类名  
        String simpleName = c.getSimpleName();  
        // 打印输入：simpleName = String  
        System.out.println("simpleName = " + simpleName);  
  
        // 获得完整类名(包含包名和类名)  
        String name = c.getName();  
        // 打印输入：name = java.lang.String  
        System.out.println("name = " + name);  
  
        // 创建字符串对象
```



```
String str = (String) c.newInstance();  
// 输出str: 空字符串 ""  
System.out.println(str);  
}  
}
```

2.4 获取Class对象的Constructor信息

一开始在阐述反射概念的时候，我们说到利用反射可以在程序运行过程中对类进行解剖并操作里面的成员。而一般常操作的成员有构造方法，成员方法，成员变量等等，那么接下来就来看看怎么利用反射来操作这些成员以及操作这些成员能干什么，先来看看怎么操作构造方法。而要通过反射操作类的构造方法，我们需要先知道一个Constructor类。

2.4.1 Constructor类概述

Constructor是构造方法类，类中的每一个构造方法都是Constructor的对象，通过Constructor对象可以实例化对象。

// public 有参构造方法

```
public Student(String name, String gender, int age) {  
    System.out.println("public 修饰有参数构造方法");  
    this.name = name;  
    this.gender = gender;  
    this.age = age;  
}
```

// public 无参构造方法 每一个构造方法都是一个Constructor对象

```
public Student() {  
    System.out.println("public 修饰无参数构造方法");  
}
```

// private 有参构造方法

```
private Student(String name, String gender) {  
    System.out.println("private 修饰构造方法");  
    this.name = name;  
    this.gender = gender;  
}
```

2.4.2 Class类中与Constructor相关方法



1. Constructor `getConstructor(Class... parameterTypes)`
根据参数类型获取构造方法对象，只能获得public修饰的构造方法。
如果不存在对应的构造方法，则会抛出 `java.lang.NoSuchMethodException` 异常。
2. Constructor `getDeclaredConstructor(Class... parameterTypes)`
根据参数类型获取构造方法对象，包括private修饰的构造方法。
如果不存在对应的构造方法，则会抛出 `java.lang.NoSuchMethodException` 异常。
3. Constructor[] `getConstructors()`
获取所有的public修饰的构造方法
4. Constructor[] `getDeclaredConstructors()`
获取所有构造方法，包括private修饰的

2.4.3 Constructor类中常用方法

1. `T newInstance(Object... initargs)`
根据指定参数创建对象。
2. `void setAccessible(true)`
暴力反射，设置为可以直接访问私有类型的构造方法。

2.4.4 示例代码

1. 学生类

```
/**
 *
 * @version 1.0
 * @description com.itheima
 * @date 2018/1/25
 */
public class Student {
    // 姓名
    private String name;
    // 性别
    public String gender;
    // 年龄
    private int age;

    // public 有参构造方法
    public Student(String name, String gender, int age) {
        System.out.println("public 修饰有参数构造方法");
        this.name = name;
        this.gender = gender;
        this.age = age;
    }

    // public 无参构造方法
    public Student() {
        System.out.println("public 修饰无参数构造方法");
    }
}
```



```
}

// private 有参构造方法
private Student(String name,String gender){
    System.out.println("private 修饰构造方法");
    this.name = name;
    this.gender = gender;
}

// getter & setter 方法
/*
 * 此处为 getter & setter方法 省略...
 */

// 普通方法
public void sleep(){
    System.out.println("睡觉");
}

public void sleep(int hour){
    System.out.println("public修饰---sleep---睡" + hour + "小时");
}

private void eat(){
    System.out.println("private修饰---eat方法---吃饭");
}

// 静态方法
public static void study(){
    System.out.println("静态方法---study方法---好好学习Java");
}

@Override
public String toString() {
    return "Student{" +
        "name='" + name + '\'' +
        ", gender='" + gender + '\'' +
        ", age=" + age +
        '}';
}
}
```

1. 测试类

```
/**
 *
 * @version 1.0
 * @description 获取Class对象的Constructor信息
 * @date 2018/1/26
 */
public class Demo03 {
    public static void main(String[] args)throws Exception{
```



