（一）单例模式详解

上一章，我们学习了设计模式的概念，以及为什么要学习设计模式，还有在进行系统设计时应当遵守的六大原则，本章我们就来开始一一的学习GOF当中的二十三钟设计模式。

我一直在思考如何去诠释这么多设计模式，因为网上有很多现成的，可供学习的资料，我在想有什么地方可以让各位跟着我的节奏去学习，而不是网上的那些资料，优势在哪里，思考很久，我觉得唯一的优势，或者说我能有的优势，就是简单通俗易懂。

遵循着中心思想通俗易懂，我们首先来回顾一下单例模式为何要出现，又或者说什么样的类可以做成单例的。

在我的工作过程中，我发现所有可以使用单例模式的类都有一个共性，那就是这个类没有自己的状态，换句话说，这些类无论你实例化多少个，其实都是一样的，而且更重要的一点是，这个类如果有两个或者两个以上的实例的话，我的程序竟然会产生程序错误或者与现实相违背的逻辑错误。

这样的话，如果我们不将这个类控制成单例的结构，应用中就会存在很多一模一样的类实例，这会非常浪费系统的内存资源，而且容易导致错误甚至一定会产生错误，所以我们单例模式所期待的目标或者说使用它的目的，是为了尽可能的节约内存空间，减少无谓的GC消耗，并且使应用可以正常运作。

我稍微总结一下，一般一个类能否做成单例，最容易区别的地方就在于，这些类，在应用中如果有两个或者两个以上的实例会引起错误，又或者我换句话说，就是这些类，在整个应用中，同一时刻，有且只能有一种状态。

一般实践当中，有很多应用级别的资源会被做成单例，比如配置文件信息，逻辑上来讲，整个应用有且只能在同在时间有一个，当然如果你有多个，这可能并不会引起程序级别错误，这里指的错误特指异常或者ERROR。但是当我们试图改变配置文件的时候，问题就出来了。

你有两种选择，第一种，将所有的实例全部更新成一模一样的状态。第二种，就是等着出现问题。

然而出现的问题大部分是逻辑层次上的错误，个人觉得这是比程序错误更加严重的错误，因为它不会告诉你空指针，不会告诉你非法参数，很多时候要等到影响到客户使用时才会被发现。

下面，我们就来看一下做成单例的几种方式。

第一种方式，我们来看一下最标准也是最原始的单例模式的构造方式。

```
public class Singleton {


    //一个静态的实例

    private static Singleton singleton;

    //私有化构造函数

    private Singleton(){}

    //给出一个公共的静态方法返回一个单一实例

    public static Singleton getInstance(){

        if (singleton == null) {

            singleton = new Singleton();

        }

        return singleton;

    }
```

（一）单例模式详解

上一章，我们学习了设计模式的概念，以及为什么要学习设计模式，还有在进行系统设计时应当遵守的六大原则，本章我们就来开始一一的学习GOF当中的二十三钟设计模式。

我一直在思考如何去诠释这么多设计模式，因为网上有很多现成的，可供学习的资料，我在想有什么地方可以让各位跟着我的节奏去学习，而不是网上的那些资料，优势在哪里，思考很久，我觉得唯一的优势，或者说我能有的优势，就是简单通俗易懂。

遵循着中心思想通俗易懂，我们首先来回顾一下单例模式为何要出现，又或者说什么样的类可以做成单例的。

在我的工作过程中，我发现所有可以使用单例模式的类都有一个共性，那就是这个类没有自己的状态，换句话说，这些类无论你实例化多少个，其实都是一样的，而且更重要的一点是，这个类如果有两个或者两个以上的实例的话，我的程序竟然会产生程序错误或者与现实相违背的逻辑错误。

这样的话，如果我们不将这个类控制成单例的结构，应用中就会存在很多一模一样的类实例，这会非常浪费系统的内存资源，而且容易导致错误甚至一定会产生错误，所以我们单例模式所期待的目标或者说使用它的目的，是为了尽可能的节约内存空间，减少无谓的GC消耗，并且使应用可以正常运作。

我稍微总结一下，一般一个类能否做成单例，最容易区别的地方就在于，这些类，在应用中如果有两个或者两个以上的实例会引起错误，又或者我换句话说，就是这些类，在整个应用中，同一时刻，有且只能有一种状态。

一般实践当中，有很多应用级别的资源会被做成单例，比如配置文件信息，逻辑上来讲，整个应用有且只能在同在时间有一个，当然如果你有多个，这可能并不会引起程序级别错误，这里指的错误特指异常或者ERROR。但是当我们试图改变配置文件的时候，问题就出来了。

你有两种选择，第一种，将所有的实例全部更新成一模一样的状态。第二种，就是等着出现问题。

然而出现的问题大部分是逻辑层次上的错误，个人觉得这是比程序错误更加严重的错误，因为它不会告诉你空指针，不会告诉你非法参数，很多时候要等到影响到客户使用时才会被发现。

下面，我们就来看一下做成单例的几种方式。

第一种方式，我们来看一下最标准也是最原始的单例模式的构造方式。

```
public class Singleton {



        //一个静态的实例

        private static Singleton singleton;

        //私有化构造函数

        private Singleton(){}

        //给出一个公共的静态方法返回一个单一实例

        public static Singleton getInstance(){

                if (singleton == null) {

                        singleton = new Singleton();

                }

                return singleton;

        }

}
```

这是在不考虑并发访问的情况下标准的单例模式的构造方式，这种方式通过几个地方来限制了我们取到的实例是唯一的。

1.静态实例，带有static关键字的属性在每一个类中都是唯一的。

2.限制客户端随意创造实例，即私有化构造方法，此为保证单例的最重要的一步。

3.给一个公共的获取实例的静态方法，注意，是静态的方法，因为这个方法是在我们未获取到实例的时候就要提供给客户端调用的，所以如果是非静态的话，那就变成一个矛盾体了，因为非静态的方法必须要拥有实例才可以调用。

4.判断只有持有的静态实例为null时才调用构造方法创造一个实例，否则就直接返回。

假如你去面试一家公司，给了你一道题，让你写出一个单例模式的例子，那么如果你是刚出大学校门的学生，你能写出上面这种示例，假设我是面试官的话，满分100的话，我会给90分，剩下的那10分算是给更优秀的人一个更高的台阶。但如果你是一个有过两三年工作经验的人，如果你写出上面的示例，我估计我最多给你30分，甚至心情要是万一不好的话可能会一分不给。

为什么同样的示例放到不同的人身上差别会这么大，就是因为前面我提到的那个情况，在不考虑并发访问的情况下，上述示例是没有问题的。

至于为什么在并发情况下上述的例子是不安全的呢，我在这里给各位制造了一个并发的例子，用来说明，上述情况的单例模式，是有可能造出来多个实例的，我自己测试了约莫100次左右，最多的一次，竟然造出了3个实例。下面给出代码，大约运行10次（并发是具有概率性的，10次只是保守估计，也可能一次，也可能100次）就会发现我们创造了不只一个实例。

```
import java.util.Collections;

import java.util.HashSet;
```

```
import java.util.Set;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;


public class TestSingleton {


	boolean lock ;


	public boolean isLock() {

		return lock;

	}


	public void setLock(boolean lock) {

		this.lock = lock;

	}


	public static void main(String[] args) throws
InterruptedException {

		final Set<String> instanceSet =
Collections.synchronizedSet(new HashSet<String>());

		final TestSingleton lock = new TestSingleton();
```

```
            lock.setLock(true);

            ExecutorService executorService =
Executors.newCachedThreadPool();

            for (int i = 0; i < 100; i++) {

                  executorService.execute(new Runnable() {



                        public void run() {

                              while (true) {

                                    if (!lock.isLock()) {

                                          Singleton singleton
= Singleton.getInstance();

instanceSet.add(singleton.toString());

                                          break;

                                    }

                              }

                        }

                  });

            }

            Thread.sleep(5000);

            lock.setLock(false);

            Thread.sleep(5000);
```

```
            System.out.println("------并发情况下我们取到的实例------");

            for (String instance : instanceSet) {

                  System.out.println(instance);

            }

            executorService.shutdown();

      }

}
```

我在程序中同时开启了100个线程，去访问getInstance方法，并且把获得实例的toString方法获得的实例字符串装入一个同步的set集合，set集合会自动去重，所以看结果如果输出了两个或者两个以上的实例字符串，就说明我们在并发访问的过程中产生了多个实例。

程序当中让main线程睡眠了两次，第一次是为了给足够的时间让100个线程全部开启，第二个是将锁打开以后，保证所有的线程都已经调用了getInstance方法。

好了，这下我们用事实说明了，上述的单例写法，我们是可以创造出多个实例的，至于为什么在这里要稍微解释一下，虽说我一直都喜欢用事实说话，包括看书的时候，我也不喜欢作者跟我解释为什么，而是希望给我一个例子，让我自己去印证。

