1、课程名称：反射与内省

2、知识点

2.1、本次预计讲解的知识点

1、 什么叫反射
2、 Class 类的作用
3、 Class 类的实例化三种方式
4、 通过 Class 类进行对象的实例化操作
5、 通过 Class 类取得类的完整信息
6、 通过 Class 类调用类中的属性或方法
7、 JavaBean 的概念
8、 内省基本概念
9、 Introspector 相关 API

3、具体内容

3.1、什么叫反射?

正常情况下，如果已经存在了一个类，那么是可以通过这个类进行对象的实例化操作的，这属于正常的情况，那么如果现在要求通过一个对象取得类的完整信息，那么此时就需要反射。

例如：现在有如下的操作代码

```
package org.classdemo.demo01;
class Person { // 定义一个类
}
public class ClassInfoDemo01 {
    public static void main(String[] args) {
        Person per = new Person();
        System.out.println(per.getClass().getName());
    }
}
```

发现，此时，取得了一个完整的对象所属于的包.类名称，也就是说现在通过反射机制，取得了反向的信息。

以上的 getClass()方法是在 Object 类中定义的方法，此方法声明如下：public final Class<?> getClass()

以上方法的返回值是 Class 类型的实例，所以，Class 类是一切反射的根源。

JAVA 反射机制是在运行状态中，对于任意一个类，都能够知道这个类的所有属性和方法；对于任意一个对象，都能够调用它的任意一个方法；这种动态获取的信息以及动态调用对象的方法的功能称为 java 语言的反射机制。

3.2、认识 Class 类

Class 类是一切的反射根源，此类的定义如下：

```
public final class Class<T> extends Object
implements Serializable, GenericDeclaration, Type, AnnotatedElement
```

Class 类表示什么？

很多的人——可以定义一个 Person 类（有年龄，性别，姓名等）

很多的车——可以定义一个 Car 类（有发动机，颜色，车轮等）

很多的类——Class 类

此类是一个最终类，是不允许有子类的，而且此类声明的时候使用了泛型声明，证明，要指明泛型类型。

而刚才的 getName()方法也是在此类中定义的。

但是，观察一下，此类中是否存在构造方法？此类的 API 文档上来就直接定义了方法的摘要。此类的构造方法已经被私有化了，如果现在要想进行此类的实例化操作，则可以通过以下三种形式：

- 第一种形式：Object 类中的 getClass()方法
- 第二种形式：类.class
- 第三种形式：通过 Class 类中定义的如下方法：
  - |- 方法：public static Class<?> forName(String className) throws ClassNotFoundException

3.2.1、第一种实例化方式

通过 getClass()方法。

```
package org.classdemo.demo01;
public class ClassInstanceDemo01 {
    public static void main(String[] args) {
        String str = "hello";
        Class<?> cls = str.getClass();
        System.out.println(cls.getName());
    }
}
```

3.2.2、第二种实例化方式

通过类.class 的形式完成。

```
package org.classdemo.demo02;
public class ClassInstanceDemo02 {
    public static void main(String[] args) {
        Class<?> cls = String.class;
        System.out.println(cls.getName());
    }
}
```

3.2.3、第三种实例化方式

第三种方式是直接使用 Class 类内部的 forName()方法完成，那么此种形式是在开发中出现最多的形式。

```
package org.classdemo.demo01;
public class ClassInstanceDemo03 {
    public static void main(String[] args) throws Exception {
        Class<?> cls = Class.forName("java.lang.String");
        System.out.println(cls.getName());
    }
}
```

以上的这种方式是在开发中使用最多的形式，应该重点掌握。

九个预定义的 Class 对象：

八大基本数据类型和 void

看看以下情况：

Class. isPrimitive()　判定指定的 Class 对象是否表示一个基本类型

Integer.getClass()

Integer.TYPE　//基本类型

int[].class

Class.isArray() //判定此 Class 对象是否表示一个数组类

3.3、使用 Class 类进行对象的实例化操作

Class 类的最大好处，是可以通过完整的包.类名称进行对象的实例化操作。

3.3.1、调用无参构造进行实例化

现在：有如下的操作类

```
package org.classdemo.demo01;
public class Person {
    private String name;
    private int age;
    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 String toString() {
        return "姓名: " + this.name + ", 年龄: " + this.age;
    }
}
```

正常情况下，如果要想使用这个类的，则必须通过关键字 new 进行对象的实例化操作，但是现在有了 Class 类之后，就可以不用这么麻烦，直接通过反射进行对象的实例化操作，使用如下的方法完成：