```
test01();
test02();
test03();
test04();
}

/**
4. Constructor[] getDeclaredConstructors()
    获取所有构造方法，包括private修饰的
*/
public static void test04() throws Exception{
    System.out.println("----- test04() -----");
    // 获取Student类的Class对象
    Class c = Student.class;
    // 获取所有的public修饰的构造方法
    Constructor[] cons = c.getDeclaredConstructors();
    // 遍历构造方法数组
    for(Constructor con:cons) {
        // 输出con
        System.out.println(con);
    }
}

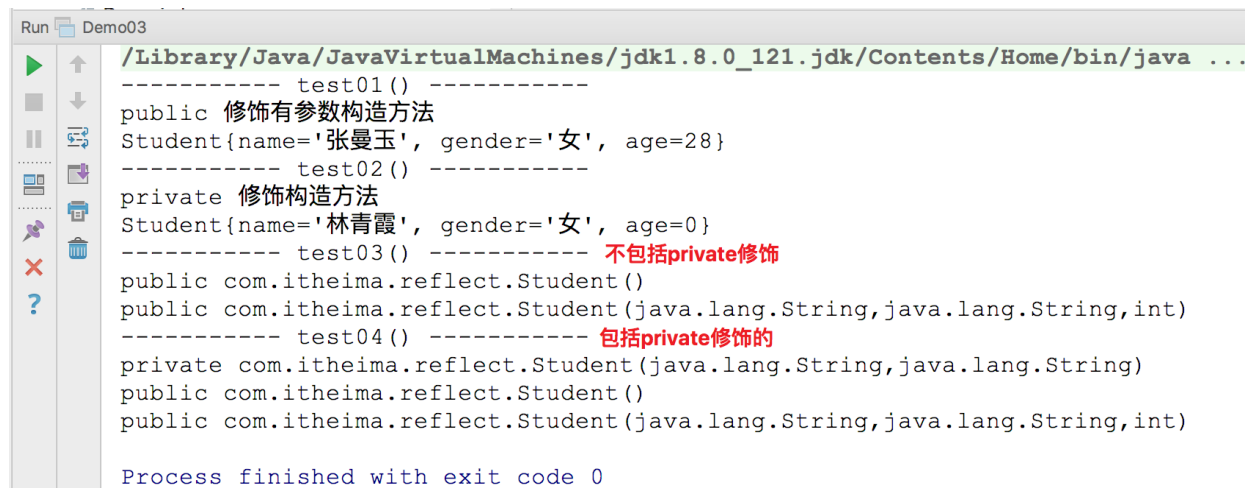
/**
3. Constructor[] getConstructors()
    获取所有的public修饰的构造方法
*/
public static void test03() throws Exception{
    System.out.println("----- test03() -----");
    // 获取Student类的Class对象
    Class c = Student.class;
    // 获取所有的public修饰的构造方法
    Constructor[] cons = c.getConstructors();
    // 遍历构造方法数组
    for(Constructor con:cons) {
        // 输出con
        System.out.println(con);
    }
}

/**
2. Constructor getDeclaredConstructor(Class... parameterTypes)
    根据参数类型获取构造方法对象，包括private修饰的构造方法。
    如果不存在对应的构造方法，则会抛出 java.lang.NoSuchMethodException 异常。
*/
public static void test02() throws Exception{
    System.out.println("----- test02() -----");
    // 获取Student类的Class对象
    Class c = Student.class;
    // 根据参数获取对应的private修饰构造方法对象
    Constructor cons = c.getDeclaredConstructor(String.class,String.class);
    // 注意：private的构造方法不能直接调用newInstance创建对象，需要暴力反射才可以

    // 设置取消权限检查（暴力反射）
```

```
cons.setAccessible(true);
// 调用Constructor方法创建学生对象
Student stu = (Student) cons.newInstance("林青霞", "女");
// 输出stu
System.out.println(stu);
}
/**
1. Constructor getConstructor(Class... parameterTypes)
根据参数类型获取构造方法对象，只能获得public修饰的构造方法。
如果不存在对应的构造方法，则会抛出 java.lang.NoSuchMethodException 异常。
*/
public static void test01() throws Exception{
    System.out.println("----- test01() -----");
    // 获取Student类的Class对象
    Class c = Student.class;
    // 根据参数获取对应的构造方法对象
    Constructor cons = c.getConstructor(String.class, String.class, int.class);
    // 调用Constructor方法创建学生对象
    Student stu = (Student) cons.newInstance("张曼玉", "女", 28);
    // 输出stu
    System.out.println(stu);
}
}
```

- 输出结果：



