13-JavaSE高级开发之反射

> ## 本节目标
>
> 1. 认识反射机制
> 2. 反射与类操作
> 3. 反射与简单Java类
> 4. ClassLoader类加载器
> 5. 反射与代理设计模式
> 6. 反射与Annotation

1.认识反射机制

1.1 初识反射

反射指的是对象的反向处理操作，既然是反向处理。我们先来观察一下"正"的操作。在默认情况下，必须要先导入一个包，而后才能产生类的实例化对象

范例：观察正常处理

```
import java.util.Date;

public class Test {
    public static void main(String[] args) {
        Date date = new Date() ;
    }
}
```

以上是我们正常的关于对象的处理流程:根据包名.类名找到类

所谓的"反"指的是根据对象来取得对象的来源信息，而这个"反"的操作核心的处理就在于Object类的一个方法:

取得Class对象:

```
public final native Class<?> getClass();
```

该方法返回的是一个Class类对象，这个Class描述的就是类。

范例：调用getClass()方法

```
import java.util.Date;

public class Test {
    public static void main(String[] args) {
        Date date = new Date() ;
        System.out.println(date.getClass());
    }
}
```

此时通过对象取得了对象的来源，这就是"反"的本质。

在反射的世界里面，看重的不再是一个对象，而是对象身后的组成(类、构造、普通、成员等)

1.2 Class类对象的三种实例化模式

Class类是描述整个类的概念，也是整个反射的操作源头，在使用Class类的时候需要关注的依然是这个类的对象。而这个类的对象的产生模式一共有三种：

> 1. 任何类的实例化对象可以通过Object类中的getClass()方法取得Class类对象。
> 2. "类.class":直接根据某个具体的类来取得Class类的实例化对象。
> 3. 使用Class类提供的方法:public static Class<?> forName(String className) throws ClassNotFoundException

范例：使用Class.forName方法

```
public class Test {
    public static void main(String[] args) throws ClassNotFoundException {
        Class<?> cls = Class.forName("java.util.Date") ;
        System.out.println(cls.getName());
    }
}
```

在以上给出的三个方法中我们可以发现，除了第一种方法会产生Date类的实例化对象之外，其他的两种都不会产生Date类的实例化对象。于是取得了Class类对象有一个最直接的好处:**可以通过反射实例化对象**，在Class类中定义有如下方法:

```
public T newInstance()
        throws InstantiationException, IllegalAccessException
```

范例：反射实例化对象

```
public class Test {
    public static void main(String[] args) throws ClassNotFoundException,
InstantiationException, IllegalAccessException {
        Class<?> cls = Class.forName("java.util.Date") ;
        Object obj = cls.newInstance() ; // 实例化对象，等价于 new java.util.Date() ;
        System.out.println(obj);
    }

}
```

现在发现除了关键字new之外，对于对象的实例化模式有了第二种做法，通过反射进行。

取得了Class对象就意味着取得了一个指定类的操作权

1.3 反射与工厂设计模式

工厂设计模式曾经给过原则：如果是自己编写的接口，要想取得本接口的实例化对象，最好使用工厂类来设计。但是也需要知道传统工厂设计所带来的问题。

范例：传统工厂类

```
interface IFruit {
    public void eat() ;
}
class Apple implements IFruit {
    @Override
    public void eat() {
        System.out.println("[Apple] 吃苹果 ");
    }
}
class FruitFactory {
    private FruitFactory() {}
    public static IFruit getInstance(String className) {
        if ("apple".equals(className)) {
            return new Apple() ;
        }
        return null ;
    }
}

public class Test {
    public static void main(String[] args) {
        IFruit fruit = FruitFactory.getInstance("apple") ;
        fruit.eat() ;
    }
}
```

以上传统工厂类在实际开发之中根本用不到。(问题就在于new)。每增加一个接口的子类就需要修改工厂类。

范例：传统工厂类增加接口子类

```
class Orange implements IFruit {
    @Override
    public void eat() {
        System.out.println("[Orange] 吃橘子 ");
    }
}
class FruitFactory {
    private FruitFactory() {}
    public static IFruit getInstance(String className) {
        if ("apple".equals(className)) {
```

```
            return new Apple() ;
        }else if ("orange".equals(className)) {
            return new Orange() ;
        }
        return null ;
    }
}
```

如果要想解决关键字new带来的问题，最好的做法就是通过反射来完成处理，因为Class类可以使用newInstance()实例化对象，同时Class.forName()能够接收类名称。

范例：修改程序

```
package www.bit.java.testthread;
interface IFruit {
    public void eat() ;
}
class Apple implements IFruit {
    @Override
    public void eat() {
        System.out.println("[Apple] 吃苹果 ");
    }
}
class Orange implements IFruit {
    @Override
    public void eat() {
        System.out.println("[Orange] 吃橘子 ");
    }
}
class FruitFactory {
    private FruitFactory() {}
    public static IFruit getInstance(String className) {
        IFruit fruit = null ;
        try {
            fruit = (IFruit) Class.forName(className).newInstance() ;
        } catch (InstantiationException | IllegalAccessException | ClassNotFoundException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
        return fruit ;
    }
}

public class Test {
    public static void main(String[] args) {
        IFruit fruit = FruitFactory.getInstance("www.bit.java.testthread.Orange") ;
        fruit.eat() ;
    }
}
```

引入反射后，每当新增接口子类，无需去修改工厂类代码就可以很方便的进行接口子类扩容。以上这种工厂类代码我们称之为*简单工厂模式*

2.反射与类操作

利用反射可以做出一个对象具备的所有操作行为，最为关键的是这一切的操作都可以基于Object进行。

2.1 取得父类信息

在java中任何的程序类都一定会有父类，在Class类中就可以通过如下方法来取得父类或者实现的父接口：

> - 取得类的包名称: public Package getPackage()

范例：取得包名称

```
package www.bit.java.testthread;
interface IFruit {}
interface IMessage{}
class CLS implements IFruit,IMessage{}

public class Test {
    public static void main(String[] args) {
        Class<?> cls = CLS.class ;  // 取得Class类对象
        System.out.println(cls.getPackage().getName());
    }
}
```

> - 取得父类的Class对象: public native Class<? super T> getSuperclass();

> - 取得实现的父接口:public Class<?>[] getInterfaces()

范例：取得父类(实现的父接口)

```
package www.bit.java.testthread;
interface IFruit {}
interface IMessage{}
class CLS implements IFruit,IMessage{}
public class Test {
    public static void main(String[] args) {
        Class<?> cls = CLS.class ;  // 取得Class类对象
        // 取得Package名称
        System.out.println(cls.getPackage().getName());
        // 取得父类名称
        System.out.println(cls.getSuperclass().getName());
        // 取得实现的父接口对象
        Class<?>[] iClass = cls.getInterfaces() ;
        for (Class<?> class1 : iClass) {
            System.out.println(class1.getName());
        }
    }
}
```

通过反射可以取得类结构上的所有关键信息。

2.2 反射调用构造

一个类中可以存在多个构造方法，如果要想取得类中构造的调用，就可以使用Class类中提供的两个方法:

> - 取得指定参数类型的构造:

```
public Constructor<T> getConstructor(Class<?>... parameterTypes)
        throws NoSuchMethodException, SecurityException
```

> - 取得类中的所有构造:

```
public Constructor<?>[] getConstructors() throws SecurityException
```

以上两个方法返回的类型都是java.lang.reflect.Constructor类的实例化对象，这个类之中大家只需要关注一个方法.

实例化对象:

```
public T newInstance(Object ... initargs)
        throws InstantiationException, IllegalAccessException,
               IllegalArgumentException, InvocationTargetException
```

范例：取得类中所有构造信息

```
package www.bit.java.testthread;

import java.lang.reflect.Constructor;

class Person {
    public Person() {}
    public Person(String name) {}
    public Person(String name,int age) {}
}
public class Test {
    public static void main(String[] args) {
        Class<?> cls = Person.class ;
        // 取得类中的全部构造
        Constructor<?>[] constructors = cls.getConstructors() ;
        for (Constructor<?> constructor : constructors) {
            System.out.println(constructor);
        }
    }
}
```

以上的操作是直接利用了Constructor类中的toString()方法取得了构造方法的完整信息(包含方法权限，参数列表)，而如果你只使用了getName()方法，只会返回构造方法的包名.类名。

在定义简单java类的时候一定要保留有一个无参构造

范例：观察Class实例化对象的问题

```
package www.bit.java.testthread;

class Person {
    private String name ;
    private int age ;
    public Person(String name,int age) {
        this.name = name ;
        this.age = age ;
    }
    @Override
    public String toString() {
        return "Person [name=" + name + ", age=" + age + "]";
    }
}
public class Test {
    public static void main(String[] args) throws InstantiationException, IllegalAccessException
{
        Class<?> cls = Person.class ;
        System.out.println(cls.newInstance());
    }
}
```

Class类通过反射实例化类对象的时候，只能够调用类中的无参构造。如果现在类中没有无参构造则无法使用Class类调用，只能够通过明确的构造调用实例化处理。

范例：通过Constructor类实例化对象

```
package www.bit.java.testthread;

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

class Person {
    private String name ;
    private int age ;
    public Person(String name,int age) {
        this.name = name ;
        this.age = age ;
    }
    @Override
    public String toString() {
        return "Person [name=" + name + ", age=" + age + "]";
    }
}
public class Test {


    public static void main(String[] args) throws InstantiationException,
```

```
IllegalAccessException, NoSuchMethodException, SecurityException, IllegalArgumentException,
InvocationTargetException {
        Class<?> cls = Person.class ;
        // 取得指定参数类型的构造方法对象
        Constructor<?> cons = cls.getConstructor(String.class,int.class) ;
        System.out.println(cons.newInstance("yuisama",29));
    }
}
```

