反射

现实生活中的反射如通过镜子可以反射出你的样子，水面可以反射出物体的形态等等，无论是哪种反射，都是通过一个虚像映射到实物，这样我们就可以获取到实物的某些形态特征。

程序中的反射也是同样的道理，它完成的是通过一个实例化对象映射到对应的类，在程序运行期间我们可以通过一个对象获取到该对象对应的类信息。

一句话简单理解反射：正常情况下我们是通过类来创建实例化对象，反射就是将这一过程进行反转，通过实例化对象来获取对应的类信息。

Class类

Class类是反射的源头，类的信息在Java中如何描述？Java也是将类的信息抽象成一个对象，Class类就是用来创建描述类信息的对象的。

Class类是专门用来描述其他类的类，Class类的每一个实例化对象对应的都是其他类的结构特征（成员变量，方法，构造函数，父类，实现的接口）。

Class对象不能通过构造函数来创建，因为Class只有一个private的构造函数，外部无法直接调用。

创建Class对象的3种方式：

- 调用Class类的镜头方法forName(String className)创建，className是目标类的全类名

```
Class clazz = Class.forName("com.southwind.entity.Student");
```

- 通过目标类的class创建，Java中的每一个类都可以调用类.class，这里的class不是属性，叫做"类字面量"，其作用是获取在内存中该类型Class对象的引用。

```
Class clazz2 = Student.class;
```

- 通过目标类的实例化对象的getClass()方法创建，getClass()是Object类中定义的方法，被所有的子类所继承，Java中的每一个类都可以调用getClass()方法，获取内存中该类的Class对象的引用。

```
Student student = new Student();
Class clazz3 = student.getClass();
```

每个目标类在内存中的Class对象（该类的运行时类）只有一份，即clazz 和clazz2，clazz3所指向的引用对象是同一个。

通过Class对象可以获取到目标类的结构，成员变量，方法，构造函数，父类，实现的接口等等。

Class类中常用的方法

public native boolean isInterface() 判断该类是否为接口

public native boolean isArray() 判断该类是否为数组

public boolean isAnnotation() 判断该类是否为注解

public String getName() 获取该类的全类名

public ClassLoader getClassLoader() 获取类加载器

public native Class<? super T> getSuperclass() 获取该类的直接父类

public Package getPackage() 获取该类的包

public String getPackageName() 获取该类的包名

public Class<?>[] getInterfaces() 获取该类的全部接口

public native int getModifiers() 获取改了的访问权限修饰符

public Filed[] getFields() 获取该类的全部公有成员变量，包括继承自父类和自定义的

public Filed[] getDeclaredFields() 获取该类的自定义成员变量

public Filed getField(String name) 通过名称获取该类的公有成员变量，包括继承自父类和自定义的

public Filed getDeclaredField(String name) 通过名称获取该类的自定义成员变量

public Method[] getMethods() 获取该类的全部公有方法，包括继承自父类和自定义的

public Method[] getDeclaredMethods() 获取该类的自定义方法

public Method getMethod(String name,Class... parameterTypes) 通过名称和参数信息获取该类的公有方法，包括继承自父类和自定义的

public Method getDeclaredMethod(String name,Class... parameterTypes) 通过名称和参数信息获取该类的自定义方法

public Constructor<?>[] getConstructors() 获取该类的公有构造函数

public Constructor<?>[] getDeclaredConstructors() 获取该类的全部构造函数

pulblic Constructor getConstructor(Class<?>... parameterTypes) 通过参数信息获取该类的公有构造函数

public Constructor getDeclaredConstructor(Class<?>... parameterTypes) 通过参数信息获取该类的构造函数

获取类的接口

```
package com.southwind.test;

import com.southwind.entity.Student;

public class Test4 {
    public static void main(String[] args) {
//      Class clazz = Student.class;
//      Class[] interfaces = clazz.getInterfaces();
//      for (Class class1 : interfaces) {
//          System.out.println(class1);
```

```
//      }
        Class clazz = String.class;
        Class[] interfaces = clazz.getInterfaces();
        for (Class class1 : interfaces) {
            System.out.println(class1);
        }
    }
}
```

获取父类

```
Class clazz = Student.class;
Class superClass = clazz.getSuperclass();
System.out.println(superClass);
```

