

java 内功进阶

--msn:xiaohuiq@hotmail.com 根据张孝祥老师视频整理

静态导入

这应该是 JDK1.5 新特性中最简单的一个特性了

比如我们现在要用到 Math 类中的方法来求取绝对值，以前的做法：

```
package org.qxh.jdk15;
```

```
public class StaticImport {  
  
    public static void main(String[] args) {  
        System.out.println(Math.abs(-3));  
    }  
}
```

如果使用静态导入特性，我们就可以直接写：

```
package org.qxh.jdk15;
```

```
import static java.lang.Math.*;
```

```
public class StaticImport {  
  
    public static void main(String[] args) {  
        System.out.println(abs(-3));  
    }  
}
```

看到不同了吗？abs 方法前的类名没有了，导入语句中增加了一个

```
import static java.lang.Math.*;
```

这句话就表示把 Math 类中的所有静态方法导入，下面的程序如果再用到 Math 类中的静态方法就可以不用再加 Math. 这部分了，比如：

```
System.out.println(abs(-3));  
System.out.println(min(-3,4));  
System.out.println(max(-3,4));
```

当然了，如果你不想导入 Math 下的所有静态方法，只想导入 abs 方法，那你的静态导入语句就可以这么写：

```
import static java.lang.Math.abs;
```

OK，静态导入就到这里~

可变参数

问题：

一个方法接受的参数个数不固定，例如

```
System.out.println(add(1,2));
```

```
System.out.println(add(1,2,3));
```

当然了，我们可以采用数组或是集合来作为参数，但是可变参数会给你提供一种更加优雅的方式

可变参数的特点：

- 1) 只能出现在参数列表的最后
- 2) ...位于变量类型和变量名之间，前后有无空格都可以
- 3) 调用可变参数的方法时，编译器为该可变参数隐含创建一个数组，在方法体中以数组的形式访问可变参数

```
package org.qxh.jdk15;
```

```
public class VariableParameter {

    public static void main(String[] args) {
        System.out.println(add(1,2));
        System.out.println(add(1,2,3));
    }

    static int add(int...args){
        int sum = 0;
        for(int arg : args){
            sum += arg;
        }
        return sum;
    }
}
```

当然了，上面这种应用已经比较典型了，我们再举一个比较常见的例子：

```
package org.qxh.jdk15;
```

```
public class DBHelper {

    public static void execute(String sql, Object... params){
        //构造PreparedStatement, 执行SQL
    }

    //使用场景
    public static void main(String[] args) {
```

```
        String sql = "update t_user set visible = ? where type = ?";
        DBHelper.execute(sql, true, 1);
    }
}
```

做 WEB 应用很常见的操作就是跟 DB 打交道，一个 BaseDao 或是 DBHelper 是必不可少的，上面的 DBHelper.execute 方法签名就比较好的运用了可变参数特性，使上层代码调用更方便

增强 for 循环

以前写 for 循环一般都是 for(int i=0;i<arr.length;i++), JDK1.5 之后出现了增强 for 循环，我们可以用一种更方便的方式来做相同的事情

语法

```
for(type 变量名 : 集合变量名) {...}
```

注意事项

迭代变量必须在 () 中定义

集合变量可以是数组或实现了 Iterable 接口的集合类

举例

可变参数那部分其实就用到了这一点

```
static int add(int...args){
    int sum = 0;
    for(int arg : args){
        sum += arg;
    }
    return sum;
}
```

额外提一点，在增强 for 循环体内有些操作是不允许的，我不知道应该如何描述这种情况，直接给大家上例子：

```
package org.qxh.jdk15;

import java.util.Arrays;
import java.util.List;
```

```

public class EnhanceFor {

    public static void main(String[] args) {
        List<String> strList = Arrays.asList("aa", "bb", "cc");
        for(String str : strList){
            System.out.println(str);
            if("bb".equals(str)){
                strList.remove("bb");
            }
        }
    }
}

```

对集合 strList 的 remove 操作这里就会报错，抛出下面的异常：
[java.lang.UnsupportedOperationException](#)

其实想想也确实不能这么操作，不知道怎么解释，只可意会不可言传啊，呵呵~

自动拆箱与装箱

自动拆箱与装箱是针对基本类型及其包装类型而言的，比如现在我们要实例化一个 Integer 类型，以前的做法可能是：Integer i = new Integer(5); 现在就直接 Integer I = 5; 即可，得，别的也不说了，直接上例子：

```
package org.qxh.jdk15;
```

```

public class AutoBox {

    public static void main(String[] args) {
        //自动装箱，把一个int类型的数字5自动装箱为Integer类型
        Integer i = 5;
        //下面是自动拆箱，把一个Integer类型自动拆箱为int类型，
        //然后和int类型的6做运算
        System.out.println(i+6);
    }
}

```

自动拆箱与装箱看上去还是蛮简单的，不过这里要给大家出个小题目，看下面打印的是 true 还是 false:

```

public void test2() {
    Integer i1 = 22;
    Integer i2 = 22;
    System.out.println(i1==i2);
}

```

我们一般理解 `Integer i1 = 22;` 相当于 `Integer i1 = new Integer(22);` 所以 `i1` 和 `i2` 是两个对象，此处应该打印 `false`，而实际上此处打印的是 `true`！`0(∩_∩)0` 不要惊讶，还有更有意思的在后边，看下面打印输出什么：

```
public void test2() {  
    Integer i1 = 222;  
    Integer i2 = 222;  
    System.out.println(i1==i2);  
}
```

如果上面的程序是输出 `true`，那这个也必然是输出 `true` 了，而实际上这个输出的是 `false`！原因如下：

如果你要自动装箱的数字比较小，位于 `-128~127` 之间，JDK 会采用享元模式，那么享元模式是什么呢？你可以 google 一下，我这里摘抄的百度百科中的一段话：

享元模式（英语：Flyweight Pattern）是一种软件[设计模式](#)。它使用共享物件，用来尽可能减少内存使用量以及分享资讯给尽可能多的相似物件；它适合用于当大量物件只是重复因而导致无法令人接受的使用大量内存。**通常物件中的部分状态是可以分享。常见做法是把它们放在外部数据结构，当需要使用时再将它们传递给享元。**

典型的享元模式的例子为文书处理器中以图形结构来表示字符。一个做法是，每个[字形](#)有其字型外观，字模 metrics，和其它格式资讯，但这会使每个字符就耗用上千字节。取而代之的是，每个字符参照到一个共享字形物件，此物件会被其它有共同特质的字符所分享；只有每个字符（文件中或页面中）的位置才需要另外储存。

不在 `-128~127` 之间的数字 JDK 会认为他们出现的频率比较小，没有必要采用享元模式，所以上面的程序返回 `false`

不知道说明白了没有，其实自动装箱和拆箱比较简单，主要是此处引入了一个享元设计模式，值得大家稍微研究一下~

枚举

枚举产生的原因

枚举其实主要就是为了规范一些变量的取值范围，比如我这有一个表示星期天的变量，`int sunday`；你觉得它应该取几？`0`？还是 `7`？每个人可能有不同的看法，不同程序员在一起协同工作，旧人走新人来，老程序员用 `7` 来代表周日，但是接手的新人可能会采用 `0` 来表示.....

如果采用枚举的话就只能取若干固定值中的一个，否则编译器就会报错，枚举可以让编译器在编译时就可以控制源程序中填写的非法值，普通变量的方式在开发阶段无法实现这一目标

枚举实现原理

java 早期是没有枚举这个概念的，后来随着 java 应用的越来越广，枚举又被加了进来，在语言层面得到了重新支持

我们先不用枚举，自己手工写一个表示星期的类，来模拟枚举的作用

```
package org.qxh.jdk15;
```

```
public class WeekDay {
```

```
    //先把构造函数私有化，防止new出新值
```

```
    private WeekDay() {  
    }  
}
```

```
    public static final WeekDay MON= new WeekDay();
```

```
    public static final WeekDay TUE= new WeekDay();
```

```
    public static final WeekDay WED= new WeekDay();
```

```
    public static final WeekDay THU= new WeekDay();
```

```
    public static final WeekDay FRI= new WeekDay();
```

```
    public static final WeekDay SAT= new WeekDay();
```

```
    public static final WeekDay SUN= new WeekDay();
```

```
}
```

这样你在其他类中为 WeekDay 变量赋值的时候，就只能取我事先为你准备好的这 7 个值：

```
package org.qxh.jdk15;
```

```
public class EnumTest {
```

```
    public static void main(String[] args) {
```

```
        WeekDay sunday = WeekDay.SUN;
```

```
    }
```

```
}
```

枚举其实就是这个原理，只是比这个更复杂一些罢了

再上一个表示性别的例子，不过要稍微复杂一些哦

```
package org.qxh.jdk15;
```

```
public abstract class Gender {
```

```
    private Gender() {  
    }  
}
```

```
    public static final Gender MALE = new Gender();
```

```

        public Gender oppositeSex() {
            return FEMALE;
        }

};

public static final Gender FEMALE = new Gender() {

    public Gender oppositeSex() {
        return MALE;
    }
};

public abstract Gender oppositeSex();

public String toString() {
    return this == MALE ? "男" : "女";
}
}

```

这是一个表示性别的类，模拟枚举功能，用了内部类，抽象方法，使用的时候可以这样用：

```

package org.qxh.jdk15;

public class EnumTest {

    public static void main(String[] args) {
        Gender male = Gender.MALE;
        System.out.println(male.oppositeSex());
    }
}

```

枚举的简单应用

下面我们就用 JDK 给我们提供的枚举类来定义上面的性别类：

```

package org.qxh.jdk15;

public class EnumTest {

    //定义枚举，比我们手工来做简单很多吧
    public enum Gender{
        MALE, FEMALE
    }

    public static void main(String[] args) {
        Gender male = Gender.MALE;
    }
}

```

```

        //枚举常用方法1: 获取字符串面值
        System.out.println(male.name());
        //枚举常用方法2: 获取该枚举所在位置,从0开始
        System.out.println(male.ordinal());
        //枚举常用方法3: 把一个字符串转变为枚举
        System.out.println(Gender.valueOf("FEMALE"));
        //枚举常用方法4: 枚举遍历
        Gender[] genders = Gender.values();
        for(Gender gender : genders){
            System.out.println(gender.name());
        }
    }
}