```
Run Demo03
/Library/Java/JavaVirtualMachines/jdk1.8.0_121.jdk/Contents/Home/bin/java ...
----- test01() -----
public 修饰有参数构造方法
Student{name='张曼玉', gender='女', age=28}
----- test02() -----
private 修饰构造方法
Student{name='林青霞', gender='女', age=0}
----- test03() ----- 不包括private修饰
public com.itheima.reflect.Student()
public com.itheima.reflect.Student(java.lang.String,java.lang.String,int)
----- test04() ----- 包括private修饰的
private com.itheima.reflect.Student(java.lang.String,java.lang.String)
public com.itheima.reflect.Student()
public com.itheima.reflect.Student(java.lang.String,java.lang.String,int)

Process finished with exit code 0
```

2.5 获取Class对象的Method信息

操作完构造方法之后，就来看看反射怎么操作成员方法了。同样的在操作成员方法之前我们需要学习一个类：Method类。

2.5.1 Method类概述

Method是方法类，类中的每一个方法都是Method的对象，通过Method对象可以调用方法。



```
// getter & setter 方法
public String getName() { return name; }  每一个成员方法都是一个Method对象

public void setName(String name) { this.name = name; }

public String getGender() { return gender; }

public void setGender(String gender) { this.gender = gender; }

public int getAge() { return age; }

public void setAge(int age) { this.age = age; }

// 普通方法
public void sleep() { System.out.println("睡觉"); }

public void sleep(int hour) { System.out.println("public修饰---sleep---睡" + hour + "小时"); }

private void eat() { System.out.println("private修饰---eat方法---吃饭"); }

// 静态方法
public static void study() { System.out.println("静态方法---study方法---好好学习Java"); }
```

2.5.2 Class类中与Method相关方法

1. Method `getMethod("方法名", 方法的参数类型... 类型)`
根据方法名和参数类型获得一个方法对象，只能是获取public修饰的
2. Method `getDeclaredMethod("方法名", 方法的参数类型... 类型)`
根据方法名和参数类型获得一个方法对象，包括private修饰的
3. Method[] `getMethods()` (了解)
获取所有的public修饰的成员方法，包括父类中。
4. Method[] `getDeclaredMethods()` (了解)
获取当前类中所有的方法，包含私有的，不包括父类中。

2.5.3 Method类中常用方法

1. `Object invoke(Object obj, Object... args)`
根据参数args调用对象obj的该成员方法
如果obj=null，则表示该方法是静态方法
2. `void setAccessible(boolean flag)`
暴力反射，设置为可以直接调用私有修饰的成员方法

2.5.4 示例代码

```
/**
 *
 * @version 1.0
 * @description 获取Class对象的Method信息
 * @date 2018/1/26
 */

public class Demo04 {
```



```
public static void main(String[] args) throws Exception{
    // 获得Class对象
    Class c = Student.class;
    // 快速创建一个学生对象
    Student stu = (Student ) c.newInstance();

    // 获得public修饰的方法对象
    Method m1 = c.getMethod("sleep",int.class);
    // 调用方法m1
    m1.invoke(stu,8);

    // 获得private修饰的方法对象
    Method m2 = c.getDeclaredMethod("eat");
    // 注意：private的成员方法不能直接调用，需要暴力反射才可以
    // 设置取消权限检查（暴力反射）
    m2.setAccessible(true);
    // 调用方法m2
    m2.invoke(stu);

    // 获得静态方法对象
    Method m3 = c.getDeclaredMethod("study");
    // 调用方法m3
    // 注意：调用静态方法时，obj可以为null
    m3.invoke(null);

    System.out.println("-----获得所有public的方法，不包括private，包括父类的-----");
    // 获得所有public的方法，包括父类的
    Method[] ms = c.getMethods();
    // 遍历方法数组
    for(Method m : ms) {
        System.out.println(m);
    }

    System.out.println("-----获得所有方法,包括private，不包括父类-----");
    // 获得所有方法,包括private，不包括父
    Method[] ms2 = c.getDeclaredMethods();
    // 遍历方法数组
    for(Method m : ms2) {
        System.out.println(m);
    }
}
```

- 输出结果：