结论：以后写简单java类要写无参构造。

2.3 反射调用普通方法(核心)

类中普通方法的反射调用你在开发之中一定会使用到，并且使用好了可以节省大量的重复编码。在Class类中有如下两种取得类中普通方法的函数:

- 取得全部普通方法:

```
public Method[] getMethods() throws SecurityException
```

- 取得指定普通方法:

```
public Method getMethod(String name, Class<?>... parameterTypes)
```

以上两个方法范辉的类型是java.lang.reflect.Method类的对象，在此类中提供有一个调用方法的支持:

- 调用:

```
public Object invoke(Object obj, Object... args)throws IllegalAccessException,
IllegalArgumentException,InvocationTargetException
```

范例：取得一个类中的全部普通方法

```
package www.bit.java.testthread;

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

class Person {
    private String name ;
    private int age ;
    public Person() {}
    public Person(String name,int age) {
        this.name = name ;
        this.age = age ;
    }
    @Override
    public String toString() {

        return "Person [name=" + name + ", age=" + age + "]";
```

```
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
}
public class Test {
    public static void main(String[] args) throws Exception {
        Class<?> cls = Person.class ;
        Method[] methods = cls.getMethods() ;
        for (Method method : methods) {
            System.out.println(method);
        }
    }
}
```

之前程序编写的简单java类中的getter、setter方法采用的都是明确的对象调用。

而现在有了反射机制处理之后，即使你没有明确的Person类型对象(依然需要实例化对象，Object对象描述，所有的普通方法必须在有实例化对象之后才可以进行调用)。就可以通过反射调用。

范例：通过反射调用setter、getter方法

```
package www.bit.java.testthread;

import java.lang.reflect.Method;

class Person {
    private String name ;
    private int age ;
    public Person() {}
    public Person(String name,int age) {
        this.name = name ;
        this.age = age ;
    }
    @Override
    public String toString() {
        return "Person [name=" + name + ", age=" + age + "]";
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;

    }
```

```
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
}
public class Test {
    public static void main(String[] args) throws Exception {
        Class<?> cls = Class.forName("www.bit.java.testthread.Person") ;
        // 任何时候调用类中的普通方法都必须有实例化对象
        Object obj = cls.newInstance() ;
        // 取得setName这个方法的实例化对象,设置方法名称与参数类型
        Method setMethod = cls.getMethod("setName", String.class) ;
        // 随后需要通过Method类对象调用指定的方法，调用方法需要有实例化对象
        // 同时传入参数
        setMethod.invoke(obj, "yuisama") ; // 相当于Person对象.setName("yuisama") ;
        Method getMethod = cls.getMethod("getName") ;
        Object result  = getMethod.invoke(obj) ;  // 相当于Person对象.getName() ;
        System.out.println(result) ;
    }
}
```

此类操作的好处是:不再局限于某一具体类型的对象，而是可以通过Object类型进行所有类的方法调用。这个操作必须掌握.

2.4 反射调用类中属性

在之前已经成功的实现了类的构造调用、方法调用，除了这两种模式之外还有类中属性调用。

前提:类中的所有属性一定在类对象实例化之后才会进行空间分配，所以此时如果要想调用类的属性，必须保证有实例化对象。通过反射的newInstance()可以直接取得实例化对象(Object类型)

在Class类中提供有两组取得属性的方法:

1. 第一组（父类中）-取得类中全部属性: public Field[] getFields() throws SecurityException
2. 第一组(父类中)-取得类中指定名称属性: public Field getField(String name) throws NoSuchFieldException, SecurityException
3. 第二组(本类中)-取得类中全部属性: public Field[] getDeclaredFields() throws SecurityException
4. 第二组(□本类中)-取得类中指定名称属性 : public Method getDeclaredMethod(String name, Class<?>... parameterTypes) throws NoSuchMethodException, SecurityException

范例：取得类中全部属性

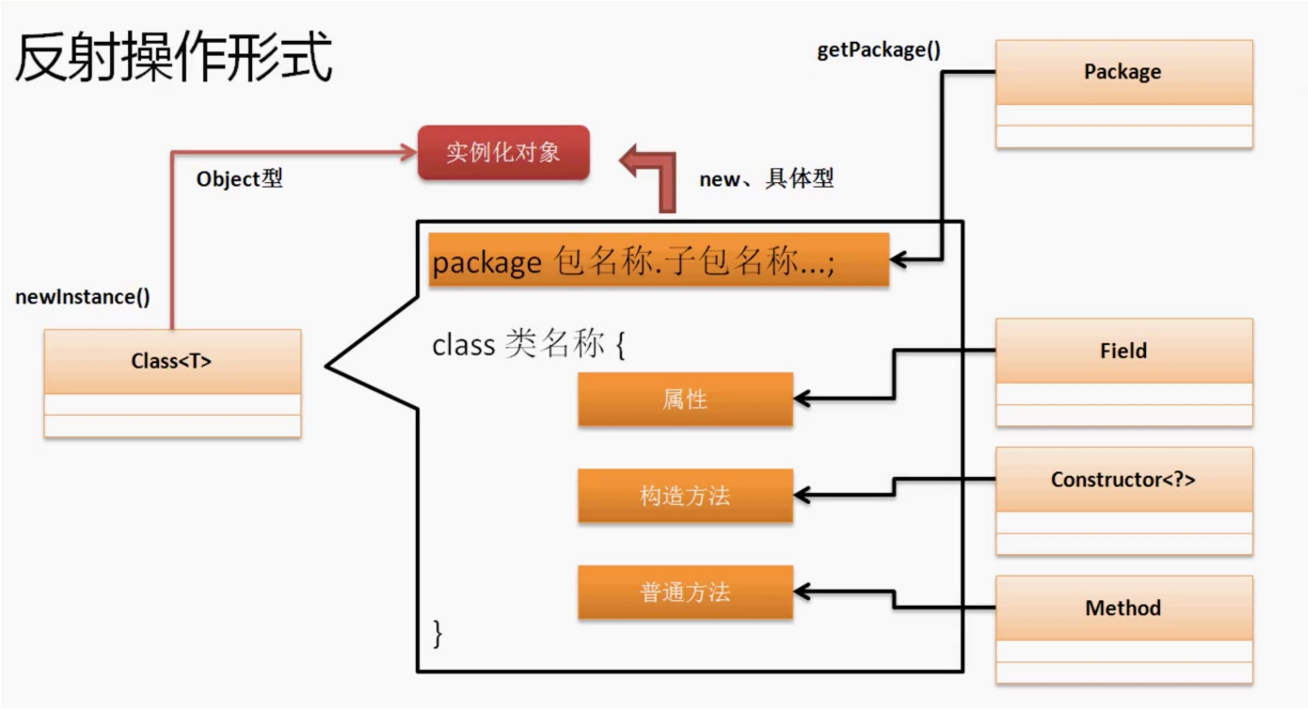
```
package www.bit.java.testthread;

import java.lang.reflect.Field;

class Person {
    public String name ;
    public int age ;

}
```

```
class Student extends Person {
    private String school ;
}
public class Test {
    public static void main(String[] args) throws Exception {
        Class<?> cls = Class.forName("www.bit.java.testthread.Student") ;
        { // 普通代码块
            // 第一组-取得类中全部属性
            Field[] fields = cls.getFields() ;
            for (Field field : fields) {
                System.out.println(field) ;
            }
        }
        System.out.println("------------------------");
        {
            // 第二组-取得类中全部属性
            Field[] fields = cls.getDeclaredFields() ;
            for (Field field : fields) {
                System.out.println(field);
            }
        }
    }
}
```

因为在实际开发之中，属性基本上都会进行封装处理，所以没有必要去关注父类中的属性。也就是说以后所取得的属性都以本类属性为主。

而后就需要关注属性的核心描述类:java.lang.reflect.Field,在这个类之中有两个重要方法：

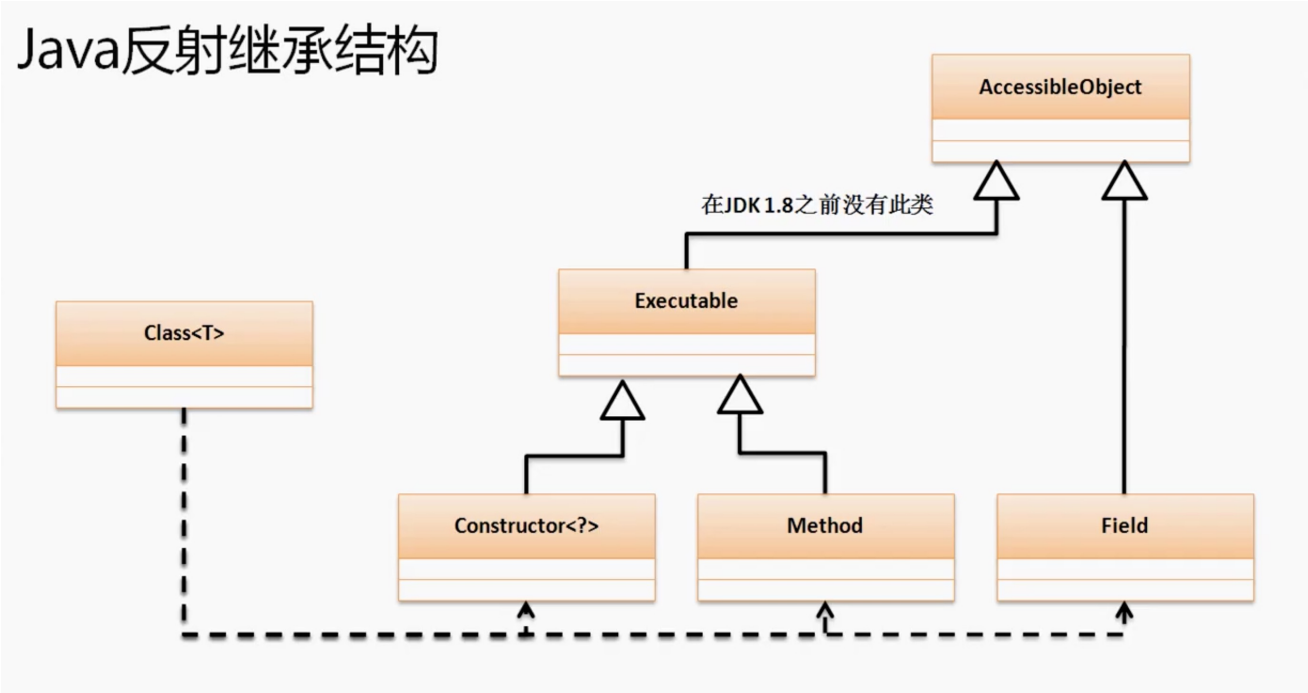
> 1. 设置属性内容 : public void set(Object obj, Object value) throws IllegalArgumentException, IllegalAccessException
> 2. 取得属性内容 : public Object get(Object obj) throws IllegalArgumentException, IllegalAccessException