```

枚举是一个类,那么它也应该有构造方法,像上面定义的 Gender 枚举是最简单的形式,它默认有一个无参的构造方法,如果我这么写:

```

public enum Gender{
    MALE,FEMALE(3);
    private Gender(){
        System.out.println("Gender()");
    }
    private Gender(int flag){
        System.out.println("Gender(flag)");
    }
}

```

MALE 这个枚举值就会调用第一个构造方法,而 FEMALE 这个枚举值就会调用第二个构造方法,即有一个 int 参数的构造方法,这个应该不难理解吧

下面我们再来看一下有抽象方法有构造函数的枚举

```

package org.qxh.jdk15;

public enum TrafficLamp {
    // GREEN,YELLOW,RED实际都是TrafficLamp的一个子类
    GREEN(30) {
        public TrafficLamp nextLamp() {
            return YELLOW;
        }
    },
    YELLOW(5) {
        public TrafficLamp nextLamp() {
            return RED;
        }
    },
    RED(45) {

```



```

        public TrafficLamp nextLamp() {
            return GREEN;
        }
    };
    // 枚举可以有抽象方法
    public abstract TrafficLamp nextLamp();

    // 枚举可以有字段
    private int time;

    // 枚举可以有构造方法
    private TrafficLamp(int time) {
        this.time = time;
    }
}

```

枚举掌握到这里基本就差不多了，还有一个比较变态的用法，就是用枚举来做单例

```
package org.qxh.jdk15;
```

```

public enum SingletonEnum {
    INSTANCE
}

```

你说上面这种方式是不是单例？仔细想想，它还真就是个单例，只是这种使用方式不多

反射

这是个很有意思的话题，java 程序员进阶必须要会的知识，平时写程序反射可能用的不多，但是你要做框架，这个知识就是必须要用到的了

Class 类

Class 类是反射的基石，为了理解这个类请慢慢的仔细的研读下面这句话：

Java 类用于描述一类事物的共性，该类事物有什么属性，没有什么属性，至于这个属性的值是什么，则是由这个类的实例对象来确定的，不同的实例对象有不同的属性值。Java 程序中的各个 java 类，它们是否属于同一类事物，是不是可以用一个类来描述这类事物呢？这个身负重任的类的名字就是 Class，要注意与小写 class 关键字的区别哦~Class 类描述了哪些方面的信息呢？类的名字，类的访问属性，类所属于的包名，字段名称的列表、方法名称的列表等等，学习反射，首先就要搞明白 Class 这个类

对比思考：

众多的人用一个什么类来表示？

人—Person

众多的 java 类用一个什么类来表示？

java 类—Class

另外，Person 类代表人，它的实例对象就是张三、李四这样一个个具体的人，Class 类代表 java 类，它的各个实例对象又分别对应什么呢？

- 对应各个类在内存中的字节码，例如 Person 类的字节码，Date 类的字节码等等
- 一个类被类加载器加载到内存中，占用一片存储空间，这个空间里面的内容就是类的字节码，不同的类的字节码是不同的，所以它们在内存中的内容是不同的，这一个个的空间可分别用一个个的对象来表示，这些对象显然具有相同的类型，这个类型是什么呢？这个类型显然就是 Class 类型喽，那么接下来的问题是如何得到各个字节码对应的实例对象（Class 类型的对象）呢？

有三种方式：

- 类名.class，例如：Date.class
- 对象.getClass()，例如：new Date().getClass()
- Class.forName(“全类名”)；例如：Class.forName(“java.util.Date”)；

上面这三种方式得到的字节码都是同一份字节码，所以下面的两个输出都是 true

```
package org.qxh.jdk15;
```

```
import java.util.Date;
```

```
public class ClassTest {
```

```
    public static void main(String[] args) throws ClassNotFoundException
    {
        Date date = new Date();
        Class clazz1 = date.getClass();
        Class clazz2 = Date.class;
        Class clazz3 = Class.forName("java.util.Date");
        System.out.println(clazz1==clazz2);
        System.out.println(clazz3==clazz2);
    }
}
```

对于 Class 类，里边有几个方法个人认为值得注意一下，就是

- isPrimitive() 判定指定的 Class 对象是否表示一个基本类型
- isAnnotation() 判定指定的 Class 对象是否表示一个注解
- isInterface() 判定指定的 Class 对象是否表示一个接口类型
- isArray() 判定此 Class 对象是否表示一个数组类
- isEnum() 当且仅当该类声明为源代码中的枚举时返回 true
- isInstance(Object obj) 判定指定的 Object 是否与此 Class 所表示的对象赋值兼

容

这几个方法不是特别重要，偶尔会用到，知道它们有的时候可以帮你省不少事

反射概述

- 反射就是把 java 类中的各种成分映射成为相应的 java 类。例如：一个 java 类用一个 Class 类的对象来表示，一个类中的组成部分：成员变量、方法、构造函数、包等等信息也用一个个 java 类来表示，就像汽车是一个类，汽车中的发动机，变速箱等等也是一个个的 java 类，表示 java 类的 Class 类显然要提供一系列的方法，来获得其中的变量，方法，构造函数，修饰符，包等信息，这些信息就是用相应类的实例对象来表示，它们是 Field、Method、Constructor、Package 等等
- 一个类中的每个成员都可以用相应的反射 API 类的一个实例对象类表示，通过调用 Class 类的方法可以得到这些实例对象，它们有什么用呢？怎么用？这正是学习和应用反射的要点

Constructor 类

- 代表某个类中的一个构造方法
- 得到某个类的所有的构造方法：

```
Class clazz = Class.forName("java.lang.String");
Constructor[] constructors = clazz.getConstructors();
```
- 得到某一个构造方法：

```
Constructor constructor = clazz.getConstructor(StringBuffer.class);
```

这就表示要得到参数为 StringBuffer 类型的那个构造方法
- 利用构造方法得到一个 String 对象

```
package org.qxh.jdk15;
```

```
import java.lang.reflect.Constructor;
```

```
public class RefectTest {
    @SuppressWarnings("unchecked")
    public static void main(String[] args) throws Exception {
        Class clazz = Class.forName("java.lang.String");
        //要拿到那个参数为StringBuffer类型的构造方法
        Constructor constructor =
clazz.getConstructor(StringBuffer.class);
        //newInstance的返回值是Object类型，所以要强转
        String strObj = (String) constructor.newInstance(new
```

```
StringBuffer("abc"));
    System.out.println(strObj.length());
}
}
```

实际上 Class 也有一个 newInstance 方法，它的内部就是调用的无参的 Constructor 的 newInstance 方法，所以要想通过这种方式来创建对象一定要保证类有一个无参构造方法

Field 类

Field 类表示类中的成员变量，比如我这有一个 RefectPoint 类：

```
package org.qxh.jdk15;

public class RefectPoint {

    private int x;
    public int y;

    public RefectPoint() {

    }

    public RefectPoint(int x, int y) {
        super();
        this.x = x;
        this.y = y;
    }
}
```

其中的 x 和 y 就是该类的成员变量，接下来我们要用反射 API 来操作它

```
package org.qxh.jdk15;

import java.lang.reflect.Field;

public class RefectPointTest {

    public static void main(String[] args) throws Exception {
        RefectPoint point = new RefectPoint(3,5);
        Class pointClass = point.getClass();
        Field fieldY = pointClass.getField("y");
        System.out.println(fieldY.get(point));
    }
}
```

上面的程序运行可以输出 5，但是下面这个就不对了

```
package org.qxh.jdk15;

import java.lang.reflect.Field;

public class RefectPointTest {

    public static void main(String[] args) throws Exception {
        RefectPoint point = new RefectPoint(3,5);
        Class pointClass = point.getClass();
        Field field = pointClass.getField("x");
        System.out.println(field.get(point));
    }

}
```

报错：

```
Exception in thread "main" java.lang.NoSuchFieldException: x
    at java.lang.Class.getField(Class.java:1520)
    at org.qxh.jdk15.RefectPointTest.main(RefectPointTest.java:10)
```

明明有这个字段却提示找不到……

呵呵，看仔细一点就知道了，是 RefectPoint 中声明的时候 x 是 private 的，而 y 是 public 的，当然了，我们还可以用这种方式来做：

```
package org.qxh.jdk15;

import java.lang.reflect.Field;

public class RefectPointTest {

    public static void main(String[] args) throws Exception {
        RefectPoint point = new RefectPoint(3,5);
        Class pointClass = point.getClass();
        //注意getDeclaredField与getField的区别
        Field field = pointClass.getDeclaredField("x");
        //暴力反射
        field.setAccessible(true);
        System.out.println(field.get(point));
    }

}
```

getDeclaredField 与 getField 这两个方法的区别：

getField: 返回一个 Field 对象，它反映此 Class 对象所表示的类或接口的指定公共成员字段

getDeclaredField: 返回一个 Field 对象，该对象反映此 Class 对象所表示的类或接口的指定已声明字段，就是说私有的也可以拿到喽~

下面有这么一个需求，给出一个 User 类：

```
package org.qxh.jdk15;

public class User {

    public String username = "qinxiaohui";
    public String password = "aa";
    public String email = "qinxhuicn@163.com";

    public String toString() {
        return username + ":" + password + ":" + email;
    }
}
```

要求对里边的字段做这么一个操作：如果该字段是 String 类型，那么就把字段中的所有 a 换成 b，想想应该如何实现呢？

```
public void changeStringValue(Object obj) throws Exception{
    Field[] fields = obj.getClass().getFields();
    for(Field field : fields){
        if(field.getType() == String.class){
            String oldValue = (String) field.get(obj);
            String newValue = oldValue.replace('a', 'b');
            field.set(obj, newValue);
        }
    }
}
```

上面的程序只强调一点：类的字节码在内存中只有一份，所以建议采用==来做比较而不是 equals！

Method 类

简介

代表某个类中的一个成员方法

- 得到类中的某一个方法

```
Method charAt = Class.forName("java.lang.String")
                        .getMethod("charAt", int.class);
```

- 调用方法

- 通常方法
String str = "abcdef";
System.out.println(str.charAt(2));

- 反射方法

```
System.out.println(charAt.invoke(str, 2));
```

如果传递给 Method 对象的 invoke() 方法的第一个参数为 null, 这意味着什么呢?