```
Run Demo04
public 修饰无参数构造方法
public 修饰---sleep---睡8小时
private 修饰---eat方法---吃饭
静态方法---study方法---好好学习Java
-----获得所有public的方法，不包括private，包括父类的-----
public java.lang.String com.itheima.reflect.Student.toString()
public java.lang.String com.itheima.reflect.Student.getName()
public void com.itheima.reflect.Student.sleep()
public void com.itheima.reflect.Student.sleep(int)
public void com.itheima.reflect.Student.setName(java.lang.String)
public static void com.itheima.reflect.Student.study()
public int com.itheima.reflect.Student.getAge()
public java.lang.String com.itheima.reflect.Student.getGender()
public void com.itheima.reflect.Student.setGender(java.lang.String)
public void com.itheima.reflect.Student.setAge(int)
public final void java.lang.Object.wait(long,int) throws java.lang.InterruptedException
public final native void java.lang.Object.wait(long) throws java.lang.InterruptedException
public final void java.lang.Object.wait() throws java.lang.InterruptedException
public boolean java.lang.Object.equals(java.lang.Object)
public native int java.lang.Object.hashCode()
public final native java.lang.Class java.lang.Object.getClass()
public final native void java.lang.Object.notify()
public final native void java.lang.Object.notifyAll()
-----获得所有方法，包括private，不包括父类-----
public java.lang.String com.itheima.reflect.Student.toString()
public java.lang.String com.itheima.reflect.Student.getName()
public void com.itheima.reflect.Student.sleep()
public void com.itheima.reflect.Student.sleep(int)
public void com.itheima.reflect.Student.setName(java.lang.String)
private void com.itheima.reflect.Student.eat()
public static void com.itheima.reflect.Student.study()
public int com.itheima.reflect.Student.getAge()
public java.lang.String com.itheima.reflect.Student.getGender()
public void com.itheima.reflect.Student.setGender(java.lang.String)
public void com.itheima.reflect.Student.setAge(int)
```

这些方法是父类Object中

2.6 获取Class对象的Field信息(了解)

2.6.1 Field类概述

Field是属性类，类中的每一个属性(成员变量)都是Field的对象，通过Field对象可以给对应的成员变量赋值和取值。

```
// 姓名
```

```
private String name;
```

```
// 性别
```

```
public String gender;
```

```
// 年龄
```

```
private int age;
```

每一个成员变量都是一个Field对象

2.6.2 Class类中与Field相关方法



1. Field `getDeclaredField(String name)`
根据属性名获得属性对象，包括private修饰的
2. Field `getField(String name)`
根据属性名获得属性对象，只能获取public修饰的
3. Field[] `getFields()`
获取所有的public修饰的属性对象，返回数组。
4. Field[] `getDeclaredFields()`
获取所有的属性对象，包括private修饰的，返回数组。

2.6.3 Field类中常用方法

```
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。

2.6.4 示例代码

```
/**
 *
 * @version 1.0
 * @description 获取Class对象的Field信息
 * @date 2018/1/26
 */
public class Demo05 {
    public static void main(String[] args) throws Exception {
        // 获得Class对象
        Class c = Student.class;
        // 快速创建一个学生对象
        Student stu = (Student) c.newInstance();

        // 获得public修饰Field对象
        Field f1 = c.getField("gender");

        // 通过f1对象给对象stu的gender属性赋值
```




```
f1.set(stu, "风清扬");  
// 通过f1对象获取对象stu的gender属性值  
String gender = (String) f1.get(stu);  
System.out.println("性别：" + gender);  
  
// 获得private修饰Field对象  
Field f2 = c.getDeclaredField("age");  
// 注意：private的属性不能直接访问，需要暴力反射才可以  
// 设置取消权限检查（暴力反射）  
f2.setAccessible(true);  
// 通过f1对象给对象stu的age属性赋值  
f2.setInt(stu, 30);  
// 通过f2对象获取对象stu的age属性值  
int age = f2.getInt(stu);  
System.out.println("年龄：" + age);  
  
System.out.println("-----获得所有public修饰的属性-----");  
// 获得所有public修饰的属性  
Field[] fs1 = c.getFields();  
// 遍历数组  
for(Field f : fs1) {  
    System.out.println(f);  
}  
  
System.out.println("-----获得所有的属性，包括private修饰-----");  
// 获得所有的属性，包括private修饰  
Field[] fs2 = c.getDeclaredFields();  
// 遍历数组  
for(Field f : fs2) {  
    System.out.println(f);  
}  
}  
}
```

