一、Java 反射机制

Java 反射（Reflective）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。

反射库（Reflection Library）提供了一个非常丰富且精心设计的工具集，以便编写能够动态操纵 Java 代码的程序。

Java 程序在运行时，Java 运行时系统一直对所有的对象进行所谓的运行时类型标识。这项信息纪录了每个对象所属的类。虚拟机通常使用运行时类型信息选准正确方法去执行，用来保存这些类型信息的类是 Class 类。Class 类封装一个对象和接口运行时的状态，当装载类时，Class 类型的对象自动创建。

Class 类没有公共构造方法。Class 对象是在加载类时由 Java 虚拟机以及通过调用类加载器中的 defineClass 方法自动构造的，因此不能显式地声明一个 Class 对象。

虚拟机为每种类型管理一个独一无二的 Class 对象。也就是说，每个类(型)都有一个 Class 对象。运行程序时，Java 虚拟机(JVM)首先检查所要加载的类对应的 Class 对象是否已经加载。如果没有加载，JVM 就会根据类名查找.class 文件，并将其 Class 对象载入。

1.1、对 Class 类的理解

在 Java 中万事万物都被抽象成为类，一个具体的事物则是某个类的实例对象，那么对 Java 中类的抽象呢（例如，每个类都有成员变量，构造方法，成员方法等）？对**类的抽象**就是类类型（class type），**即 Class 类**。

Class 类的实例就是 Java 中的类，Class 类的构造方法是私有的，只能由 Java 虚拟机创建，并且每个类只有一个与之对应的 Class 类实例（**不管用何种方式获取的都一样**）。

当一个类由 JVM 加载到内存中时，都会生成类对应的一个 Class 类的对象，（更恰当地说，是被保存在一个同名的.class 文件中）。为了生成这个类的对象，运行这个程序的 Java 虚拟机将使用被称为“类加载器”的子系统。该 Class 类的对象就保存了运行时 Java 类的类型信息。

运行时类型信息（Run-Time Type Information，RTTI），使得你可以在程序运行时发现和使用类型信息。多态实现的关键就是使用运行时类型信息（多态有两种，一种是编译时多态，通过方法的重载实现，另一种是运行时多态，通过方法的覆盖来实现）。

1.2、获取 Class 的对象的方法

1、调用 Object 类的 getClass()方法来得到 Class 对象，这也是最常见的产生 Class 对象的方法。

```
MyObject  x;
Class c1 = x.getClass();
```

2、使用 Class 类的中静态 forName()方法获得与字符串对应的 Class 对象。

```
Class c2 = Class.forName("Employ"); //Employ 必须是接口或者类的名字。
```

forName()方法获得与字符串对应的 Class 对象一般要放在 try-catch 语句块中使用，因为字符串不一定就能对应上一个合法的类。

3、如果 T 是一个 Java 类型，那么 T.class 就代表了匹配的类对象。

```
Class c3 = Manager.class;
```

这样做不仅简单，而且安全，因为它在编译时就会受到检查（因此不需要置于 try 语句块中）。并且它根除了对 forName()方法的调用，所以也更高效。

类字面常量不仅可以应用于普通的类，也可以使用于接口，数组，以及基本数据类型。另外基本类型的包装器类，还有一个标准字段 TYPE（例如 Integer.TYPE）。TYPE 字段是一个引用，指向对应的基本数据类型的 class 对象。

需要注意：System.out.println(c1 == c2 && c2 == c3)为 true。这是因为每个类对应的 Class 类实例只有一份。

1.3、Class 的常用方法

1、getName()：一个 Class 对象描述了一个特定类的属性，Class 类中最常用的方法 getName，以 String 的形式返回此 Class 对象所表示的实体(类、接口、数组类、基本类型或 void)名称。

2、newInstance()：Class 还有一个有用的方法可以为类创建一个实例，这个方法叫做 newInstance()。newInstance()方法调用默认构造器(无参数构造器)初始化新建对象。

3、getClassLoader()：返回该类的类加载器。

4、getComponentType()：返回表示数组组件类型的 Class。

5、getSuperclass()：返回表示此 Class 所表示的实体(类、接口、基本类型或 void)的超类的 Class。

6、isArray()：判定此 Class 对象是否表示一个数组类。

1.4、Class 的一些使用技巧

1、forName 和 newInstance 结合起来使用，可以根据存储在字符串中的类名创建对象。

```
Object obj = Class.forName(s).newInstance();
```

2、虚拟机为每种类型管理一个独一无二的 Class 对象。因此可以使用==操作符来比较类对象。

```
if(e.getClass() == Employee.class) {…}  // 如果相等则表示 e 是 Employee 类的的实例对象。
```

3、newInstance()方法和 new 关键字的区别：newInstance：弱类型，低效率。只能调用无参构造。new：强类型。相对高效。能调用任何 public 构造。