说明该 Method 对象对应的是一个静态方法

反射方式调用 main 方法

上面的程序看上去还是蛮简单的, 接下来的需求是根据用户提供的类名, 去执行该类的 main 方法。用户提供的类名可能是在配置文件中配置的, 也可能是通过启动参数传递过来的, 但都是字符串, 所以只能用反射的方式才能调用

```
package org.qxh.jdk15.reflect;
```

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

```
public class MainMethodReflect {
```

```
    public static void main(String[] args) throws Exception {
        String className = "org.qxh.jdk15.reflect.Test";
        Method mainMethod = Class.forName(className).getMethod("main",
String[].class);
        mainMethod.invoke(null, new String[]{"111", "222"});
    }
}
```

```
class Test{
    public static void main(String[] args) {
        for(String arg : args){
            System.out.println(arg);
        }
    }
}
```

上面这程序看上去没错吧, 我也觉得没错, 不过 eclipse 还是在 mainMethod.invoke 这一行下面给出了 warning 提示, 不理它, 运行之, 报错如下:

```
Exception in thread "main" java.lang.IllegalArgumentException: wrong
number of arguments
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at
sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.
java:39)
    at
sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAcces
sorImpl.java:25)
```

```
at java.lang.reflect.Method.invoke(Method.java:597)
at
org.qxh.jdk15.reflect.MainMethodReflect.main(MainMethodReflect.java:10)
```

提示说参数的个数不对，main 方法需要一个 String 数组我就传过一个数组，为啥还提示说个数不对呢？

这是 JDK1.5 为了兼容 JDK1.4 所产生的问题

我们知道 JDK1.5 加入了可变参数，而 1.4 是没有的，所以 Method.invoke 这个方法在 JDK1.4 中的签名是：Method.invoke(Object obj, Object[] args)；而在 JDK1.5 中的签名是 Method.invoke(Object obj, Object... args)。JDK1.5 为了兼容 1.4，会把参数打散，比如我传递的是 `new String[]{"111", "222"}`，因为 invoke 的第二个参数是 Object 类型的数组，所以 String 数组就被看成是 `new Object[]{"111", "222"}`，而 main 方法本身是接收一个 String 类型的数组的，String[] 是 Object 的子类，所以 main 方法应该接收一个 Object 类型的对象而不是 Object 数组

找到了问题的根源我们就可以找到解决办法了~

➤ 方法一

这样调用：`mainMethod.invoke(null, (Object)new String[]{"111", "222"})`；做一个类型强转

➤ 方法二

这样调用：`mainMethod.invoke(null, new Object[]{new String[]{"111", "222"}})`；也是把 String[] 看做了一个 Object

数组的反射

请看下面程序的输出：

```
package org.qxh.jdk15.reflect;
```

```
public class ArrayReflect {
```

```
    public static void main(String[] args) {
        int[] a1 = new int[3];
        int[] a2 = new int[4];
        int[][] a3 = new int[3][4];
        String[] a4 = new String[3];
        System.out.println(a1.getClass() == a2.getClass());
        //System.out.println(a1.getClass() == a4.getClass());
        //System.out.println(a1.getClass() == a3.getClass());
    }
}
```

上面的程序会输出 true，看 JDK 文档就知道，只要类型一样，维度一样的数组，其字节码就是相同的，程序中的两行注释如果打开注释编译时就会报错，是因为我的 JDK 版本比较高：

A screenshot of a Windows command prompt window. The title bar shows the path 'C:\Windows\system32\cmd.exe'. The window contains the following text: 'Microsoft Windows [版本 6.1.7601]', '版权所有 (c) 2009 Microsoft Corporation。保留所有权利。', 'C:\Users\qinxiaohui01>java -version', 'java version "1.6.0_26"', 'Java(TM) SE Runtime Environment (build 1.6.0_26-b03)', 'Java HotSpot(TM) Client VM (build 20.1-b02, mixed mode, sharing)', and 'C:\Users\qinxiaohui01>'. The window has a scrollbar on the right side.

如果你的版本没有这么高，这两行可能就不报错了，至于从哪个版本开始修正了这个问题我还不知道……如果编译期间不报错，运行时这两行也会输出 false 而不是 true
另外下面这个程序很建议你运行一下：

```
package org.qxh.jdk15.reflect;
```

```
public class ArrayReflect {  
  
    public static void main(String[] args) {  
        int[] a1 = new int[3];  
        System.out.println(a1.getClass().getName());  
    }  
}
```

输出的什么我这就暂时不说了，非常建议读者看一下 getName() 这个方法的 javadoc，这个知识对以后的调试程序看错误信息很有用（个人观点）

小练习

利用数组的反射原理，来做一个工具类，打印对象的内容

```
package org.qxh.jdk15.reflect;
```

```
import java.lang.reflect.Array;
```

```
public class ArrayReflect {  
  
    public static void main(String[] args) {  
        int[] a1 = new int[] { 1, 2, 3 };  
        String str = "qinxiaohui";  
        String[] strArr = { "111", "222" };  
        Integer[] intArr = { 1, 2, 3 };  
        printObject(a1);  
        printObject(str);  
    }  
}
```

```

        printObject(strArr);
        printObject(intArr);
    }

    private static void printObject(Object obj) {
        if (obj.getClass().isArray()) {
            for (int i = 0; i < Array.getLength(obj); i++) {
                System.out.print(Array.get(obj, i) + ",");
            }
            System.out.println();
        } else {
            System.out.println(obj);
        }
    }
}

```

请注意一下 Array 这个类

内省 (Introspector)

内省主要用于对 JavaBean 的操作，JavaBean 是一种特殊的 java 类，其中某些方法符合某种命名规则，如果一个 java 类中的一些方法符合某种命名规则，则可以把它当做 JavaBean 来使用，至于是什么规则，请 google 一下~（其实个人认为没有必要纠结于 JavaBean 这个概念，你就把它看做是个普通的 java 类也无伤大雅）

属性的概念

有些人对属性和成员变量分不清，我这里简单强调一下~，看下面的类：

```
package org.qxh.jdk15.introspector;
```

```

public class Person {

    private int x = 23;

    public int getAge() {
        return x;
    }

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

```

```
}
```

我们称 Person 类中有一个属性叫 age, 没有人说 Person 有个属性叫 x 吧, 属性其实就是 get 和 set 后边的那部分

如果要在两个模块之间传递多个信息, 可以将这些信息封装到一个 JavaBean 中, 这种 JavaBean 的实例对象通常称之为值对象 (Value Object, 简称 VO)。这些信息在类中用私有字段来存储, 如果读取或设置这些字段的值, 则需要通过一些相应的方法来访问, 大家觉得这些方法的名称叫什么好? JavaBean 的属性是根据其中的 getter 和 setter 方法来确定的, 而不是根据其中的成员变量。如果方法名位 setId, 中文意思即为设置 id, 至于你把它存到哪个变量上, 用管吗? 如果方法名位 getId, 中文意思即为获取 id, 至于你把它存到哪个变量上, 用管吗? 去掉 set 前缀, 剩余部分就是属性名, 如果剩余部分的第二个字母是小写的, 则把剩余部分的首字母改成小写的

getId() 的属性名—id

isLast() 的属性名—last

getCPU() 的属性名—CPU

总之, 一个类被当做 JavaBean 来使用时, JavaBean 的属性是根据方法名推断出来的, 咱根本看不到 JavaBean 内部的成员变量

另外一点:

一个符合 JavaBean 特点的类可以当做普通类一样使用, 但把它当做 JavaBean 来用会带来一些额外的好处:

- 在 javaee 开发中, 经常要用到 JavaBean, 很多环境就要求按 JavaBean 方式进行操作, 别人都这么用和要求这么做, 那你就没必要不这么做喽~
- JDK 中提供了对 JavaBean 进行操作的一些 API, 这套 API 就称为内省 API, 用内省 API 来操作 JavaBean 比用普通方式更方便

内省 DEMO

还是上面的这个 Person 类, 现在我要用内省 API 来操作这个类, 代码如下:

简单工具类:

```
package org.qxh.jdk15.introspector;

import java.beans.PropertyDescriptor;
import java.lang.reflect.Method;

public class JavaBeanUtil {

    public static Object getProperty(Object obj, String propertyName) {
        try {
```

```

       PropertyDescriptor pd = new PropertyDescriptor(propertyName,
            obj.getClass());
        Method readMethod = pd.getReadMethod();
        return readMethod.invoke(obj);
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}

public static void setProperty(Object obj, String propertyName, Object
propertyValue) {
    try {
        PropertyDescriptor pd = new PropertyDescriptor(propertyName,
            obj.getClass());
        Method readMethod = pd.getWriteMethod();
        readMethod.invoke(obj, propertyValue);
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}
}

```

小\ DEMO:

```
package org.qxh.jdk15.introspector;
```

```

public class IntrospectorTest {

    public static void main(String[] args) {
        Person person = new Person();
        JavaBeanUtil.setProperty(person, "age", 33);
        System.out.println(JavaBeanUtil.getProperty(person, "age"));
    }
}

