11. 反射与内省

11.1. 什么是反射（Reflection）

- 通过对象去获取类信息

11.2. Class类

- Class类是一切的反射根源类
- Class类表示什么？
  - 很多的人——可以定义一个Person类
- 很多的车——可以定义一个Car类
- 很多的类——Class类
- 得到Class类的对象有三种方式
  - Object类中的getClass()方法
  - 类.class
  - 通过Class的forName()方法
- 使用Class类进行对象的实例化操作

```
// 调用无参数的构造方法进行实例化
public T newInstance() throws InstantiationException,
IllegalAccessException

// 调用有参数的构造方法进行实例化
public Constructor<?>[] getConstructors() throws SecurityException
```

- 示例

```
/**
 * 获取Class对象的三种形式
 */
@Test
```

```
public void test1() {
    // 通过对象的getClass()方法
    Dog dog = new Dog("二哈", 5, "白色");
    Class dogClass = dog.getClass();

    // 通过类.class
    Class aClass = Dog.class;

    // 通过Class.forName()方法
    try {
        Class forName = Class.forName("day11_反射与内省.reflection.Dog");
    } catch (ClassNotFoundException e) {
        e.printStackTrace();
    }
}
```

11.3. 通过Class类取得类信息

- 通过反射来实例化对象

```
/**
 * 通过反射来实例化对象
 */
@Test
public void test2() {
    Class<Dog> dogClass = Dog.class;
    try {
        // 通过Class对象实例化类对象，调用了默认无参的构造方法
        Dog dog = (Dog) dogClass.newInstance();
    } catch (InstantiationException e) {
        e.printStackTrace();
    } catch (IllegalAccessException e) {
        e.printStackTrace();
    }
}
```

- 获取所有的构造方法

```
/**
 * 获取所有的构造方法
 */
@Test
public void test3() {
    Class<Dog> dogClass = Dog.class;
    Constructor<?>[] constructors = dogClass.getConstructors();
    for (int i = 0; i < constructors.length; i++) {
        System.out.println(constructors[i].getName());
        System.out.println(constructors[i].getParameterCount());
    }
    try {
        // 获取一个指定的构造方法
        Constructor<Dog> constructor =
dogClass.getConstructor(String.class, int.class, String.class);
        // 调用有参数的构造方法来实例化对象
        Dog dog = constructor.newInstance("哈士奇", 4, "黑色");
    } catch (NoSuchMethodException e) {
```

```
            e.printStackTrace();
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        } catch (InstantiationException e) {
            e.printStackTrace();
        } catch (InvocationTargetException e) {
            e.printStackTrace();
        }
    }
```

- 获取属性

```
/**
 * 获取的所有属性
 */
@Test
public void test4() {
    Class<Dog> dogClass = Dog.class;
    // 获取非私有的属性
    Field[] fields = dogClass.getFields();
    System.out.println(fields.length);
    System.out.println("------------------");
    // 获取所有属性（包括私有属性）
    Field[] declaredFields = dogClass.getDeclaredFields();
    System.out.println(declaredFields.length);
    System.out.println("--------------------");
    for (int i = 0; i < declaredFields.length; i++) {
        int modifiers = declaredFields[i].getModifiers();
        Class<?> type = declaredFields[i].getType();
        String name = declaredFields[i].getName();
        System.out.println(Modifier.toString(modifiers) + " " + type +
" " + name);
    }
}
```

- 获取包名

```
@Test
public void test5() {
    Dog dog = new Dog("拉布拉多", 5, "黄色");
    Class<Dog> dogClass = Dog.class;
    // 获取类的包名
    Package aPackage = dogClass.getPackage();
    System.out.println(aPackage.getName());
}
```

11.4. 通过Class类调用属性或方法

- 调用类中的方法

```
@Test
public void test5() {
    Dog dog = new Dog("拉布拉多", 5, "黄色");
    Class<Dog> dogClass = Dog.class;
    // 获取公共的方法，包括继承的公有方法
```

```
    Method[] methods = dogClass.getMethods();
    for (int i = 0; i < methods.length; i++) {
        System.out.println(methods[i]);
        if (methods[i].getName().equals("toString")) {
            try {
                String s = (String) methods[i].invoke(dog);
                System.out.println(s);
            } catch (IllegalAccessException e) {
                e.printStackTrace();
            } catch (InvocationTargetException e) {
                e.printStackTrace();
            }
        }
    }

    System.out.println("--------------------");
    // 访问私有方法，获取到本类中定义的所有方法（不包括父类）
    Method[] declaredMethods = dogClass.getDeclaredMethods();
    for (int i = 0; i < declaredMethods.length; i++) {
        System.out.println(declaredMethods[i]);
        if (declaredMethods[i].getName().equals("set")) {
            // 设置方法可以被访问（去除访问修饰符的检查）
            declaredMethods[i].setAccessible(true);
            try {
                declaredMethods[i].invoke(dog);
            } catch (IllegalAccessException e) {
                e.printStackTrace();
            } catch (InvocationTargetException e) {
                e.printStackTrace();
            }
        }
    }
}
```

- 调用类中的属性

```
@Test
public void test6() {
    Dog dog = new Dog("拉布拉多", 4, "灰色");
    Class<Dog> dogClass = Dog.class;
    try {
        Field name = dogClass.getDeclaredField("name");
        name.setAccessible(true);
        name.set(dog, "二哈");
        System.out.println("修改后的名字为: " + dog.getName());
    } catch (NoSuchFieldException e) {
        e.printStackTrace();
    } catch (IllegalAccessException e) {
        e.printStackTrace();
    }
}
```