二、Java 动态加载类

1、静态加载类，在编译时就需要加载，当缺少可能用到的类时便编译不过。如下：

```
package com.zxt.classload;

public class Office {

    /**
     * @Description: new 创建对象是静态加载类，在编译时刻就需要加载所有可能使用到的类，
     * 因此当没有Word、Excel类时，便无法编译通过。（那怕只缺少一个类，其他存在的类也无法使用）
     * @param args
     */
    public static void main(String[] args) {
        if("Word".equals(args[0])) {
            Word w = new Word();
            w.strat();
        }

        if("Excel".equals(args[0])) {
            Excel e = new Excel();
            e.strat();
        }
    }
}
```

只有在 Word 类和 Excel 类都创建之后，上述程序才能编译通过。

2、动态加载类，可以解决上述问题，只有当类在运行时使用到的时候才去加载它，编译时不需要加载。

```
package com.zxt.classload;

public class OfficeBetter {

    /**
     *
     * @Description: 动态加载类，在运行时加载
     *
     * @param args
     */
    public static void main(String[] args) {

        try {
            Class c = Class.forName(args[0]);
```

```
            // 动态生成的类需要进行类型转换，因此我们使用的Word、Excel需要遵循相同的标准
            OfficeInterface oi = (OfficeInterface) c.newInstance();

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

即使没有创建对应 `OfficeInteface` 的实现类，上述程序也能编译通过，需要那个类创建即可使用。当有新的功能添加进来时，上述代码也不需要修改，而静态加载方式则需要修改源程序。

三、通过 Class 类获取类型信息

可以通过 Class 类的一些操作获取某个类型的信息，包括类的成员变量、方法、构造方法等。

```
import java.lang.reflect.Constructor;
import java.lang.reflect.Field;
import java.lang.reflect.Method;
import java.lang.reflect.Modifier;

/**
 * @Description: 通过反射机制获取类的信息（方法、属性、构造方法）
 * @time: 2018年9月1日 下午2:49:13
 */
public class ClassUtil {
    /**
     * @Description: 类的基本信息
     * @param obj 该对象所属类的信息
     */
    public static void printClassMessage(Object obj) {
        // 1、首先获取类的类类型
        // 传递的是那个子类的对象，clazz就是该子类的类类型
        Class<? extends Object> clazz = obj.getClass();

        // 获取类的名称
        System.out.println("类的名称：" + clazz.getName());

        // 类的所属包信息
        System.out.println(clazz.getPackage());
    }

    /**
     * @Description: 类的成员变量的信息
     * @param obj 该对象所属类的信息
     */
    public static void printClassFieldMessage(Object obj) {
        Class<? extends Object> clazz = obj.getClass();

        /**
         * 成员变量也是对象，是java.lang.reflect.Field类的对象
         * getFields()：获取的是所有的public的属性，包括父类继承而来的
         * getDeclaredFields()：获取的是该类自己声明的属性，不论访问权限
         */
        System.out.println("该类自己声明的属性：");
```

```
        Field[] fs = clazz.getDeclaredFields();
        for(Field field : fs) {
            // 得到成员变量的类型的类类型
            Class<?> fieldType = field.getType();
            String typeName = fieldType.getName();
            // 得到成员变量的名称
            String fieldName = field.getName();
            System.out.println(typeName + " " + fieldName);
        }
    }

    /**
     * @Description: 类的方法信息
     * @param obj 该对象所属类的信息
     */
    public static void printMethodClassMessage(Object obj) {
        Class<? extends Object> clazz = obj.getClass();

        // 获取类的方法
        /**
         * Method类，方法类型，一个成员方法就是一个Method对象
         * getMethods(): 获取的是所有的public的方法，包括从父类继承而来的方法
         * getDeclaredMethods(): 获取的是所有该类自己声明的方法，不论访问权限
         */
        System.out.println("该类的所有public方法: ");
        Method[] ms = clazz.getMethods();
        for(int i = 0; i < ms.length; i++) {
            // 获取方法的访问权限，Method.getModifiers()返回的是访问修饰符的整数表示
            System.out.print(Modifier.toString(ms[i].getModifiers()) + " ");

            // 得到方法的返回值类型---->getReturnType()得到的是返回值类型的类类型
            Class<?> returnType = ms[i].getReturnType();
            System.out.print(returnType.getName() + " ");

            // 获取方法的名称
            System.out.print(ms[i].getName() + "( ");

            // 获取方法的参数列表----->参数列表的类类型
            @SuppressWarnings("rawtypes")
            Class[] paramTypes = ms[i].getParameterTypes();
            for(int j = 0; j < paramTypes.length; j++) {
                if(j != paramTypes.length - 1) {
                    System.out.print(paramTypes[j].getName() + ", ");
                } else {
```

```
                    System.out.print(paramTypes[j].getName());
                }
            }

            System.out.println(" )");
        }
    }

    /**
     * @Description: 对象的构造函数信息
     */
    public static void printConstructorMessage(Object obj) {
        Class<? extends Object> clazz = obj.getClass();