No.	方法名称	类型	描述
1	public T newInstance() throws InstantiationException,IllegalAccessException	普通	根据指定的包.类名称进行对象的实例化操作

范例：进行对象的实例化操作

```
package org.classdemo.demo01;
public class ClassInstanceObjectDemo01 {
    public static void main(String[] args) throws Exception {
        Class<?> cls = Class.forName("org.classdemo.demo01.Person");
        Person per = (Person) cls.newInstance();
        per.setName("张三");
        per.setAge(10);
        System.out.println(per);
    }
}
```

此时，可以发现，非常方便的通过另外一种形式进行了对象的实例化操作。有时候在开发中为了方便，也会将以上的代码写成如下的形式：

```
package org.classdemo.demo01;
public class ClassInstanceObjectDemo02 {
    public static void main(String[] args) throws Exception {
        Person per = (Person) Class.forName("org.classdemo.demo01.Person")
                .newInstance();
        per.setName("张三");
        per.setAge(10);
        System.out.println(per);
    }
}
```

但是，以上的操作只适合于类中存在无参构造方法的情况。如果，现在在 Person 类中增加了有两个参数的构造方法：

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

则，再次运行以上程序的时候将出现以下的问题：

```
Exception in thread "main" java.lang.InstantiationException: org.classdemo.demo03.Person
    at java.lang.Class.newInstance0(Unknown Source)
    at java.lang.Class.newInstance(Unknown Source)
    at
org.classdemo.demo03.ClassInstanceObjectDemo02.main(ClassInstanceObjectDemo02.java:7)
```

根本就找不到 Person 类中的无参构造方法，所以一直在强调，类中必须存在无参构造方法。

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

3.3.2、调用有参构造进行实例化

如果现在在一个类中只存在一个有两个参数的构造方法，如下：

```
package org.classdemo.demo01;
public class Person {
    private String name;
    private int age;
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
    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 String toString() {
        return "姓名：" + this.name + "，年龄：" + this.age;
    }
}
```

那么这个时候就必须按照如下的步骤完成：

1、 还是实例化 Class 类的对象

2、 通过 Class 类找到此类中的构造方法

3、 通过构造方法明确的给出需要的参数，并进行对象的实例化操作。

Class 类可以通过以下的方法取得一个类中的全部构造：

No.	方法名称	类型	描述
1	public Constructor<?>[] getConstructors() throws SecurityException	普通	得到全部的构造方法

一个类中可能存在多个构造方法，所以以上的方法，将返回一个 Constructor 的对象数组。

java.lang.reflect.Constructor 是在反射包中进行定义的，证明使用此方法可以进行反射的操作。使用此类的如下方法：

No.	方法名称	类型	描述
1	public T newInstance(Object... initargs) throws InstantiationException,IllegalAccessException, IllegalArgumentException,InvocationTargetException	普通	此方法可以进行对象的实例化操作

此方法接收的一个可变参数，所以，在调用以上方法的时候，需要指明参数的顺序及个数。

范例：通过 Constructor 进行对象的实例化操作

```
package org.classdemo.demo01;
import java.lang.reflect.Constructor;
public class ClassInstanceObjectDemo03 {
    public static void main(String[] args) throws Exception {
        Class<?> cls = Class.forName("org.classdemo.demo01.Person");
        Constructor<?>[] con = cls.getConstructors();// 得到全部的构造
        Person per = (Person)con[0].newInstance("张三", 30);
        System.out.println(per);
    }
}
```

此时，已经可以正常的调用成功了，但是以上的操作明显比第一种更加复杂，所以在开发中如果使用了此种形式的操作，最好的建议是类中保留无参构造方法。

3.4、小结

请总结出现在已经学习过的对象实例化方法：

1、 通过关键字 new，此种方法比较直观，直接由 java 低层给予支持，但是此种操作会造成代码的耦合度加深，中间加入了工厂设计，来解决此种问题。

2、 clone()：对象克隆

3、 通过 Class 类，进行反射加载实例化。此种方式最方便，直接传入完整的包.类名称就可以完成功能。

4、 引用传递也是一种

3.5、通过反射取得类的完整信息

反射机制最大好处，是可以通过一个 Class 类取得一个类中的完整信息，包括类所在的包，继承的父类，类中的所有方法、属性等等，为了测试方便，现在建立如下的一个类：

```
package org.classdemo.demo01;
interface Info {
    public String AUTHOR = "HELLO";
    public void sayHello();
    public String say(String name, int age);
}
public class Person implements Info { // 实现接口
    private String name;
    private int age;
    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 String toString() {
        return "姓名：" + this.name + "，年龄：" + this.age;
    }
    public String say(String name, int age) {
        return "你好，" + name + "，我的年龄是：" + age;
    }
    public void sayHello() {
        System.out.println("Hello World!!!");
    }}
```