```

有人可能会问，我直接 `person.setAge(33);` 设置值，`person.getAge()` 获取值就可以了干嘛要这么麻烦呢？！主要是使用场景的问题，个人认为对象拷贝和做框架的时候用这个用的比较多，以后遇到了就知道了，先记住这种用法

apache 有一个开源类库叫做 BeanUtils，这个工具包可以简化 JavaBean 的操作，大家可以了解一下：

BeanUtils.copyProperties

BeanUtils.getProperty

BeanUtils.setProperty

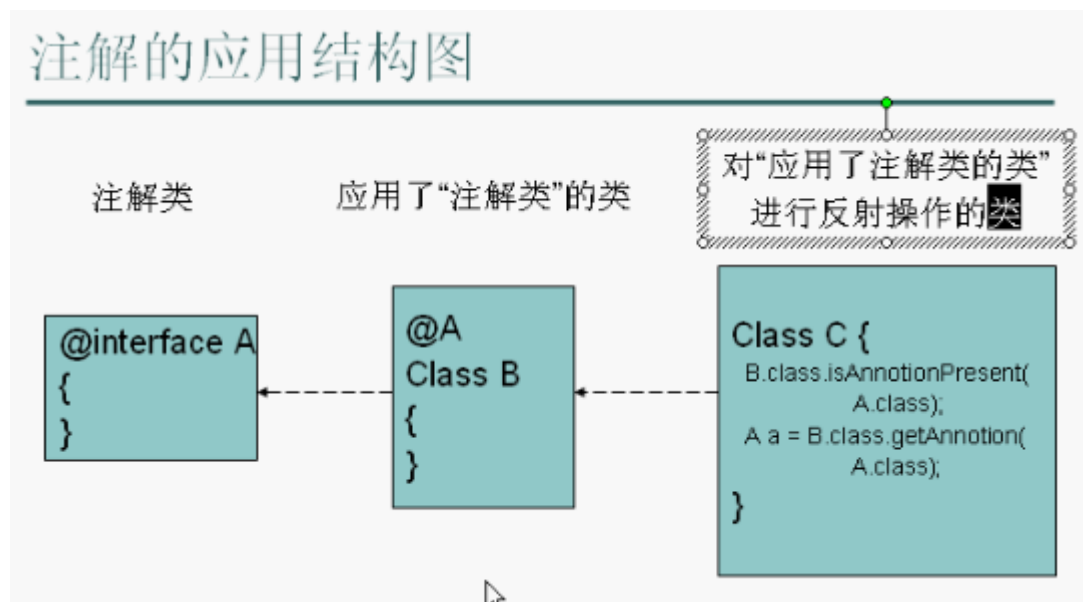
这三个方法灰常常用，一定要掌握

注解 (Annotation)

注解相当于一种标记，在程序中加了注解就相当于给程序打上了某种标记，没加，则相当于没打这种标记，以后，javac 编译器，开发工具和其他程序可以用反射来了解你的类及各种元素上有无任何标记，看你有什么标记，就去干相应的事。标记可以加在包，类，字段，方法，方法的参数以及局部变量上

JDK 的 java.lang 包中提供有三个基本的 Annotation：
@SuppressWarnings, @Override, @Deprecated，这三个注解分别是什么作用大家应该很清楚吧

注解应用结构图



自定义注解类：

```
package org.qxh.jdk15.annotation;
```

```
public @interface MyAnnotation {  
  
}
```

使用注解的类：

```
package org.qxh.jdk15.annotation;
```

```
@MyAnnotation  
public class ClassUseAnnotation {  
  
    public static void main(String[] args) {
```

```

        if(ClassUseAnnotation.class.isAnnotationPresent(MyAnnotation.class)) {
            MyAnnotation myAnnotation =
ClassUseAnnotation.class.getAnnotation(MyAnnotation.class);
            System.out.println(myAnnotation);
        }
    }
}

```

我现在运行上面的 main 方法，但是什么都没有打印出来，原因何在？？

如果我把 MyAnnotation 这个类这么改一下就会有输出：

```

package org.qxh.jdk15.annotation;

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

@Retention(RetentionPolicy.RUNTIME)
public @interface MyAnnotation {

}

```

@Retention(RetentionPolicy.RUNTIME) 这句代码引出了一个新的概念，就是注解的生命周期问题

注解的生命周期分三个阶段：

- 源代码阶段：RetentionPolicy.SOURCE
- CLASS 字节码阶段：RetentionPolicy.CLASS
- 运行时阶段：RetentionPolicy.RUNTIME

我们写的 java 代码经过编译成为.class 文件，最终由类加载器 load 到内存里成为二进制字节码，然后才能运行，注解就是根据这个过程定义的这三个生命周期，默认的 Retention 是 RetentionPolicy.SOURCE，所以第一次运行程序没有输出，直到把 Retention 改为 RetentionPolicy.RUNTIME 之后才看到输出结果，就是这个原因

另外对于 java.lang 包中的三个基本注解

- @SuppressWarnings 的 Retention 为：RetentionPolicy.SOURCE
- @Override 的 Retention 为：RetentionPolicy.SOURCE
- @Deprecated 的 Retention 为：RetentionPolicy.RUNTIME

@Retention 叫做元注解，还有一个元注解是 @Target，表示你定义的注解可以放到哪些成分上面：

```

package org.qxh.jdk15.annotation;

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

```

```

@Retention(RetentionPolicy.CLASS)
@Target({ElementType.TYPE, ElementType.METHOD})
public @interface MyAnnotation {

}

```

上面的@Target 就表示咱自己的枚举 MyAnnotation 可以放到类（或与类平级的 enum、@interface 等）上面和方法上面，之所以取名为 ElementType.*TYPE*，而不是 ElementType.*CLASS*，是因为 JDK1.5 之后出现了一些特殊的类，比如 enum、@interface 等，这些都是和 class 平级的，他们的共同父类是 Type，大家可以看一下 javadoc，查阅一下这几个类和接口的继承关系。

注解的属性

上面讲到的注解只是给大家入个门，注解最有用的是因为它有属性，给注解定义属性的方式如下：

```

package org.qxh.jdk15.annotation;

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

@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE, ElementType.METHOD})
public @interface MyAnnotation {

    String color() default "blue";
    String value();

}

```

属性的定义方式就跟接口里的方法一样，可以有默认值，其中的 value 属性比较特殊，一般我们写属性的时候都是 @MyAnnotation(color="red", value="test")

属性名=属性值，但是我们上面的 color 有默认值了，所以就可以使用默认值，属性名如果是 value 就可以省略 value=这部分而写作：@MyAnnotation("test")

```

package org.qxh.jdk15.annotation;

@MyAnnotation("test")
public class ClassUseAnnotation {

    public static void main(String[] args) {

        if(ClassUseAnnotation.class.isAnnotationPresent(MyAnnotation.class

```

```

s)) {
    MyAnnotation myAnnotation =
ClassUseAnnotation.class.getAnnotation(MyAnnotation.class);
    System.out.println(myAnnotation.color());
    System.out.println(myAnnotation.value());
}
}
}

```

@MyAnnotation("test")这部分我们完全可以写成: @MyAnnotation(color = "red", value = "test") 不过如果你的颜色就打算采用默认值的话就没必要这么写了。另外取得属性值得时候就跟调用方法的方式是一样的:

```

System.out.println(myAnnotation.color());
System.out.println(myAnnotation.value());

```

注解的属性可以有很多种, 比如下面:

```

package org.qxh.jdk15.annotation;

```

```

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

```

```

@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE, ElementType.METHOD})
public @interface MyAnnotation {

```

```

    String color() default "blue";
    String value();
    int[] intArr() default {1,2,3,4};
    Gender enumAttr() default Gender.MALE;
    MetaAnnotation annotationAttr() default @MetaAnnotation();

```

```

    enum Gender{
        MALE, FEMALE
    }

```

```

    public @interface MetaAnnotation{
        String value() default "MetaAnnotation";
    }

```

测试类如下:

```

package org.qxh.jdk15.annotation;

```

```

@MyAnnotation(color = "red", value = "test")
public class ClassUseAnnotation {

```



```

    public static void main(String[] args) {

        if(ClassUseAnnotation.class.isAnnotationPresent(MyAnnotation.class)) {

            MyAnnotation myAnnotation =
ClassUseAnnotation.class.getAnnotation(MyAnnotation.class);
            System.out.println(myAnnotation.color());
            System.out.println(myAnnotation.value());
            System.out.println(myAnnotation.annotationAttr().value());
            System.out.println(myAnnotation.intArr().length);
            System.out.println(myAnnotation.enumAttr().name());

        }

    }
}

```

根据查阅 java 语言规范我们会发现，注解的属性可以允许一下类型：

八种基本类型、String、Class、enum、annotation 以及前面这些类型的数组

注解其实就相当于一种标记，我们自己编写一个注解，然后设置一些属性来传递一些信息，到时候什么地方用到了这个注解我们就可以通过反射拿到这些信息，然后做相应的处理。说白了，注解就是一种传递信息的手段而已（个人观点）

泛型

说实话泛型是 JDK1.5 中相当重要也是相当难的一个新特性，当然了，如果只是应用泛型还算简单，我们就先从简单着手，看看泛型的简单应用

入门

```

package org.qxh.jdk15.generic;

import java.util.ArrayList;
import java.util.List;

public class GenericeCompare {
    public static void main(String[] args) {
        // 没有应用泛型的情况，任何类型都可以往集合里塞，类型不安全
        List oldList = new ArrayList();
        oldList.add(2);
        oldList.add(2L);
    }
}

```

```

oldList.add("String");
// 假如现在想遍历这个集合，取出的每一个元素都只能当成Object
for (Object obj : oldList) {
    // 不能把这个obj强制转换为Integer|Long|String
    System.out.println(obj);
}
// 采用了泛型的情况，为List集合规定了一种可以存放的类型，其他
// 类型都不能存放进去，我们称这种集合是类型安全的
List<Integer> genericList = new ArrayList<Integer>();
genericList.add(2);
genericList.add(3);
// 由于集合是类型安全的，那我们就可以使用参数类型特有的方法来操作了
int sum = 0;
for (Integer intElement : genericList) {
    sum += intElement;
}
System.out.println("sum = " + sum);
}
}

```

上面就是对泛型的简单应用，为了让编译器帮助我们来检查装进集合中的元素的类型，我们就可以提前通过泛型参数来告诉编译器，这样就避免了一些类型安全问题

泛型擦除

上面刚说可以避免类型安全问题，现在就要颠覆这个观点。我们姑且称泛型的这个特点为泛型擦除。

是这样，泛型是给编译器看的，当程序编译的时候，为了性能考虑，会把泛型信息擦除，运行的时候 JVM 实际是不知道泛型这个概念的，看下面的例子

```

package org.qxh.jdk15.generic;

import java.util.ArrayList;
import java.util.List;

public class GenericTest {

    public static void main(String[] args) throws Exception {
        List<String> strList = new ArrayList<String>();
        strList.add("abc");
        List<Integer> intList = new ArrayList<Integer>();
        intList.add(3);
        //下面的语句输出true，说明不管是String类型的List
        //还是Integer类型的List，都对应了同一份字节码，也就是说
    }
}

