单元测试

单元测试概述

- 概念

单元测试就是针对最小的功能单元编写测试代码，Java程序最小的功能单元是方法，因此，单元测试就是针对Java方法的测试，进而检查方法的正确性

- **目前测试方法是怎么进行的，存在什么问题**

1. 只有一个main方法，如果一个方法的测试失败了，其他方法测试会受到影响。
2. 无法得到测试的结果报告，需要程序员自己去观察测试是否成功。
3. 无法实现自动化测试。

- **Junit单元测试框架**

JUnit是使用Java语言实现的单元测试框架，它是开源的，Java开发者都应当学习并使用JUnit编写单元测试。
此外，几乎所有的IDE工具都集成了JUnit，这样我们就可以直接在IDE中编写并运行JUnit测试，JUnit目前最新版本是5。

- **Junit优点**

1. JUnit可以灵活的选择执行哪些测试方法，可以一键执行全部测试方法。
2. Junit可以生成全部方法的测试报告。
3. 单元测试中的某个方法测试失败了，不会影响其他测试方法的测试。

- 小结

Junit单元测试是做什么的?
　测试类中方法的正确性的。
Junit单元测试的优点是什么?
　JUnit可以选择执行哪些测试方法，可以一键执行全部测试方法的测试。
　JUnit可以生测试报告，如果测试良好则是绿色；如果测试失败，则是红色。
　单元测试中的某个方法测试失败了，不会影响其他测试方法的测试。

单元测试快速入门

- 步骤

1. 将JUnit的jar包导入到项目中
2. IDEA通常整合好了Junit框架，一般不需要导入。
   如果IDEA没有整合好，需要自己手工导入如下2个JUnit的jar包到模块（hamcrest-core1.3、junit-4.12）
3. 编写测试方法：该测试方法必须是公共的无参数无返回值的非静态方法。
   public void testxxx(){}
4. 在测试方法上使用@Test注解：标注该方法是一个测试方法
5. 在测试方法中完成被测试方法的预期正确性测试。
6. 选中测试方法，选择“JUnit运行”，如果测试良好则是绿色；如果测试失败，则是红色

- 代码实现

```
package test_demo;

import org.junit.Assert;
import org.junit.Test;


public class JunitDemo {
    public boolean isRunning(int number) {
        if (number >= 0) return false;
        return true;
    }

    public String isActive(int number) {
        if (number >= 0) return "登录失败";
        return "登录成功";
    }
}
```

```
package test_demo;

import org.junit.Assert;
import org.junit.Test;

public class TestDemo {
    private JunitDemo junitDemo = new JunitDemo();

    @Test
```

```
    public void test1() {
        final boolean rs = junitDemo.isRunning(-4);
        System.out.println(rs);
        //结果是真的就返回真，否则返回报错，信息为填写的信息 java.lang.AssertionError: the result is false
        Assert.assertTrue("the result is false", rs);
        //结果如果是真就返回真，如果是假就报错 java.lang.AssertionError
        //Assert.assertTrue(rs);
        //结果是假就返回Boolean的假，结果是真就报错 java.lang.AssertionError
//        Assert.assertFalse(rs);
        //结果是false的就返回false，结果是true就报错，信息为填写的信息 java.lang.AssertionError: the answer is
true!
//        Assert.assertFalse("the answer is true! ", rs);
    }


    @Test
    public void test() {
        /*   不匹配就报错
             org.junit.ComparisonFailure:
            预期:登录失败
            实际:登录成功
        * */
        final String rs = junitDemo.isActive(1);
        Assert.assertEquals("登录失败", rs);
    }
}
```

- 小结

JUnit单元测试的实现过程是什么样的
必须导入Junit框架的jar包。
定义的测试方法必须是无参数无返回值，且公开的方法。
测试方法使用@Test注解标记。

JUnit测试某个方法，测试全部方法怎么处理？成功的标志是什么
测试某个方法直接右键该方法启动测试。
测试全部方法，可以选择类或者模块启动。
红色失败，绿色通过

单元测试常用注解

- **Junit4的常用注解**

注解	说明
@Test	测试方法
@Before	用来修饰实例方法，该方法会在每一个测试方法执行之前执行一次。
@After	用来修饰实例方法，该方法会在每一个测试方法执行之后执行一次。
@BeforeClass	用来静态修饰方法，该方法会在所有测试方法之前只执行一次。
@AfterClass	用来静态修饰方法，该方法会在所有测试方法之后只执行一次。

开始执行的方法:初始化资源。
执行完之后的方法:释放资源

- 代码实现

```
package test_demo;

import org.junit.*;

public class JunitDemo2 {

    @Test
    public void test1(){
        System.out.println("@Test注解1");
    }

    @Test
    public void test2(){
        System.out.println("@Test注解2");
    }
    @Test
    public void test3(){
        System.out.println("@Test注解3");
    }

    @Before
    public void test4(){
        System.out.println("@Before注解");
    }

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


    /*
    *   1. Method testBeforeClass() should be static
    * */
    @BeforeClass
    public static void testBeforeClass(){
        System.out.println("@BeforeClass注解");
    }
```

```
    /*
    *   2. Method testAfterClass() should be static
    * */
    @AfterClass
    public static void testAfterClass(){
        System.out.println("@AfterClass注解");
    }

}
```

```
测试整个类：

@BeforeClass注解

@Before注解
@Test注解1
@After注解

@Before注解
@Test注解2
@After注解

@Before注解
@Test注解3
@After注解

@AfterClass注解
```

```
测试单个实例方法

@BeforeClass注解
@Before注解
@Test注解3
@After注解
@AfterClass注解
```