3.5.1、取得类所在的包

可以通过 Class 类中的如下方法取得一个类所在的包：

No.	方法名称	类型	描述
1	public Package getPackage()	普通	得到一个类所在的包

此方法返回一个 Package 类型的实例化对象。取得包的名称：

No.	方法名称	类型	描述
1	public String getName()	普通	得到名字

范例：取得所在的包

```
package org.classdemo.demo01.case01;
public class ClassInfoDemo01 {
    public static void main(String[] args) throws Exception {
        Class<?> cls = Class.forName("org.classdemo.demo01.Person");
        Package pack = cls.getPackage(); // 取得包
        System.out.println(pack);
    }
}
```

3.5.2、取得一个类中的全部方法

可以通过 Class 类中的如下方法取得类中的全部方法：

No.	方法名称	类型	描述
1	public Method[] getMethods() throws SecurityException	普通	得到全部的方法

以上的方法以 Method 对象数组的形式返回，Method 本身就表示全部的方法，可以通过如下的操作取得方法的完整信息：

No.	方法名称	类型	描述
1	public int getModifiers()	普通	得到全部的修饰符
2	public Class<?> getReturnType()	普通	得到返回值类型
3	public Class<?>[] getParameterTypes()	普通	得到全部的参数类型
4	public Class<?>[] getExceptionTypes()	普通	得到全部的异常

但是，对于以上的操作中，发现得到修饰符号的方法返回的是整型，但是一般来讲修饰符都是 public、private 之类的，因为在整个 JAVA 中，所有的修饰符可以按照一定的数字进行相加操作。如果要想正确的还原修饰符，则需要借助 Modifer 类完成。

No.	方法名称	类型	描述
1	public static String toString(int mod)	普通	传入修饰符的数字，并转换为修饰符描述

范例：取得类中的全部方法

```
package org.classdemo.demo01.case02;
import java.lang.reflect.Method;
import java.lang.reflect.Modifier;
public class ClassInfoDemo02 {
    public static void main(String[] args) throws Exception {
        Class<?> cls = Class.forName("org.classdemo.demo01.Person");
        Method met[] = cls.getMethods();
        for (int x = 0; x < met.length; x++) {
            int mod = met[x].getModifiers();
            System.out.print(Modifier.toString(mod) + " "); // 还原修饰符
            String ret = met[x].getReturnType().getName();// 得到返回值类型
            System.out.print(ret + " ");
            System.out.print(met[x].getName() + "(");
            Class<?> param[] = met[x].getParameterTypes();// 取得全部的参数
            for (int y = 0; y < param.length; y++) {
                System.out.print(param[y].getName() + " arg-" + y);
                if (y < param.length - 1) {
                    System.out.print(", ");
                }
            }
            System.out.print(") ");
            Class<?> exp[] = met[x].getExceptionTypes();
            if (exp.length > 0) {
                System.out.print("throws ");
                for (int y = 0; y < exp.length; y++) {
                    System.out.print(exp[y].getName());
                    if (y < exp.length - 1) {
                        System.out.print(", ");
                    }
                }
            }
            System.out.println();
```

```
        }
    }
}
```

3.5.3、取得一个类中的全部属性

如果要想取得一个类中的全部属性，则可以依靠以下的方法：

No.	方法名称	类型	描述
1	public Field[] getFields() throws SecurityException	普通	取得全部属性，继承来的属性
2	public Field[] getDeclaredFields() throws SecurityException	普通	取得全部属性，本类中的属性

以上的方法返回的是 Filed 类的对象数组，此类的相关操作方法与 Method 类是非常一致的。Filed 类的常用方法如下：

No.	方法名称	类型	描述
1	public Class<?> getType()	普通	取得类型
2	public int getModifiers()	普通	取得修饰符
3	public String getName()	普通	取得名称

范例：取得一个类中的全部属性

```
package org.classdemo.demo01.case03;
import java.lang.reflect.Field;
import java.lang.reflect.Modifier;
public class ClassInfoDemo03 {
    public static void main(String[] args) throws Exception {
        Class<?> cls = Class.forName("org.classdemo.demo01.Person");
        {
            Field[] field = cls.getFields();
            for (int x = 0; x < field.length; x++) {
                int mod = field[x].getModifiers();
                System.out.print(Modifier.toString(mod) + " "); // 还原修饰符
                System.out.print(field[x].getType().getName() + " ");
                System.out.print(field[x].getName());
                System.out.println();
            }
        }
        {
            Field[] field = cls.getDeclaredFields();
            for (int x = 0; x < field.length; x++) {
                int mod = field[x].getModifiers();
                System.out.print(Modifier.toString(mod) + " "); // 还原修饰符
                System.out.print(field[x].getType().getName() + " ");
                System.out.print(field[x].getName());
                System.out.println();
            }
        }
    }
}
```