造成这种情况的原因是因为，当并发访问的时候，第一个调用getInstance方法的线程A，在判断完singleton是null的时候，线程A就进入了if块准备创造实例，但是同时另外一个线程B在线程A还未创造出实例之前，就又进行了singleton是否为null的判断，这时singleton依然为null，所以线程B也会进入if块去创造实例，这时问题就出来了，有两个线程都进入了if块去创造实例，结果就造成单例模式并非单例。

为了避免这种情况，我们就要考虑并发的情况了，我们最容易想到的方式应该是下面这样的方式，直接将整个方法同步。

```
public class BadSynchronizedSingleton {

	//一个静态的实例

	private static BadSynchronizedSingleton
synchronizedSingleton;

	//私有化构造函数

	private BadSynchronizedSingleton(){}

	//给出一个公共的静态方法返回一个单一实例

	public synchronized static BadSynchronizedSingleton
getInstance(){

		if (synchronizedSingleton == null) {

			synchronizedSingleton = new
BadSynchronizedSingleton();

		}

		return synchronizedSingleton;

	}

}
```

上面的做法很简单，就是将整个获取实例的方法同步，这样在一个线程访问这个方法时，其它所有的线程都要处于挂起等待状态，倒是避免了刚才同步访问创造出多个实例的危险，但是我只想说，这样的设计实在是糟糕透了，这样会造成很多无谓的等待，所以为了表示我的愤怒，我在类名上加入Bad。

其实我们同步的地方只是需要发生在单例的实例还未创建的时候，在实例创建以后，获取实例的方法就没必要再进行同步控制了，所以我们将上面的示例改为很多教科书中标准的单例模式版本，也称为双重加锁。

```
public class SynchronizedSingleton {


	//一个静态的实例
	private static SynchronizedSingleton
synchronizedSingleton;
	//私有化构造函数
	private SynchronizedSingleton(){}
	//给出一个公共的静态方法返回一个单一实例
	public static SynchronizedSingleton getInstance(){
		if (synchronizedSingleton == null) {
			synchronized (SynchronizedSingleton.class)
{
				if (synchronizedSingleton == null) {
					synchronizedSingleton = new
SynchronizedSingleton();
				}
			}
		}
		return synchronizedSingleton;
	}
}
```

这种做法与上面那种最无脑的同步做法相比就要好很多了，因为我们只是在当前实例为null，也就是实例还未创建时才进行同步，否则就直接返回，这样就节省了很多无谓的线程等待时间，值得注意的是在同步块中，我们再次判断了synchronizedSingleton是否为null，解释下为什么要这样做。

假设我们去掉同步块中的是否为null的判断，有这样一种情况，假设A线程和B线程都在同步块外面判断了synchronizedSingleton为null，结果A线程首先获得了线程锁，进入了同步块，然后A线程会创造一个实例，此时synchronizedSingleton已经被赋予了实例，A线程退出同步块，直接返回了第一个创造的实例，此时B线程获得线程锁，也进入同步块，此时A线程其实已经创造好了实例，B线程正常情况应该直接返回的，但是因为同步块里没有判断是否为null，直接就是一条创建实例的语句，所以B线程也会创造一个实例返回，此时就造成创造了多个实例的情况。

经过刚才的分析，貌似上述双重加锁的示例看起来是没有问题了，但如果再进一步深入考虑的话，其实仍然是有问题的。

如果我们深入到JVM中去探索上面这段代码，它就有可能（注意，只是有可能）是有问题的。

因为虚拟机在执行创建实例的这一步操作的时候，其实是分了好几步去进行的，也就是说创建一个新的对象并非是原子性操作。在有些JVM中上述做法是没有问题的，但是有些情况下是会造成莫名的错误。

首先要明白在JVM创建新的对象时，主要要经过三步。

1.分配内存

2.初始化构造器

3.将对象指向分配的内存的地址

这种顺序在上述双重加锁的方式是没有问题的，因为这种情况下JVM是完成了整个对象的构造才将内存的地址交给了对象。但是如果2和3步骤是相反的（2和3可能是相反的是因为JVM会针对字节码进行调优，而其中的一项调优便是调整指令的执行顺序），就会出现问题了。

因为这时将会先将内存地址赋给对象，针对上述的双重加锁，就是说先将分配好的内存地址指给synchronizedSingleton，然后再进行初始化构造器，这时候后面的线程去请求getInstance方法时，会认为synchronizedSingleton对象已经实例化了，直接返回一个引用。如果在初始化构造器之前，这个线程使用了synchronizedSingleton，就会产生莫名的错误。

所以我们在语言级别无法完全避免错误的发生，我们只有将该任务交给JVM，所以有一种比较标准的单例模式。如下所示。

```
package com.oneinstance;



public class InnerClassSingleton {

	public static Singleton getInstance(){

		return Singleton.singleton;

	}



	private static class Singleton{

		protected static Singleton singleton = new
Singleton();

	}

}
```

首先来说一下，这种方式为何会避免了上面莫名的错误，主要是因为一个类的静态属性只会在第一次加载类时初始化，这是JVM帮我们保证的，所以我们无需担心并发访问的问题。所以在初始化进行一半的时候，别的线程是无法使用的，因为JVM会帮我们强行同步这个过程。另外由于静态变量只初始化一次，所以singleton仍然是单例的。

上面这种写法是我们使用静态的内部类作为单例，这样不太符合我们的习惯。所以我们改为以下形式。

```
public class Singleton {

	private Singleton(){}

	public static Singleton getInstance(){
		return SingletonInstance.instance;
	}

	private static class SingletonInstance{

		static Singleton instance = new Singleton();

	}
}
```

好了，进行到这里，单例模式算是已经完成了。最终的产物就是如上述的形式。上述形式保证了以下几点。

1.Singleton最多只有一个实例，在不考虑反射强行突破访问限制的情况下。

2.保证了并发访问的情况下，不会发生由于并发而产生多个实例。

3.保证了并发访问的情况下，不会由于初始化动作未完全完成而造成使用了尚未正确初始化的实例。

以下为不太常用的方式，这里给出来只是给各位参考，不建议使用下述方式。

第一种，就是俗称的饿汉式加载。

```
public class Singleton {



	private static Singleton singleton = new Singleton();



	private Singleton(){}



	public static Singleton getInstance(){

		return singleton;

	}

}
```

上述方式与我们最后一种给出的方式类似，只不过没有经过内部类处理，这种方式最主要的缺点就是一旦我访问了Singleton的任何其他的静态域，就会造成实例的初始化，而事实是可能我们从始至终就没有使用这个实例，造成内存的浪费。

不过在有些时候，直接初始化单例的实例也无伤大雅，对项目几乎没什么影响，比如我们在应用启动时就需要加载的配置文件等，就可以采取这种方式去保证单例。

第二种我就不贴了，与双重锁定一模一样，只是给静态的实例属性加上关键字volatile，标识这个属性是不需要优化的。

这样也不会出现实例化发生一半的情况，因为加入了volatile关键字，就等于禁止了JVM自动的指令重排序优化，并且强行保证线程中对变量所做的任何写入操作对其他线程都是即时可见的。这里没有篇幅去介绍volatile以及JVM中变量访问时所做的具体动作，总之volatile会强行将对该变量的所有读和取操作绑定成一个不可拆分的动作。如果读者有兴趣的话，可以自行去找一些资料看一下相关内容。

不过值得注意的是，volatile关键字是在JDK1.5以及1.5之后才被给予了意义，所以这种方式要在JDK1.5以及1.5之后才可以使用，但仍然还是不推荐这种方式，一是因为代码相对复杂，二是因为由于JDK版本的限制有时候会有诸多不便。

好了，以上基本上就是常见的所有单例模式的构造方式，如果下次再有面试让你去写一个单例模式，有时间的话就把上面所有的全部写给面试官并一一将优劣讲给他听吧，这样的话估计offer已经离你不远了。

本次单例模式的分享就到此结束了，感谢各位的收看。

（二）代理模式详解（包含原理详解）

我特意将本系列改了下名字，原名是《设计模式学习之路》，原因是因为之前写过一篇《spring源码学习之路》，但是我感觉本次写二十三种设计模式，更多的还是分享给各位自己的理解，所以感觉学习之路有点不合适，另外，从本章开始，正式启用本人称呼，LZ。

好了，废话至此，本章接着讨论第二种要介绍的设计模式，代理模式。

LZ不希望写的东西与网络上的资料千篇一律，所以这一系列不会像很多典型文章一章，只是列出这个模式的定义以及一堆适用的情况，然后就是一堆这个模式的各个角色，对于这种罗列LZ并不反对，相比之下会比较清晰，但是如果脱离了实际，就会导致看的人特别是初学者觉得设计模式很陌生很遥远。