        /**
         * 构造函数也是对象，是java.lang.reflect.Constructor类的对象
         * getConstructors()获取所有的public的构造函数
         * getDeclaredConstructors()获取自己定义的所有的构造函数
         */
        System.out.println("该类的构造方法：");
        @SuppressWarnings("rawtypes")
        Constructor[] cs = clazz.getDeclaredConstructors();
        for(int i = 0; i < cs.length; i++) {
            // 获取构造方法的访问权限，Constructor.getModifiers()返回的是访问修饰符的整数表示
            System.out.print(Modifier.toString(cs[i].getModifiers()) + " ");
            // 构造方法没有返回值，获取构造方法的名称
            System.out.print(cs[i].getName() + "( ");

            // 获取方法的参数列表----->参数列表的类类型
            @SuppressWarnings("rawtypes")
            Class[] paramTypes = cs[i].getParameterTypes();
            for(int j = 0; j < paramTypes.length; j++) {
                if(j != paramTypes.length - 1) {
                    System.out.print(paramTypes[j].getName() + ", ");
                } else {
                    System.out.print(paramTypes[j].getName());
                }
            }

            System.out.println(" )");
        }
    }
}
```

四、方法的反射调用

反射之中包含了一个「反」字，所以想要解释反射就必须先从「正」开始解释。

一般情况下，我们使用某个类时必定知道它是什么类，是用来做什么的。于是我们直接对这个类进行实例化，之后使用这个类对象进行操作。

```
Test test = new Test(); //直接初始化，「正射」
test.setNum(4);
```

上面这样子进行类对象的初始化，我们可以理解为「正」。

而反射则是一开始并不知道我要初始化的类对象是什么，自然也无法使用 new 关键字来创建对象了。

这时候，我们使用 JDK 提供的反射 API 进行反射调用：

```
// 通过反射的机制调用类中的方法（在无法直接创建类的对象的时候）
Class clazz = null;
try {
    clazz = Class.forName("com.zxt.classload.Test");
    Test test1 = (Test) clazz.newInstance();
    // 获取方法
    Method method = clazz.getMethod("setNum", int.class);
    // 或者
    // Method method = clazz.getMethod("setNum", new Class[]{int.class});
    method.invoke(test1, 4);

} catch (Exception e) {
    e.printStackTrace();
}
```

上面两段代码的执行结果，其实是完全一样的。但是其思路完全不一样，第一段代码在未运行时就已经确定了要运行的类（Test），而第二段代码则是在运行时通过字符串值才得知要运行的类（com.zxt.classload.Test）。

一个方法由方法名称和参数列表唯一决定，反射机制调用方法的方式：method.invoke(对象，参数列表)；method 表示该方法 Method 类的对象。

程序实例

```
public class Test {
    private int num = 0;
    public int getNum() {
        return num;
    }
```

```
    public void setNum(int num) {
        this.num = num;
    }

    public void printNum() {
        System.out.println("Num is: " + num);
    }
}

import java.lang.reflect.Method;
/**
 * @Description: 通过反射机制调用类的方法
 */
public class MethodUtil {
    // 方法由方法名称和参数列表唯一确定
    public static void main(String[] args) {
        // 传统调用对象类中的方法的方式（也可以理解为正射），条件是需要构造Test类的对象（未运行的编译期就要创建），才能调用其中的方法
        Test test = new Test();
        test.setNum(4);
        test.printNum();

        // 通过反射的机制调用类中的方法（在无法直接创建类的对象的时候）
        Class clazz = null;
        try {
            clazz = Class.forName("com.zxt.classload.Test");
            Test test1 = (Test) clazz.newInstance();
            // 获取方法
            Method method = clazz.getMethod("setNum", int.class);
            // 或者
            // Method method = clazz.getMethod("setNum", new Class[]{int.class});
            // invoke()函数的返回值即method对象代表的方法的返回值，无返回值则返回null
            method.invoke(test1, 4);

            Method printMethod = clazz.getMethod("printNum");
            // Method printMethod = clazz.getMethod("printNum", new Class[]{});
            printMethod.invoke(test1);
            // printMethod.invoke(test1, new Class[]{});