以上的全部操作都只作为了解应用，开发中基本上是不会应用到的。

3.6、通过反射操作类

通过反射机制，不光只可以取得类的结构信息那么简单，还可以使用反射机制进行类中方法的调用或属性的调用。

3.6.1、调用类中的方法

在正常情况下，都是通过对象.方法()进行方法调用的，那么现在如果使用反射的话也可以完成同样的功能，具体的操作形式如下。

No.	方法名称	类型	描述
1	public Method getMethod(String name,Class<?>... parameterTypes) throws NoSuchMethodException,SecurityException	普通	取得类中的一个方法

以上的操作将返回一个具体的方法的 Method 对象，同时指定要操作的方法名称，参数的类型，传入之后再通过 Method 类中的以下方法直接调用类中的操作：

No.	方法名称	类型	描述
1	public Object invoke(Object obj,Object... args) throws IllegalAccessException, IllegalArgumentException,InvocationTargetException	普通	调用类中的方法，传入实例化对象，以及具体的参数内容

范例：调用 Person 类中的 sayHello()方法

```
package org.classdemo.demo01.case04;
import java.lang.reflect.Method;
public class MethodInvokeDemo01 {
    public static void main(String[] args) throws Exception {
        Class<?> cls = Class.forName("org.classdemo.demo01.Person");
        Method met = cls.getMethod("sayHello");// 调用sayHello()方法
        met.invoke(cls.newInstance());// 调用类中的方法
    }
}
```

此时的调用，完全没有具体的实例化对象的操作，都是通过 Object 类完成的。

范例：调用 Person 类中的 say()方法

- 此方法中存在参数及返回值类型

```
package org.classdemo.demo01.case04;
import java.lang.reflect.Method;
public class MethodInvokeDemo02 {
    public static void main(String[] args) throws Exception {
        Class<?> cls = Class.forName("org.classdemo.demo01.Person");
        Method met = cls.getMethod("say", String.class, int.class);// 调用say()方法，需要两个参数
        Object obj = met.invoke(cls.newInstance(), "张三", 30);// 调用类中的方法
        System.out.println(obj);
    }
}
```

此时，发现方法也可以正常的调用成功了。

3.6.2、调用 setter 及 getter

从最早开始接触到面向对象的概念就一直强调类中的全部属性都必须进行封装，封装之后的属性必须通过 setter 及 getter 方法设置和取得，那么之所以这样做，就是因为在以后的程序开发中，都可以使用反射机制进行方法的自动调用，只需要传入属性的名称即可。

```
package org.classdemo.demo01.case04;
import java.lang.reflect.Method;
public class MethodInvokeDemo03 {
    public static void main(String[] args) throws Exception {
        Class<?> cls = Class.forName("org.classdemo.demo01.Person");
        Object obj = cls.newInstance();
        setter(obj, "name", "张三", String.class);
        setter(obj, "age", 30, int.class);
        getter(obj, "name");
        getter(obj, "age");
    }
    public static void setter(Object inst, String attr, Object value,
            Class<?> valtype) throws Exception {
        Method met = inst.getClass().getMethod("set" + initCap(attr), valtype);
        met.invoke(inst, value);
    }
    public static void getter(Object inst, String attr) throws Exception {
        Method met = inst.getClass().getMethod("get" + initCap(attr));
        Object re = met.invoke(inst);
        System.out.println(re);
    }
    public static String initCap(String att) {
        StringBuffer buf = new StringBuffer();
        buf.append(att.substring(0, 1).toUpperCase()).append(att.substring(1));
        return buf.toString();
    }
}
```

以上的操作代码将是在所有的开源框架中所采用的核心设计思想。此处只需要了解即可。

3.6.3、直接调用属性

刚才的操作是通过 Method 调用的类中的属性，那么在 Java 的反射中也可以通过属性直接进行操作。

在 Field 类中定义了以下的方法：

No.	方法名称	类型	描述
1	public Object get(Object obj) throws IllegalArgumentException, IllegalAccessException	普通	取得属性
2	public void set(Object obj,Object value) throws	普通	设置属性，等同于使用“=”完成操作