LZ并不反对这种教学式的标准模式，但说实话，LZ本人看这种帖子从来都感觉收获不大，看一遍看一遍，到现在都没记住那些各个适用的情况与一堆乱七八糟的角色。

所以LZ探讨代理模式，不会再按这个步骤进行，而是跟着自己的思维进行。

首先代理模式，可以分为两种，一种是静态代理，一种是动态代理。

两种代理从虚拟机加载类的角度来讲，本质上都是一样的，都是在原有类的行为基础上，加入一些多出的行为，甚至完全替换原有的行为。

静态代理采用的方式就是我们手动的将这些行为换进去，然后让编译器帮我们编译，同时也就将字节码在原有类的基础上加入一些其他的东西或者替换原有的东西，产生一个新的与原有类接口相同却行为不同的类型。

说归说，我们来真实的去试验一下，实验的话需要找一个示例，就拿我们的数据库连接来做例子吧。

我们都知道，数据库连接是很珍贵的资源，频繁的开关数据库连接是非常浪费服务器的CPU资源以及内存的，所以我们一般都是使用数据库连接池来解决这一问题，即创造一堆等待被使用的连接，等到用的时候就从池里取一个，不用了再放回去，数据库连接在整个应用启动期间，几乎是不关闭的，除非是超过了最大闲置时间。

但是在程序员编写程序的时候，会经常使用connection.close()这样的方法，去关闭数据库连接，而且这样做是对的，所以你并不能告诉程序员们说，你们使用连接都不要关了，去调用一个其他的类似归还给连接池的方法吧。这是不符合程序员的编程思维的，也很勉强，而且具有风险性，因为程序员会忘的。

解决这一问题的办法就是使用代理模式，因为代理模式可以替代原有类的行为，所以我们要做的就是替换掉connection的close行为。

下面是connection接口原有的样子，我去掉了很多方法，因为都类似，全贴上来占地方。

```
import java.sql.SQLException;

import java.sql.Statement;

import java.sql.Wrapper;



public interface Connectionextends Wrapper {

      Statement createStatement() throws SQLException;

      void close() throws SQLException;

}
```

这里只贴了两个方法，但是我们代理的精髓只要两个方法就能掌握，下面使用静态代理，采用静态代理我们通常会使用组合的方式，为了保持对程序猿是透明的，我们实现Connection这个接口， 如下所示。

```
import java.sql.SQLException;
```

```
import java.sql.Statement;


public class ConnectionProxy implements Connection{

	private Connection connection;

	public ConnectionProxy(Connection connection) {

		super();

		this.connection = connection;

	}


	public Statement createStatement() throws SQLException{

		return connection.createStatement();

	}


	public void close() throws SQLException{

		System.out.println("不真正关闭连接，归还给连接
池");

	}


}
```

我们在构造方法中让调用者强行传入一个原有的连接，接下来我们将我们不关心的方法，交给真正的Connection接口去处理，就像

createStatement方法一样，而我们将真正关心的close方法用我们自己希望的方式去进行。

此处为了更形象，LZ给出一个本人写的非常简单的连接池，意图在于表明实现的思路。下面我们来看一下连接池的变化，在里面注明了变化点。

```
import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.SQLException;

import java.util.LinkedList;



public class DataSource {



      private static LinkedList<Connection> connectionList =
new LinkedList<Connection>();



      static{

            try {

          Class.forName("com.mysql.jdbc.Driver");

            } catch (ClassNotFoundException e) {

                  e.printStackTrace();

            }

      }
```

```
	private static Connection createNewConnection() throws
SQLException{

		return
DriverManager.getConnection("url","username", "password");

	}



	private DataSource(){

		if (connectionList == null ||
connectionList.size() == 0) {

			for (int i = 0; i < 10; i++) {

				try {


connectionList.add(createNewConnection());

				} catch (SQLException e) {

					e.printStackTrace();

				}

			}

		}

	}



	public Connection getConnection() throws Exception{
```

```
            if (connectionList.size() > 0) {
                  //return connectionList.remove();   这是原有的方式，直接返回连接，这样可能会被程序员把连接给关闭掉
                  //下面是使用代理的方式，程序员再调用close时，就会归还到连接池
                  return new ConnectionProxy(connectionList.remove());
            }
            return null;
      }

      public void recoveryConnection(Connection connection){
            connectionList.add(connection);
      }

      public static DataSource getInstance(){
            return DataSourceInstance.dataSource;
      }

      private static class DataSourceInstance{
```

```
            private static DataSource dataSource = new
DataSource();



      }



}
```

连接池我们把它做成单例，所以假设是上述连接池的话，我们代理中的close方法可以再具体化一点，就像下面这样，用归还给连接池的动作取代关闭连接的动作。

```
public void close() throws SQLException{

DataSource.getInstance().recoveryConnection(connection);

}
```

好了，这下我们的连接池返回的连接全是代理，就算程序员调用了close方法也只会归还给连接池了。

我们使用代理模式解决了上述问题，从静态代理的使用上来看，我们一般是这么做的。

1，代理类一般要持有一个被代理的对象的引用。

2，对于我们不关心的方法，全部委托给被代理的对象处理。

3，自己处理我们关心的方法。

这种代理是死的，不会在运行时动态创建，因为我们相当于在编译期，也就是你按下CTRL+S的那一刻，就给被代理的对象生成了一个不可动态改变的代理类。

静态代理对于这种，被代理的对象很固定，我们只需要去代理一个类或者若干固定的类，数量不是太多的时候，可以使用，而且其实效果

比动态代理更好，因为动态代理就是在运行期间动态生成代理类，所以需要消耗的时间会更久一点。就像上述的情况，其实就比较适合使用静态代理。

下面介绍下动态代理，动态代理是JDK自带的功能，它需要你去实现一个InvocationHandler接口，并且调用Proxy的静态方法去产生代理类。

接下来我们依然使用上面的示例，但是这次该用动态代理处理，我们来试一下看如何做。

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

import java.lang.reflect.Method;

import java.lang.reflect.Proxy;

import java.sql.Connection;

public class ConnectionProxy implements InvocationHandler{

	private Connection connection;

	public ConnectionProxy(Connection connection) {

		super();

		this.connection = connection;

	}



	public Object invoke(Object proxy, Method method,
Object[] args) throws Throwable {

		//这里判断是Connection接口的close方法的话
```

```
            if
(Connection.class.isAssignableFrom(proxy.getClass()) &&
method.getName().equals("close")) {

                    //我们不执行真正的close方法

                    //method.invoke(connection, args);

                    //将连接归还连接池


DataSource.getInstance().recoveryConnection(connection);

                    return null;

            }else {

                    return method.invoke(connection, args);

            }

      }



      public Connection getConnectionProxy(){

            return (Connection)
Proxy.newProxyInstance(getClass().getClassLoader(), new
Class[]{Connection.class}, this);

      }

}
```

上面是我们针对connection写的动态代理，InvocationHandler接口只有一个invoke方法需要实现，这个方法是用来在生成的代理类用回调使用的，关于动态代理的原理一会做详细的分析，这里我们先只关注用法。很显然，动态代理是将每个方法的具体执行过程交给了我们在

invoke方法里处理。而具体的使用方法，我们只需要创造一个ConnectionProxy的实例，并且将调用getConnectionProxy方法的返回结果作为数据库连接池返回的连接就可以了。

上述便是我们针对connection做动态代理的方式，但是我们从中得不到任何好处，除了能少写点代码以外，因为这个动态代理还是只能代理Connection这一个接口，如果我们写出这种动态代理的方式的话，说明我们应该使用静态代理处理这个问题，因为它代表我们其实只希望代理一个类就好。从重构的角度来说，其实更简单点，那就是在你发现你使用静态代理的时候，需要写一大堆重复代码的时候，就请改用动态代理试试吧。

通常情况下，动态代理的使用是为了解决这样一种问题，就是我们需要代理一系列类的某一些方法，最典型的应用就是我们前段时间讨论过的springAOP，我们需要创造出一批代理类，切入到一系列类当中的某一些方法中。下面给出一个经常使用的动态代理方式。

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

import java.lang.reflect.Method;

import java.lang.reflect.Proxy;




public class DynamicProxy implements InvocationHandler{



	private Object source;



	public DynamicProxy(Object source) {

		super();
```

```
        this.source = source;
    }

    public void before(){
        System.out.println("在方法前做一些事，比如打开事务");
    }

    public void after(){
        System.out.println("在方法返回前做一些事，比如提交事务");
    }

public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
//假设我们切入toString方法，其他其实也是类似的，一般我们这里大部分是针对特定的方法做事情的，通常不会对类的全部方法切入
//比如我们常用的事务管理器，我们通常配置的就是对save,update,delete等方法才打开事务
        if (method.getName().equals("toString")) {
            before();
        }
```

```
            Object result = method.invoke(source, args);

            if (method.getName().equals("toString")) {

                  after();

            }

            return result;

      }



      public Object getProxy(){

            return
Proxy.newProxyInstance(getClass().getClassLoader(),
source.getClass().getInterfaces(), this);

      }

}
```