```

```

//泛型信息被擦除了
System.out.println(strList.getClass()==intList.getClass());
//接下来我们通过反射的方式往Integer类型的List中存放进String类型的数据
intList.getClass().getMethod("add",
Object.class).invoke(intList, "xyz");
//下面的语句会输出xyz，说明字符串"xyz"确实被加进了集合里
System.out.println(intList.get(1));
}

}

```

上面的注释已经很清楚了，我这就不再罗嗦了

泛型术语

ArrayList<E>类定义和 ArrayList<Integer>类引用中涉及如下术语：

- ArrayList<E>整个称为泛型类型
- ArrayList<E>中的 E 称为类型变量和类型参数
- ArrayList<Integer>整个称为参数化的类型，参数化的类型在 java 中有一个专门的类来表示：ParameterizedType，这个类你应该在 BaseDao 这种类里看到过吧，呵呵
- ArrayList<Integer>中的 Integer 称为类型参数的实例或实际类型参数
- ArrayList<Integer>中的<>念作 typeof
- ArrayList 称为原始类型

泛型进阶

- 参数化类型与原始类型的兼容性
 - 参数化类型可以引用一个原始类型的对象，编译报告警告，例如：


```
Collection<String> c = new Vector();
```
 - 原始类型可以引用一个参数化类型的对象，编译报告警告，例如：


```
Collection c = new Vector<String>();
```
- 参数化类型不考虑类型参数的继承关系
 - Vector<String> v = new Vector<Object>();//错误，不写<Object>可以，但是你写了就是明知故犯
 - Vector<Object> v = new Vector<String>();//也错误，不要以为 Object 是 String 的父类就可以这么写，参数化类型不考虑类型参数的继承关系
- ? 通配符

现在有这么一个需求：打印泛型集合中的所有元素

方法签名你打算如何设计？

```
public void printCollection(Collection<String> collection)?
public void printCollection(Collection<Integer> collection)?
public void printCollection(Collection<Object> collection)?
```

上面的方法显然都是不可以的，这时我们就需要？通配符了

```
public static void printCollection(Collection<?> collection){
    for(Object obj : collection){
        System.out.println(obj);
    }
}
```

上面的？就表示可以接收任何泛型参数类型，但是注意一点，我们在 printCollection 方法里是不能调用跟参数化类型有关的方法的，比如：collection.add("String");就会报错滴。因为 java 编译器此时闹不清你的？代表什么类型，所以不能随便往集合中加入元素。

- 限定通配符的上边界
 - 正确：Vector<? extends Number> v = new Vector<Integer>();
 - 错误：Vector<? extends Number> v = new Vector<String>();
- 限定通配符的下边界
 - 正确：Vector<? super Integer> v = new Vector<Number>();
 - 错误：Vector<? super Integer> v = new Vector<Byte>();
- 限定通配符总是包括自己

泛型 DEMO

下面给出一个泛型中套泛型的小例子，如果把这个看明白了，应用别人写好的泛型应该就不成问题了：

```
package org.qxh.jdk15.generic;

import java.util.HashMap;
import java.util.Map;
import java.util.Set;

public class MapEntryTest {

    public static void main(String[] args) {
        Map<String, Integer> ageMap = new HashMap<String, Integer>();
        ageMap.put("qinxiaohui", 23);
        ageMap.put("morflame", 20);
        Set<Map.Entry<String, Integer>> entrySet = ageMap.entrySet();
        for (Map.Entry<String, Integer> entry : entrySet){
            System.out.println(entry.getKey()+"："+entry.getValue());
        }
    }
}
```

```

    }
}
}

```

其实此处的最佳实践是 `entrySet` 这个局部变量直接省略，`for` 循环的：后面直接写 `ageMap.entrySet()` 即可，之所以这么写主要是为了让大家更深入的了解泛型

自定义泛型方法

上面是在应用别人定义好的泛型方法，接下来我们要自己定义泛型方法让别人来使用
先看摘抄的一句话：

用过 C++ 的人都知道泛型模板的作用，java 中的泛型跟这个类似，但是这种相似性仅限于表面，Java 语言中的泛型基本上完全是在编译器中实现，用于编译器执行类型检查和类型推断，然后生成普通的非泛型字节码，这就是上面提到的泛型擦除。Java 中泛型没有 C++ 的牛掰，主要原因是扩展虚拟机指令集来支持泛型被认为是无法接受的，这会让 Java 厂商升级其 JVM 造成难以逾越的障碍。所以，java 的泛型采用了可以完全在编译器中实现的擦除方法。

虽说没有 C++ 的泛型模板那么好用，Java 还是在努力去模仿，下面就让我们一起来模仿一下：

需求：定义一个方法，可以交换任意类型数组中的两个位置的元素

```

package org.qxh.jdk15.generic;

import java.util.Arrays;

public class GenericSwap {

    public static void main(String[] args) {
        String[] arr = { "abc", "xyz", "qinxiaohui" };
        swap(arr, 1, 2);
        System.out.println(Arrays.asList(arr));
    }

    //定义泛型方法时，泛型参数<T>要在方法返回值之前声明
    static <T> void swap(T[] arr, int i, int j) {
        T tmp = arr[i];
        arr[i] = arr[j];
        arr[j] = tmp;
    }
}

```

注意：

只有引用类型才能作为泛型方法的实际参数，上面的 `swap` 方法如果这样调用就会报错：

```
swap(new int[] {1, 2, 3}, 1, 2);
```

另外:

- 除了在应用泛型时可以使用 extends 限定符,在定义泛型时也可以使用 extends 限定符,例如: Class.getAnnotation() 方法的定义。并且可以用&来指定多个边界,如:

```
<V extends Serializable&Cloneable> void method() {}
```

- 普通方法、静态方法、构造方法中都可以使用泛型
- 也可以用类型变量表示异常,称为参数化的异常,可以用于方法的 throws 列表中,但是不能用在 catch 子句中

```
private static <T extends Exception> void sayHello() throws T {  
    try {  
  
    } catch (Exception e) {  
        throw (T) e;  
    }  
}
```

- 在泛型中可以有多多个类型参数,在定义它们的尖括号内用逗号分开,例如:

```
public static <K, V> V getValue(K key) {return map.get(key);}
```

还有一个问题就是泛型中类型参数的类型推断,这个比较复杂,真正遇到复杂情况的时候也不多,暂且略过.....

泛型类型

泛型类型就是把类型参数放到类级别上,使用场景:一个类中很多地方都要用到相同的类型参数,放在方法级别上没有问题,但是麻烦。一个典型应用就是 BaseDao 的设计:

```
package org.qxh.jdk15.generic;  
  
import java.io.Serializable;  
  
public interface BaseDao<E> {  
  
    public void persist(E entity);  
  
    public void update(E entity);  
  
    public void delete(E entity);  
  
    public E findById(Serializable id);  
}
```

如果以前接触过这种类但是没有看得很明白,现在就可以回去仔细看一下了,如果你还没有

接触过这种类，不用担心，很快就会接触到了~

泛型的实际类型参数

有的时候我们会有这么一种需求，`Vector<Date> v = new Vector<Date>()`；我们希望通过反射获取 `Vector` 中存放的实际类型，想想这个该如何解决呢？

第一反应可能是 `v.getClass()` 得到 `v` 的字节码，然后通过这个字节码去获取实际类型参数，但是这个方法行不通.....

曲径通幽，我们换一个角度考虑问题，先上代码：

```
package org.qxh.jdk15.generic;

import java.lang.reflect.Method;
import java.lang.reflect.ParameterizedType;
import java.lang.reflect.Type;
import java.util.Date;
import java.util.Vector;

public class RealGenericParam {

    public static void main(String[] args) throws Exception {
        Method applyMethod =
RealGenericParam.class.getMethod("applyVector", Vector.class);
        Type[] types = applyMethod.getGenericParameterTypes();
        ParameterizedType pType = (ParameterizedType) types[0];
        Type realType = pType.getActualTypeArguments()[0];
        System.out.println(realType);
        System.out.println(realType == java.util.Date.class);
    }

    public static void applyVector(Vector<Date> v) {

    }
}
```

这里是把这个泛型化参数放到了方法参数位置，然后利用反射得到这个方法的信息，进而挖掘出实际类型参数，绕了一圈，总算解决了。另外，如果 `Vector<Date> v` 是作为类的成员变量的话也可以通过反射拿到实际类型参数，大家可以尝试一把~

类加载器

- 类加载器就是加载类的工具，要运行一个类，首先要把硬盘上的.class 文件 load 到内存里，这个工作就是由类加载器来做的
- JVM 中可以安装多个类加载器，系统默认三个主要类加载器，每个类负责加载特定位置的类：
Bootstrap、ExtClassLoader、AppClassLoader
- 类加载器也是 java 类，因为类加载器本身也要由类加载器来加载，那么 java 类的第一个妈妈是谁呢？显然第一个类加载器不是 java 类，这正是 Bootstrap，它是用 C++写成的，当 JVM 内核启动时随之启动
- JVM 中所有类加载器采用具有父子关系的树形结构进行组织，在实例化每个类装载器对象时，需要为其指定一个父级类加载器对象或是默认采用系统类加载器为其父级类加载器

入门演示 DEMO

```
package org.qxh.classloader;

import org.junit.Test;

public class ClassLoaderTest {

    @Test
    public void test1() {
        //打印出: sun.misc.Launcher$AppClassLoader
        System.out.println(ClassLoaderTest.class.getClassLoader()
            .getClass().getName());
        //打印出null，说明System这个的类加载器是Bootstrap
        System.out.println(System.class.getClassLoader());
    }
}
```