范例：通过反射操作属性

```
package www.bit.java.testthread;

import java.lang.reflect.Field;

class Person {
    private String name ;
}
public class Test {
    public static void main(String[] args) throws Exception {
        Class<?> cls = Class.forName("www.bit.java.testthread.Person") ;
        // 实例化本类对象
        Object obj = cls.newInstance() ;
        // 操作name属性
        Field nameField = cls.getDeclaredField("name") ;
        nameField.set(obj, "yuisama") ; // 相当于对象.name = "yuisama"
        System.out.println(nameField.get(obj)); // 取得属性
    }
}
```

来看Java反射继承结构



在AccessibleObject类中提供有一个方法:

动态设置封装:

```
public void setAccessible(boolean flag) throws SecurityException
```

范例:动态设置封装

```
package www.bit.java.testthread;

import java.lang.reflect.Field;

class Person {
    private String name ;
}
public class Test {
    public static void main(String[] args) throws Exception {
        Class<?> cls = Class.forName("www.bit.java.testthread.Person") ;
        // 实例化本类对象
        Object obj = cls.newInstance() ;
        // 操作name属性
        Field nameField = cls.getDeclaredField("name") ;
        // 取消封装
        nameField.setAccessible(true) ;
        // ----------------------------
        nameField.set(obj, "yuisama") ; // 相当于对象.name = "yuisama"
        System.out.println(nameField.get(obj)); // 取得属性
    }
}
```

在Field类之中有一个特别有用的方法:

取得属性类型:

```
public Class<?> getType()
```

范例：取得属性类型

```
package www.bit.java.testthread;

import java.lang.reflect.Field;

class Person {
    private String name ;
}
public class Test {
    public static void main(String[] args) throws Exception {
        Class<?> cls = Class.forName("www.bit.java.testthread.Person") ;
        // 实例化本类对象
        Object obj = cls.newInstance() ;
        // 操作name属性
        Field nameField = cls.getDeclaredField("name") ;
        // 包.类
        System.out.println(nameField.getType().getName()) ;
        // 类名称
        System.out.println(nameField.getType().getSimpleName()) ;
    }
}
```

将Field取得属性与Method取得方法配合使用就可以编写出非常灵活的代码。

3.反射与简单Java类

以后JavaEE使用到的所有开发框架里面到处都是反射的身影，没有反射就没有开发框架。

下面主要结合简单Java类来进行反射开发的深入研究，这些也是后期SpringMVC开发框架的主要操作原理。

3.1 反射与单级VO操作

如果现在有一个简单Java类，按照原始的做法使用getter与setter对属性进行操作。

范例：基本程序

```
class Emp{
    private String ename ;
    private String job ;

    public String getEname() {
        return ename;
    }

    public void setEname(String ename) {

        this.ename = ename;
```

```
    }

    public String getJob() {
        return job;
    }

    public void setJob(String job) {
        this.job = job;
    }
}
```

现在Emp类中存在有无参构造，于是按照传统调用则编写如下:

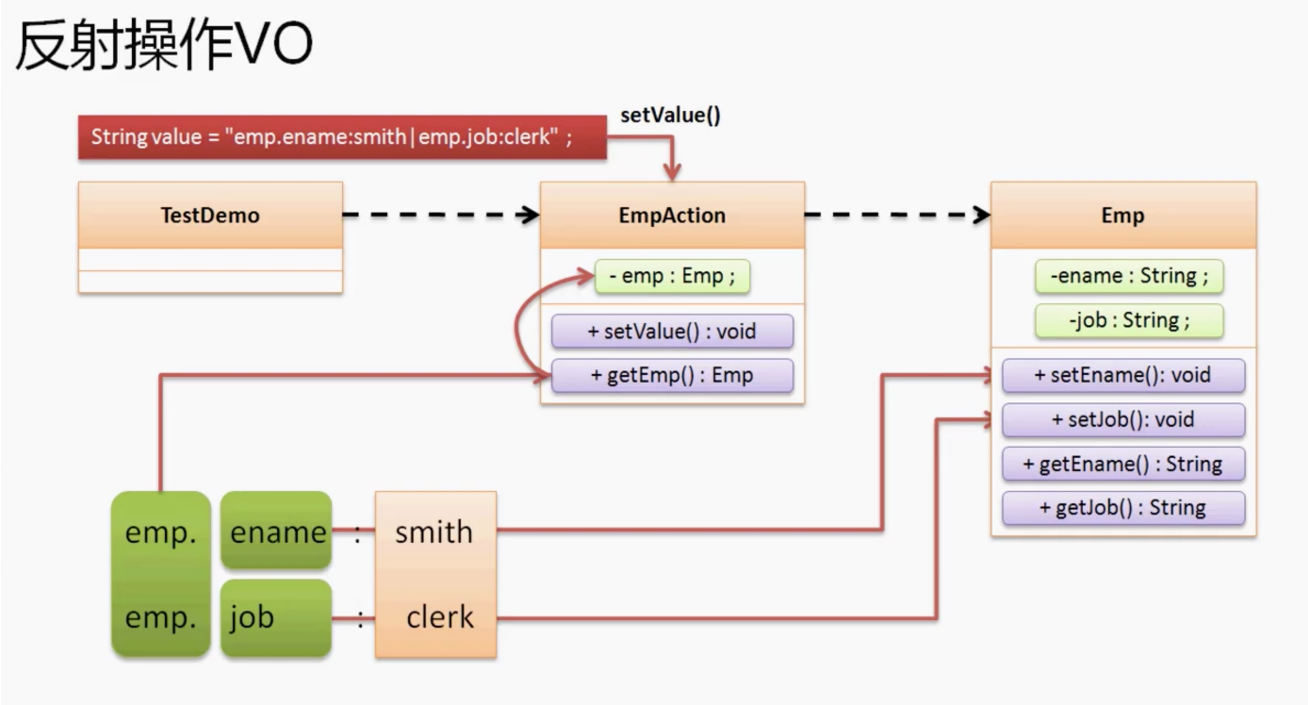
```
package www.bit.java.vo;

class Emp{
    private String ename ;
    private String job ;

    public String getEname() {
        return ename;
    }

    public void setEname(String ename) {
        this.ename = ename;
    }

    public String getJob() {
        return job;
    }

    public void setJob(String job) {
        this.job = job;
    }

    @Override
    public String toString() {
        return "Emp{" +
                "ename='" + ename + '\'' +
                ", job='" + job + '\'' +
                '}';
    }
}


public class TestDemo {
    public static void main(String[] args) {
        Emp emp = new Emp() ;
        emp.setEname("yuisama") ;
        emp.setJob("Java Coder") ;

        System.out.println(emp);
```

```
    }
}
```

假设一个类中存在有几十个属性，要按照原始做法，要调用几十次setter方法，这样操作就太麻烦了。

现在希望能对程序做简化，输入字符串"属性名称:属性值|属性名称:属性值|属性名称:属性值|....", 就可以将类中的属性设置好。希望通过此程序实现任意的简单Java类的属性设置。

设计思路如下:



现在所有的操作是通过TestDemo类调用EmpAction类实现的，而EmpAction类的主要作用是在于定位要操作的属性类型。同时该程序应该符合于所有简单的Java类开发形式，因此对于我们的设计而言必须有一个单独的类(BeanOperation)(实现此适配。

范例：Emp类设计（简单Java类）

```
package www.bit.java.vo;

public class Emp {
    private String ename ;
    private String job ;

    public String getEname() {
        return ename;
    }

    public void setEname(String ename) {
        this.ename = ename;
    }

    public String getJob() {

        return job;
```

```
    }

    public void setJob(String job) {
        this.job = job;
    }

    @Override
    public String toString() {
        return "Emp{" +
                "ename='" + ename + '\'' +
                ", job='" + job + '\'' +
                '}';
    }
}
```

范例：EmpAction类设计

```
package www.bit.java.vo;

import www.bit.java.util.BeanOperation;

public class EmpAction {
    private Emp emp = new Emp() ;
    public void setValue(String value) throws Exception{
        BeanOperation.setBeanValue(this,value) ;
    }
    public Emp getEmp(){
        return emp ;
    }
}
```