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

总结反射的一些作用：

1、如果我知道一个类的名称/或者它的一个实例对象，我就能把这个类的所有方法和变量的信息找出来(方法名，变量名，方法，修饰符，类型，方法参数等等所有信息)。

2、如果我还明确知道这个类里某个变量的名称，我还能得到这个变量当前的值。

3、当然，如果我明确知道这个类里的某个方法名+参数个数类型，我还能通过传递参数来运行那个类里的那个方法。

4.1、集合泛型的本质理解

```
import java.lang.reflect.Method;
import java.util.ArrayList;

/**
 * @Description: 反射机制来理解Java泛型
 */
public class GenericTest {

    public static void main(String[] args) {
        // 没有指定ArrayList集合的类型时，可以添加任意Object
        ArrayList list = new ArrayList();
        list.add(12);
        list.add("hello");

        // 指定String类型之后，list1中只能添加String类型的元素
        ArrayList<String> list1 = new ArrayList<String>();
        list1.add("hello");
        // list1.add(10); 编译出错，可以通过反射机制来实现
        // 因此集合泛型是为了避免错误输入

        Class c1 = list.getClass();
        Class c2 = list1.getClass();
        System.out.println(c1 == c2); // true

        /**
         *  c1 == c2 为true，说明编译之后集合的泛型被擦除（去泛型化）
         *  Java中集合的泛型，是防止错误输入的，只有在编译阶段有效，绕过编译就无效了
         *  而反射的操作都是在编译之后的操作，因此可以利用反射机制实现list1.add(10)
         */
        // 验证：通过方法的反射操作来绕过编译
        try {
            Method addMethod = c2.getMethod("add", Object.class);
            // 绕过编译就绕过了泛型
```

```
            addMethod.invoke(list1, 10);
            System.out.println(list1.size());
            // 此时list1无法通过foreach来遍历

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

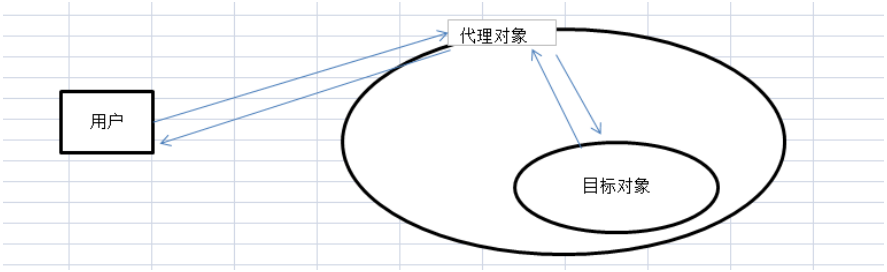
五、代理模式

代理模式的定义：给某一个对象提供一个代理，并由代理对象控制对原对象的引用。

代理(Proxy)是一种设计模式，提供了对目标对象另外的访问方式；即通过代理对象访问目标对象。这样做的好处是：可以在目标对象实现的基础上，增强额外的功能操作，即扩展目标对象的功能。

这里使用到编程中的一个思想：**不要随意去修改别人已经写好的代码或者方法**，如果需改修改，可以通过代理的方式来扩展该方法。

举个例子来说明代理的作用：假设我们想邀请一位明星，那么并不是直接联系明星，而是联系明星的经纪人，来达到同样的目的。明星就是一个目标对象，他只要负责活动中的节目，而其他琐碎的事情就交给他的代理人(经纪人)来解决。这就是代理思想在现实中的一个例子。



代理模式的关键点是：代理对象与目标对象。**代理对象是对目标对象的扩展**，并会调用目标对象。

代理模式包含如下角色：

ISubject：抽象主题角色，是一个接口。该接口是对象和它的代理共用的接口。

RealSubject：真实主题角色，是实现抽象主题接口的类。

Proxy：代理角色，内部含有对真实对象 RealSubject 的引用，从而可以操作真实对象。代理对象提供与真实对象相同的接口，以便在任何时刻都能代替真实对象。同时，代理对象可以在执行真实对象操作时，附加其他的操作，相当于对真实对象进行封装。

代理模式的应用：

远程代理：也就是为一个对象在不同的地址空间提供局部代表。这样可以隐藏一个对象存在于不同地址空间的事实。

虚拟代理：是根据需要创建开销很大的对象。通过它来存放实例化需要很长时间的真实对象。

安全代理：用来控制真实对象访问时的权限。

智能代理：是指当调用真实的对象时，代理处理一些另外的事情。

一般将代理分类的话，可分为静态代理和动态代理两种。

5.1、静态代理

静态代理比较简单，是由程序员编写的代理类，并在程序运行前就编译好的，而不是由程序动态产生代理类，这就是所谓的静态。

考虑这样的场景，管理员在网站上执行操作，在生成操作结果的同时需要记录操作日志，这是很常见的。此时就可以使用代理模式，代理模式可以通过聚合和继承两种方式实现：

```
/**
 * @Description: 抽象主题接口
 * @author:  zxt
 * @time: 2018年7月7日 下午2:29:46
 */
public interface Manager {
    public void doSomething();
}

/**
 * @Description: 真实的主题类
 * @author:  zxt
 * @time: 2018年7月7日 下午2:31:21
 */
public class Admin implements Manager {
    @Override
    public void doSomething() {
        System.out.println("这是真实的主题类：Admin doSomething!!!");
    }
}