- **Junit5的常用注解**

注解	说明
@Test	测试方法
@BeforeEach	用来修饰实例方法，该方法会在每一个测试方法执行之前执行一次。
@AfterEach	用来修饰实例方法，该方法会在每一个测试方法执行之后执行一次。
@BeforeAll	用来静态修饰方法，该方法会在所有测试方法之前只执行一次。
@AfterAll	用来静态修饰方法，该方法会在所有测试方法之后只执行一次。

一 一对应

反射

反射概述

- 概念

反射是指对于任何一个Class类，在"运行的时候"都可以直接得到这个类全部成分。
在运行时,可以直接得到这个类的构造器对象：Constructor
在运行时,可以直接得到这个类的成员变量对象：Field
在运行时,可以直接得到这个类的成员方法对象：Method

- **Java语言的反射机制**

主要是指程序可以访问、检测和修改它本身状态或行为的一种能力
运行时 动态获取类信息 以及 动态调用类中成分 的能力

- **反射的关键**

反射的第一步都是先得到编译后的Class类对象，然后就可以得到Class的全部成分

```
HelloWorld.java -> javac -> HelloWorld.class
Class c = HelloWorld.class;
```

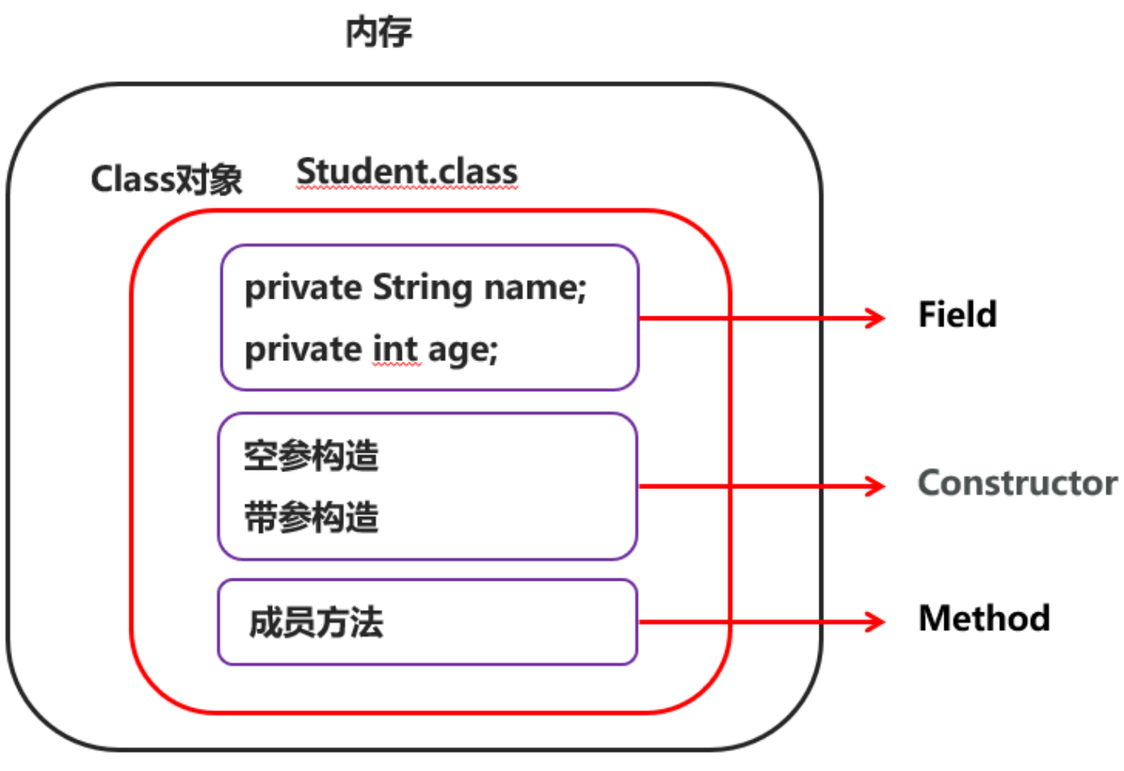
- 小结

反射是在运行时获取类的字节码文件对象：然后可以解析类中的全部成分。
反射的核心思想和关键就是:得到编译以后的class文件对象。

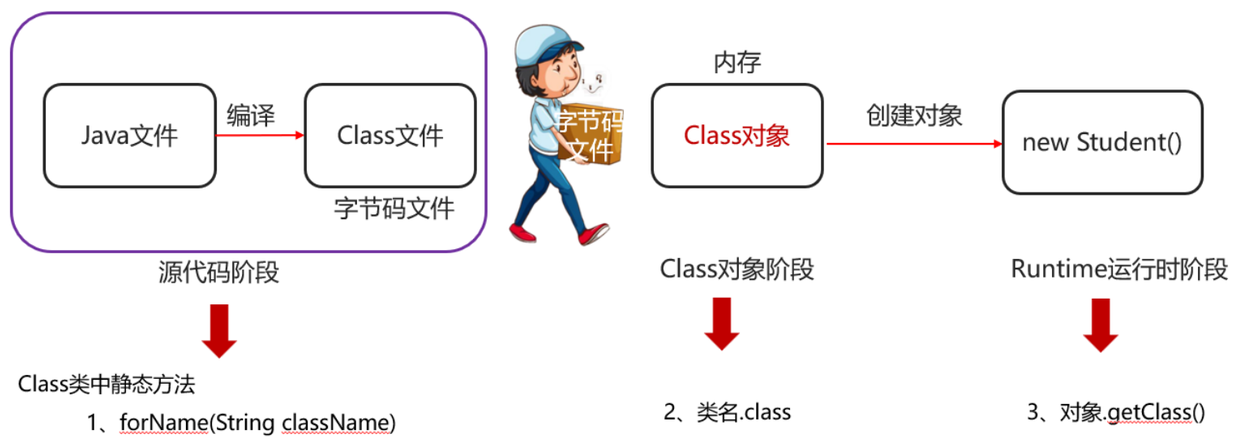
反射获取类对象

- **反射的第一步：获取Class类的对象**

- **Class类结构分析**



- **三种获取Class对象的方法对应的阶段**



- 代码实现

```
package reflect_demo;


/**
 * ClassName: ReflectDemoGetClass <br/>
 * Description: <br/>
 * date: 2022/3/17 15:55<br/>
 *
 * @author fgcy<br />
 * @since JDK 1.8
 */
public class ReflectDemoGetClass {
    //获取Class对象
    public static void main(String[] args) throws Exception {
        //方式一:Class类中的一个静态方法  Class.forName()
        final Class c1 = Class.forName("reflect_demo.Student");
        System.out.println(c1);
        //方式二:
        final Class c2 = Student.class;
        System.out.println(c2);
        //方式三:Object类有一个getClass的实例方法，所以任何个类的实例都有这个方法
        final Class c3 = new Student().getClass();
        System.out.println(c3);
    }
}
```

```
class Student {

}
```

```
class reflect_demo.Student
class reflect_demo.Student
class reflect_demo.Student
```

- 小结

反射的第一步是什么
　获取Class类对象，如此才可以解析类的全部成分

获取Class类的对象的三种方式

```
// 方式一：
 Class c1 = Class.forName("全类名");
 // 方式二：
 Class c2 = 类名.class
  //方式三：
 Class c3 = 对象.getClass();
```

反射获取构造器对象

- 步骤

反射的第一步是先得到类对象，然后从类对象中获取类的成分对象。
Class类中用于获取构造器的方法，如下：

方法	说明
Constructor<?>[] getConstructors()	返回所有构造器对象的数组（只能拿public的）
Constructor<?>[] getDeclaredConstructors()	返回所有构造器对象的数组，存在就能拿到
Constructor<T> getConstructor(Class<?>... parameterTypes)	返回单个构造器对象（只能拿public的）
Constructor<T> getDeclaredConstructor(Class<?>... parameterTypes)	返回单个构造器对象，存在就能拿到

- 代码实现

```
public class Students {
    private String name;
    private int age;
    private char sex;
    private static String userInfo;
    private static final String LIKE = "WAITING";

    public Students() {

    }

    private Students(String name) {
        this.name = name;
    }

    public Students(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public Students(String name, int age, char sex) {
        this.name = name;