- 输出结果

```
Run Demo05  
/Library/Java/JavaVirtualMachines/jdk1.8.0_121.jdk/Contents/Home/bin/java ...  
public 修饰无参数构造方法  
性别：风清扬  
年龄：30  
-----获得所有public修饰的属性----- Student类中只有gender属性是public修饰的  
public java.lang.String com.itheima.reflect.Student.gender  
-----获得所有的属性，包括private修饰-----  
private java.lang.String com.itheima.reflect.Student.name  
public java.lang.String com.itheima.reflect.Student.gender  
private int com.itheima.reflect.Student.age  
  
Process finished with exit code 0
```

2.7 小结

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方法，属性，构造方法等成员。

我们使用反射的时候，都需要先获取类的字节码对象，最常用的方式就是 `Class.forName("类的全限定名")`。

我们可以通过字节码对象就可以获取类的构造器，从而创建对象，最常用的方式是通过字节码对象获取无参构造器 `getConstructor`，就可以调用构造器对象的 `newInstance()` 方法创建对象，当然这种方式简化为：字节码对象直接调用 `newInstance()` 方法创建对象。

也可以通过字节码对象获取指定的方法，然后调用方法对象的 `invoke(Object obj, Object... args)` 来执行此方法。

也可以通过字节码对象获取类中的所有字段，从而给字段赋值和获取字段的值。

有了以上的基础，我们就可以通过一个指定全限定名，就可以创建该类对象的实例对象，执行里面的方法，给其字段进行赋值或者获取值。

第3章 注解

3.1 注解的概述

3.1.1 注解的概念

- 注解是JDK1.5的特性。
- 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什么种 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。

3.1.2 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. 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

```
public class Student extends Object {  
  
    @Override  
    public String toString() {  
        return "重写父类Object的toString方法";  
    }  
  
    @Override  
    public void study() {  
    }  
}
```

此处没有报错，因为父类中有toString方法。

此处编译失败，因为父类中没有study方法。

2. 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。

3.1.3 常见注解

1. **@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. **@Deprecated**:用来表示不赞成使用

3.2 自定义注解

3.2.1 定义格式

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

- 以上定义出来的注解就是一个最简单的注解了，但这样的注解意义不大，因为注解中没有任何内容，就好像我们定义一个类而这个类中没有任何成员变量和方法一样，这样的类意义也是不大的，所以在定义注解时会在里面添加一些成员来让注解功能更加强大，这些成员就是属性。接下来就看看怎么给注解添加属性。

3.2.2 注解的属性

1. 属性的作用

- 可以让用户在使用注解时传递参数，让注解的功能更加强大。

2. 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;

3. 属性定义示例



```
public @interface Student {  
    String name(); // 姓名  
    int age() default 18; // 年龄  
    String gender() default "男"; // 性别  
}  
// 该注解就有了三个属性：name, age, gender
```

4. 属性适用的数据类型

- 八种基本数据类型 (int, float, boolean, byte, double, char, long, short)
- String类型, Class类型, 枚举类型, 注解类型
- 以上所有类型的一维数组

3.3 使用自定义注解

3.3.1 定义注解

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

```
public @interface Book {  
    // 书名  
    String value();  
    // 价格  
    double price() default 100;  
    // 多位作者  
    String[] authors();  
}
```

3.3.2 使用注解

1. 定义类在成员方法上使用Book注解

```
/**  
 *  
 * @version 1.0  
 * @description 书架类  
 * @date 2018/1/26  
 */  
public class BookShelf {  
  
    @Book(value = "西游记", price = 998, authors = {"吴承恩", "白求恩"})  
    public void showBook(){  
  
    }  
}
```



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。

3.3.3 特殊属性value

1. 当注解中只有一个属性且名称是value，在使用注解时给value属性赋值可以直接给属性值，无论value是单值元素还是数组类型。

```
// 定义注解Book
public @interface Book {
    // 书名
    String value();
}

// 使用注解Book
public class BookShelf {
    @Book("西游记")
    public void showBook(){

    }
}
或
public class BookShelf {
    @Book(value="西游记")
    public void showBook(){

    }
}
```