/**
 * @Description: 以聚合的方式实现代理主题
 * @author:  zxt
 * @time: 2018年7月7日 下午2:37:08
 */
public class AdminPoly implements Manager {
    // 真实主题类的引用
    private Admin admin;

    public AdminPoly(Admin admin) {
        this.admin = admin;
```

```
    }

    @Override
    public void doSomething() {
        System.out.println("聚合方式实现代理：Admin操作开始！！");
        admin.doSomething();
        System.out.println("聚合方式实现代理：Admin操作结束！！");
    }
}

/**
 * @Description: 继承方式实现代理
 * @author:  zxt
 * @time: 2018年7月7日 下午2:40:39
 */
public class AdminProxy extends Admin {
    @Override
    public void doSomething() {
        System.out.println("继承方式实现代理：Admin操作开始！！");
        super.doSomething();
        System.out.println("继承方式实现代理：Admin操作结束！！");
    }
}

public static void main(String[] args) {
    // 1、聚合方式的测试
    Admin admin = new Admin();
    Manager manager = new AdminPoly(admin);
    manager.doSomething();

    System.out.println("============================");
    // 2、继承方式的测试
    AdminProxy proxy = new AdminProxy();
    proxy.doSomething();
}
```

聚合实现方式中代理类聚合了被代理类，且**代理类及被代理类都实现了同一个接口，可实现灵活多变。**继承式的实现方式则不够灵活。

比如，在管理员操作的同时需要进行权限的处理，操作内容的日志记录，操作后数据的变化三个功能。三个功能的排列组合有 6 种，也就是说使用继承要编写 6 个继承了 Admin 的代理类，而使用聚合，仅需要针对权限的处理、日志记录和数据变化三个功能编写代理类，在业务逻辑中根据具体需求改变代码顺序即可。

缺点：

1)、代理类和委托类实现了相同的接口，代理类通过委托类实现了相同的方法。这样就出现了大量的代码重复。如果接口增加一个方法，除了所有实现类需要实现这个方法外，所有代理类也需要实现此方法。增加了代码维护的复杂度。

2)、代理对象只服务于一种类型的对象，如果要服务多类型的对象。势必要为每一种对象都进行代理，静态代理在程序规模稍大时就无法胜任了。

5.2、动态代理

实现动态代理的关键技术是反射。

一般来说，对代理模式而言，一个主题类与一个代理类一一对应，这也是静态代理模式的特点。

但是，也存在这样的情况，有 n 各主题类，但是代理类中的“前处理、后处理”都是一样的，仅调用主题不同。也就是说，多个主题类对应一个代理类，共享“前处理，后处理”功能，动态调用所需主题，大大减小了程序规模，这就是动态代理模式的特点。动态代理主要有两种：JDK 自带的动态代理和 CGLIB 动态代理。

首先是另一个静态代理的实例：

1、一个可移动接口

```
public interface Moveable {

    public void move();
}
```

2、一个实现了该接口的 Car 类

```
public class Car implements Moveable {

    @Override
    public void move() {
        try {
            Thread.sleep(new Random().nextInt(1000));
            System.out.println("汽车行驶中----");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

3、现在需要有一个代理类来记录 Car 的运行时间：

```
public class CarTimeProxy implements Moveable {
    private Moveable m;

    public CarTimeProxy(Moveable m) {
        super();
        this.m = m;
    }

    @Override
    public void move() {
        long startTime = System.currentTimeMillis();
        System.out.println("汽车行驶前----");
        m.move();
        long endTime = System.currentTimeMillis();
        System.out.println("汽车行驶结束----行驶时间为: " + (endTime - startTime) + "
毫秒！");
    }
}
```

4、另一个代理类记录 Car 的日志：

```
public class CarLogProxy implements Moveable {
    private Moveable m;

    public CarLogProxy(Moveable m) {
        super();
        this.m = m;
    }

    @Override
    public void move() {
        System.out.println("日志开始");
        m.move();
        System.out.println("日志结束");
    }
}
```

5、客户端的调用：

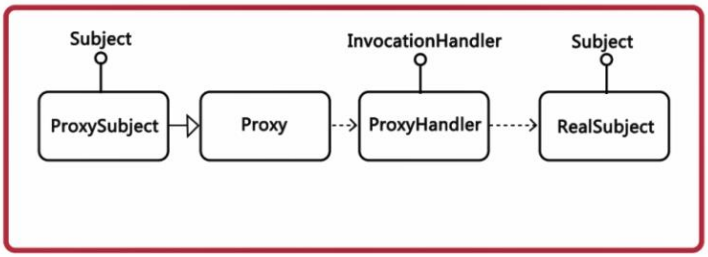
```
public class CarTest {

    public static void main(String[] args) {
        Car car = new Car();
        // 先写日志，再计时
        CarTimeProxy ctp = new CarTimeProxy(car);
        CarLogProxy clp = new CarLogProxy(ctp);
```

```
        clp.move();

        System.out.println();
        // 先计时，再写日志
        CarLogProxy clp1 = new CarLogProxy(car);
        CarTimeProxy ctp1 = new CarTimeProxy(clp1);
        ctp1.move();
    }
}
```