范例：BeanOperation类设计(公共程序类)

```
package www.bit.java.util;

import java.lang.reflect.Field;
import java.lang.reflect.Method;

/**
 * 本类主要负责实现自动的VO匹配处理操作
 */
public class BeanOperation {
    private BeanOperation() {}

    /**
     * 负责设置类中的属性操作
     * @param actionObject 表示当前发出设置请求的程序类的当前对象
     * @param msg 属性的具体内容，格式为:"属性名称:|内容|属性名称:|内容..."

     */
```

```
public static void setBeanValue(Object actionObject,String msg) throws Exception{
    // 要想进行内容的设置，必须将字符串拆分
    // 按照竖线拆分，取出所有要设置的内容
    String[] result = msg.split("\\|") ;
    // 每次执行后只剩下"属性名称:内容"
    for (int i = 0; i < result.length ; i++) {
        // 需要针对属性名称及内容做进一步拆分
        String[] temp = result[i].split(":") ;
        // 属性名称 "emp.ename"
        String attribute = temp[0] ;
        // 属性内容
        String value = temp[1] ;
        String[] fields = attribute.split("\\.") ;
        // 获取当前操作的简单Java类对象
        Object currentObject = getObject(actionObject,fields[0]) ;
        // 调用简单Java类的setter方法
        setObjectValue(currentObject,fields[1],temp[1]);
    }
}

/**
 * 将给定字符串的首字母大写
 * @param str 给定的字符串
 * @return 返回首字母大写的字符串
 */
public static String initCap(String str) {
    return str.substring(0,1).toUpperCase()+str.substring(1) ;
}

/**
 * 负责调用XXXAction类中的getter方法取得简单Java类对象
 * @param obj 表示调用方法的所在类对象
 * @param attribute 表示属性名称
 * @return 调用对象结果
 */
public  static Object getObject(Object obj,String attribute) throws Exception{
    String methodName = "get" + initCap(attribute) ;
    // 调用指定属性的Field对象，目的是取得对象类型，如果没有此属性意味着操作无法继续进行
    Field field = obj.getClass().getDeclaredField(attribute) ;
    if (field == null) {
        // 第二次尝试从父类中取得该属性
        field = obj.getClass().getField(attribute) ;
    }
    if (field == null) {
        // 两次都未取得指定属性的对象，表示该对象一定不存在
        return null ;
    }
    Method method = obj.getClass().getMethod(methodName) ;
    return method.invoke(obj) ;
}

/**

 * 根据指定的类对象设置指定类中的属性，调用setter方法
```

```
     * @param obj 属性所在类的实例化对象
     * @param attribute 属性名称
     * @param value 属性内容
     */
    public static void setObjectValue(Object obj,String attribute,String value) throws
Exception{
        Field field = obj.getClass().getDeclaredField(attribute) ;
        // 判断属性是否存在
        if (field == null) {
            // 第二次尝试从父类中取得该属性
            field = obj.getClass().getField(attribute) ;
        }
        if (field == null) {
            // 两次都未取得指定属性的对象，表示该对象一定不存在
            return ;
        }
        String methodName = "set" + initCap(attribute) ;
        Method setMethod = obj.getClass().getMethod(methodName,field.getType()) ;
        setMethod.invoke(obj,value) ;
    }
}
```

范例：测试类设计

```
package www.bit.java.vo;

public class TestDemo {
    public static void main(String[] args) throws Exception {
       String value = "emp.ename:yuisama|emp.job:Java Coder" ;
       EmpAction empAction = new EmpAction() ;
       empAction.setValue(value) ;
       System.out.println(empAction.getEmp());
    }
}
```

3.2 多级VO设置实现

对3.1节的需求做一个更改，假设现在一个雇员(EMP)属于一个部门(dept)，一个部门属于一个公司。这种类似的关系都可以通过字符串的配置来设置内容。

范例：建立Company类

```
package www.bit.java.vo;

public class Company {
    private String cname ;
    private String address ;

    public String getCname() {
        return cname;
```

```
    }

    public void setCname(String cname) {
        this.cname = cname;
    }

    public String getAddress() {
        return address;
    }

    public void setAddress(String address) {
        this.address = address;
    }

    @Override
    public String toString() {
        return "Company{" +
                "cname='" + cname + '\'' +
                ", address='" + address + '\'' +
                '}';
    }
}
```

范例：修改Dept类

```
    private String dname ;
    private String loc ;
    private Company company = new Company() ;
```

范例：修改Emp类

```
    private String ename ;
    private String job ;
    private Dept dept = new Dept() ;
```

此时所有的引用关系上都自动进行了对象实例化。而现在希望程序既可以满足单级VO或多级VO操作，所以对于内容的设置可能采用如下代码出现：

范例：定义TestDemo测试类

```
String value = "emp.ename:yuisama|emp.job:Java Coder|emp.dept.name:教务
部|emp.dept.company.cname=bit" ;
```

多级VO设置即:给定如上字符串把属性都正确设置到各个类中。

范例：修改setBeanValue()方法，支持多级VO设置

```
/**
     * 负责设置类中的属性操作
```

```
     * @param actionObject 表示当前发出设置请求的程序类的当前对象
     * @param msg 属性的具体内容，格式为:"属性名称:|内容|属性名称:|内容..."
     */
    public static void setBeanValue(Object actionObject,String msg) throws Exception{
        // 要想进行内容的设置，必须将字符串拆分
        // 按照竖线拆分，取出所有要设置的内容
        String[] result = msg.split("\\|") ;
        // 每次执行后只剩下"属性名称:内容"
        for (int i = 0; i < result.length ; i++) {
            // 需要针对属性名称及内容做进一步拆分
            String[] temp = result[i].split(":") ;
            // 属性名称 "emp.ename"
            String attribute = temp[0] ;
            // 属性内容
            String value = temp[1] ;
            // 通过属性的拆分就可以区分出是单级VO还是多级
            String[] fields = attribute.split("\\.") ;
            if (fields.length > 2 ) {
                // 多级VO操作
                // 如果要想通过多级确定出属性的操作对象，那么就应该一层层找出每一个getter方法返回的内容
                Object currentObject = actionObject ; // 确定当前操作的对象
                for(int x = 0 ; x < fields.length-1 ; x++ ){
                    // 对应getter的返回对象
                    // 循环结束一定能拿到最后一层的当前对象
                    currentObject = getObject(currentObject,fields[x]) ;
                }
                setObjectValue(currentObject,fields[fields.length - 1],value) ;
            }else {
                // 单级VO
                // 获取当前操作的简单Java类对象
                Object currentObject = getObject(actionObject,fields[0]) ;
                // 调用简单Java类的setter方法
                setObjectValue(currentObject,fields[1],temp[1]);
            }

        }
    }
```

上面程序属于多级VO配置属性。

3.3 设置各种数据类型

在实际开发之中使用最多的几种类型:int、double、long、Date、String等。

范例:修改Emp类(getter、setter方法略)

```
    private String ename ;
    private String job ;
    private Double salary ;
    private Date hireDate ;
```

范例：修改Dept类

```
    private String dname ;
    private String loc ;
    private Long count ; // 总员工数量
```

范例：修改Company类

```
    private Integer cid ;
    private String cname ;
    private String address ;
    private Date createDate ;
```

现在的程序支持的数据类型都是在开发中常用的类型。

要想支持各种数据类型，那么不能再使用String来进行内容的接收与设置了，需要更换为Object。

范例：修改setObjectValue方法

```
    public static void setObjectValue(Object obj,String attribute,Object value) throws
Exception{
        Field field = obj.getClass().getDeclaredField(attribute) ;
        // 判断属性是否存在
        if (field == null) {
            // 第二次尝试从父类中取得该属性
            field = obj.getClass().getField(attribute) ;
        }
        if (field == null) {
            // 两次都未取得指定属性的对象，表示该对象一定不存在
            return ;
        }
        String methodName = "set" + initCap(attribute) ;
        Method setMethod = obj.getClass().getMethod(methodName,field.getType()) ;
        setMethod.invoke(obj,value) ;
    }
```

进而修改setBeanValue方法，不能再使用String进行数值保存了，需要依靠传入的对象和属性的类型来判断到底传入的是什么数据类型，为了方便数据转型，再定义一组相关方法

范例：

```
/**
     * 将字符串的内容根据属性类型变为各种数据类型。
     * 支持的类型:int(Integer)、double(Double)、long(Long)、String、Date
     * @param wrapObject 包装类
     * @param attribute 属性
     * @param value 属性值
     * @return 根据属性类型进行转型处理
     */


    public static Object getValue(Object wrapObject,String attribute,String value) throws
```

```
Exception{
        Field field = wrapObject.getClass().getDeclaredField(attribute) ;
        // 判断属性是否存在
        if (field == null) {
            // 第二次尝试从父类中取得该属性
            field = wrapObject.getClass().getField(attribute) ;
        }
        if (field == null) {
            // 两次都未取得指定属性的对象，表示该对象一定不存在
            return null ;
        }
       return stringToType(field.getType().getSimpleName(),value) ;
    }

    /**
     * 根据指定类型将字符串做转型处理
     * @param type 数据类型
     * @param value 数据内容
     * @return 转换为具体类型
     */
    private static Object stringToType(String type,String value) throws Exception{
        if ("String".equals(type)){
            if (isNotNull(value)){
                return value ;
            }else {
                return null ;
            }
        } else if ("int".equals(type)||"Integer".equals(type)){
            if (isInt(value)) {
                // 数据内容为整型
                return Integer.parseInt(value);
            }
        }else if ("double".equals(type)||"Double".equals(type)){
            if (isDouble(value)) {
                // 数据内容为小数
                return Double.parseDouble(value) ;
            }
        }else if ("long".equals(type)||"Long".equals(type)){
            if (isDouble(value)) {
                // 数据内容为长整型
                return Long.parseLong(value) ;
            }
        }else if ("Date".equals(type)){
            // 数据类型为Date
            String pattern = null ;
            if (isDate(value)) {
              pattern = "yyyy-MM-dd" ;
            }
            if (pattern!=null){
                return new SimpleDateFormat(pattern).parse(value) ;
            }
        }

        return null ;
```

```
    }

    /**
     * 判断字符串是否为空
     * @param str 要判断的字符串
     * @return 如果字符串为空，返回false；否则返回true
     */
    private static boolean isNotNull(String str) {
        return !(str == null || str.isEmpty() || "".equals(str)) ;
    }

    /**
     * 判断给定字符串是否是一个整数
     * @param str 给定字符串
     * @return 由整数组成，返回true；否则返回false
     */
    private static boolean isInt(String str){
        if (isNotNull(str)){
            return str.matches("\\d+") ;
        }
        return false ;
    }

    /**
     * 判断给定的字符串是否为小数
     * @param str 给定的字符串
     * @return 由小数组成，返回true；否则返回false
     */
    private static boolean isDouble(String str){
        if (isNotNull(str)){
            return str.matches("\\d+(\\.\\d+)?") ;
        }
        return false ;
    }

    /**
     * 判断给定字符串是否为日期类型(yyyy-MM-dd)
     * @param str 给定字符串
     * @return 是日期类型返回true；否则返回false
     */
    private static boolean isDate(String str){
        if (isNotNull(str)){
            return str.matches("\\d{4}-\\d{2}-\\d{2}") ;
        }
        return false ;
    }
```