下面我们查看系统默认提供的三个类加载器的父子关系：

```
@Test
public void test2() {
    ClassLoader loader = ClassLoaderTest.class.getClassLoader();
    while(loader!=null){
        System.out.println(loader.getClass().getName());
        loader = loader.getParent();
    }
}
```

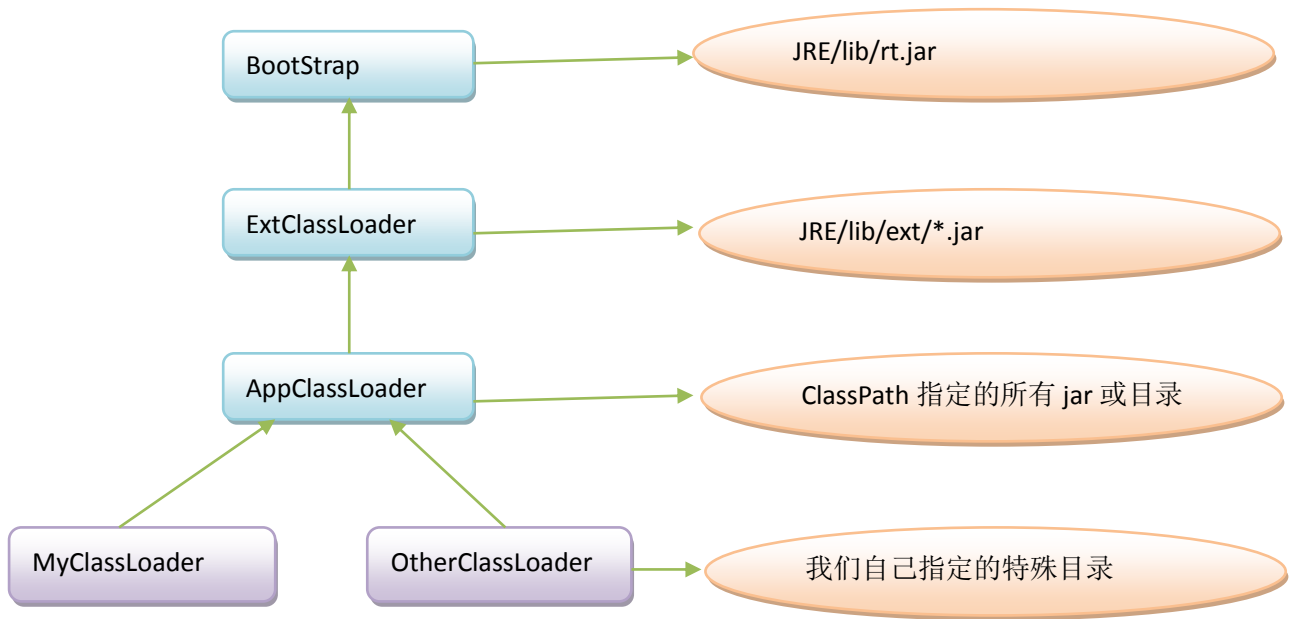

上面的程序打印内容如下：

```
sun.misc.Launcher$AppClassLoader
```

```
sun.misc.Launcher$ExtClassLoader
```

说明 ExtClassLoader 是 AppClassLoader 的父类，最终的 loader 变为 null，说明 ExtClassLoader 的父类是 Bootstrap

类加载器之间的父子关系和管辖范围图



类加载器的委托机制

- 当 Java 虚拟机要加载一个类时，到底派出哪个类加载器去加载呢？
 - 首先是当前线程的类加载器去加载线程中的第一个类
 - 如果类 A 引用了类 B，JVM 将使用加载类 A 的类装载器去加载类 B
 - 还可以直接调用 `ClassLoader.loadClass()` 方法来指定某个类加载器去加载某个类
- 每个类加载器加载类时，又先委托给其上级类加载器
 - 当所有祖宗类加载器没有加载到类，回到发起者类加载器，还加载不了，则抛 `ClassNotFoundException`，不是再去找发起者类加载器的儿子，因为没有 `getChildren()` 方法，即使有，那么多个儿子，找哪一个呢？

思考：可否自己写一个 `java.lang.System` 类来覆盖 JDK 提供的那个 `System` 类？

通常这么做是不可以的，由于类加载器的父类委托机制，Bootstrap 这个类会去加载 JDK 提供的 `java.lang.System` 这个类，而不会加载程序员自己写的 `System` 类，不过我们可以编写自己的类加载器来加载自己的 `System` 类~

编写自己的类加载器

编写自己的类加载器要继承自 `ClassLoader` 抽象类，但是不用覆盖 `ClassLoader` 中的 `loadClass` 方法，由于 `ClassLoader` 的编写采用了模板设计模式，子类自己应该去做的事情都可以通过覆盖父类的 `findClass` 方法来实现，也就是说 `ClassLoader` 的 `loadClass` 方法应该大体是这样的：

```
protected Class<?> loadClass(String name, boolean resolve)
    throws ClassNotFoundException{
    boolean loadOkByParent = 让父类去加载;
    if(loadOkByParent){
        //父类成功加载咱就不需要做什么了
    } else {
        //父类没能加载成功，自己来加载
        findClass();
    }
}
```

就是说 `loadClass` 中有一些父类委托的代码，因为所有的类加载器的这部分代码都是一样的，所以就做成了模板，模板方法中调用了 `findClass()` 方法，这个方法是子类真正应该做的逻辑处理。

下面的例子是从 JDK 帮助文档中扒过来的，希望对大家有帮助：

有些类可能并非源自一个文件；它们可能源自其他来源（如网络），也可能是由应用程序构造的。`defineClass` 方法将一个 `byte` 数组转换为 `Class` 类的实例。这种新定义的类的实例可以使用 `Class.newInstance` 来创建。

类加载器所创建对象的方法和构造方法可以引用其他类。为了确定引用的类，Java 虚拟机将调用最初创建该类的类加载器的 `loadClass` 方法。

例如，应用程序可以创建一个网络类加载器，从服务器中下载类文件。示例代码如下所示：

```
ClassLoader loader = new NetworkClassLoader(host, port);
Object main = loader.loadClass("Main", true).newInstance();
...
```

网络类加载器子类必须定义方法 `findClass` 和 `loadClassData`，以实现从网络加载类。下载组成该类的字节后，它应该使用方法 `defineClass` 来创建类实例。示例实现如下：

```
class NetworkClassLoader extends ClassLoader {
```

```
String host;
```

```
int port;
```

```
public Class findClass(String name) {
```

```
    byte[] b = loadClassData(name);
```

```
    return defineClass(name, b, 0, b.length);
```

```
}
```

```
private byte[] loadClassData(String name) {
```

```
    // load the class data from the connection
```

```
    . . .
```

```
}
```

```
}
```

tomcat 中的类加载机制

接下来这个问题比较有意思，稍微有一点深度，慢慢来

tomcat 是我们常用的 WEB 容器 (Jsp、servlet 容器)，tomcat 自己定义了一些类加载器，运行在 tomcat 中的 Servlet 是如何加载的呢？我们写一个 servlet 测试一下.....

Servlet 类名为: TestServlet, url-pattern 为: /servlet/TestServlet, 源码为:

```
package org.qxh.classloadweb.servlets;
```

```
import java.io.IOException;
```

```
import java.io.PrintWriter;
```

```
import javax.servlet.ServletException;
```

```
import javax.servlet.http.HttpServlet;
```

```
import javax.servlet.http.HttpServletRequest;
```

```
import javax.servlet.http.HttpServletResponse;
```

```
public class TestServlet extends HttpServlet {
```

```
    public void doGet(HttpServletRequest request, HttpServletResponse  
response)
```

```
        throws ServletException, IOException {
```

```
        response.setContentType("text/html");
```

```
        PrintWriter out = response.getWriter();
```

```
        ClassLoader loader = this.getClass().getClassLoader();
```

```
        while (loader!=null) {
```

```
            out.println(loader.getClass().getName()+"<br>");
```

```
        loader = loader.getParent();
    }
    out.close();
}
}
```

访问: <http://localhost/servlet/TestServlet>

输出内容为:

org.apache.catalina.loader.WebappClassLoader

org.apache.catalina.loader.StandardClassLoader

sun.misc.Launcher\$AppClassLoader

sun.misc.Launcher\$ExtClassLoader

说明这个 Servlet 是由 apache 自己写的 WebappClassLoader 这个类加载器来加载的, 它的父类是 StandardClassLoader, StandardClassLoader 的父类才是 AppClassLoader

OK, 接下来玩点花样:

把咱刚才编写的 TestServlet 这个类打成 jar 包, 放到 ext 目录中, 重启 tomcat, 访问原来的地址, 结果如何?

报找不到 HttpServlet 的错误!

什么原因?

上面有句话不知道你记得吧:

如果类 A 引用了类 B, JVM 将使用加载类 A 的类装载器去加载类 B

由于父类委托机制, 我们把 TestServlet 打成的 jar 包放在 ext 目录下之后, ExtClassLoader 类加载器就可以找到 TestServlet 这个类了, 由于 TestServlet 是继承自 HttpServlet 的, 所以 JVM 将使用加载 TestServlet 的类装载器去加载 HttpServlet, 而 HttpServlet 所在的 jar 包并没有在 ext 目录下, 所以 ExtClassLoader 找不到这个类, 所以就会报错

接下来我们找到 tomcat 的 lib 目录中的 servlet-api.jar, 然后把它放到 ext 目录中, 重启 tomcat, 再访问刚才的 URL, 问题解决了, 打印输出:

sun.misc.Launcher\$ExtClassLoader

大家看明白了吗?