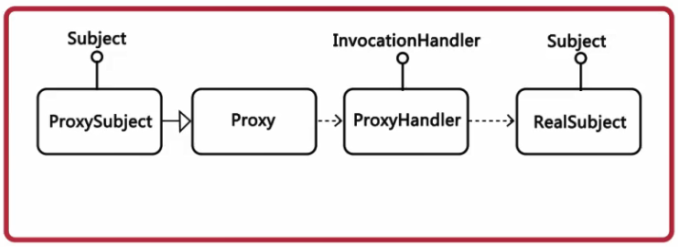
```
Problems  Console  Javadoc  JUnit
<terminated> CarTest [Java Application] D:\Program Java\jre1.8.0_60\
日志开始
汽车行驶前----
汽车行驶中----
汽车行驶结束----行驶时间为：420毫秒!
日志结束

汽车行驶前----
日志开始
汽车行驶中----
日志结束
汽车行驶结束----行驶时间为：977毫秒!
```

5.2.1、JDK 的动态代理

在 java 的动态代理机制中，有两个重要的类或接口，一个是 InvocationHandler(Interface)、另一个则是 Proxy(Class)，这一个类和接口是实现我们动态代理所必须用到的。



Java动态代理类位于java.lang.reflect包下，一般主要涉及到以下两个类：
(1)Interface InvocationHandler：该接口中仅定义了一个方法
public object invoke(Object obj,Method method, Object[] args)
在实际使用时，第一个参数obj一般是指代理类，method是被代理的方法，args为该方法的参数数组。 这个抽象方法在代理类中动态实现。



(2)Proxy：该类即为动态代理类
static Object newProxyInstance(ClassLoader loader, Class[] interfaces, InvocationHandler h)：返回代理类的一个实例，返回后的代理类可以当作被代理类使用(可使用被代理类的在接口中声明过的方法)

JDK 动态代理的实现

1、创建一个实现接口 InvocationHandler 的类，它必须实现 invoke 方法。

使用 JDK 动态代理类时，需要实现 InvocationHandler 接口，所有动态代理类的方法调用，都会交由 InvocationHandler 接口实现类里的 invoke()方法去处理。这是动态代理的关键所在。

2、创建被代理的类以及接口。

3、调用 Proxy 的静态方法，创建代理类。

newProxyInstance(ClassLoader loader, Class[] interfaces, InvocationHandler h);

4、通过代理调用方法。

使用 JDK 动态代理的方式实现上面 Car 的时间代理：

1、首先是 InvocationHandler 接口的实现类：

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

public class TimeHandler implements InvocationHandler {
    // 被传递过来的要被代理的对象
    private Object object;

    public TimeHandler(Object object) {
        super();
        this.object = object;
    }

    /**
     * proxy：被代理的对象
     * method：被代理的方法
```

```
     * args：被代理方法的参数
     * 函数返回：method的返回
     *
     */
    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable
{
        long startTime = System.currentTimeMillis();
        System.out.println("汽车行驶前----");
        method.invoke(object, args);
        long endTime = System.currentTimeMillis();
        System.out.println("汽车行驶结束----行驶时间为：" + (endTime - startTime) + "
毫秒！");
        return null;
    }
}
```

2、创建动态代理类：

```
/**
 * @Description: JDK动态代理的测试类
 * @author:  zxt
 * @time: 2019年3月1日 下午7:59:29
 */
public class TimeHandlerTest {

    public static void main(String[] args) {
        // 需要被代理的对象
        Car car = new Car();
        InvocationHandler h = new TimeHandler(car);

        Class<?> clazz = car.getClass();

        /**
         * 参数一：类加载器
         * 参数二：被代理类实现的接口
         * 参数三：InvocationHandler实例
         *
         * 函数返回：返回由InvocationHandler接口接收的被代理类的一个动态代理类对象
         */
        Moveable m = (Moveable) Proxy.newProxyInstance(clazz.getClassLoader(),
clazz.getInterfaces(), h);
        m.move();
    }
}
```

5.2.2、cglib 动态代理

JDK 动态代理可以在运行时动态生成字节码，主要使用到了一个接口 InvocationHandler 与 Proxy.newProxyInstance 静态方法。使用内置的 Proxy 实现动态代理有一个问题：被代理的类必须要实现某接口，未实现接口则没办法完成动态代理。

如果项目中有些类没有实现接口，则不应该为了实现动态代理而刻意去抽象出一些没有实际意义的接口，通过 cglib 可以解决该问题。

CGLIB(Code Generation Library)是一个开源项目，是一个强大的，高性能，高质量的 Code 生成类库，**它可以在运行期扩展 Java 类与实现 Java 接口**，通俗地说 cglib 可以在运行时动态生成字节码。

使用 cglib 完成动态代理，大概的原理是：cglib 继承被代理的类，重写方法，织入通知，动态生成字节码并运行。对指定目标类产生一个子类，通过方法拦截技术拦截所有父类的方法调用，因为是继承实现所以 final 类是没有办法动态代理的。

CGLIB 动态代理实例：