上述我做了一些注释，其实已经说明一些问题，这个代理类的作用是可以代理任何类，因为它被传入的对象是Object，而不再是具体的类，比如刚才的Connection，这些产生的代理类在调用toString方法时会被插入before方法和after方法。

动态代理有一个强制性要求，就是被代理的类必须实现了某一个接口，或者本身就是接口，就像我们的Connection。

道理其实很简单，这是因为动态代理生成的代理类是继承Proxy类的，并且会实现被你传入newProxyInstance方法的所有接口，所以我们可以将生成的代理强转为任意一个代理的接口或者Proxy去使用，但是Proxy里面几乎全是静态方法，没有实例方法，所以转换成Proxy意义不大，几乎没什么用。假设我们的类没有实现任何接口，那么就意味着你只能将生成的代理类转换成Proxy，那么就算生成了，其实也没什

么用，而且就算你传入了接口，可以强转，你也用不了这个没有实现你传入接口的这个类的方法。

你可能会说，假设有个接口A，那我将接口A传给newProxyInstance方法，并代理一个没实现接口A的类B，但类B与接口A有一样的方法可以吗？

答案是可以的，并且JDK的动态代理只认你传入的接口，只要你传入，你就可以强转成这个接口，这个一会解释，但是你无法在invoke方法里调用method.invoke方法，也就是说，你只能全部替换A接口的方法，而不能使用类B中原有与接口A方法描述相同的方法，这是因为invoke中传入的Method的class信息是接口A，而类B因为没实现接口A，所以无法执行传入的Method，会抛出非法参数异常。

下面我贴出测试代码，各位可以自己试一下，具体为何会这样是在后面解释的，这里不再多做解释。

先是一个普通接口。

```
public interface TestInterface {

      void method1();

      void method2();

      void method3();

}
```

然后是一个类，和接口一模一样的方法，但是就是没实现这个接口。

```
public class TestClass{

      public void method1() {

      System.out.println("TestClass.method1");

      }
```

```
	public void method2() {
System.out.println("TestClass.method2");
	}

	public void method3() {
System.out.println("TestClass.method3");
	}

}
```

下面是测试类。

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

public class DynamicProxy implements InvocationHandler{

	private Object source;

	public DynamicProxy(Object source) {
```

```
        super();

        this.source = source;

    }



    public Object invoke(Object proxy, Method method,
Object[] args) throws Throwable {

        System.out.println("接口的方法全部变成这样了");

        //这里source是TestClass，但是我们不能使用反射调用
它的方法，像下面这样，放开这一行会抛异常

        //return method.invoke(source, args);

        return null;

    }



    public static void main(String[] args) {

        //只要你传入就可以强转成功

        TestInterface object =   (TestInterface)
Proxy.newProxyInstance(ClassLoader.getSystemClassLoader(),
new Class[]{TestInterface.class}, new DynamicProxy(new
TestClass()));

        object.method1();

        object.method2();

        object.method3();
```

```
    }

}
```

上面我们运行就会发现接口的方法全部都只能输出一个很2的字符串了。如果是要继续使用TestClass的方法也不是不行，只要你确认你传入的类包括了所有你传入的接口的方法，只是没实现这些接口而已，那么你可以在invoke中这样使用。

```
public Object invoke(Object proxy, Method method, Object[] 
args) throws Throwable {

            System.out.println("before");

            Method sourceMethod = 
source.getClass().getDeclaredMethod(method.getName(), 
method.getParameterTypes());

            sourceMethod.setAccessible(true);

            Object result = sourceMethod.invoke(source, 
args);

            System.out.println("after");

            return result;

    }
```

这就与你实现接口的表现行为一致了，但是我们本来就只需要一句method.invoke就可以了，就因为没实现接口就要多写两行，所以这种突破JDK动态代理必须实现接口的行为就有点画蛇添足了。因为你本来就实现了该接口的方法，只是差了那一句implements而已。

上面写这个例子只是为了解释LZ当初的疑惑，因为LZ曾一度认为不实现接口就不能使用动态代理，现在想想那时候LZ有点2，呵呵。

好了，从现在开始，我们开始详细讲解动态代理的原理，这算是进阶篇，如果是新手的话，可以跳过下面的内容，因为现在还没必要知道

这些，而且弄不好会越看越蒙，不过仅仅是LZ个人建议，你要有耐心，完全可以继续看下去。

接下来我们结合源码去看一下，代理类是如何产生的，首先当然就是要进入Proxy的newProxyInstance方法，这里是产生代理的入口，源码如下

```
public static Object newProxyInstance(ClassLoader loader,
                                      Class<?>[] interfaces,
                                      InvocationHandler h)
    throws IllegalArgumentException
    {
    if (h == null) {
        throw new NullPointerException();
    }


    /*
     * Look up or generate the designated proxy class.
     */
    Class cl = getProxyClass(loader, interfaces);


    /*
     * Invoke its constructor with the designated
invocation handler.
```

```
        */
      try {
            Constructor cons =
cl.getConstructor(constructorParams);
            return (Object) cons.newInstance(new Object[] { h
});
      } catch (NoSuchMethodException e) {
            throw new InternalError(e.toString());
      } catch (IllegalAccessException e) {
            throw new InternalError(e.toString());
      } catch (InstantiationException e) {
            throw new InternalError(e.toString());
      } catch (InvocationTargetException e) {
            throw new InternalError(e.toString());
      }
}
```

这个方法其实很简单，首先获取了代理类的运行时Class引用，然后调用了这个Class中的构造方法，这个构造方法只有一个参数，正是InvocationHandler接口，由此产生了一个代理类的实例。那么关键的地方就在于如何获取的代理类运行时的class信息的呢？我们进入getProxyClass方法看一下。为了方便起见，我直接加注释，这个方法需要解释的地方比较多。

```
public static Class<?> getProxyClass(ClassLoader loader,

                  Class<?>... interfaces) throws
IllegalArgumentException {

            //如果传入的接口长度大于65535就抛出异常，我去你
妹。。。

            if (interfaces.length > 65535) {

                  throw new
IllegalArgumentException("interface limit exceeded");

            }



            Class proxyClass = null;



            /* collect interface names to use as key for
proxy class cache */

            String[] interfaceNames = new
String[interfaces.length];



            Set interfaceSet = new HashSet(); // for
detecting duplicates



            for (int i = 0; i < interfaces.length; i++) {

                  /*
```

```
                    * Verify that the class loader resolves
the name of this interface

                    * to the same Class object.

                    */

                  String interfaceName =
interfaces[i].getName();

                  Class interfaceClass = null;

                  try {

                        //加载每一个接口的运行时Class信息

                        interfaceClass =
Class.forName(interfaceName, false, loader);

                  } catch (ClassNotFoundException e) {

                  }

                  //如果采用你传入的类加载器载入的Class和你传
入的Class不相等则抛出异常

                  if (interfaceClass != interfaces[i]) {

                        throw new
IllegalArgumentException(interfaces[i]

                                    + " is not visible from
class loader");

                  }



                  //如果你传入的不是接口抛出异常
```

```
                /*
                  * Verify that the Class object actually
represents an interface.
                  */
                if (!interfaceClass.isInterface()) {
                      throw new
IllegalArgumentException(interfaceClass.getName()
                                    + " is not an
interface");
                }


                //如果你传入的接口重复抛异常
                /*
                  * Verify that this interface is not a
duplicate.
                  */
                if (interfaceSet.contains(interfaceClass))
{
                      throw new
IllegalArgumentException("repeated interface: "
                                    +
interfaceClass.getName());
                }
```

```
                interfaceSet.add(interfaceClass);


                interfaceNames[i] = interfaceName;
            }


            /*
             * Using string representations of the proxy interfaces as keys in the
             * proxy class cache (instead of their Class objects) is sufficient
             * because we require the proxy interfaces to be resolvable by name
             * through the supplied class loader, and it has the advantage that
             * using a string representation of a class makes for an implicit weak
             * reference to the class.
             */
            Object key = Arrays.asList(interfaceNames);


            /*
             * Find or create the proxy class cache for the class loader.
```

```
            */
        Map cache;
        synchronized (loaderToCache) {
            //这个是为了存储每一个类加载器所载入过的代理接口的代理类
            cache = (Map) loaderToCache.get(loader);
            if (cache == null) {
                cache = new HashMap();
                loaderToCache.put(loader, cache);
            }
            /*
             * This mapping will remain valid for the duration of this method,
             * without further synchronization, because the mapping will only be
             * removed if the class loader becomes unreachable.
             */
        }

        /*
```

```
             * Look up the list of interfaces in the proxy
class cache using the

             * key. This lookup will result in one of three
possible kinds of

             * values: null, if there is currently no proxy
class for the list of

             * interfaces in the class loader, the
pendingGenerationMarker object,

             * if a proxy class for the list of interfaces
is currently being

             * generated, or a weak reference to a Class
object, if a proxy class

             * for the list of interfaces has already been
generated.