        this.age = age;
        this.sex = sex;
    }

    @Override
    public String toString() {
        return "Students{" +
                "name='" + name + '\'' +
                ", age=" + age +
                ", sex=" + sex +
                '}';
    }

    public String getName() {
        return name;
    }

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

    public int getAge() {
        return age;
    }

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

    public char getSex() {
        return sex;
    }

    public void setSex(char sex) {
        this.sex = sex;
    }
}
```

```
    @Test
    public void test1() {
        final Class c = Students.class;

        //通过Class类对象获取该类的所有用public修饰的构造器 返回一个构造器数组
        final Constructor[] constructors1 = c.getConstructors();
```

```
        final StringBuilder sb1 = new StringBuilder("(");
        for (Constructor constructor : constructors1) {
            System.out.println(constructor.getName() + "参数是" + constructor.getParameterCount() + "个");
        }
        System.out.println("----------------------------");

        //通过Class类对象获取该类的所有的构造器 返回一个构造器数组
        final Constructor[] declaredConstructors = c.getDeclaredConstructors();
        for (Constructor declaredConstructor : declaredConstructors) {
            System.out.println(declaredConstructor.getName() + "参数是" +
declaredConstructor.getParameterCount() + "个");
        }
        System.out.println("----------------------------");

        try {
            //通过Class类对象获取该类的某个用public修饰的构造器 入参为类对象 此处为无参构造
            final Constructor constructor1 = c.getConstructor();
            System.out.println(constructor1.getName() + "参数是" + constructor1.getParameterCount() + "个");
            System.out.println("----------------------------");
            /*
              private Students(String name) {
                    this.name = name;
                }
            * */
            //通过Class类对象获取该类的某个构造器 入参为类对象 此处是获取一个参数的私有构造
            final Constructor declaredConstructor = c.getDeclaredConstructor(String.class);
            System.out.println(constructor1.getName() + "参数是" + declaredConstructor.getParameterCount() +
"个");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
```

```
reflect_demo.Students参数是3个
reflect_demo.Students参数是2个
reflect_demo.Students参数是0个
----------------------------
reflect_demo.Students参数是3个
reflect_demo.Students参数是2个
reflect_demo.Students参数是1个
reflect_demo.Students参数是0个
----------------------------
reflect_demo.Students参数是0个
----------------------------
reflect_demo.Students参数是1个
```