```
import java.util.Random;

// 不实现接口的被代理类
public class Train {

    public void move() {
        try {
            Thread.sleep(new Random().nextInt(1000));
            System.out.println("火车行驶中----");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

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

import net.sf.cglib.proxy.Enhancer;
import net.sf.cglib.proxy.MethodInterceptor;
import net.sf.cglib.proxy.MethodProxy;

public class CglibProxy implements MethodInterceptor {
    private Enhancer enhancer = new Enhancer();
```

```
    // 得到代理类的方法
    public Object getProxy(Class<?> clazz) {
        // 设置创建子类的类  （即我们需要为哪个类产生代理类）
        enhancer.setSuperclass(clazz);
        enhancer.setCallback(this);

        return enhancer.create();
    }

    /**
     * 拦截所有目标类方法的调用
     * object：目标类的实例
     * method：目标类的目标方法的反射实例
     * args：目标方法的参数
     * proxy：代理类的实例
     */
    @Override
    public Object intercept(Object object, Method method, Object[] args, MethodProxy
proxy) throws Throwable {
        long startTime = System.currentTimeMillis();
        System.out.println("火车行驶前----");

        // 代理类调用父类的方法 (由于Cglib动态代理的实现是通过继承被代理类，因此代理类这里
需要调用父类的方法)
        proxy.invokeSuper(object, args);

        long endTime = System.currentTimeMillis();
        System.out.println("火车行驶结束----行驶时间为：" + (endTime - startTime) + "
毫秒！");
        return null;
    }
}
```

```
public class CglibProxyTest {

    public static void main(String[] args) {
        CglibProxy cglibProxy = new CglibProxy();
        Train train = (Train) cglibProxy.getProxy(Train.class);
        train.move();
    }
}
```

5.3、JDK 动态代理的模拟实现

模拟 JDK 动态代理的实现，根据 Java 源代码动态生成代理类。

```
package com.zxt.jdkproxy;

import java.io.File;
import java.lang.reflect.Constructor;
import java.lang.reflect.Method;

import javax.tools.JavaCompiler;
import javax.tools.JavaCompiler.CompilationTask;
import javax.tools.StandardJavaFileManager;
import javax.tools.ToolProvider;

import org.apache.commons.io.FileUtils;

import com.zxt.staticproxy.Car;

/**
 *
 * @Description: 模拟JDK动态代理的实现
 * 动态代理的实现思路：
 * 实现功能：通过自定义的Proxy的newProxyInstance方法返回代理对象
 * 1、声明一段源码（动态产生代理）
 * 2、编译源码（JDK Compiler API），产生新的类（代理类）
 * 3、将这个类load到内存当中，产生一个新的对象（代理对象）
 * 4、return 代理对象
 *
 * @author:  zxt
 *
 * @time: 2019年4月18日 下午3:44:58
 *
 */
public class MyProxy {

    @SuppressWarnings({ "rawtypes", "unchecked" })
    public static Object newProxyInstance(Class<?> inteface) throws Exception {
        // 1、声明一段源码（动态产生代理）
        String rt = "\r\n";
        String methodStr = "";
        for(Method m : inteface.getMethods()) {
            methodStr += "    @Override" + rt
                       + "    public void " + m.getName() + "() {" + rt
                       + "        System.out.println(\"日志开始\");" + rt
```

```
                        + "        m." + m.getName() + "();" + rt
                        + "        System.out.println(\"日志结束\");" + rt
                        + "    }";
        }
        String code =
            "package com.zxt.jdkproxy;" + rt + "\n"
            + "import com.zxt.staticproxy.Moveable;" + rt + "\n"
            + "public class $MyProxy0 implements " + inteface.getSimpleName() + " {"
+ rt + "\n"
            + "  private " + inteface.getSimpleName() + " m;" + rt + "\n"
            + "  public $MyProxy0(" + inteface.getSimpleName() + " m) {" + rt
            + "      super();" + rt
            + "      this.m = m;" + rt
            + "  }" + rt + "\n"
            + methodStr + rt + "\n"
            + "}";


        // 由源代码生成java类文件
        String filename = System.getProperty("user.dir") +
"/bin/com/zxt/jdkproxy/$MyProxy0.java";
        File file = new File(filename);
        // 使用commons-io里面的简便的工具类来写文件
        FileUtils.writeStringToFile(file, code, "UTF-8");


        // 2、编译源码（JDK Compiler API），产生新的类（代理类）
        // 拿到编译器
        JavaCompiler compiler = ToolProvider.getSystemJavaCompiler();
        // 文件管理者
        StandardJavaFileManager fileManager = compiler.getStandardFileManager(null,
null, null);
        // 获取文件
        Iterable units = fileManager.getJavaFileObjects(filename);
        // 获取编译任务
        CompilationTask task = compiler.getTask(null, fileManager, null, null, null,
units);
        // 编译
        task.call();
        fileManager.close();