2. 如果注解中除了value属性还有其他属性，且至少有一个属性没有默认值，则在使用注解给属性赋值时，value属性名不能省略。

```
// 定义注解Book
public @interface Book {
    // 书名
    String value();
    // 价格
    double price() default 100;
    // 多位作者
    String[] authors();
}

// 使用Book注解：正确方式
@Book(value="红楼梦",authors = "曹雪芹")
public class BookShelf {
    // 使用Book注解：正确方式
    @Book(value="西游记",authors = {"吴承恩","白求恩"})
    public void showBook(){

    }
}
```



```
// 使用Book注解：错误方式
public class BookShelf {
    @Book("西游记", authors = {"吴承恩", "白求恩"})
    public void showBook(){

    }
}
// 此时value属性名不能省略了。
```

3.3.4 问题分析

现在我们已经学会了如何定义注解以及如何使用注解了，可能细心的同学会发现一个问题：我们定义的注解是可以使用在任何成员上的，比如刚刚Book注解的使用：

```
// 定义注解Book
public @interface Book {
    // 书名
    String value();
    // 价格
    double price() default 100;
    // 多位作者
    String[] authors();
}
// 使用Book注解：正确方式
@Book(value="红楼梦", authors = "曹雪芹")
public class BookShelf {
    // 使用Book注解：正确方式
    @Book(value="西游记", authors = {"吴承恩", "白求恩"})
    public void showBook(){

    }
}
```

- 此时Book同时使用在了类定义上或成员方法上，编译器也没有报错，因为默认情况下，注解可以用在任何地方，比如类，成员方法，构造方法，成员变量等地方。
- 如果要限制注解的使用位置怎么办？那就要学习一个新的知识点：**元注解**。接下来就来看看什么是元注解以及怎么使用。

3.4 注解之元注解

3.4.1 元注解的概述

- Java API提供的注解
- 专门用来定义注解的注解。
- 任何Java官方提供的非元注解的定义中都使用到了元注解。

3.4.2 常用元注解

- @Target
- @Retention

3.4.2.1 元注解之@Target

- 作用：指明此注解用在哪个位置，如果不写默认是任何地方都可以使用。
 - 可选的参数值在枚举类ElementType中包括：

TYPE：用在类、接口上
FIELD：用在成员变量上
METHOD：用在方法上
PARAMETER：用在参数上
CONSTRUCTOR：用在构造方法上
LOCAL_VARIABLE：用在局部变量上

3.4.2.2 元注解之@Retention

- 作用：定义该注解的生命周期(有效范围)。
 - 可选的参数值在枚举类型RetentionPolicy中包括

SOURCE：注解只存在于Java源代码中，编译生成的字节码文件中就不存在了。
CLASS：注解存在于Java源代码、编译以后的字节码文件中，运行的时候内存中没有，默认值。
RUNTIME：注解存在于Java源代码中、编译以后的字节码文件中、运行时内存中，程序可以通过反射获取该注解。

3.4.3 元注解使用示例

```
@Target({ElementType.METHOD, ElementType.TYPE})
@interface Stu{
    String name();
}

// 类
@Stu(name="jack")
public class AnnotationDemo02 {

    // 成员变量
    @Stu(name = "lily") // 编译失败
    private String gender;

    // 成员方法
    @Stu(name="rose")
    public void test(){

    }

    // 构造方法
    @Stu(name="lucy") // 编译失败
    public AnnotationDemo02(){}
}
```



```
@Target({ElementType.METHOD, ElementType.TYPE})
@interface Stu {
    String name();
}

// 类
@Stu(name="jack")
public class AnnotationDemo02 {

    // 成员变量
    @Stu(name = "lily")
    private String gender;

    // 成员方法
    @Stu(name="rose")
    public void test() {

    }

    // 构造方法
    @Stu(name="lucy")
    public AnnotationDemo02() {}
}
```