- **使用反射技术获取构造器对象并使用**

获取构造器的作用依然是初始化一个对象返回

Constructor类中用于创建对象的方法

符号	说明
T newInstance(Object... initargs)	根据指定的构造器创建对象
public void setAccessible(boolean flag)	设置为true,表示取消访问检查，进行暴力反射

```
public class Students {
    private String name;
    private int age;
    private char sex;
    private static String userInfo;
    private static final String LIKE = "WAITING";

    public Students() {

    }

    private Students(String name) {
        this.name = name;
    }

    public Students(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public Students(String name, int age, char sex) {
        this.name = name;
        this.age = age;
        this.sex = sex;
    }

    @Override
    public String toString() {
        return "Students{" +
                "name='" + name + '\'' +
                ", age=" + age +
                ", sex=" + sex +
                '}';
    }

    public String getName() {
        return name;
    }

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

    public int getAge() {
        return age;
    }
```

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

    public char getSex() {
        return sex;
    }

    public void setSex(char sex) {
        this.sex = sex;
    }
}
```

```
    @Test
    public void test2() {
        try {
            final Class c = Students.class;
             /*
               private Students(String name) {
                  this.name = name;
               }
            * */
            //获取一个私有类型的有参构造
            final Constructor constructor = c.getDeclaredConstructor(String.class);

            //暴力反射，获取访问该构造器的权限
            constructor.setAccessible(true);
            /*
             * java.lang.IllegalAccessException: Class reflect_demo.ConstructorDemo1 can not access
             * a member of class reflect_demo.Students with modifiers "private"
             * */
            //解决方法，暴力反射，赋权（本次有效）
            //newInstance(Object...args)用可变长参数
            //反射：以前是new 构造器，现在是构造器调用方法来new自己
            final Object o = constructor.newInstance("张氏情歌");//如果访问一个用private修饰的类成员，就会报错
            System.out.println(o);//多态，使用的是子类重写的方法

            //获取一个public修饰的无参构造
            final Constructor noArgsConstructor = c.getConstructor();

            //公开的成员可以直接访问
            final Object o1 = noArgsConstructor.newInstance();
            System.out.println(o1);//多态
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
```

```
Students{name='张氏情歌', age=0, sex= }
Students{name='null', age=0, sex= }
```

- 小结

利用反射技术获取构造器对象的方式
  getDeclaredConstructors()
  getDeclaredConstructor (Class<?>... parameterTypes)
反射得到的构造器可以做什么?
  依然是创建对象的
  public newInstance(Object... initargs)
如果是非public的构造器，需要打开权限（暴力反射），然后再创建对象
  setAccessible(boolean)
  反射可以破坏封装性，私有的也可以执行了

反射获取成员变量对象

- **使用反射技术获取成员变量对象并使用**

反射的第一步是先得到类对象，然后从类对象中获取类的成分对象。
获取成员变量的作用依然是在某个对象中取值、赋值
Class类中用于获取成员变量的方法,如下：

方法	说明
Field[] getFields()	返回所有成员变量对象的数组（只能拿public的）
Field[] getDeclaredFields()	返回所有成员变量对象的数组，存在就能拿到
Field getField(String name)	返回单个成员变量对象（只能拿public的）
Field getDeclaredField(String name)	返回单个成员变量对象，存在就能拿到

- 代码实现1

```
public class Students {
    private String name;
    private int age;
    private char sex;
    private static String userInfo;
    private static final String LIKE = "WAITING";

    public Students() {

    }

    private Students(String name) {
        this.name = name;
```

```
    }

    public Students(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public Students(String name, int age, char sex) {
        this.name = name;
        this.age = age;
        this.sex = sex;
    }

    @Override
    public String toString() {
        return "Students{" +
                "name='" + name + '\'' +
                ", age=" + age +
                ", sex=" + sex +
                '}';
    }

    public String getName() {
        return name;
    }

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

    public int getAge() {
        return age;
    }

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

    public char getSex() {
        return sex;
    }

    public void setSex(char sex) {
        this.sex = sex;
    }
}
```

```
    @Test
    public void test6() {
        final Class c = Students.class;
        //获取全部的成员属性（静态成员属性，实例成员属性）
        final Field[] fields = c.getDeclaredFields();
        for (Field field : fields) {
            System.out.println(field.getType() + " " + field.getName());
        }
    }
```

```
class java.lang.String name
int age
char sex
class java.lang.String userInfo
class java.lang.String LIKE
```

Field类中用于取值、赋值的方法

符号	说明
void set(Object obj, Object value)：	赋值
Object get(Object obj)	获取值。

- 代码实现2

```
public class Students {
    private String name;
    private int age;
    private char sex;
    private static String userInfo;
    private static final String LIKE = "WAITING";

    public Students() {

    }

    private Students(String name) {
        this.name = name;
    }

    public Students(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public Students(String name, int age, char sex) {
        this.name = name;
        this.age = age;
        this.sex = sex;
    }