总结一下, 刚开始的情况是默认情况, TestServlet 和 HttpServlet 这两个类都是由 tomcat 提供的 WebappClassLoader 来加载的, 后来改动之后正常了, 这两个类又都是由 ExtClassLoader 这个类来加载的。这些操作无非就是想说明白三件事:

- 1) 不同的类加载器加载不同的类
- 2) 类加载器的父类委托机制
- 3) 如果类 A 引用了类 B, JVM 将使用加载类 A 的类装载器去加载类 B

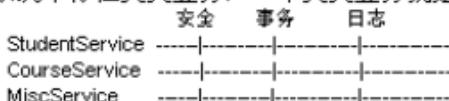

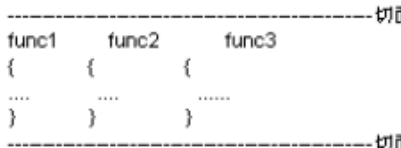
代理

接下来步入一个新的主题：代理。很重要，静下心来慢慢来~

代理的概念和作用

- 生活中的代理
比如现在我要买一台联想的本子，一种方式是我跑到北京联想总部来买，第二种方式是从代理商那里买，最终结果都是买到一台 lenovo 笔记本。但是这两种方式是存在不同的，不同点就是中间的代理
- 程序中的代理
 - 要为己存在的多个具有相同接口的目标类的各个方法增加一些系统功能, 比如异常处理, 日志, 记录运行时间, 事务管理等等, 你准备如何做?
 - 编写一个与目标类具有相同接口的代理类, 代理类的每个方法调用目标类的相同方法, 并在调用方法时加上系统功能的代码
 - 如果采用工厂方法和配置文件的方式进行管理, 则不需要修改客户端程序, 在配置文件中配置是使用目标类还是代理类, 这样以后很容易切换, 譬如, 想要日志功能时就配置代理类, 否则配置目标类, 这样, 增加系统功能很容易, 以后运行一段时间后, 又想去掉系统功能也很容易。当前喽, 前提是要针对接口编程哦~

AOP

- 系统中存在交叉业务, 一个交叉业务就是要切入到系统中的一个方面, 如下所示:

- 用具体的程序代码描述交叉业务:

- 交叉业务的编程问题即为面向方面的编程 (Aspect oriented program, 简称AOP), AOP的目标就是要使交叉业务模块化。可以采用将切面代码移动到原始方法的周围, 这与直接在方法中编写切面代码的运行效果是一样的, 如下所示:

- 使用代理技术正好可以解决这种问题, 代理是实现AOP功能的核心和关键技术。

动态代理

- 要位系统中的各种接口的类增加代理功能，那将需要太多的代理类，全部采用静态代理方式，将是一件非常麻烦的事情！写成百上千个代理确实太累了
- JVM 可以在运行期动态生成出类的字节码，这种动态生成的类往往被用作代理类，即动态代理类
- JVM 生成的动态类必须实现一个或多个接口，所以，JVM 生成的动态类，只能用作具有相同接口的目标类的代理
- 大名鼎鼎的 CGLIB 库可以动态生成一个类的子类，一个类的子类也可以用作该类的代理，所以，如果要为一个没有实现接口的类生成动态代理类，那么可以使用 CGLIB 库
- 代理类的各个方法中通常除了要调用目标类的相应方法和对外返回目标返回的结果外，还可以在代理方法中的如下四个位置加上系统功能代码：
 - 在调用目标方法之前
 - 在调用目标方法之后
 - 在调用目标方法前后
 - 在处理目标方法异常的 catch 块中

创建动态类及查看其方法列表信息

接下来我们打算做一个代理出来，这个代理实现 Collection 接口，代码不多，不过还是尽量写的质量高一点，抽取一个工具类出来：

```
package org.qxh.proxy;

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

public class ClassUtil {
    /**
     * 打印某个Class的所有公有方法签名
     * @param clazz
     */
    public static void printMethods(Class clazz) {
        Method[] methods = clazz.getMethods();
        for (Method method : methods) {
            String name = method.getName();
            StringBuilder sBuilder = new StringBuilder(name);
            sBuilder.append('(');
            Class[] clazzParams = method.getParameterTypes();
            for (Class clazzParam : clazzParams) {
                sBuilder.append(clazzParam.getName()).append(',');
            }
        }
    }
}
```

```

    }
    if (clazzParams != null && clazzParams.length != 0)
        sBuilder.deleteCharAt(sBuilder.length() - 1);
    sBuilder.append(' ');
    System.out.println(sBuilder.toString());
}

/**
 * 打印某个Class的所有构造函数签名
 * @param clazz
 */
public static void printConstructors(Class clazz) {
    Constructor[] constructors = clazz.getConstructors();
    for (Constructor constructor : constructors) {
        String name = constructor.getName();
        StringBuilder sBuilder = new StringBuilder(name);
        sBuilder.append('(');
        Class[] clazzParams = constructor.getParameterTypes();
        for (Class clazzParam : clazzParams) {
            sBuilder.append(clazzParam.getName()).append(',');
        }
        if (clazzParams != null && clazzParams.length != 0)
            sBuilder.deleteCharAt(sBuilder.length() - 1);
        sBuilder.append(')');
        System.out.println(sBuilder.toString());
    }
}

```

接下来做实现 Collection 接口的代理

```

package org.qxh.proxy;

import java.lang.reflect.Proxy;
import java.util.Collection;

public class ProxyTest {
    public static void main(String[] args) throws Exception {
        //创建代理类可以采用Proxy, 我们做一个实现了Collection接口的代理
        Class clazzProxy = Proxy.getProxyClass(Collection.class,
            getClassLoader(), Collection.class);
        System.out.println(clazzProxy.getName());

        System.out.println("-----构造函数列表-----");
        ClassUtil.printConstructors(clazzProxy);
    }
}

```

```

        System.out.println("\n-----方法列表-----");
        ClassUtil.printMethods(clazzProxy);
    }
}

```

打印内容如下:

\$Proxy0

-----构造函数列表-----

\$Proxy0 (java.lang.reflect.InvocationHandler)

-----方法列表-----

add (java.lang.Object)

equals (java.lang.Object)

toString()

hashCode()

clear()

contains (java.lang.Object)

isEmpty()

addAll (java.util.Collection)

iterator()

size()

toArray ([Ljava.lang.Object;)

toArray()

remove (java.lang.Object)

containsAll (java.util.Collection)

removeAll (java.util.Collection)

retainAll (java.util.Collection)

isProxyClass (java.lang.Class)

getProxyClass (java.lang.ClassLoader, [Ljava.lang.Class;)

getInvocationHandler (java.lang.Object)

newProxyInstance (java.lang.ClassLoader, [Ljava.lang.Class;, java.lang.reflect.InvocationHandler)

wait (long)

wait()

wait (long, int)

getClass()

notify()

notifyAll()

类名怪怪的, 暂且不去管它, 构造函数接收的类型是 `InvocationHandler`, 后边会有应用, 主要看方法列表, 是不是包含了所有 `Collection` 接口的方法?

接下来我们利用上面拿到的字节码 `clazzProxy` 来 `new` 一个实现了 `Collection` 接口的对象出来:


```

package org.qxh.proxy;

import java.lang.reflect.Constructor;
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;
import java.util.Collection;

public class ProxyTest {
    public static void main(String[] args) throws Exception {
        Class clazzProxy = Proxy.getProxyClass(Collection.class
            .getClassLoader(), Collection.class);
        // 这样new对象是不行的，因为通过打印clazzProxy的构造函数我们已经发现
        // 它没有无参构造函数
        // clazzProxy.newInstance();
        Constructor constructor = clazzProxy
            .getConstructor(InvocationHandler.class);
        Collection collection = (Collection) constructor
            .newInstance(new InvocationHandler() {

                @Override
                public Object invoke(Object proxy, Method method,
                    Object[] args) throws Throwable {
                    System.out.println("invoke()");
                    return null;
                }
            });
        System.out.println(collection);
    }
}

```

咦？看上去程度没错，怎么打印的是 null ？

不要着急，如果你这样打印就知道原因了：

```
System.out.println(collection.toString());
```

还是输出 null，说明 collection 不为 null，因为它可以调用 toString 方法，第一次打印 null 完全是因为 toString 方法返回 null！

完成 InvocationHandler 对象的内部功能

上面的创建代理的方式还是有点复杂，而且 InvocationHandler 内部啥也没做，接下来我们会采用一种更常用也更方便的方式来创建动态代理，同时还会在 InvocationHandler 内部做一点处理：

```

package org.qxh.proxy;

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;
import java.util.ArrayList;
import java.util.Collection;
import java.util.List;

public class InvocationHandlerTest {

    public static void main(String[] args) {
        Collection collection = (Collection) Proxy.newProxyInstance(
            Collection.class.getClassLoader(),
            new Class[] { Collection.class }, new InvocationHandler()
        {

            List targetList = new ArrayList();

            public Object invoke(Object proxy, Method method,
                                Object[] args) throws Throwable {
                long beginTime = System.currentTimeMillis();
                Object retVal = method.invoke(targetList, args);
                Thread.sleep(10);
                long endTime = System.currentTimeMillis();
                System.out.println(method.getName()+"运行时间为:
"+(endTime-beginTime));
                return retVal;
            }

        });

        collection.add("123");
        collection.add("456");
        collection.add("789");
        System.out.println(collection.size());
    }
}

```

输出为:

```

add运行时间为: 10
add运行时间为: 10
add运行时间为: 10
size运行时间为: 13
3

```

说明对 collection 的每个方法的调用都是会经过 InvocationHandler.invoke 的处理, 所以在调用目标之前 method.invoke(targetList, args);, 就可以加入一些系统功能代码, 比如上面是计算运行时间的。由于现在的 CPU 太快, 为了看出效果加了 Thread.sleep(10); 这句