             */

            synchronized (cache) {

                /*

                  * Note that we need not worry about
reaping the cache for entries

                  * with cleared weak references because if
a proxy class has been

                  * garbage collected, its class loader
will have been garbage

                  * collected as well, so the entire cache
will be reaped from the
```

```
                * loaderToCache map.
                */
            do {
                //检查是否有生成好的代理
                Object value = cache.get(key);
                if (value instanceof Reference) {
                    proxyClass = (Class)
((Reference) value).get();
                }
                //有的话直接返回
                if (proxyClass != null) {
                    // proxy class already
generated: return it
                    return proxyClass;
                //否则看一下这个代理类是不是正在构造
中，是的话就在cache对象上等待
                } else if (value ==
pendingGenerationMarker) {
                    // proxy class being generated:
wait for it
                    try {
                        cache.wait();
```

```
                            } catch (InterruptedException
e) {
                                /*
                                 * The class generation
that we are waiting for should
                                 * take a small, bounded
time, so we can safely ignore
                                 * thread interrupts
here.
                                 */
                            }
                            continue;
                        //如果没有现成的，也没有创造中的，那
就开始创造代理类
                        } else {
                            /*
                             * No proxy class for this
list of interfaces has been
                             * generated or is being
generated, so we will go and
                             * generate it now. Mark it as
pending generation.
                             */
```

```
                                //将当前代理类置为正在构造中，
并直接退出循环

                                cache.put(key,
pendingGenerationMarker);

                                break;

                        }

                } while (true);

            }



            try {

                String proxyPkg = null; // package to
define proxy class in



                //这一段是看你传入的接口中有没有不是public
的接口，如果有，这些接口必须全部在一个包里定义的，否则抛异常

                /*

                 * Record the package of a non-public
proxy interface so that the

                 * proxy class will be defined in the same
package. Verify that all

                 * non-public proxy interfaces are in the
same package.

                 */
```

```
                for (int i = 0; i < interfaces.length; i++)
{
                    int flags =
interfaces[i].getModifiers();
                    if (!Modifier.isPublic(flags)) {
                        String name =
interfaces[i].getName();
                        int n = name.lastIndexOf('.');
                        String pkg = ((n == -1) ? "" :
name.substring(0, n + 1));
                        if (proxyPkg == null) {
                            proxyPkg = pkg;
                        } else if
(!pkg.equals(proxyPkg)) {
                            thrownew
IllegalArgumentException(
"non-public interfaces from different packages");
                        }
                    }
                }


                if (proxyPkg == null) { // if no non-public
proxy interfaces,
```

```
                        proxyPkg = ""; // use the unnamed
package
                    }


                    {
                        /*
  * Choose a name for the proxy class to generate.
                          */
                        long num;
                        synchronized (nextUniqueNumberLock) {
                              num = nextUniqueNumber++;
                        }
                        //生成一个随机代理类名
                        String proxyName = proxyPkg +
proxyClassNamePrefix + num;
                        /*
                          * Verify that the class loader
hasn't already defined a class
                          * with the chosen name.
                          */
```

//这一句就是重中之重了，生成代理类的class文件，这就是JDK动态代理的原理了，通过动态生成class文件来产生的代理类

//这个generateProxyClass方法下面会着重介绍

/*

* Generate the specified proxy class.

*/

byte[] proxyClassFile = ProxyGenerator.generateProxyClass(

proxyName, interfaces);

try {

//得到class文件二进制流后，直接载入代理类

proxyClass = defineClass0(loader, proxyName,

proxyClassFile, 0, proxyClassFile.length);

} catch (ClassFormatError e) {

/*

* A ClassFormatError here means that (barring bugs in the

* proxy class generation code) there was some other invalid

```
                                * aspect of the arguments
supplied to the proxy class

                                * creation (such as virtual
machine limitations exceeded).

                                */

                              throw new
IllegalArgumentException(e.toString());

                        }

                  }

                  //proxyClasses这个Map是为了来判断是不是代理
的Class

                  // add to set of all generated proxy
classes, for isProxyClass

                  proyClasses.put(proxyClass, null);



            } finally {

                  /*

                    * We must clean up the "pending
generation" state of the proxy

                    * class cache entry somehow. If a proxy
class was successfully

                    * generated, store it in the cache (with
a weak reference);
```

```
                    * otherwise, remove the reserved entry.
In all cases, notify all
                    * waiters on reserved entries in this
cache.
                    */
                  synchronized (cache) {
                        if (proxyClass != null) {
                              //最终将生成的代理用弱引用包装
起来放到cache当中
                              cache.put(key, new
WeakReference(proxyClass));
                        } else {
                              //如果代理类是空则移除代理的接
口所代表的key值
                              cache.remove(key);
                        }
                        //通知正在等待在cache对象上的线程，告
诉他们可以继续了，代理Class加载完毕了
                        cache.notifyAll();
                  }
            }
            return proxyClass;
      }
```

上面基本上已经解释的很清楚了，下面就是去看一下byte[] proxyClassFile = ProxyGenerator.generateProxyClass(proxyName, interfaces)这句话是如何处理的，也就是如何得到的代理类的class文件的，我们进去源码看一下，我依然会加上注释。

```
public static byte[] generateProxyClass(String paramString, Class[] paramArrayOfClass) {

            //新建一个ProxyGenerator实例，传入类名和接口数组

            ProxyGenerator localProxyGenerator = new ProxyGenerator(paramString, paramArrayOfClass);

            //这个才是真正生成class文件的地方

            final byte[] arrayOfByte = localProxyGenerator.generateClassFile();

            //看保存生成文件的标志是否为真，如果是就将class文件生成到本地，生成时要检查权限

            if (saveGeneratedFiles) {

                  AccessController.doPrivileged(new PrivilegedAction() {

                        public Object run() {

                              try {

                                    FileOutputStream localFileOutputStream = new FileOutputStream(ProxyGenerator.dotToSlash(this.val$name) + ".class");

localFileOutputStream.write(arrayOfByte);

localFileOutputStream.close();

                                    return null;
```

```
                                } catch (IOException
localIOException) {

                                        thrownew InternalError(
"I/O exception saving generated file: " + localIOException);

                                }

                        }



                });

            }



            return arrayOfByte;

      }
```

我们继续跟踪到localProxyGenerator.generateClassFile()这一句当中，依然会加上注释。

```
private byte[] generateClassFile() {

            //addProxyMethod方法，就是将方法都加入到一个列表
中，并与对应的class对应起来

            //这里给Object对应了三个方法hashCode，toString和
equals

            addProxyMethod(hashCodeMethod, Object.class);

            addProxyMethod(equalsMethod, Object.class);

            addProxyMethod(toStringMethod, Object.class);
```

```
            //同样将接口与接口下的方法对应起来

            for (int i = 0; i < this.interfaces.length; i++)
{

                  localObject1 =
this.interfaces[i].getMethods();

                  for (int k = 0; k < localObject1.length;
k++) {

                        addProxyMethod(localObject1[k],
this.interfaces[i]);

                  }



            }

            //检查所有代理方法的返回类型

            for (Iterator localIterator1 =
this.proxyMethods.values().iterator(); localIterator1

                        .hasNext();) {

                  localObject1 = (List)
localIterator1.next();

                  checkReturnTypes((List) localObject1);

            }



            Object localObject2;
```

try {

//方法中加入构造方法，这个构造方法只有一个，就是一个带有InvocationHandler接口的构造方法

//这个才是真正给class文件，也就是代理类加入方法了，不过还没真正处理，只是先加进来等待循环，构造方法在class文件中的名称描述是<init>

this.methods.add(generateConstructor());

//循环代理方法

for (localIterator1 = this.proxyMethods.values().iterator(); localIterator1

.hasNext();) {

localObject1 = (List) localIterator1.next();

for (localIterator2 = ((List) localObject1).iterator(); localIterator2

.hasNext();) {

localObject2 = (ProxyMethod) localIterator2.next();

//给每一个代理方法加一个Method类型的属性，数字10是class文件的标识符，代表这些属性都是private static的

this.fields.add(new FieldInfo(((ProxyMethod) localObject2).methodFieldName,"Ljava/lang/reflect/Method;", 10));

//将每一个代理方法都加到代理类的方法中

this.methods.add(((ProxyMethod) localObject2).generateMethod());

}

}

Iterator localIterator2;

//加入一个静态初始化块，将每一个属性都初始化，这里静态代码块也叫类构造方法，其实就是名称为<clinit>的方法，所以加到方法列表

this.methods.add(generateStaticInitializer());

} catch (IOException localIOException1) {

throw new InternalError("unexpected I/O Exception");

}

//方法和属性个数都不能超过65535，包括刚才的接口个数也是这样，

//这是因为在class文件中，这些个数都是用4位16进制表示的，所以最大值是2的16次方-1

if (this.methods.size() > 65535) {

throw new IllegalArgumentException("method limit exceeded");

}