范例：修改setBeanValue方法

```
/**

     * 负责设置类中的属性操作
```

```
     * @param actionObject 表示当前发出设置请求的程序类的当前对象
     * @param msg 属性的具体内容，格式为:"属性名称:|内容|属性名称:|内容..."
     */
    public static void setBeanValue(Object actionObject,String msg) throws Exception{
        // 要想进行内容的设置，必须将字符串拆分
        // 按照竖线拆分，取出所有要设置的内容
        String[] result = msg.split("\\|") ;
        // 每次执行后只剩下"属性名称:内容"
        for (int i = 0; i < result.length ; i++) {
            // 需要针对属性名称及内容做进一步拆分
            String[] temp = result[i].split(":") ;
            // 属性名称 "emp.ename"
            String attribute = temp[0] ;

            // 通过属性的拆分就可以区分出是单级VO还是多级
            String[] fields = attribute.split("\\.") ;
            if (fields.length > 2 ) {
                // 多级VO操作
                // 如果要想通过多级确定出属性的操作对象，那么就应该一层层找出每一个getter方法返回的内容
                Object currentObject = actionObject ; // 确定当前操作的对象
                for(int x = 0 ; x < fields.length-1 ; x++ ){
                    // 对应getter的返回对象
                    // 循环结束一定能拿到最后一层的当前对象
                    currentObject = getObject(currentObject,fields[x]) ;
                }
                // 属性内容
                Object value = getValue(currentObject,fields[fields.length -1 ],temp[1]) ;
                setObjectValue(currentObject,fields[fields.length - 1],value) ;
            }else {
                // 单级VO
                // 获取当前操作的简单Java类对象
                // 属性内容
                Object value = getValue(actionObject,attribute,temp[1]) ;
                Object currentObject = getObject(actionObject,fields[0]) ;
                // 调用简单Java类的setter方法
                setObjectValue(currentObject,fields[1],value) ;
            }

        }
    }
```

随后修改测试类进行内容设置

范例：修改测试类

```
package www.bit.java.vo;

public class TestDemo {
    public static void main(String[] args) throws Exception {
       String value = "emp.ename:yuisama|emp.job:Java Coder|emp.dept.dname:教务
部|emp.dept.company.cname:bit"+
               "|emp.salary:1999.12|emp.hireDate:2017-10-
10|emp.dept.count:100000|emp.dept.company.cid:10|emp.dept.company.createDate:1999-09-10" ;
       EmpAction empAction = new EmpAction() ;
       empAction.setValue(value) ;
       System.out.println(empAction.getEmp());
    }
}
```

3.4 自动实例化关联类对象

目前已经实现的程序代码之中有一处最明显的问题在于，类之间必须明确的实例化好对象。一旦其中多级Vo之中有一个类没有进行对象的实例化操作，整个程序就会报错。所以为了保证程序的正确执行，当发现没有实例化关联类对象时自动实例化，已经实例化了则保持与之前同样的操作。

范例:修改getObject()方法

```
public  static Object getObject(Object obj,String attribute) throws Exception{
        String methodName = "get" + initCap(attribute) ;
        // 调用指定属性的Field对象，目的是取得对象类型，如果没有此属性意味着操作无法继续进行
        Field field = obj.getClass().getDeclaredField(attribute) ;
        if (field == null) {
            // 第二次尝试从父类中取得该属性
            field = obj.getClass().getField(attribute) ;
        }
        if (field == null) {
            // 两次都未取得指定属性的对象，表示该对象一定不存在
            return null ;
        }
        Method method = obj.getClass().getMethod(methodName) ;
        // 调用getter方法取得实例化对象
        Object objectInstance = method.invoke(obj) ;
        if (objectInstance==null) {
            // 现在没有实例化关联类对象，必须自己手工实例化关联类对象
            // 所有程序类都可以根据反射来取得实例化对象,只需要取得类的Class类型
            // 实例化新对象
            objectInstance = field.getType().newInstance() ;
            setObjectValue(obj,attribute,objectInstance);
        }
            return objectInstance ;
    }
```

4.ClassLoader类加载器

Class类描述的是整个类的信息，在Class类中提供的forName()方法,这个方法根据ClassPath配置的路径进行类的加载，如果说现在你的类的加载路径可能是网络、文件，这个时候就必须实现类加载器，也就是ClassLoader类的主要作用。

4.1 认识ClassLoader

首先通过Class类观察如下方法:

```
 public ClassLoader getClassLoader()
```

范例：编写一个简单的反射程序，来观察ClassLoader的存在

```
// 自定义类，这个类一定在CLASSPATH中
class Member{}
public class TestDemo {
    public static void main(String[] args) {
        Class<?> cls = Member.class ;
        System.out.println(cls.getClassLoader()) ;
        System.out.println(cls.getClassLoader().getParent()) ;
        System.out.println(cls.getClassLoader().getParent().getParent());
    }
}
```

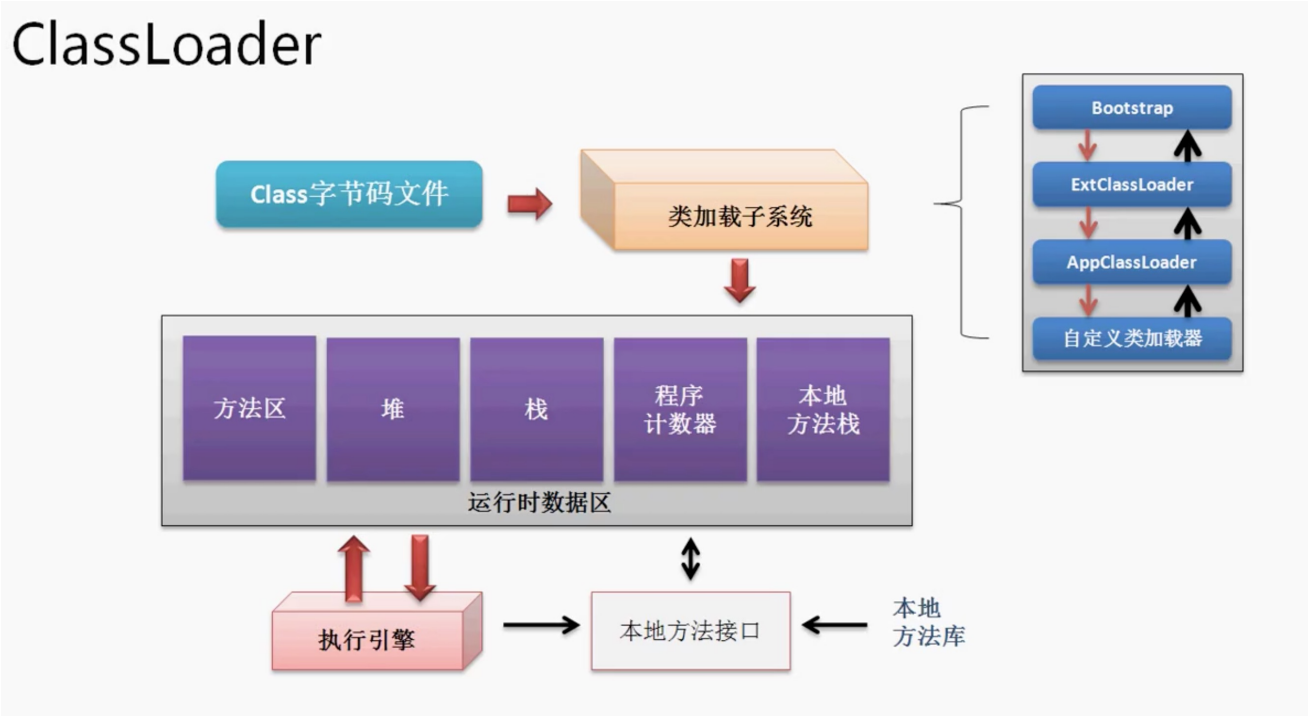
运行结果如下:

```
sun.misc.Launcher$AppClassLoader@58644d46
sun.misc.Launcher$ExtClassLoader@6a38e57f
null
```

此时出现了两个类加载器:ExtClassLoader(扩展类加载器)、AppClassLoader(应用程序类加载器)。

那么，什么是类加载器?

JVM设计团队把类加载阶段中的"通过一个类的全限定名来获取描述此类的二进制字节流"这个动作放在Java虚拟机外部去实现，以便让应用程序自己决定如何去获取所需要的类。实现这个动作的代码模块称之为"类加载器".