指定了注解的使用位置：成员方法和类，接口上，其他地方就不能使用

注解Stu不能使用在成员变量上了

注解Stu不能使用在构造方法上了

3.5 注解解析

3.5.1 什么是注解解析

- 通过Java技术获取注解数据的过程则称为注解解析。

3.5.2 与注解解析相关的接口

- Anotation**：所有注解类型的公共接口，类似所有类的父类是Object。
- AnnotatedElement**：定义了与注解解析相关的方法，常用方法以下几个：

```
boolean isAnnotationPresent(Class annotationClass); 判断当前对象是否有指定的注解，有则返回true，否则返回false。
```

```
T getAnnotation(Class<T> annotationClass); 获得当前对象上指定的注解对象。
```

```
Annotation[] getAnnotations(); 获得当前对象及其从父类上继承的所有的注解对象。
```

3.5.3 获取注解数据的原理

注解作用在那个成员上，就通过反射获得该成员的对象来得到它的注解。

- 如注解作用在方法上，就通过方法(Method)对象得到它的注解

```
// 得到方法对象
Method method = clazz.getDeclaredMethod("方法名");
// 根据注解名得到方法上的注解对象
Book book = method.getAnnotation(Book.class);
```

- 如注解作用在类上，就通过Class对象得到它的注解



```
// 获得Class对象
Class c = 类名.class;
// 根据注解的Class获得使用在类上的注解对象
Book book = c.getAnnotation(Book.class);
```

3.5.4 使用反射获取注解的数据

3.5.4.1 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

3.5.4.2 代码实现

1. 注解Book

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

2. BookStore类

```
@Book(value = "红楼梦", authors = "曹雪芹", price = 998)
public class BookStore {

    @Book(value = "西游记", authors = "吴承恩")
    public void buyBook(){

    }
}
```

3. TestAnnotation类

```
public class TestAnnotation {
    public static void main(String[] args) throws Exception{

        System.out.println("-----获取类上注解的数据-----");
    }
}
```



```
test01();
System.out.println("-----获取成员方法上注解的数据-----");
test02();
}

/**
 * 获取BookStore类上使用的Book注解数据
 */
public static void test01(){
    // 获得BookStore类对应的Class对象
    Class c = BookStore.class;
    // 判断BookStore类是否使用了Book注解
    if(c.isAnnotationPresent(Book.class)) {
        // 根据注解Class对象获取注解对象
        Book book = (Book) c.getAnnotation(Book.class);
        // 输出book注解属性值
        System.out.println("书名：" + book.value());
        System.out.println("价格：" + book.price());
        System.out.println("作者：" + Arrays.toString(book.authors()));
    }
}

/**
 * 获取BookStore类成员方法buyBook使用的Book注解数据
 */
public static void test02() throws Exception{
    // 获得BookStore类对应的Class对象
    Class c = BookStore.class;
    // 获得成员方法buyBook对应的Method对象
    Method m = c.getMethod("buyBook");
    // 判断成员方法buyBook上是否使用了Book注解
    if(m.isAnnotationPresent(Book.class)) {
        // 根据注解Class对象获取注解对象
        Book book = (Book) m.getAnnotation(Book.class);
        // 输出book注解属性值
        System.out.println("书名：" + book.value());
        System.out.println("价格：" + book.price());
        System.out.println("作者：" + Arrays.toString(book.authors()));
    }
}
}
```

3.6 注解案例

3.5.1 案例说明

模拟JUnit测试的@Test

3.5.2 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。

3.5.3 案例代码

1. 注解MyTest

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

1. 目标类MyTestDemo

```
public class MyTestDemo {
    @MyTest
    public void test01(){
        System.out.println("test01");
    }

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

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

1. 调用类TestMyTest

```
public class TestMyTest {
    public static void main(String[] args) throws Exception{
        // 获得MyTestDemo类Class对象
        Class c = MyTestDemo.class;
        // 获得所有的成员方法对象
        Method[] methods = c.getMethods();
        // 创建MyTestDemo类对象
        Object obj = c.newInstance();
        // 遍历数组
        for (Method m:methods) {
            // 判断方法m上是否使用注解MyTest
            if(m.isAnnotationPresent(MyTest.class)){
                // 执行方法m
                m.invoke(obj);
            }
        }
    }
}
```



- 输出结果：

```
Run TestMyTest
/Library/Java/JavaVirtualMachines/jdk1.8.0_121.jdk/Contents/Home/bin/java ...
test01
test03
Process finished with exit code 0
```