```
            if (this.fields.size() > 65535) {
                  throw new IllegalArgumentException("field
limit exceeded");
            }
            //这里是将类名中的.转成成斜线为了写入class文件。
            this.cp.getClass(dotToSlash(this.className));
            this.cp.getClass("java/lang/reflect/Proxy");
            for (int j = 0; j < this.interfaces.length; j++)
{

this.cp.getClass(dotToSlash(this.interfaces[j].getName()));
            }


            this.cp.setReadOnly();
            //这里开始真正的写class文件
            ByteArrayOutputStream localByteArrayOutputStream
= new ByteArrayOutputStream();
            Object localObject1 = new
DataOutputStream(localByteArrayOutputStream);
            try {
                  //写入class文件的标识号，标识这是一个class
文件
```

((DataOutputStream) localObject1).writeInt(-889275714);

//次版本号0

((DataOutputStream) localObject1).writeShort(0);

//主版本号，49代表的是JDK1.5

((DataOutputStream) localObject1).writeShort(49);

//这里写入的是常量池，包括一些属性名称，类名称，方法描述符，属性描述符等等，常量池在加载时会被放到方法区或者说永久代。

this.cp.write((OutputStream) localObject1);

//这里写入的是这个类的访问标识，49代表的是public final，也就是说JDK动态代理生成的代理类都是final的

((DataOutputStream) localObject1).writeShort(49);

//写入代理类的类名

((DataOutputStream) localObject1).writeShort(this.cp

.getClass(dotToSlash(this.className)));

//写入代理类的父类类名，也就是Proxy类，这个位置的类如果说是JAVA文件，相当于extend后面的类，也就是父类

((DataOutputStream) localObject1).writeShort(this.cp

```
.getClass("java/lang/reflect/Proxy"));
                    //写入代理类所实现的接口数量
                    ((DataOutputStream) localObject1)

.writeShort(this.interfaces.length);
                    //写入代理类所实现的接口类名，同样的，对于
JAVA文件来说，相当于implements后面的接口，也就是实现的接口
                    for (int m = 0; m < this.interfaces.length;
m++) {
                         ((DataOutputStream)
localObject1).writeShort(this.cp

.getClass(dotToSlash(this.interfaces[m].getName())));
                    }
                    //写入属性个数
                    ((DataOutputStream)
localObject1).writeShort(this.fields.size());
                    //写入属性描述
                    for (Iterator localIterator3 =
this.fields.iterator(); localIterator3
                              .hasNext();) {
                         localObject2 = (FieldInfo)
localIterator3.next();
```

```
                        ((FieldInfo) localObject2)
.write((DataOutputStream) localObject1);
                    }
                    //写入方法个数
                    ((DataOutputStream)
localObject1).writeShort(this.methods.size());
                    //写入方法描述，方法的code属性，以及构造方
法和类构造方法都在这里被写入了。
                    for (localIterator3 =
this.methods.iterator(); localIterator3
                                .hasNext();) {
                        localObject2 = (MethodInfo)
localIterator3.next();
                        ((MethodInfo) localObject2)
                                    .write((DataOutputStream)
localObject1);
                    }
                    //结束
                    ((DataOutputStream)
localObject1).writeShort(0);
            } catch (IOException localIOException2) {
                    throw new InternalError("unexpected I/O
Exception");
```

```
            }


            return localByteArrayOutputStream.toByteArray();
        }
```

其实代理类的class文件并不复杂，还是有很多规律可循的，所以上述过程基本上可以让各位了解下JDK动态代理生成代理类时都生成了什么东西。

下面我们可以调用下JDK中生成Class文件的方法，并且写入到本地文件，然后使用反编译工具来看一下生成的代理类到底是什么样子的。下面是生成文件的测试类。我们暂且将生成的类名写成TestProxy，代理的接口就是我们上面的TestInterface。如下。

```
import java.io.File;

import java.io.FileOutputStream;

import java.io.IOException;



import sun.misc.ProxyGenerator;





public class CreateClassTest {



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

```
            byte[] classFile =
ProxyGenerator.generateProxyClass("TestProxy", new Class[]
{TestInterface.class});

            File file = new File("F:/TestProxy.class");

            FileOutputStream fos = new
FileOutputStream(file);

            fos.write(classFile);

            fos.flush();

            fos.close();

      }



}
```

生成后，我们反编译过来会是如下格式的JAVA文件。我加入了注释，大致说明了下文件中生成的部分与刚才分析的时候写入的过程的对应关系。

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

import java.lang.reflect.Method;

import java.lang.reflect.Proxy;

import java.lang.reflect.UndeclaredThrowableException;



//public final的，继承Proxy，实现你传入的接口

public final class TestProxy extends Proxy
```

```
    implements TestInterface
{
    //private static 的Method属性，对应所有方法
    private static Method m1;
    private static Method m5;
    private static Method m3;
    private static Method m4;
    private static Method m0;
    private static Method m2;
    //唯一的构造方法，需要一个InvocationHandler接口传入
    public TestProxy(InvocationHandler paramInvocationHandler)
        throws
    {
        super(paramInvocationHandler);
    }
    //重写Object的三个方法
    public final boolean equals(Object paramObject)
        throws
    {
        try
```

```
      {
         return ((Boolean)this.h.invoke(this, m1, new
Object[] { paramObject })).booleanValue();
      }
      catch (Error|RuntimeException localError)
      {
         throw localError;
      }
      catch (Throwable localThrowable)
      {
         throw new
UndeclaredThrowableException(localThrowable);
      }
   }


   public final void method3()
      throws
   {
      try
      {
         this.h.invoke(this, m5, null);
```

```
            return;
        }
        catch (Error|RuntimeException localError)
        {
            throw localError;
        }
        catch (Throwable localThrowable)
        {
            throw new
UndeclaredThrowableException(localThrowable);
        }
    }
    //代理的三个方法，回调传入的InvocationHandler的invoke方法
    public final void method1()
        throws
    {
        try
        {
            this.h.invoke(this, m3, null);
            return;
        }
```

```
      catch (Error|RuntimeException localError)
      {
         throw localError;
      }
      catch (Throwable localThrowable)
      {
         throw new
UndeclaredThrowableException(localThrowable);
      }
   }


   public final void method2()
      throws
   {
      try
      {
         this.h.invoke(this, m4, null);
         return;
      }
      catch (Error|RuntimeException localError)
      {
```

```
            throw localError;
        }
        catch (Throwable localThrowable)
        {
            throw new
UndeclaredThrowableException(localThrowable);
        }
    }


    public final int hashCode()
        throws
    {
        try
        {
            return ((Integer)this.h.invoke(this, m0,
null)).intValue();
        }
        catch (Error|RuntimeException localError)
        {
            throw localError;
        }
```

```
      catch (Throwable localThrowable)
      {
         throw new
UndeclaredThrowableException(localThrowable);
      }
   }


   public final String toString()
      throws
   {
      try
      {
         return (String)this.h.invoke(this, m2, null);
      }
      catch (Error|RuntimeException localError)
      {
         throw localError;
      }
      catch (Throwable localThrowable)
      {
```

```
        throw new
UndeclaredThrowableException(localThrowable);

      }

   }

   //这个就是刚才
this.methods.add(generateStaticInitializer());这一句话所加入
的静态初始化块，初始化每一个属性

   static

   {

      try

      {//每一个属性所代表的Method都是与上面加入代理方法列表时
与固定类绑定的，这是class文件中的格式，方法要与固定的类绑定

         m1 =
Class.forName("java.lang.Object").getMethod("equals", new
Class[] { Class.forName("java.lang.Object") });

         m5 =
Class.forName("TestInterface").getMethod("method3", new
Class[0]);

         m3 =
Class.forName("TestInterface").getMethod("method1", new
Class[0]);

         m4 =
Class.forName("TestInterface").getMethod("method2", new
Class[0]);

         m0 =
Class.forName("java.lang.Object").getMethod("hashCode", new
Class[0]);
```

```
        m2 =
Class.forName("java.lang.Object").getMethod("toString", new
Class[0]);

        return;