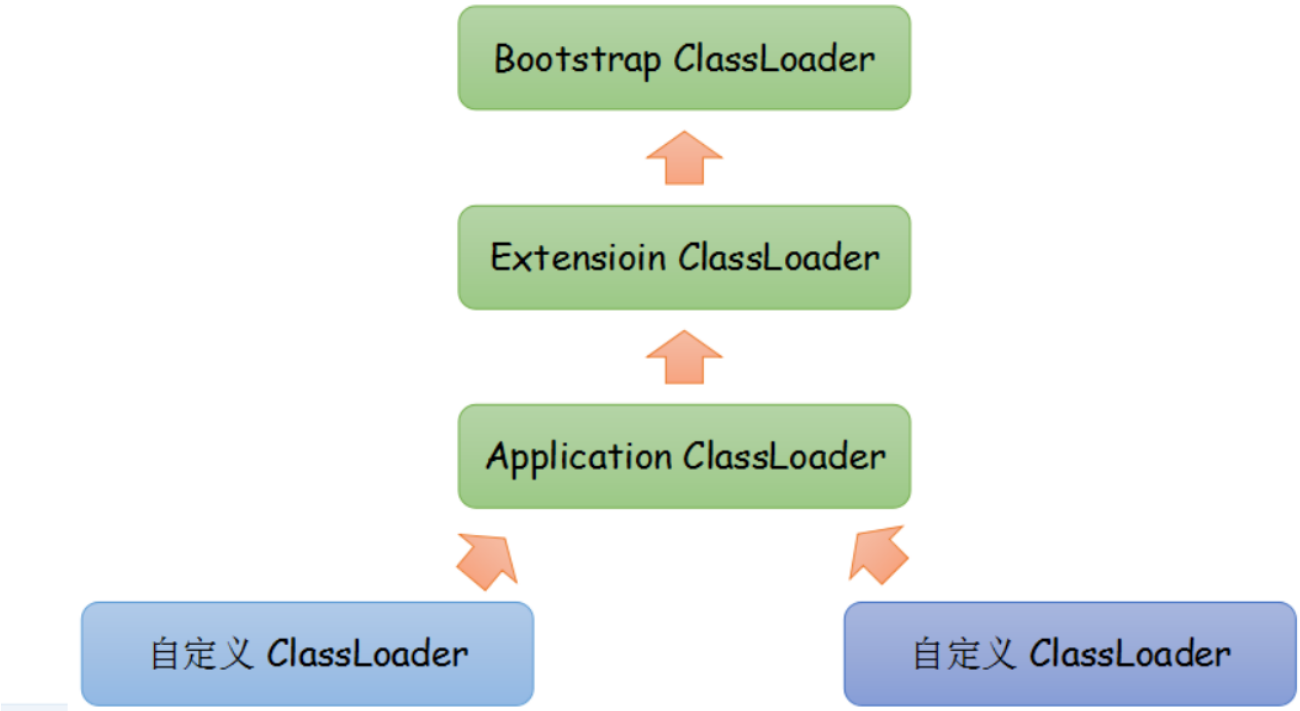
Bootstrap(启动类加载器)：这个类加载器使用C++实现，是虚拟机自身的一部分；其他的类加载器都由Java语言实现，独立于JVM外部并且都继承于java.lang.ClassLoader.BootStrap类加载器负责将存放于<Java_HOME>\lib目录中(或者被-Xbootclasspath参数指定路径中)能被虚拟机识别的(仅按照文件名识别，如rt.jar，名字不符合的类库即使放在lib目录中也不会被加载)类库加载到JVM内存中。启动类加载器无法被Java程序直接引用。

ExtClassLoader(扩展类加载器)：它负责加载<Java_HOME>\lib\ext目录中，或者被java.ext.dirs系统变量指定的路径中的类库。开发者可以直接使用扩展类加载器。

AppClassLoader(应用程序类加载器):负责加载用户类路径(ClassPath)上指定的类库，如果应用程序中没有自定义自己的类加载器，则此加载器就是程序中默认的类加载器。

4.2 双亲委派模型

我们的应用程序都是由这三种加载器互相配合进行加载的，如果有必要，还可以加入自定义的类加载器。这些类加载器的关系一般如下图所示:



上图展示的类加载器之间的这种层次关系，就称为类加载器的**双亲委派模型**。双亲委派模型要求除了顶层的父类加载器外，其余的类加载器都应有自己的父类加载器。

双亲委派模型的工作流程是：如果一个类加载器收到了类加载请求，它首先不会自己去尝试加载这个类，而是把这个请求委托给父类加载器去完成，每一个层次的类加载器都是如此。**因此，所有的加载请求都应当传送到顶层的BootStrap加载器中，只有当父加载器反馈无法完成这个加载请求时(在自己搜索范围中没有找到此类)，子加载器才会尝试自己去加载。**

类加载器的双亲委派模型从JDK1.2引入后被广泛应用于之后几乎所有的Java程序中，但它并不是强制性约束，甚至可以破坏双亲委派模型来进行类加载，最典型的就是OSGI技术。

双亲委派模式对于保证Java程序的稳定运行很重要。有一个显而易见的好处就是Java类随着它的类加载器一起具备了一种带有优先级的层次关系。例如java.lang.Object类，它存放在rt.jar中，无论哪一个类加载器要加载这个类，最终都是委派给处于顶端的启动类加载器进行加载。因此，**Object类在程序的各种类加载器环境中都是同一个类**

范例：观察CLassLoader.loadClass()方法

```
// First, check if the class has already been loaded
            Class<?> c = findLoadedClass(name);
            if (c == null) {
                long t0 = System.nanoTime();
                try {
                    if (parent != null) {
                        c = parent.loadClass(name, false);
                    } else {
                        c = findBootstrapClassOrNull(name);
                    }
                } catch (ClassNotFoundException e) {
                    // ClassNotFoundException thrown if class not found
                    // from the non-null parent class loader

                }
```

```
                if (c == null) {
                    // If still not found, then invoke findClass in order
                    // to find the class.
                    long t1 = System.nanoTime();
                    c = findClass(name);
                }
            }
            if (resolve) {
                resolveClass(c);
            }
```

范例:编写自己的java.lang.Object，观察是否被加载。

4.3 自定义类加载器

自定义类加载器：用户决定类从哪里加载。

ClassLoader类中提供有如下方法（进行类的加载）：

```
protected Class<?> loadClass(String name, boolean resolve)
        throws ClassNotFoundException
```

范例：观察默认类加载器

```
// 自定义类，这个类一定在CLASSPATH中
class Member{
    @Override
    public String toString() {
        return "Member";
    }
}
public class TestDemo {
    public static void main(String[] args) throws Exception{

 System.out.println(Class.forName("Member").getClassLoader().loadClass("Member").newInstance());
    }
}
```

范例：在Desktop上建立Member.java文件

```
// 自定义类，这个类一定在CLASSPATH中
class Member{
    @Override
    public String toString() {
        return "Member";
    }
}
```

随后将此文件用javac编译后生成class文件。现在希望通过自定义的类加载器实现/Desktop/Member.class文件的加载。

范例：实现自定义类加载器

ClassLoader提供的类加载:

```
    protected final Class<?> defineClass(String name, byte[] b, int off, int len)
        throws ClassFormatError
```

```
import java.io.ByteArrayOutputStream;
import java.io.FileInputStream;
import java.io.InputStream;

// 自定义类加载器
class MyClassLoader extends ClassLoader {
    /**
     * 实现一个自定义的类加载器，传入类名称，通过指定路径加载
     * @param className 类名称
     * @return 返回的Class对象
     * @throws Exception
     */
    public Class<?> loadData(String className) throws Exception {
        // 加载类文件的信息
        byte[] classData = this.loadClassData() ;
        return super.defineClass(className,classData,0,classData.length) ;
    }
    /**
     * 通过指定的文件路径进行类的文件加载，实际上就是进行二进制文件读取
     * @return 类文件数据
     * @throws Exception
     */
    private byte[] loadClassData() throws Exception {
        InputStream input = new FileInputStream("/Users/yuisama/Desktop/Member.class") ;
        // 取得所有字节内容，放到内存中
        ByteArrayOutputStream bos = new ByteArrayOutputStream() ;
        // 读取缓冲区
        byte[] data = new byte[20] ;
        int temp = 0 ;
        while ((temp = input.read(data))!=-1){
            bos.write(data,0,temp) ;
        }
        byte[] result = bos.toByteArray() ;
        input.close() ;
        bos.close() ;
        return result ;
    }

}
public class TestDemo {
    public static void main(String[] args) throws Exception{

        Class<?> cls = new MyClassLoader().loadData("Member") ;
```

```
        System.out.println(cls.getClassLoader()) ;
        System.out.println(cls.getClassLoader().getParent()) ;
        System.out.println(cls.getClassLoader().getParent().getParent()) ;
        System.out.println(cls.newInstance());
    }
}
```

类加载器给用户提供最大的帮助为：可以通过动态的路径进行类的加载操作。

比较两个类相等的前提：必须是由同一个类加载器加载的前提下才有意义。否则，即使两个类来源于同一个Class文件，被同一个虚拟机加载，只要加载他们的类加载器不同，那么这两个类注定不想等。