    @Override
```

```
    public String toString() {
        return "Students{" +
                "name='" + name + '\'' +
                ", age=" + age +
                ", sex=" + sex +
                '}';
    }

    public String getName() {
        return name;
    }

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

    public int getAge() {
        return age;
    }

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

    public char getSex() {
        return sex;
    }

    public void setSex(char sex) {
        this.sex = sex;
    }
}
```

```
    @Test
    public void test7() {
        try {
            final Class c = Students.class;
            //获取一个无参构造器对象
            final Constructor constructor = c.getDeclaredConstructor();
            final Object o = constructor.newInstance();
            //获取一个私有成员
            final Field nameF = c.getDeclaredField("name");
            //暴力反射
            nameF.setAccessible(true);
            nameF.set(o, "桃燃里");
            System.out.println(o);
            System.out.println(nameF.get(o));
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
```

```
Students{name='桃燃里', age=0, sex= }
桃燃里
```

- 小结

利用反射技术获取成员变量的方式
  获取类中成员变量对象的方法
  getDeclaredFields()
  getDeclaredField (String name)
反射得到成员变量可以做什么?
  依然是在某个对象中取值和赋值。
  void set(Object obj, Object value):
  Object get(Object obj)
如果某成员变量是非public的，需要打开权限（暴力反射），然后再取值、赋值
  setAccessible(boolean)

反射获取方法对象

- **使用反射技术获取方法对象并使用**

反射的第一步是先得到类对象，然后从类对象中获取类的成分对象。
获取成员方法的作用依然是在某个对象中进行执行此方法
Class类中用于获取成员方法的方法，如下：

方法	说明
Method[] getMethods()	返回所有成员方法对象的数组（只能拿public的）
Method[] getDeclaredMethods()	返回所有成员方法对象的数组，存在就能拿到
Method getMethod(String name, Class<?>... parameterTypes)	返回单个成员方法对象（只能拿public的）
Method getDeclaredMethod(String name, Class<?>... parameterTypes)	返回单个成员方法对象，存在就能拿到

- 代码实现

```
public class Dog {
    private String name;
    private int age;
    private char sex;

    public Dog() {
    }

    public Dog(String name, int age, char sex) {
```

```
        this.name = name;
        this.age = age;
        this.sex = sex;
    }

    @Override
    public String toString() {
        return "Dog{" +
                "name='" + name + '\'' +
                ", age=" + age +
                ", sex=" + sex +
                '}';
    }

    private String eat(int times) {
        return "狗一天吃" + times + "次";
    }

    void eat(String foodName) {
        System.out.println("狗吃" + foodName);
    }

    public static int do_() {
        return 1;
    }
}
```

```
    @Test
    public void test1() {
        final Class c = Dog.class;
        //获取全部方法
        final Method[] methods = c.getDeclaredMethods();
        for (Method method : methods) {
            System.out.println("方法名：" + method.getName() + " 返回值：" + method.getReturnType() + " 参数个
数：" +
                    method.getParameterCount());
        }
    }
```

```
方法名：toString 返回值：class java.lang.String 参数个数：0
方法名：eat 返回值：void 参数个数：1
方法名：eat 返回值：class java.lang.String 参数个数：1
方法名：do_ 返回值：int 参数个数：0
```

Method类中用于触发执行的方法

符号	说明
Object invoke(Object obj, Object... args)	运行方法 参数一：用obj对象调用该方法 参数二：调用方法的传递的参数（如果没有就不写） 返回值：方法的返回值（如果没有就不写）

- 代码实现

```
public class Dog {
    private String name;
    private int age;
    private char sex;

    public Dog() {
    }

    public Dog(String name, int age, char sex) {
        this.name = name;
        this.age = age;
        this.sex = sex;
    }

    @Override
    public String toString() {
        return "Dog{" +
                "name='" + name + '\'' +
                ", age=" + age +
                ", sex=" + sex +
                '}';
    }

    private String eat(int times) {
        return "狗一天吃" + times + "次";
    }

    void eat(String foodName) {
        System.out.println("狗吃" + foodName);
    }

    public static int do_() {
        return 1;
    }
}
```

```
    @Test
    public void test3() {
        try {
            final Class c = Dog.class;
            //获取一个无参构造实例
            final Constructor constructor = c.getDeclaredConstructor();
            //通过调用构造实例的newInstance方法获取狗对象
```

```
            final Dog dog = (Dog) constructor.newInstance();
            //根据方法名和形参列表获得一个方法实例
            final Method eat1 = c.getDeclaredMethod("eat", int.class);
            //暴力反射
            eat1.setAccessible(true);
            //通过方法对象调用invoke方法，调用狗对象的方法
            //java.lang.IllegalAccessException: Class reflect_demo.MethodDemo
            // can not access a member of class reflect_demo.Dog with modifiers "private"
            System.out.println(eat1.invoke(dog, 3));

            final Method eat2 = c.getDeclaredMethod("eat", String.class);
            //方法返回值为void是，此处返回值为null
            //不是私有方法可以访问，包权限
            System.out.println(eat2.invoke(dog, "人血馒头"));
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
```