      }

      catch (NoSuchMethodException
localNoSuchMethodException)

      {

        throw new
NoSuchMethodError(localNoSuchMethodException.getMessage());

      }

      catch (ClassNotFoundException
localClassNotFoundException)

      {

        throw new
NoClassDefFoundError(localClassNotFoundException.getMessage()
);

      }

   }

}
```

看到这里就知道invoke方法是干嘛的了，其实就是生成的代理类对每一个方法的处理就是回调invoke方法。从生成的代理类源文件中也可以发现，每一个Method除了hashCode，toString和equals外，都是与所属的接口绑定的，所以这也就解释了为什么我们不实现这个接口，只传入进入的话，不能直接使用method.invoke，而是要转成source对应的method才可以调用。

好了，代理模式就分析到这里了，这里讲的更多的是代理模式的原理，对于如何使用并没有讲述太多，是因为代理模式在平时工作中用的虽然很多，但我们大多是使用的现成的，原因很简单，就是因为spring的AOP已经给我们弄了一个很好的动态代理的框架，所以我们几乎不需要自己去写，只要明白其原理，知道动态代理和静态代理主要处理的问题是那种的，知道在何处用，也能够用起来得心应手就可以了，当然这只是LZ个人之见，仅供参考。

（三）简单工厂模式详解

上一章我们着重讨论了代理模式，以及其实现原理，相信如果你看完了整篇博文，应该就对代理模式很熟悉了。

本章我们讨论简单工厂模式，LZ当初不小心夸下海口说不和网络上传播的教学式模式讲解雷同，所以现在感觉写一篇博文压力颇大。

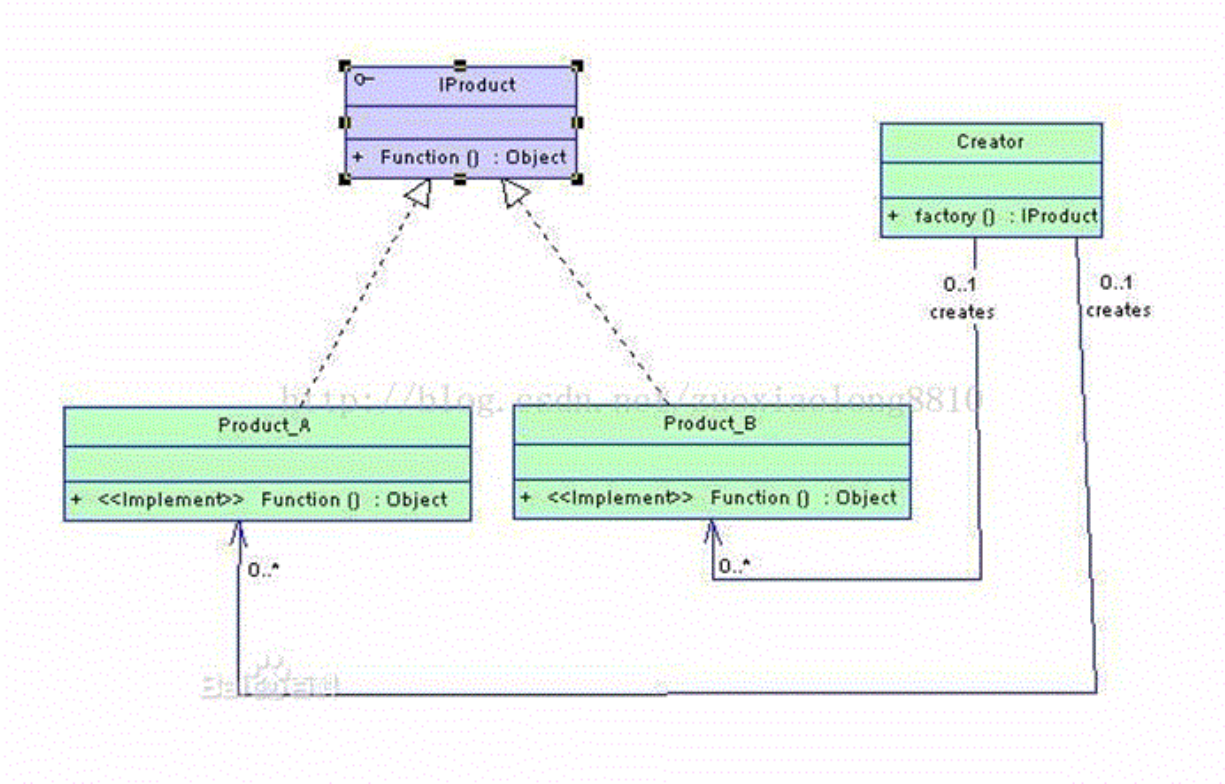
如何来介绍简单工厂呢，LZ着实费了不少心思，这个模式本身不复杂，但其实越是不复杂的模式越难写出特点，因为它太简单。

为了便于各位看后面例子的时候容易理解，LZ这里先给出引自其它地方的简单工厂模式的定义。

定义：从设计模式的类型上来说，简单工厂模式是属于创建型模式，又叫做静态工厂方法（Static Factory Method）模式，但不属于23种GOF设计模式之一。简单工厂模式是由一个工厂对象决定创建出哪一种产品类的实例。简单工厂模式是工厂模式家族中最简单实用的模式，可以理解为是不同工厂模式的一个特殊实现。

定义中最重要的一句话就是，由一个工厂对象决定创建出哪一种产品类的实例，这个LZ在下面会专门举一个现实应用中的例子去展现。

另外给出简单工厂模式的类图，本类图以及上面的定义都引自百度百科。



可以看出，上面总共有三种类，一个是工厂类Creator，一个是产品接口IProduct，一个便是具体的产品，例如产品A和产品B，这之中，工厂类负责整个创建产品的逻辑判断，所以为了使工厂类能够知道我们需要哪一种产品，我们需要在创建产品时传递给工厂类一个参数，去表明我们想要创建哪种产品。

下面LZ将上面的类图转化为更为简单的JAVA代码，便于清晰的展示上面类图中各个类之间的关系。

首先是产品接口。

```
public interface IProduct {

      public void method();

}
```

两个具体的产品。

```
public class ProductA implements IProduct{
```

```
	public void method() {
		System.out.println("产品A方法");
	}
}

public class ProductB implements IProduct{

	public void method() {
		System.out.println("产品B方法");
	}
}
```

下面是工厂类。

```
public class Creator {
	private Creator(){}
	public static IProduct createProduct(String productName){
		if (productName == null) {
			return null;
		}
		if (productName.equals("A")) {
			return new ProductA();
```

```
            }else if (productName.equals("B")) {
                  return new ProductB();
            }else {
                  return null;
            }
      }
}
```

最终客户端调用，并显示结果。

```
public class Client {

      public static void main(String[] args) {
            IProduct product1 = Creator.createProduct("A");
            product1.method();

            IProduct product2 = Creator.createProduct("B");
            product2.method();
      }
}
```

上面便是整个类图转换为JAVA代码的简单例子，将上面标准的简单工厂模式的类图和代码给出，是为了一些新手先熟悉一下这个设计模式

的大体框架，方便我们下面使用实际的例子去阐述的时候更加容易理解。

下面LZ就找一个各位基本上都使用过或者将来要使用的一个例子来说明简单工厂模式，我们去模拟一个简单的struts2的功能。

LZ会自己制作一个简单的WEB项目来做例子，其中会忽略掉很多细节，目的是为了突出我们的简单工厂模式。

众所周知，我们平时开发web项目大部分是以spring作为平台，来集成各个组件，比如集成struts2来完成业务层与表现层的逻辑，集成hibernate或者ibatis来完成持久层的逻辑。

struts2在这个过程当中提供了分离数据持久层，业务逻辑层以及表现层的责任，有了Struts2，我们不再需要servlet，而是可以将一个普通的Action类作为处理业务逻辑的单元，然后将表现层交给特定的视图去处理，比如JSP,template等等。

我们来尝试着写一个非常非常简单的web项目，来看看在最原始的时候，也就是没有spring，struts2等等这些个开源框架的时候，我们都是怎么过的。

由于LZ会省略WEB架构过程当中的很多细节，所以最好是各位亲手做过一些项目，相对而言看起来会更有体会一些，不过LZ相信既然有兴趣来看设计模式，应该都基本上有过这种锻炼。

下面LZ把我们一个简单的WEB项目中需要的类都列出来，并加上简单的注释。

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



//假设这是一个小型的WEB项目，我们通常里面会有这些类



//这个类在代理模式出现过，是我们的数据源连接池，用来生产数据库连接。
```

```
class DataSource{}


//我们一般会有这样一个数据访问的基类，这个类要依赖于数据源

class BaseDao{}


//一般会有一系列这样的DAO去继承BaseDao，这一系列的DAO类便是数据持久层

class UserDao extends BaseDao{}

class PersonDao extends BaseDao{}

class EmployeeDao extends BaseDao{}


//我们还会有一系列这样的servlet，他们通常依赖于各个Dao类，这一系列servlet便是我们的业务层

class LoginServlet extends HttpServlet{}

class LoginOutServlet extends HttpServlet{}

class RegisterServlet extends HttpServlet{}


//我们通常还会有HTML页面或者JSP页面，但是这个本次不在考虑范围内，这便是表示层。
```