动态代理原理浅析

动态代理的运行原理应该也是比较简单的

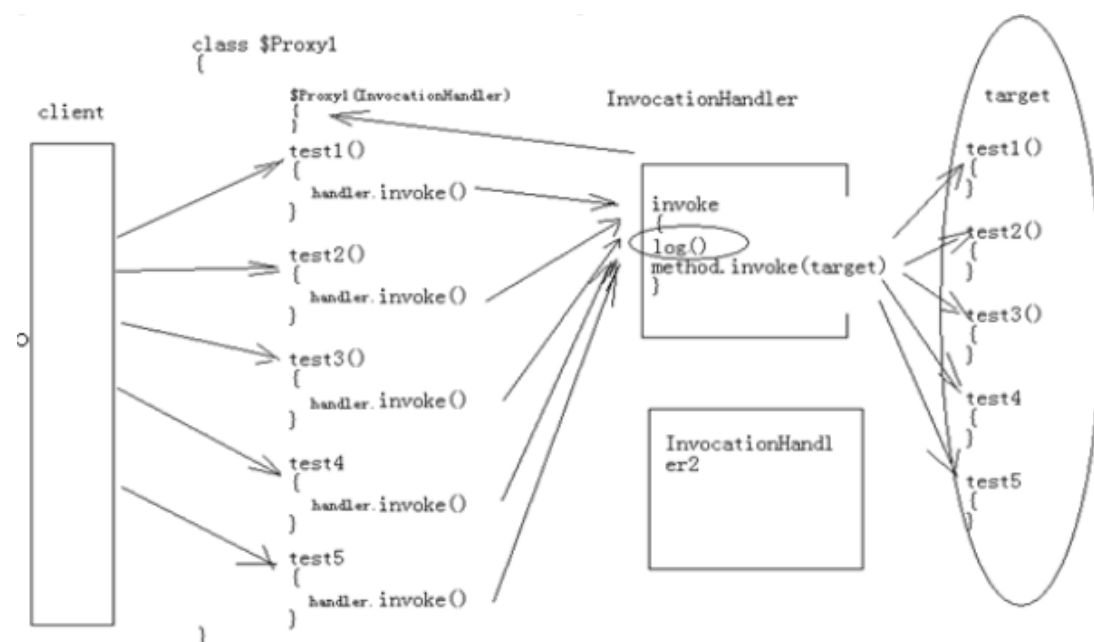
首先，在构造方法中传入的是 `InvocationHandler` 对象，Proxy 内部肯定有个成员变量来记住传入的值。

然后客户代码调用 `size` 方法或是 `clear` 方法的时候，代理类的内部应该是类似这样的代码：

```
int size() {  
    return this.handle.invoke(this, this.getClass().getMethod("size"), null);  
}  
  
void clear() {  
    this.handle.invoke(this, this.getClass().getMethod("clear"), null);  
}
```

这样的代码是很有规律性的，有规律的代码当然就可以用程序来动态生成，所以动态代理就产生了~

下面是原理图：



上面的 `InvocationHandler` 的内部代码都是硬编码的，这样显然是没有太大用处的，我们希望复用！

最终是想做成：被代理的对象和系统功能代码都是可以由用户来指定的

第一步重构：

把获取代理对象的代码抽取为一个方法：

```
package org.qxh.proxy;
```

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

```

import java.lang.reflect.Method;
import java.lang.reflect.Proxy;
import java.util.ArrayList;
import java.util.Collection;
import java.util.List;

public class InvocationHandlerTest {

    public static void main(String[] args) {
        List targetList = new ArrayList();
        Collection collection = (Collection) getProxy(targetList);
        collection.add("123");
        collection.add("456");
        collection.add("789");
        System.out.println(collection.size());
    }

    private static Object getProxy(final Object target) {
        return Proxy.newProxyInstance(
            target.getClass().getClassLoader(),
            target.getClass().getInterfaces(),
            new InvocationHandler() {
                public Object invoke(Object proxy, Method method,
                    Object[] args) throws Throwable {
                    long beginTime = System.currentTimeMillis();
                    Object retVal = method.invoke(target, args);
                    Thread.sleep(10);
                    long endTime = System.currentTimeMillis();
                    System.out.println(method.getName() + "运行时间为:
"
                        + (endTime - beginTime));
                    return retVal;
                }
            });
    }
}

```

第二步重构:

把系统功能代码抽取出来

要抽取这部分代码我们需要做一个契约, 也就是一个接口, 到时候我们的代码就会调用接口中的固定方法, 而用户要做的就是写接口的实现类

契约接口:

```

package org.qxh.proxy;

import java.lang.reflect.Method;

```

```

public interface Advice {
    //调用target之前执行的契约方法
    void beforeMethod(Method method);
    //调用target之后执行的契约方法
    void afterMethod(Method method);
}

```

然后用户要做的就是写自己的对 Advice 的实现类：

```

package org.qxh.proxy;

import java.lang.reflect.Method;

public class MyAdvice implements Advice {

    long beginTime;

    public void afterMethod(Method method) {
        long endTime = System.currentTimeMillis();
        System.out.println(method.getName() + "运行时间为: "
            + (endTime - beginTime));
    }

    public void beforeMethod(Method method) {
        beginTime = System.currentTimeMillis();
    }

}

```

最后就是更改 InvocationHandler 内部实现：

```

package org.qxh.proxy;

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;
import java.util.ArrayList;
import java.util.Collection;
import java.util.List;

public class InvocationHandlerTest {

    public static void main(String[] args) {
        List targetList = new ArrayList();
        Collection collection = (Collection) getProxy(targetList, new
MyAdvice());
        collection.add("123");
    }
}

```

```

        collection.add("456");
        collection.add("789");
        System.out.println(collection.size());
    }

    private static Object getProxy(final Object target, final Advice
advice) {
        return Proxy.newProxyInstance(
            target.getClass().getClassLoader(),
            target.getClass().getInterfaces(),
            new InvocationHandler() {
                public Object invoke(Object proxy, Method method,
                    Object[] args) throws Throwable {
                    advice.beforeMethod(method);
                    Object retVal = method.invoke(target, args);
                    Thread.sleep(10);
                    advice.afterMethod(method);
                    return retVal;
                }
            });
    }
}

```

第三步重构，就是把 getProxy 方法放到一个单独的类中，呵呵，这么简单的重构方式也好意思拿出来，其实把方法移动一下位置真的是一个很重要也很常用的重构方法，这样做算是给大家提个醒：

```

package org.qxh.proxy;

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

public class ProxyUtil {
    public static Object getProxy(final Object target, final Advice advice)
    {
        return
Proxy.newProxyInstance(target.getClass().getClassLoader(),
            target.getClass().getInterfaces(), new
InvocationHandler() {
                public Object invoke(Object proxy, Method method,
                    Object[] args) throws Throwable {
                    advice.beforeMethod(method);
                    Object retVal = method.invoke(target, args);
                    advice.afterMethod(method);
                    return retVal;
                }
            });
    }
}

```

```

        }
    });
}
}

```

当然了，此处 Advice 的方法参数仅仅有 Method 其实是不够的，不过这里仅仅是说明这种用法而已，大家了解思想就好

实现一个简单的 Spring

我们最后要做的是一个类似 Spring 的小框架

- 首先有个工厂类 BeanFactory，负责创建目标类和代理类的实例对象，并通过配置文件进行切换。其 getBean 方法参数字符串返回一个相应的实例对象，如果参数字符串在配置文件中对应的类名不是 ProxyFactoryBean，则直接返回该类的实例对象，否则，返回该类实例对象的 getProxy() 方法返回的对象。
- BeanFactory 的构造方法接收代表配置文件的输入流对象，配置文件格式如下：

```

#xxx=java.util.ArrayList
xxx=org.qxh.proxy.ProxyFactoryBean
xxx.advice=org.qxh.proxy.MyAdvice
xxx.target=java.util.ArrayList

```

BeanFactory 的代码：

```

package org.qxh.proxy;

import java.io.IOException;
import java.io.InputStream;
import java.util.Properties;

public class BeanFactory {
    Properties props = new Properties();
    public BeanFactory(InputStream ips) {
        try {
            props.load(ips);
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }

    public Object getBean(String name) {
        String className = props.getProperty(name);
        Object bean = null;
        try {

```

```

        Class clazz = Class.forName(className);
        bean = clazz.newInstance();
    } catch (Exception e) {
        e.printStackTrace();
    }

    if(bean instanceof ProxyFactoryBean){
        Object proxy = null;
        ProxyFactoryBean proxyFactoryBean = (ProxyFactoryBean)bean;
        try {
            Advice advice =
(Advice)Class.forName(props.getProperty(name +
".advice")).newInstance();
            Object target = Class.forName(props.getProperty(name +
".target")).newInstance();
            proxyFactoryBean.setAdvice(advice);
            proxyFactoryBean.setTarget(target);
            proxy = proxyFactoryBean.getProxy();
        } catch (Exception e) {
            e.printStackTrace();
        }
        return proxy;
    }
    return bean;
}
}

```

ProxyFactoryBean 的代码:

```

package org.qxh.proxy;

public class ProxyFactoryBean {

    private Advice advice;
    private Object target;

    public Advice getAdvice() {
        return advice;
    }

    public void setAdvice(Advice advice) {
        this.advice = advice;
    }

    public Object getTarget() {
        return target;
    }
}

```



```

    public void setTarget(Object target) {
        this.target = target;
    }

    public Object getProxy() {
        return ProxyUtil.getProxy(target, advice);
    }
}

最后搞一个测试类:
package org.qxh.proxy;

import java.io.InputStream;
import java.util.Collection;

public class AopFrameworkTest {

    public static void main(String[] args) throws Exception {
        InputStream ips =
AopFrameworkTest.class.getResourceAsStream("config.properties");
        Object bean = new BeanFactory(ips).getBean("xxx");
        System.out.println(bean.getClass().getName());
        ((Collection)bean).clear();
    }
}

```

如果配置文件这样配置:

```

#xxx=java.util.ArrayList
xxx=org.qxh.proxy.ProxyFactoryBean
xxx.advice=org.qxh.proxy.MyAdvice
xxx.target=java.util.ArrayList

```

输出为:

```
$Proxy0
```

clear 运行时间为: 0

如果配置文件这样配置:

```

xxx=java.util.ArrayList
#xxx=org.qxh.proxy.ProxyFactoryBean
xxx.advice=org.qxh.proxy.MyAdvice
xxx.target=java.util.ArrayList

```

输出为:

```
java.util.ArrayList
```

这样一来就可以根据配置文件自由切换, 想用原始类就用原始类, 想用代理就用代理

OK, Java 进阶内功修炼所需知识掌握这些就差不多了。太高级的这里没有涉及, 个人水平有限, 主要是根据张孝祥老师的视频整理而成, 在此感谢张老师的无私奉献~

希望这个文档对你有点帮助, 也不枉费我十月一假期的辛苦~