```
狗一天吃3次
狗吃人血馒头
null
```

- 小结

利用反射技术获取成员方法对象的方式
  获取类中成员方法对象
  getDeclaredMethods()
  getDeclaredMethod (String name, Class<?>... parameterTypes)

反射得到成员方法可以做什么?
  依然是在某个对象中触发该方法执行。
  Object invoke(Object obj, Object... args)

如果某成员方法是非public的，需要打开权限（暴力反射），然后再触发执行
  setAccessible(boolean)

反射的作用-绕过编译阶段为集合添加数据

- 概念

反射是作用在运行时的技术，此时集合的泛型将不能产生约束了，此时是可以为集合存入其他任意类型的元素的。
泛型只是在编译阶段可以约束集合只能操作某种数据类型，在编译成Class文件进入运行阶段的时候，其真实类型都是ArrayList了，泛型相当于被擦除了

- 代码实现

```
@Test
    public void test4() throws Exception {
        List<Integer> list1 = new ArrayList<>();
        List<Integer> list2 = new ArrayList<>();
        System.out.print("由ArrayList类实例出来的对象，属于同一份class字节码生成的:");
        System.out.println(list1.getClass() == list2.getClass());
        list1.add(12);
        list1.add(1);
        list1.add(88);
        //编译的时候就报错
        //list1.add("88");

        final Class c = list1.getClass();
        //在运行阶段往ArrayList中添加元素，绕过编译阶段的泛型约束
        final Method add = c.getDeclaredMethod("add", Object.class);
        System.out.println("在泛型为Integer的List中添加字符串:" + add.invoke(list1, "字符串"));
        System.out.println(list1);
    }
```

```
由ArrayList类实例出来的对象，属于同一份class字节码生成的:true
在泛型为Integer的List中添加字符串:true
[12, 1, 88, 字符串]
```

- 小结

反射为何可以给约定了泛型的集合存入其他类型的元素?
  编译成Class文件进入运行阶段的时候，泛型会自动擦除。
  反射是作用在运行时的技术，此时已经不存在泛型了

反射的作用-通用框架的底层原理

- 使用反射做一个简易的框架

需求
  给你任意一个对象，在不清楚对象字段的情况可以，可以把对象的字段名称和对应值存储到文件中去。
分析
  定义一个方法，可以接收任意类的对象。
  每次收到一个对象后，需要解析这个对象的全部成员变量名称。
  这个对象可能是任意的，那么怎么样才可以知道这个对象的全部成员变量名称呢?
  使用反射获取对象的Class类对象，然后获取全部成员变量信息。
  遍历成员变量信息，然后提取本成员变量在对象中的具体值
  存入成员变量名称和值到文件中去即可。

- 代码实现

```
package reflect_demo;

import org.junit.Test;

import java.io.FileWriter;
import java.io.PrintWriter;
import java.lang.reflect.Field;

public class Autils {
```

```
    public static void saveData(Object object) {
        try (
                final PrintWriter pw = new PrintWriter(new FileWriter("src/reflect_demo/data.txt", true));
        ) {
            final Class c = object.getClass();
            pw.println("=======================" + c.getSimpleName() + "==============================");
            final Field[] fields = c.getDeclaredFields();
            for (Field field : fields) {
                field.setAccessible(true);
                final String name = field.getName();
                final String value = field.get(object) + "";
                pw.println(name + "=" + value);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
```

```
    @Test
    public void test1() {
       Autils.saveData(new Dog("旺财", 3, '雄'));
        Autils.saveData(new Students("王小明", 24, '男'));
    }
}
```

```
=======================Dog==============================
name=旺财
age=3
sex=雄
=======================Students==============================
name=王小明
age=24
sex=男
userInfo=null
LIKE=WAITING
```

- 小结

反射的作用?
可以在运行时得到一个类的全部成分然后操作。
可以破坏封装性。（很突出）
也可以破坏泛型的约束性。（很突出）
更重要的用途是适合：做Java高级框架
基本上主流框架都会基于反射设计一些通用技术功能

注解

注解概述

- 概念

Java 注解（Annotation）又称 Java 标注，是 JDK5.0 引入的一种注释机制。
Java 语言中的类、构造器、方法、成员变量、参数等都可以被注解进行标注。
注释给人观看的，注解给代码观看的

- **注解的作用是什么呢?**

对Java中类、方法、成员变量做标记，然后进行特殊处理，至于到底做何种处理由业务需求来决定。
例如：JUnit框架中，标记了注解@Test的方法就可以被当成测试方法执行，而没有标记的就不能当成测试方法执行。

- 小结

注解的作用
对Java中类、方法、成员变量做标记，然后进行特殊处理。
例如：JUnit框架中，标记了注解@Test的方法就可以被当成测试方法执行，而没有标记的就不能当成测试方法执行

自定义注解

- 格式

```
public @interface 注解名称 {
    public 属性类型 属性名() default 默认值 ;
}
//Java支持的数据类型 基本上都支持
```

- **特殊属性**

value属性，如果只有一个value属性的情况下，使用value属性的时候可以省略value名称不写!!
但是如果有多个属性，且多个属性没有默认值，那么value名称是不能省略的。

```
//自定义注解
public @interface MyBook {
    //public String name();//默认自带public，可以省略
    String name();

    double price();

    int number();
}
```

```
@interface Book {
    String value();//当只有一个值时，用value作为属性名，使用该注解时，可以省略键不写

    String author() default "衣冠禽兽";//当注解中的属性不止一个时，就不能省略键，除非除了value外，其他的属性都有默认值
}
```

```
/*
 * 在没有用元注解规定该注解的使用范围时，该注解可以在类中的任何成员上使用，包括类，成员变量，局部变量，方法
 * */
//当自定义的注解中，属性没有默认值时，在使用该注解时，必须声明全部没有默认值的属性
```