	IllegalArgumentException, IllegalAccessException		
3	public void setAccessible(boolean flag) throws SecurityException	普通	让属性对外部可见

范例：直接调用属性

```
package org.classdemo.demo01.case04;
import java.lang.reflect.Field;
public class FieldInvokeDemo01 {
    public static void main(String[] args) throws Exception {
        Class<?> cls = Class.forName("org.classdemo.demo01.Person");
        Object obj = cls.newInstance();
        Field name = cls.getDeclaredField("name");
        Field age = cls.getDeclaredField("age");
        name.setAccessible(true);// 取消了private
        age.setAccessible(true);// 取消了private
        name.set(obj, "张三");
        age.set(obj, 10);
        System.out.println(name.get(obj));
        System.out.println(age.get(obj));
    }
}
```

发现，使用反射机制，可以完成很多其他的同样类型的操作，而且在以上的操作代码中，根本不用去关系具体对象的的具体类型，一切都是以 Object 为操作的标准。

3.7、 JavaBean 的概念

什么是 JavaBean?

- Bean 理解为组件意思，JavaBean 就是 Java 组件，在广泛的理解就是一个类，对于组件来说，关键在于要具有“能够被 IDE 构建工具侦测其属性和事件”的能力，通常在 Java 中。
- **一个 JavaBean 要具备这样的命名规则：**
- **1**、对于一个名称为 xxx 的属性，通常你要写两个方法：getXxx()和 setXxx()。任何浏览这些方法的工具，都会把 get 或 set 后面的第一个字母自动转换为大写。
- **2**、对于布尔型属性，可以使用以上 get 和 set 的方式，不过也可以把 get 替换成 is。
- **3**、Bean 的普通方法不必遵循以上的命名规则，不过它们必须是 public 的。
- **4**、对于事件，要使用 Swing 中处理监听器的方式。如 addWindowListener， removeWindowListener

3.8、 内省基本概念

内省(IntroSpector)是 Java 语言对 Bean 类属性、事件的一种缺省处理方法。例如类 A 中有属性 name，那我们可以通过 getName,setName 来得到其值或者设置新的值。

通过 getName/setName 来访问 name 属性，这就是默认的规则。

Java 中提供了一套 API 用来访问某个属性的 getter/setter 方法，通过这些 API 可以使你不需要了解这个规则，这些 API 存放于包 java.beans 中，一般的做法是通过类 Introspector 的 getBeanInfo 方法 来获取某个对象的 BeanInfo 信息，然后通过 BeanInfo 来获取属性的描述器(PropertyDescriptor)，通过这个属性描述器就可以获取某个属性对应的 getter/setter 方法，然后我们就可以通过反射机制来调用这些方法。

3.9、 Introspector 相关 API

1、Introspector 类：

Introspector 类为通过工具学习有关受目标 Java Bean 支持的属性、事件和方法的知识提供了一个标准方法。

static BeanInfo **getBeanInfo**(Class<?> beanClass)

在 Java Bean 上进行内省，了解其所有属性、公开的方法和事件。

2、BeanInfo 类：

该类实现此 BeanInfo 接口并提供有关其 bean 的方法、属性、事件等显式信息。

MethodDescriptor[] **getMethodDescriptors**()

获得 beans MethodDescriptor。

PropertyDescriptor[] **getPropertyDescriptors**()

获得 beans PropertyDescriptor。

3、PropertyDescriptor 类：

PropertyDescriptor 描述 Java Bean 通过一对存储器方法导出的一个属性。

Method **getReadMethod**()

获得应该用于读取属性值的方法。

Method **getWriteMethod**()

获得应该用于写入属性值的方法。

4、MethodDescriptor 类：

MethodDescriptor 描述了一种特殊方法，

即 Java Bean 支持从其他组件对其进行外部访问。

Method **getMethod**()

获得此 MethodDescriptor 封装的方法。

4、总结

1、 理解反射的基本概念
2、 理解 Class 类的作用
3、 理解 Class 类的实例化三种方式
4、 掌握通过 Class 类进行对象的实例化操作
5、 掌握通过 Class 类取得类的完整信息
6、 掌握通过 Class 类调用类中的属性或方法（理解），此块内容将在日后的高端课程中发现其具体应用
7、 内省及相关 API

5、作业

1、 创建一个 Student 类，该类中有若干个构造方法和属性的 get/set 方法
<1>使用 `Constructor` 分别调用各个构造方法生成实例
<2>使用反射调用 `set` 方法传值，并调用 `get` 方法输出值
<3>使用反射直接获取私有成员变量