获取构造函数

```
package com.southwind.reflect;

import java.lang.reflect.Constructor;

import com.southwind.entity.Student;

public class Test {
    public static void main(String[] args) {
        Class clazz = Student.class;
        /*
         * 获取Student类的全部公有构造函数
         */
        Constructor<Student>[] constructors = clazz.getConstructors();
        for (Constructor<Student> constructor : constructors) {
            System.out.println(constructor);
        }
        System.out.println("****************************************");
        /*
         * 获取Student类的全部构造函数
         */
        Constructor<Student>[] constructors2 =
clazz.getDeclaredConstructors();
        for (Constructor<Student> constructor : constructors2) {
            System.out.println(constructor);
        }
        System.out.println("****************************************");
        /*
         * 获取Student(int id)构造函数
         */
        try {
```

```
            Constructor<Student> constructor =
clazz.getDeclaredConstructor(int.class);
            System.out.println(constructor);
        } catch (NoSuchMethodException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (SecurityException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
        System.out.println("**************************************");
        /*
         * 获取Student(String name,double score)构造函数
         */
        try {
            Constructor<Student> constructor2 =
clazz.getConstructor(String.class,double.class);
            System.out.println(constructor2);
        } catch (NoSuchMethodException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (SecurityException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

    }
}
```

获取方法

```
package com.southwind.reflect;

import java.lang.reflect.Method;

public class Test2 {
    public static void main(String[] args) {
        try {
            Class clazz = Class.forName("com.southwind.entity.Student");
            Method[] methods = clazz.getMethods();
            for (Method method : methods) {
                System.out.println(method);
            }

            Method[] methods = clazz.getDeclaredMethods();
            for (Method method : methods) {
                System.out.println(method);
            }
```

```
            Method method = clazz.getMethod("hashCode", null);
            System.out.println(method);

            Method method = clazz.getDeclaredMethod("getName", null);
            System.out.println(method);

        } catch (ClassNotFoundException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (NoSuchMethodException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (SecurityException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```

获取成员变量

```
package com.southwind.reflect;

import java.lang.reflect.Field;
import java.lang.reflect.Modifier;

import com.southwind.entity.Student;

public class Test3 {
    public static void main(String[] args) {
        Student student = new Student();
        Class clazz = student.getClass();
//      Field[] fields = clazz.getFields();
//      for (Field field : fields) {
//          System.out.println(field);
//      }

//      Field[] fields = clazz.getDeclaredFields();
//      for (Field field : fields) {
//          System.out.println(field);
//      }

        try {
//          Field field = clazz.getField("age");
//          System.out.println(field);

            Field field = clazz.getDeclaredField("id");
            System.out.println(field.getName());
            System.out.println(field.getModifiers());
```

```
            String str = Modifier.toString(field.getModifiers());
            System.out.println(str);
            System.out.println(field.getType());
        } catch (NoSuchFieldException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (SecurityException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

    }
}
```

反射的应用

反射调用方法

常规情况下，先创建对象，再通过对象来调用方法，现有对象，再调方法。

反射情况下，先获取方法对象，再调用方法对象的invoke()方法来完成目标方法的调用。

```
package com.southwind.reflect;

import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;

import com.southwind.entity.Student;

public class Test4 {
    public static void main(String[] args) {
        Student student = new Student();
//      System.out.println(student.getId(""));
//      int id = student.getId(null);
//      System.out.println(id);

        Class clazz = Student.class;
        try {
//          Method method = clazz.getDeclaredMethod("getId", String.class);
//          int id = (int) method.invoke(student, "");
//          System.out.println(id);

            Method method = clazz.getDeclaredMethod("getName", null);
            //暴力反射
            method.setAccessible(true);
            String str = (String) method.invoke(student, null);
            System.out.println(str);
        } catch (NoSuchMethodException e) {
            // TODO Auto-generated catch block
```

```
            e.printStackTrace();
        } catch (SecurityException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (IllegalArgumentException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (IllegalAccessException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (InvocationTargetException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```

常规情况下，private的方法在外部无法调用，如果通过反射机制是在外部完成对private方法的调用的，需要调用Method.setAccessible(true)来完成暴力修改。

反射访问成员变量

```
package com.southwind.reflect;

import java.lang.reflect.Field;

import com.southwind.entity.Student;

public class Test5 {
    public static void main(String[] args) {
        Class clazz = Student.class;
        Student student = new Student();
        try {
//          Field field = clazz.getDeclaredField("name");
//          field.set(student, "张三");
//          System.out.println(student);

            Field field = clazz.getDeclaredField("id");
            field.setAccessible(true);
            field.set(student, "0X123");
            System.out.println(student);

        } catch (NoSuchFieldException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (SecurityException e) {
```

```
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (IllegalArgumentException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (IllegalAccessException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

    }
}
```