```
@MyBook(name = "swx", price = 12.4, number = 2)
class TestAnnotation {

    @MyBook(name = "SSS", price = 14.4, number = 0)
    private int number;

    @MyBook(name = "SSS", price = 14.4, number = 0)
    //有两个属性，一个是value，一个是有默认值的属性
    public TestAnnotation(@Book("可以省略前面的键") int number) {//省略键不写的情况
        this.number = number;

        @Book(value = "aaa", author = "误人子弟")
        String name;
    }
}
```

元注解

- 定义

元注解：就是注解注解的注解

- 常用的两个元注解

@Target: 约束自定义注解只能在哪些地方使用，
@Retention：申明注解的生命周期
@Documented：描述在使用 javadoc 工具为类生成帮助文档时是否要保留其注解信息
@Inherited：inherited注解的作用是：使被它修饰的注解具有继承性（如果某个类使用了被@Inherited修饰的注解，则其子类将自动具有该注解）

- @Target中可使用的值定义在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(入参)
CONSTRUCTOR, 构造器
LOCAL_VARIABLE, 局部变量

- @Retention中可使用的值定义在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（开发常用）

- 代码实现

```
package annotation_demo;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;


//@Target({ElementType.TYPE})//也可以使用，通过数组接一个
//@Target(ElementType.TYPE)//可以作用于类或接口上
@Retention(RetentionPolicy.RUNTIME)//只能选择一种

// 1构造器  2成员方法 3成员变量   4局部变量     5类、接口；     6方法参数(入参)
@Target({ElementType.CONSTRUCTOR, ElementType.METHOD, ElementType.FIELD, ElementType.LOCAL_VARIABLE,
ElementType.TYPE,
         ElementType.PARAMETER})
public @interface MyTest {
}


@MyTest
class User {
    @MyTest
    private String name;

    @MyTest
    public User() {
    }

    public void test_(@MyTest String a_) {
        @MyTest
        int number;
    }
}
```

@Inherited

```
//----------------MyInheritedAnnotation注解-----------------------------
@Inherited
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface MyInheritedAnnotation {
    public String name() default "pengjunlee";
}
```

```
//-------------Parent类---------------------------------------
    @MyInheritedAnnotation(name="parent")
    public class Parent {}
```

```
//--------------Child类-----------------------------------------
public class Child extends Parent{
    public static void main(String[] args) {
        Class<Child> child=Child.class;
        MyInheritedAnnotation annotation = child.getAnnotation(MyInheritedAnnotation.class);
        System.out.println(annotation.name());
    }
}
```

```
parent
```

- 小结

元注解是什么
  注解注解的注解

@Target约束自定义注解可以标记的范围
  @Retention用来约束自定义注解的存活范围。

注解解析

- 概念

注解的操作中经常需要进行解析，注解的解析就是判断是否存在注解，存在注解就解析出内容

- **与注解解析相关的接口**

Annotation: 注解的顶级接口，注解都是Annotation类型的对象
AnnotatedElement:该接口定义了与注解解析相关的解析方法
所有的类成分Class, Method , Field , Constructor，都实现了AnnotatedElement接口他们都拥有解析注解的能力

方法	说明
Annotation[] getDeclaredAnnotations()	获得当前对象上使用的所有注解，返回注解数组。
T getDeclaredAnnotation(Class<T> annotationClass)	根据注解类型获得对应注解对象
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

- **解析注解的技巧**

注解在哪个成分上，我们就先拿哪个成分对象。
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，再来拿上面的注解
比如注解作用在类上，则要该类的Class对象，再来拿上面的注解
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，再来拿上面的注解

- 代码体现

```
package annotation_demo;


import java.lang.annotation.*;
import java.lang.reflect.Method;
import java.util.Arrays;


@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE, ElementType.METHOD})
public @interface Bookk {
    String name() default "无名";

    String[] authors() default {"佚名"};

    double price();
}
```

```
@Bookk(name = "斗罗大陆", price = 9.9)
class BookStore {

    @Bookk(price = 444.11, authors = {"宅猪", "猫腻", "我吃西红柿", "佛前献花"})
    private void te() {
    }
}

class TestBook {
    public static void main(String[] args) {
        try {
            test1();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
```

```
 public static void test1() throws NoSuchMethodException {
        final Class c = BookStore.class;
        //首先判断该类上有没有Bookk的注解
        if (c.isAnnotationPresent(Bookk.class)) {
            //AnnotatedElement extends Annotation(顶级接口)
            //final Annotation annotation = c.getDeclaredAnnotation(Bookk.class);
            //向下转型(Book是注解的实现)
            final Bookk bookk = (Bookk) c.getDeclaredAnnotation(Bookk.class);
            System.out.println(bookk.name());
            System.out.println(Arrays.toString(bookk.authors()));
            System.out.println(bookk.price());
        }
```

```
        System.out.println("----------------------");
        final Method te = c.getDeclaredMethod("te");
        if (te.isAnnotationPresent(Bookk.class)) {
            final Bookk bookk = te.getDeclaredAnnotation(Bookk.class);
            System.out.println(bookk.price());
            System.out.println(Arrays.toString(bookk.authors()));
            System.out.println(bookk.name());
        }
    }
}
```