        // 3、加载到内存
        ClassLoader cl = ClassLoader.getSystemClassLoader();
        Class c = cl.loadClass("com.zxt.jdkproxy.$MyProxy0");
```

```
        // 4、返回代理类
        Constructor ctr = c.getConstructor(inteface);
        return ctr.newInstance(new Car());
    }


    public static void main(String[] args) {

    }
}
```

```
public class MyProxyTest {
    public static void main(String[] args) throws Exception {
        Moveable m = (Moveable) MyProxy.newProxyInstance(Moveable.class);
        m.move();
    }
}
```

可以发现上述实现中的源代码是写死在类中的，因此无法对任意类进行动态代理，所以仿照 InvocationHandler 接口，定义自己的 InvocationHandler 接口从而实现对不同的类进行动态代理。

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

public interface MyInvocationHandler {

    public void invoke(Object o, Method m);
}
```

实现该接口的类

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

public class MyLogHandler implements MyInvocationHandler {
    // 需要被代理的对象
    private Object target;

    public MyLogHandler(Object target) {
        super();
        this.target = target;
    }

    @Override
    public void invoke(Object o, Method m) {
        try {
```

```
            System.out.println("日志开始");
            m.invoke(target);
            System.out.println("日志结束");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

对动态代理 MyProxy 类进行改进

```
/**
 *
 * @Description: 模拟JDK动态代理的实现
 * 动态代理的实现思路:
 * 实现功能：通过自定义的Proxy的newProxyInstance方法返回代理对象
 * 1、声明一段源码（动态产生代理）
 * 2、编译源码（JDK Compiler API），产生新的类（代理类）
 * 3、将这个类load到内存当中，产生一个新的对象（代理对象）
 * 4、return 代理对象
 *
 * @author:  zxt
 *
 * @time: 2019年4月18日 下午3:44:58
 *
 */
public class MyProxy {

    @SuppressWarnings({ "rawtypes", "unchecked" })
    public static Object newProxyInstance(Class<?> inteface, MyInvocationHandler h)
throws Exception {
        // 1、声明一段源码（动态产生代理）
        String rt = "\r\n";
        String methodStr = "";
        for(Method m : inteface.getMethods()) {
            methodStr += "   @Override" + rt
                    + "   public void " + m.getName() + "() {" + rt
                    + "       try { " + rt
                    + "           Method md = " + inteface.getSimpleName() +
".class.getMethod(\""
                                        + m.getName() + "\");" + rt
                    + "           h.invoke(this, md);" + rt
                    + "       } catch (Exception e) { " + rt
                    + "           e.printStackTrace();" + rt
                    + "       }" + rt
                    + "   }";
```

```
        }
        String code =
            "package com.zxt.jdkproxy;" + rt + "\n"
            + "import java.lang.reflect.Method;" + rt
            + "import com.zxt.staticproxy.Moveable;" + rt + "\n"
            + "public class $MyProxy0 implements " + inteface.getSimpleName() + " {"
+ rt + "\n"
            + " private MyInvocationHandler h;" + rt + "\n"
            + " public $MyProxy0( MyInvocationHandler h ) {" + rt
            + "     this.h = h;" + rt
            + " }" + rt + "\n"
            + methodStr + rt + "\n"
            + "}";


        // 由源代码生成java类文件
        String filename = System.getProperty("user.dir") +
"/bin/com/zxt/jdkproxy/$MyProxy0.java";
        File file = new File(filename);
        // 使用commons-io里面的简便的工具类来写文件
        FileUtils.writeStringToFile(file, code, "UTF-8");


        // 2、编译源码（JDK Compiler API），产生新的类（代理类）
        // 拿到编译器
        JavaCompiler compiler = ToolProvider.getSystemJavaCompiler();
        // 文件管理者
        StandardJavaFileManager fileManager = compiler.getStandardFileManager(null,
null, null);
        // 获取文件
        Iterable units = fileManager.getJavaFileObjects(filename);
        // 获取编译任务
        CompilationTask task = compiler.getTask(null, fileManager, null, null, null,
units);
        // 编译
        task.call();
        fileManager.close();

        // 3、加载到内存
        ClassLoader cl = ClassLoader.getSystemClassLoader();
        Class c = cl.loadClass("com.zxt.jdkproxy.$MyProxy0");

        // 4、返回代理类
        Constructor ctr = c.getConstructor(MyInvocationHandler.class);
```

```
        return ctr.newInstance(h);
    }
}
```

测试类：

```
public class MyProxyTest {

    public static void main(String[] args) throws Exception {
        // 需要被代理的对象
        Car car = new Car();
        MyInvocationHandler h = new MyLogHandler(car);

        Moveable m = (Moveable) MyProxy.newProxyInstance(Moveable.class, h);
        m.move();
    }
}
```