范例：观察不同类加载器加载同一个class的情况

5.反射与代理设计模式

实际开发之中有两大核心设计模式：工厂设计模式与代理设计模式。

5.1 基础代理设计模式

代理设计模式的核心本质在于：一个接口有两个子类，一个负责真实业务，一个负责与真实业务有关的所有辅助性操作。按照这样的原则，一个基础的代理设计模式如下：

```
package www.bit.java.agency;

import java.lang.reflect.Constructor;

interface ISubject { // 核心操作接口
    public void eat() ; // 吃饭是核心业务
}
class RealSubject implements ISubject {
    @Override
    public void eat() {
        System.out.println("饿了要吃饭") ;
    }
}
class ProxySubject implements ISubject {
    private ISubject subject ;

    public ProxySubject(ISubject subject) {
        this.subject = subject;
    }
    public void prepare() {
        System.out.println("饭前收拾食材") ;
    }
    public void afterEat() {
        System.out.println("洗刷刷") ;
    }
    @Override
    public void eat() {
        this.prepare() ;
        this.subject.eat() ; // 核心吃
```

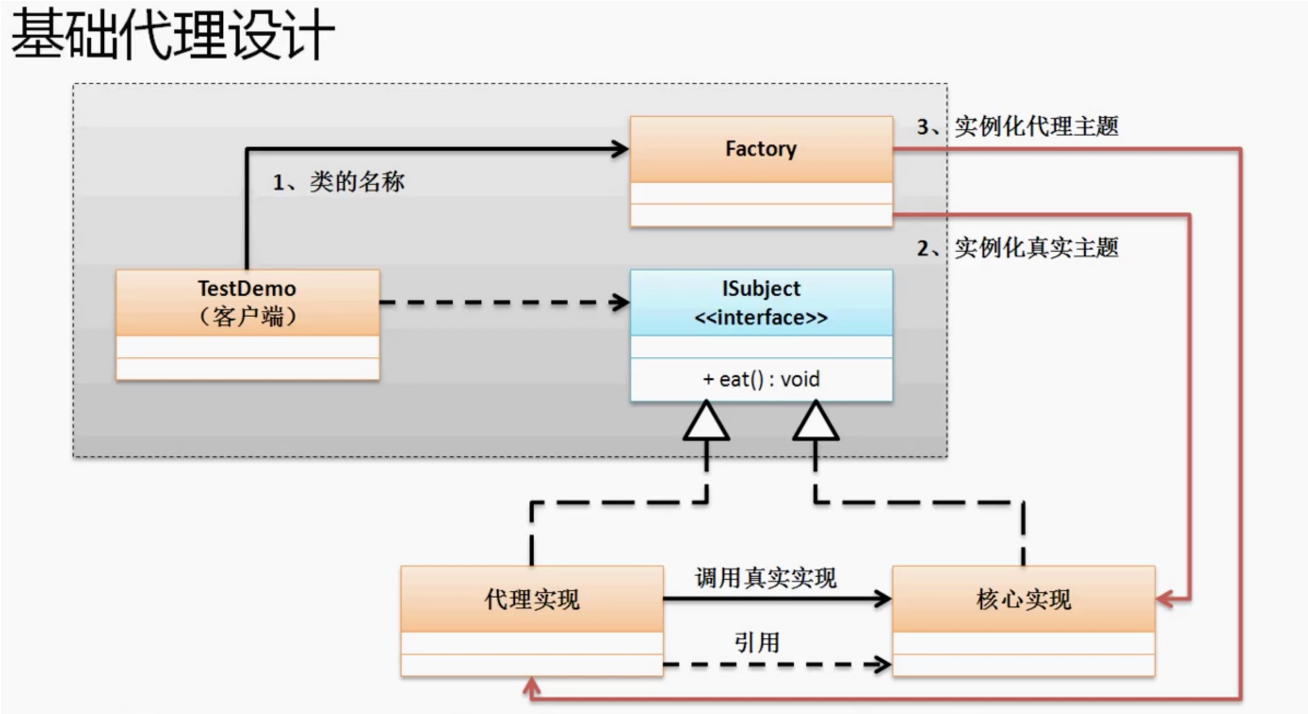
```
        this.afterEat() ;
    }
}
class Factory {
    private Factory(){}
    public static <T> T getInstance(String className) {
        T t = null ;
        try {
            t = (T) Class.forName(className).newInstance() ;
        } catch (InstantiationException e) {
            e.printStackTrace();
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
        return t ;
    }
    public static <T> T getInstance(String className,Object obj) {
        T t = null ;
        try {
            Constructor<?> cons =
Class.forName(className).getConstructor(obj.getClass().getInterfaces()[0]) ;
            t = (T) cons.newInstance(obj) ;
        } catch (Exception e) {
            e.printStackTrace();
        }
        return t ;
    }
}
public class TestDemo {
    public static void main(String[] args) {
        ISubject subject =
Factory.getInstance("www.bit.java.agency.ProxySubject",Factory.getInstance("www.bit.java.agency.
RealSubject"))  ;
        subject.eat() ;
    }
}
```

以上程序如果结合反射之后，整体的处理会非常繁琐。不光开发端，使用者使用起来也很麻烦。对于以上操作，客户端最多只需要关系代理是谁，实际业务是谁即可。

范例:修改工厂类与测试类

```
public static <T> T getInstance(String proxyClassName, String realClassName) {
        T t = null ;
        try {
            // 取得真实接口对象
            T realObj = getInstance(realClassName) ;
            Constructor<?> cons =
Class.forName(proxyClassName).getConstructor(realObj.getClass().getInterfaces()[0]) ;
            t = (T) cons.newInstance(realObj) ;
```

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

public static void main(String[] args) {
        ISubject subject =
Factory.getInstance("www.bit.java.agency.ProxySubject","www.bit.java.agency.RealSubject") ;
        subject.eat();

    }
```

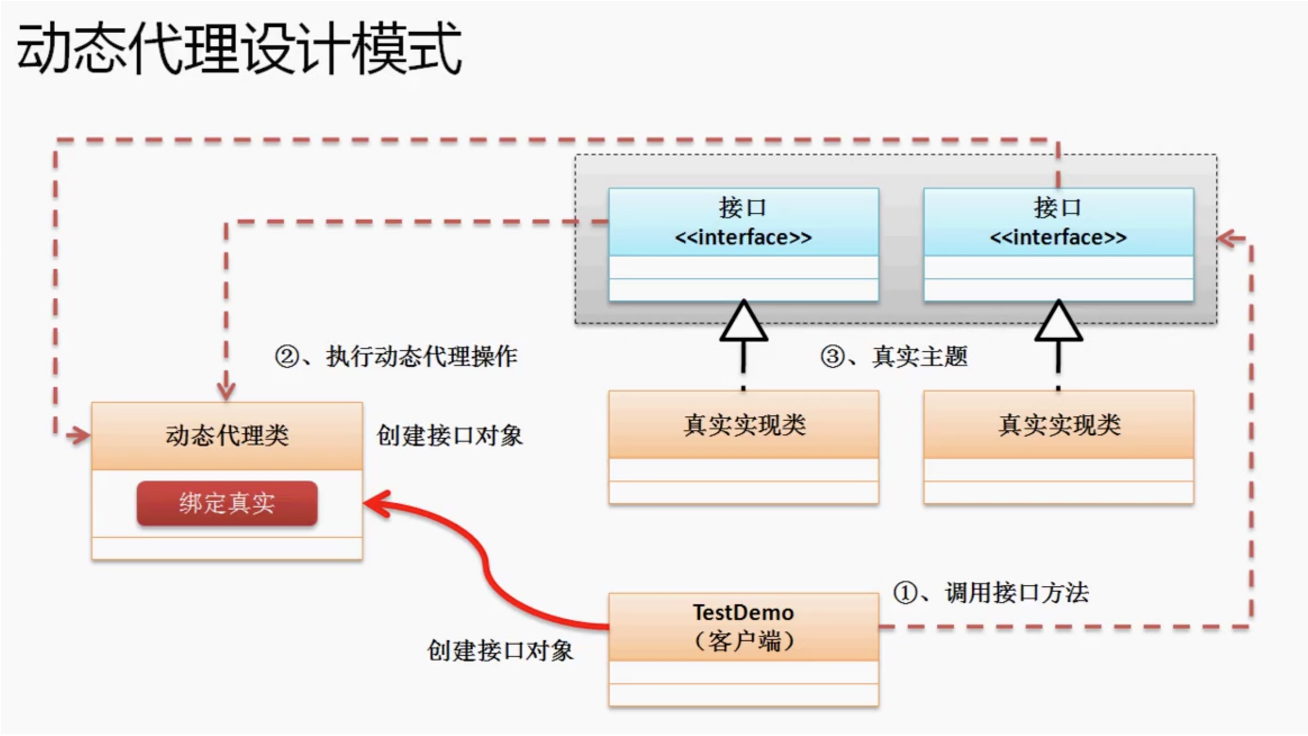


现在的问题是：在开发中并不知道项目会有多少个接口，如果这些接口都需要使用到代理模式，那么就意味着每一个接口都需要编写两个子类，再假设这些接口的代理类的功能几乎都一样。

之前的这种代理设计只是一种最简单的代理设计，所以这种代理设计只能够代理一个接口的子类对象，无法代理更多的接口子类对象。要想真正使用代理设计模式，我们需要引入动态代理设计模式

5.2 动态代理设计模式(核心)

动态代理模式的核心特点：*一个代理类可以代理所有需要被代理的接口的子类对象*



要想进行动态代理设计的实现，代理类不再具体实现于某一个接口。

```
/**
 * 动态代理实现的标识接口，只有实现此接口才具备有动态代理的功能
 */
public interface InvocationHandler {
    /**
     * invoke表示的是调用执行的方法，但是所有的代理类返回给用户的接口对象都属于代理对象
     * 当用户执行接口方法的时候所调用的实例化对象就是该代理主题动态创建的一个接口对象
     * @param proxy 表示被代理的对象信息
     * @param method 返回的是被调用的方法对象，取得了Method对象则意味着可以使用invoke()反射调用方法
     * @param args 方法中接收的参数
     * @return 方法的返回值
     * @throws Throwable 可能产生的异常
     */
    public Object invoke(Object proxy, Method method, Object[] args)
            throws Throwable;
}
```

如果要想进行对象的绑定，那么就需要使用一个Proxy程序类，这个程序类的功能是可以绑定所有需要绑定的接口子类对象，而且这些对象都是根据接口自动创建的,该类有一个动态创建绑定对象的方法:

```
    public static Object newProxyInstance(ClassLoader loader,Class<?>
[]interfaces,InvocationHandler h) throws IllegalArgumentException
```

范例：动态代理设计实现

```
package www.bit.java.agency;

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

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

interface ISubject { // 核心操作接口
    public void eat(String msg, int num) ; // 吃饭是核心业务
}
class RealSubject implements ISubject {
    @Override
    public void eat(String msg ,int num) {
        System.out.println("我要吃 "+num + "分量的 "+msg) ;
    }
}

/**
 * 动态代理类
 */
class ProxySubject implements InvocationHandler {
    // 绑定任意接口的对象，使用Object描述
    private Object target ;
    /**
     * 实现真实对象的绑定处理，同时返回代理对象
     * @param target
     * @return 返回一个代理对象(这个对象是根据接口定义动态创建生成的代理对象)
     */
    public Object bind(Object target) {
        // 保存真实主题对象
        this.target = target ;
        return
Proxy.newProxyInstance(target.getClass().getClassLoader(),target.getClass().getInterfaces(),this
) ;
    }

    public void preHandle() {
        System.out.println("[ProxySubject] 方法处理前") ;
    }

    public void afterHandle(){
        System.out.println("[ProxySubject] 方法处理后") ;
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        this.preHandle() ;
        // 反射调用方法
        Object ret = method.invoke(this.target,args) ;
        this.afterHandle() ;
        return ret;
    }
}

public class TestDemo {
    public static void main(String[] args) {

        ISubject subject =(ISubject) new ProxySubject().bind(new RealSubject()) ;
```

```
        subject.eat("宫保鸡丁",20) ;
    }
}
```

如果还想进一步的实现，就可以实现工厂类对实例化接口对象部分做进一步的隐藏处理。

5.3 cglib实现动态代理(了解)

动态代理实现完成了，但是所有的代理设计模式都会存在一个问题:离不开接口。

如果现在有新的需求：实现没有接口的动态代理模式。上面第二节我们讲了Proxy类提供的创建动态代理对象的方法:

```
    public static Object newProxyInstance(ClassLoader loader,Class<?>
[]interfaces,InvocationHandler h) throws IllegalArgumentException
```

此时如果要想实现这样的要求，就必须依靠另外的第三方组件包:cglib。这个开发包才能帮用户实现这类的要求。

范例：实现基于类操作的动态代理模式

```
package test ;

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

import java.lang.reflect.Method;

class Message {
    public void send() {
        System.out.println("www.bit.java.reflact") ;
    }
}
class ClassProxy implements MethodInterceptor { // 定义一个拦截器
    private Object target ; // 真实主题对象

    public ClassProxy(Object target) {
        this.target = target;
    }
    public void preHandle() {
        System.out.println("[ProxySubject] 方法处理前") ;
    }

    public void afterHandle(){
        System.out.println("[ProxySubject] 方法处理后") ;
    }

    public Object intercept(Object o, Method method, Object[] objects, MethodProxy methodProxy)
throws Throwable {
        this.preHandle() ;

        Object ret = method.invoke(this.target,objects) ;
```

```
        this.afterHandle() ;
        return ret ;
    }
}


public class TestClassProxy {
    public static void main(String[] args) {
        Message msg = new Message() ;
        // 负责代理关系的代理处理类
        Enhancer enhancer = new Enhancer() ;
        enhancer.setSuperclass(msg.getClass()) ;
        // 代理对象,以上就动态配置好了类之间的代理关系
        enhancer.setCallback(new ClassProxy(msg)) ;
        Message temp = (Message) enhancer.create() ;
        temp.send() ;
    }
}
```

该设计我们有所了解即可，我们要掌握的还是JDK本身支持的代理设计，包括以后Spring默认使用的依然是传统的代理设计模式。

总结

> 1. 动态代理设计模式必须掌握，必须清楚每一个类和接口的作用以及彼此之间的操作关系。
> 2. 有面试题让你去编写代理设计类:写上JDK提供的动态代理类，写上cglib实现的特点(绝对是亮点)

6.反射与Annotation

Annotation是颠覆性的开发技术，在以后的开发之中我们会见到大量的Annotation。Annotation的设计有一个前提：需要有代码容器，才可以实现自定义的Annotation。

6.1 反射取得Annotation信息

Annotation注解可以定义在类或方法上，在学习反射的概念后，此时我们可以通过反射取得所定义的Annotation信息。在java.lang.reflect.AccessibleObject(java.lang.Class)类中提供有如下与Annotation有关的方法：

> 1. 取得全部Annotation: public Annotation[] getAnnotations()
> 2. 取得指定的Annotation: public T getAnnotation(Class annotationClass)

范例：取得定义在类上的Annotation

```
package www.bit.java.annotation;

import java.io.Serializable;
import java.lang.annotation.Annotation;

@SuppressWarnings("serial")
@Deprecated
class Member implements Serializable {}


public class TestAnnotation {
```

```
    public static void main(String[] args) {
        Annotation[] ant = Member.class.getAnnotations() ;
        for (Annotation a: ant) {
            System.out.println(a);
        }
    }
}
```

Annotation本身有自己的保存范围，不同的Annotation的返回也不同。所以此处只出现了一个Annotation。

范例：在方法上使用Annotation

```
package www.bit.java.annotation;

import java.io.Serializable;
import java.lang.annotation.Annotation;

@SuppressWarnings("serial")
@Deprecated
class Member implements Serializable {
    @Deprecated
    @Override
    public String toString() {
        return super.toString();
    }
}

public class TestAnnotation {
    public static void main(String[] args) {
        Annotation[] ant = new Annotation[0];
        try {
            ant = Member.class.getMethod("toString").getAnnotations();
        } catch (NoSuchMethodException e) {
            e.printStackTrace();
        }
        for (Annotation a: ant) {
            System.out.println(a);
        }
    }
}
```

通过上述代码可以发现，反射可以取得结构上定义的Annotation，Annotation的设计是离不开反射的。

6.2 自定义Annotation

要想自定义Annotation，首先需要解决的就是Annotation的作用范围。通过第一节的范例我们可以发现，不同的Annotation有自己的运行范围，而这些范围就在一个枚举类(java.lang.annotation.RetentionPolicy)中定义:

> 1. SOURCE:在源代码中出现的Annotation
> 2. CLASS:在*.class中出现的Annotation
> 3. RUNTIME:在类执行的时候出现的Annotation

范例：定义一个在运行时生效的Annotation

```
package www.bit.java.annotation;

import java.io.Serializable;
import java.lang.annotation.Annotation;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;

/**
 * 自定义一个Annotation
 */
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnnotation{
    public String name() ;
    public int age() ;
}

@Deprecated
@MyAnnotation(name = "yuisama" , age = 25)
class Member implements Serializable {
}

public class TestAnnotation {
    public static void main(String[] args) {

        Annotation[] ant = new Annotation[0];
        ant = Member.class.getAnnotations();
        for (Annotation a: ant) {
            System.out.println(a);
        }
    }
}
```

此时自定义的Annotation强制要求设置name和age属性内容，这样不方便用户使用。所以可以设置默认值，在定义Annotation时使用default设置默认值。

范例：使用默认值自定义Annotation

```
/**
 * 自定义一个Annotation
 */
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnnotation{
    public String name() default "yuisama" ;
    public int age() default 25 ;
}

@Deprecated
@MyAnnotation
class Member implements Serializable {
}
```

之前的操作都是取得全部Annotation信息，现在也可以取得某一个具体的Annotation信息。

范例：取得某个具体的Annotation

```
package www.bit.java.annotation;

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

/**
 * 自定义一个Annotation
 */
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnnotation{
    public String name() default "yuisama" ;
    public int age() default 25 ;
}

@Deprecated
@MyAnnotation
class Member implements Serializable {
}

public class TestAnnotation {
    public static void main(String[] args) {
        MyAnnotation ma = Member.class.getAnnotation(MyAnnotation.class) ;
        System.out.println("姓名 :" +ma.name()) ;
        System.out.println("年龄 :"+ma.age()) ;
    }
}
```

可以在Annotation中编写许多的属性信息。这里需要注意一点：Annotation的使用需要特殊环境，不是随意编写的，目前我们知道自定义的语法和形式即可。

6.3 Annotation与工厂设计模式

在之前编写的工厂类都是通过明确的类信息传递实现的实例化对象。

范例：用注解的形式配置工厂类

```
package www.bit.java.annotation;

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

/**
 * 自定义一个Annotation
 */
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnnotation{
   public Class<?> target() ;
}
interface IFruit {
    public void eat() ;
}
class Apple implements IFruit {
    @Override
    public void eat() {
        System.out.println("吃苹果") ;
    }
}
@MyAnnotation(target = Apple.class)
class Factory {
    public static <T> T getInstance() throws Exception{
        MyAnnotation mt = Factory.class.getAnnotation(MyAnnotation.class) ;
        return (T) mt.target().newInstance() ;
    }
}
public class TestAnnotation {
    public static void main(String[] args) throws Exception{
        IFruit fruit = Factory.getInstance() ;
        fruit.eat() ;
    }
}
```

通过代码可以发现Annotation可以很好的解决程序配置项的问题。程序可以通过Annotation配置的信息来实现不同的操作效果。

对于Annotation的开发大概清楚流程即可，暂时不作为重点。在JavaEE中会接触到许多开发框架提供的Annotation，到时会用即可。