```
斗罗大陆
[佚名]
9.9
----------------------
444.11
[宅猪, 猫腻, 我吃西红柿, 佛前献花]
无名
```

注解的应用场景一：junit框架

- 需求

定义若干个方法，只要加了YourTest注解，就可以在启动时被触发执行

- 代码体现

```
package annotation_demo;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
import java.lang.reflect.Method;


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

```
class T {
    @YourTest
    public void test1() {
        System.out.println("test1 running---");
    }

    public void test2() {
        System.out.println("test2 running---");
    }
    @YourTest
    public void test3() {
        System.out.println("test3 running---");
    }
}
```

```
    public static void main(String[] args) throws Exception {
        final T t = new T();
        final Class c = T.class;
        final Method[] methods = c.getDeclaredMethods();
        //遍历所有的方法
        for (Method method : methods) {
            //判断这个方法上有没有注解
            if (method.isAnnotationPresent(YourTest.class)) {
                //没有返回值，没有入参,是成员方法(但反射可以调用所有的成员方法，静态和实例)
                //final Object invoke = method.invoke(t);
                method.invoke(t);//直接调，不用给入参，不用给返回值
            }
        }
    }
```

```
test3 running---
test1 running---
进程已结束，退出代码为 0
```

动态代理

- 概念

代理就是被代理者没有能力或者不愿意去完成某件事情，需要找个人代替自己去完成这件事，动态代理就是用来对业务功能（方法）进行代理的

- 实现步骤

必须有接口，实现类要实现接口（代理通常是基于接口实现的）。
创建一个实现类的对象，该对象为业务对象(被代理对象)，紧接着为业务对象做一个代理对象

- 代码实现

```
/**
 * ClassName: UserService
 * Description:业务接口
 * date:2022/3/18
 *
 * @author fgcy
 * @since JDK 1.8
 */
public interface UserService {
    //接口中的方法默认都是抽象公开的，可以省略(成员属性都是常量:public static final修饰的)
    public abstract String login(String name, String password);
    //省略public abstract
```

```
    boolean updateUser(int id, String name);

    boolean deleteUser(int id);
}
```

```
/**
 * ClassName: UserServiceImpl
 * Description:业务接口的实现类
 * date:2022/3/18
 *
 * @author fgcy
 * @since JDK 1.8
 */
public class UserServiceImpl implements UserService {

    @Override
    public String login(String name, String password) {
        try {
            Thread.sleep(200);
            if ("admin".equals(name) && "password".equals(password)) {
                return "登录成功";
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
        return "登录失败";
    }

    @Override
    public boolean updateUser(int id, String name) {
        try {
            System.out.println("更新数据成功! ");
            Thread.sleep(100);
        } catch (Exception e) {
            e.printStackTrace();
            return false;
        }
        return true;
    }

    @Override
    public boolean deleteUser(int id) {
        try {
            System.out.println("数据删除成功! ");
            Thread.sleep(100);
        } catch (Exception e) {
            e.printStackTrace();
            return false;
        }
        return true;
    }
}
```

```
package proxy_demo;


import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;

/**
 * ClassName: ProxyUtils
 * Description: 获取代理的类
 * date:2022/3/18
 *
 * @author fgcy
 * @since JDK 1.8
 */
//泛型方法 可以接收任意类型的对象，生成一个同类型的代理对象返回
public class ProxyUtils {
    //<T>申明这是一个泛型方法，然后由返回值类型决定入参类型
    public static <T> T getProxy(T object) {
        /*
         public static Object newProxyInstance(ClassLoader loader,Class<?>[] interfaces,InvocationHandler
h)
         第一个参数是类加载器，负责将需要被代理的类加载进内存中，以便以后生成代理对象
         第二个参数是获取被代理对象的全部接口
         第三个参数是代理的核心逻辑

        * */
        final T t = (T) Proxy.newProxyInstance(object.getClass().getClassLoader(),
object.getClass().getInterfaces(), new InvocationHandler() {
            @Override
            public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
                //参数一:代理对象本身(基本没用)
                //参数二:正在被代理的方法
                //参数三:被代理方法应该传入的参数
                final long start = System.currentTimeMillis();

                //触发真正的业务方法
                final Object result = method.invoke(object, args);
                final long end = System.currentTimeMillis();
                System.out.println(method.getName() + "方法耗时: " + (end - start) / 1000.0);
                //返回方法的返回结果【void----->null】
                return result;
            }
        });
        //返回被代理类的类型
        return t;
    }
}
```

```
/**
 * ClassName: Test
 * Description:测试类
 * date:2022/3/18
 *
```

```
 * @author fgcy
 * @since JDK 1.8
 */
public class Test {
    public static void main(String[] args) {
        UserService userService = ProxyUtils.getProxy(new UserServiceImpl());
        userService.deleteUser(12);
        userService.updateUser(12, "swx");
    }
}
```

```
数据删除成功!
deleteUser方法耗时：0.114
更新数据成功!
updateUser方法耗时：0.111

进程已结束，退出代码为 0
```

- 总结

1. 非常的灵活，支持任意接口类型的实现类对象做代理，也可以直接为接口本身做代理。
2. 可以为被代理对象的所有方法做代理。
3. 可以在不改变方法源码的情况下，实现对方法功能的增强。
4. 不仅简化了编程工作、提高了软件系统的可扩展性，同时也提高了开发效率。
5. 动态代理其实是一种AOP思想；