常规情况下，private修饰的成员变量在外部是无法访问的，但是通过反射机制可以修改成员变量的访问权限，暴力修改，通过调用Filed.setAccessible(true)完成。

反射调用构造函数

```
package com.southwind.reflect;

import java.lang.reflect.Constructor;
import java.lang.reflect.InvocationTargetException;

import com.southwind.entity.Student;

public class Test6 {
    public static void main(String[] args) {
        Class clazz = Student.class;
        try {
//          Constructor<Student> constructor = clazz.getConstructor(null);
//          Student student = constructor.newInstance(null);
//          System.out.println(student);

            Constructor<Student> constructor =
clazz.getDeclaredConstructor(String.class,String.class);
            constructor.setAccessible(true);
            Student student = constructor.newInstance("0X123","张三");
            System.out.println(student);

        } catch (NoSuchMethodException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (SecurityException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (InstantiationException e) {
```

```
                // TODO Auto-generated catch block
                e.printStackTrace();
            } catch (IllegalAccessException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            } catch (IllegalArgumentException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            } catch (InvocationTargetException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }
    }
```

常规情况下，private的构造函数在外部是无法调用的，但是在反射机制下，可以通过暴力修改的方式来设置构造函数的访问权限，以便在外部可以调用该private的构造函数。

动态代理

动态代理是反射的一个重要应用。

Java中的代理模式的特点是委托类和代理类实现了同样的接口，即委托类和代理类都具备完成需求的能力，代理类可以为委托类进行消息预处理，过滤消息，以及事后处理消息等。

代理类和委托类之间存在注入的关联关系，即在设计程序时需要将委托类定义为代理类的成员变量。

代理类本身并不会真正的去执行业务逻辑，而是通过调用委托类的方法来完成。

简单来说就是我们在访问委托对象时，是通过代理对象来间接访问的。代理模式就是通过这种间接访问的方式，为程序预留出可处理的空间，利用此空间，在不影响核心业务的基础上可以附加其他的业务，这就是代理模式的好处。

代理模式又可以分为静态代理和动态代理，静态代理需要预先写好代理类的代码，在编译期代理类的class文件就已经生成。

动态代理是指在编译期并没有确定具体的代理类，在程序运行期间根据Java的指示动态生成的方式。

通过java.lang.reflect.InvocationHandler接口和java.lang.reflect.Proxy类完成动态代理模式，动态指在编程代码的时候并不知道具体的代理类是什么结构，在程序运行期间生成JDK动态代理类和动态代理对象。

动态代理类：自定义类，实现InvocationHandler接口

```
package com.southwind.proxy3;

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

```
public class MyInvocationHandler implements InvocationHandler {
    private Object object;

    public Object bind(Object object) {
        this.object = object;
        return
Proxy.newProxyInstance(MyInvocationHandler.class.getClassLoader(),
object.getClass().getInterfaces(), this);
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
        // TODO Auto-generated method stub
        System.out.println("动态代理开始执行");
        return method.invoke(this.object, args);
    }
}
```

```
package com.southwind.proxy3;

import com.southwind.proxy.Apple;
import com.southwind.proxy.Phone;
import com.southwind.proxy2.BMW;
import com.southwind.proxy2.Benz;
import com.southwind.proxy2.Car;

public class Test3 {
    public static void main(String[] args) {
        MyInvocationHandler myInvocationHandler = new
MyInvocationHandler();
        BMW bmw = new BMW();
        Benz benz = new Benz();
        Apple apple = new Apple();
        Car car = (Car) myInvocationHandler.bind(benz);
        System.out.println(car.saleCar());
    }
}
```

代理模式

1.委托类和代理类需要实现同一个接口。

2.在代理类中定义一个委托类的成员变量，在创建代理对象时需要将委托对象传入到代理对象中。

3.在代理类的接口方法中调用委托对象的接口方法。

静态代理：接口，委托类，静态代理类

动态代理：接口，委托类，代理模版类（不是代理类，程序运行期间借助于此模式动态生成一个代理类）

- 代理模版类需要实现InvocationHandler接口