11.5. 动态代理设计模式

- 所谓动态代理，即通过运行代理类：Proxy的代理，接口和实现类之间可以不直接发生联系，而可以在运行期（Runtime）实现动态关联
- Java动态代理主要是使用java.lang.reflect包中的两个类
- InvocationHandler类

```
public Object invoke(Object obj, Method method, Object[] obs)
// 其中第一个参数obj指的是代理类，method是被代理的方法，obs是指被代理的方法的参数组。此方法由代理类来实现
```

- Proxy类

```
protected Proxy(InvocationHandler h);
static Class getProxy(ClassLoader loader, Class[] interface);
static Object newProxyInstance(ClassLoader loader, Class[] interfaces, InvocationHandler h);
// 动态代理其实是在运行时生成class，所以，我们必须提供一组interface，然后告诉他class已经实现了这些interface，而且在生成Proxy的时候，我们必须给他提供一个handler，让他来接管实际的工作
```

11.6. 类加载器原理分析

- 类加载的过程
  - JVM将类加载过程分为三个步骤：装载（Load）、链接（Link）和初始化（Initialize）。链接又分为三个步骤：验证、准备和解析
  - 装载：查找并加载类的二进制数据
  - 链接：
    - 验证：确保被加载类的正确性
    - 准备：为类的静态变量分配内存，并将其初始化为默认值
    - 解析：把类中的符号引用转换为直接引用
  - 初始化：为类的静态变量赋予正确的初始值
- 类的初始化，类什么时候被初始化
  - 创建类的实例，也就是new一个对象
  - 访问某个类或者接口的静态变量，或者对该静态变量赋值
  - 调用类的静态方法
  - 反射（Class.forName("com.reflect.Dog")）
  - 初始化一个类的子类（会首先初始化子类的父类）
  - JVM启动时标明的启动类，即文件名和类名相同的那个类
- 类的加载
  - 指的是将类的.class文件中的二进制数据读入到内存中，将其放在运行时数据区的方法区内，然后在堆区创建一个这个类的Java.lang.Class对象，用来封装在方法区类的对象

11.7. 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
- JavaBean的命名规则
  - 对于一个名称为xxx的属性，通常要写两个方法：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小写
  - 对于布尔属性，可以使用以上get和set方式，不过也可以把get替换成is
  - Bean的普通方法不必遵循以上的命名规则，不过他们必须是public的
  - 对于事件，要使用Swing中处理监听器的方式。如addWindowListener，reemoveWindowListener
- BeanUtils工具类：http://apache.org/

11.8. 内省基本概念

- 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对应属性的getter/setter方法，然后我们就可以通过反射机制来调用这些方法

11.9. Introspector相关API

- Introspector类
  - Introspector类为通过工具学习有关受目标JavaBean支持的属性、事件和方法的只是提供了一个标准方法t

```
// 在JavaBean上进行内省，了解其所有属性、公开的方法和事件
static BeanInfo getBeanInfo(Class<?> beanClass);
```

- 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

```
// 获得beans MethodDescriptor
MethodDescriptor[] getMethodDescriptors();

// 获得 beans PropertyDescriptor
PropertyDescriptor[] getPropertyDescriptors();
```

- PropertyDescriptor类
  - PropertyDescriptor描述JavaBean通过一对存储器方法导出的一个属性

```
// 获取应该用于读取属性值的方法
Method getReadMethod();

// 获取应该用于写入属性值得方法
Method getWriteMethod();
```

- MethodDescriptor类
  - MethodDescriptor描述了一种特殊的方法，即JavaBean支持从其他组件对其进行外部访问

```
// 获得此MethodDescriptor封装的方法
Method getMethod();
```

11.10. 理解可配置的AOP框架

- 补充知识
  - AOP的概念：Aspect Oriented Programming（面向切面编程）
  - 可理解AOP框架实现
  - AOP用来封装横切关注点，具体可以在以下场景使用：
    - 权限
    - 缓存
    - 错误处理
    - 调试
    - 记录跟踪
    - 持久化
    - 同步
    - 事务

11.11. 单例模式优化

- 保证同步保证线程安全synchronized
- 使用volatile关键字
  - volatile提醒编译器它后面定义的变量随时都有可能改变，因此编译后的程序每次需要存储或读取这个变量的时候，都会直接从变量地址中读取数据。如果没有volatile关键字，则编译器可能优化调用和存储，可能暂时使用寄存器中的值，如果这个变量由别的程序更新了的话，将出现不一致的情况
- 防止反射调用私有构造方法
- 让单例类序列化安全

```
import java.io.Serializable;

```

```java
/**
 * @author xiao儿
 * @date 2019/9/10 20:10
 * @Description SingleTon
 * 单例模式
 * 1.多线程访问的安全问题
 * 2.加上volatile关键字，保证变量的一致性
 * 3.防止反射私有化构造方法
 * 4.让单例类序列化安全
 */
public class Singleton implements Serializable {
    private volatile static Singleton singleton = null;

    private Singleton() {
        if (singleton != null) {
            throw new RuntimeException("此类对象为单例模式，已经被实例化");
        }
    }

    public static Singleton getInstance() {
        if (singleton == null) {
            synchronized (Singleton.class) {
                if (singleton == null) {
                    singleton = new Singleton();
                }
            }
        }
        return singleton;
    }
}
```

####