以上是我们小型WEB项目大体的结构，可以看到LZ写了三个Servlet，没有写具体的实现到底如何，但是不难猜测，三个servlet的功能分别是进行登录，注销，以及注册新用户的功能。我们的servlet一般都是继承自HttpServlet，因为我们在web.xml配置servlet时，所写入的Class需要实现servlet接口，而我们通常采用的传输协议都是HTTP，

所以HttpServlet就是我们最好的选择了，它帮我们完成了基本的实现。

但是这样我们有很多限制，比如我们一个servlet一般只能负责一个单一的业务逻辑，因为我们所有的业务逻辑通常情况下都集中在doPost这样一个方法当中，可以想象下随着业务的增加，我们的servlet数量会高速增加，这样不仅项目的类会继续增加，最最恶心的是，我们每添加一个servlet就要在web.xml里面写一个servlet配置。

但是如果我们让一个Servlet负责多种业务逻辑的话，那我们需要在doPost方法中加入很多if判断，去判断当前的操作。

比如我们将上述三个servlet合一的话，你会在doPost出现以下代码。

```
protected void doPost(HttpServletRequest req,
HttpServletResponse resp) throws ServletException,
IOException {

        //我们加入一个操作的参数，来让servlet做出不同的业务
处理

            String operation = req.getParameter("operation");

            if (operation.equals("login")) {

System.out.println("login");

            }else if (operation.equals("register")) {

System.out.println("register");

            }else if (operation.equals("loginout")) {

System.out.println("loginout");

            }else {

                  throw new RuntimeException("invalid
operation");
```

```
        }

    }
```

这实在是非常烂的代码，因为每次你新加一个操作，都要修改doPost这个方法，而且多个业务逻辑都集中在这一个方法当中，会让代码很难维护与扩展，最容易想到的就是下列做法。（小提示：如果你的项目中出现了这种代码结构，请务必想办法去掉它，你完全可以尽量忘掉Java里还有elseif和swich）

```
protected void doPost(HttpServletRequest req,
HttpServletResponse resp) throws ServletException,
IOException {

        //我们加入一个操作的参数，来让servlet做出不同的业
务处理

        String operation = req.getParameter("operation");

        if (operation.equals("login")) {

            login();

        }else if (operation.equals("register")) {

            register();

        }else if (operation.equals("loginout")) {

            loginout();

        }else {

            throw new RuntimeException("invalid
operation");

        }

    }
```

```
	private void login(){
System.out.println("login");
	}

	private void register(){
		System.out.println("register");
	}

	private void loginout(){
System.out.println("loginout");
	}
```

这样会比第一种方式好一点，一个方法太长，实在不是什么好征兆，等到你需要修改这部分业务逻辑的时候，你就会后悔你当初的写法了，如果这段代码不是亲手写的，那请你放心的在心中吐糟吧，因为这实在不是一个合格的程序员应该写出的程序。

虽然我们已经将各个单一的业务逻辑拆分成方法，但这依然是违背单一原则这个小萝莉的，因为我们的servlet应该只是处理业务逻辑，而不应该还要负责与业务逻辑不相关的处理方法定位这样的责任，这个责任应该交给请求方，原本在三个servlet分别处理登陆，注销和注册的时候，其实就是这样的，作为请求方，只要是请求LoginServlet，就说明请求的人是要登陆，处理这个请求的servlet不需要再出现有关判断请求操作的代码。

所以我们需要想办法把判断的业务逻辑交给请求方去处理，回想下struts2的做法，我们可以简单模拟下struts2的做法。相信不少

同学应该都用过struts2，那么你肯定对以下配置很熟悉。

```
  <filter>

            <filter-name>struts2</filter-name>

            <filter-
class>org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAn
dExecuteFilter</filter-class>

      </filter>



      <filter-mapping>

            <filter-name>struts2</filter-name>

            <url-pattern>/*</url-pattern>

      </filter-mapping>
```

这是struts2最核心的filter，它的任务就是分派各个请求，根据请求的URL地址，去找到对应的处理该请求的Action。

我们来模拟一个分配请求的过滤器，它的任务就是根据用户的请求去产生响应的servlet处理请求，而这些servlet其实就是上面的例子当中的productA和productB这类的角色，也就是具体的产品，而它们实现的接口正是Servlet这个抽象的产品接口。

我们用这个过滤器来消除servlet在web.xml的配置，帮我们加快开发的速度，我们写出如下filter。

```
package com.web.filter;



import java.io.IOException;

import javax.servlet.Filter;
```

```
import javax.servlet.FilterChain;

import javax.servlet.FilterConfig;

import javax.servlet.Servlet;

import javax.servlet.ServletException;

import javax.servlet.ServletRequest;

import javax.servlet.ServletResponse;

import javax.servlet.http.HttpServletRequest;

import com.web.factory.ServletFactory;

//用来分派请求的filter

public class DispatcherFilter implements Filter{

      private static final String URL_SEPARATOR = "/";

      private static final String SERVLET_PREFIX =
"servlet/";

      private String servletName;

      public void init(FilterConfig filterConfig) throws
ServletException {}

public void destroy() {}



      public void doFilter(ServletRequest servletRequest,
ServletResponse servletResponse,FilterChain filterChain)
throws IOException, ServletException {

        parseRequestURI((HttpServletRequest) servletRequest);
```

```
            //这里为了体现我们本节的重点，我们采用一个工厂来
帮我们制造Action

            if (servletName != null) {

                  //这里使用的正是简单工厂模式，创造出一个
servlet，然后我们将请求转交给servlet处理

                  Servlet servlet =
ServletFactory.createServlet(servletName);

                  servlet.service(servletRequest,
servletResponse);

            }else {

filterChain.doFilter(servletRequest, servletResponse);

            }

      }



      //负责解析请求的URI，我们约定请求的格式必须
是/contextPath/servlet/servletName

      //不要怀疑约定的好处，因为LZ一直坚信一句话，约定优于配
置

      private void parseRequestURI(HttpServletRequest
httpServletRequest){

            String validURI =
httpServletRequest.getRequestURI().replaceFirst(httpServletRe
quest.getContextPath() + URL_SEPARATOR, "");

            if (validURI.startsWith(SERVLET_PREFIX)) {
```

```
                    servletName = validURI.split(URL_SEPARATOR)
[1];

            }

        }



}
```

这个filter需要在web.xml中加入以下配置，这个不多做介绍，直接贴上来。

```
<filter>

        <filter-name>dispatcherFilter</filter-name>

        <filter-
class>com.web.filter.DispatcherFilter</filter-class>

   </filter>

   <filter-mapping>

        <filter-name>dispatcherFilter</filter-name>

        <url-pattern>/servlet/*</url-pattern>

   </filter-mapping>
```

LZ在filter中稍微加入了一些注释，由于本章的重点是简单工厂模式，所以我们这里突出我们本章的主角，使用简单工厂来创造servlet去处理客户的请求，当然如果你是一个JAVA造诣比较深的程序猿，出于好奇进来一观，或许会对这种简单工厂模式的处理方式不屑一顾，不过我们不能偏离主题，我们的目的不是模拟一个struts2，而是介绍简单工厂。

下面给出我们的主角，我们的servlet工厂，它就相当于上面的Creator。

```
package com.web.factory;


import javax.servlet.Servlet;


import com.web.exception.ServletException;

import com.web.servlet.LoginServlet;

import com.web.servlet.LoginoutServlet;

import com.web.servlet.RegisterServlet;


public class ServletFactory {


	private ServletFactory(){}

	//一个servlet工厂，专门用来生产各个servlet，而我们生产的依据就是传入的servletName，

	//这个serlvetName由我们在filter截获，传给servlet工厂。

	public static Servlet createServlet(String servletName) {

		if (servletName.equals("login")) {

			return new LoginServlet();

		}else if (servletName.equals("register")) {
```

```
                return new RegisterServlet();
            }else if (servletName.equals("loginout")) {
                return new LoginoutServlet();
            }else {
                throw new ServletException("unknown servlet");
            }
        }
}
```

看到这里，是不是有点感觉了呢，我们一步一步去消除servlet的XML配置的过程，其实就是在慢慢的写出一个简单工厂模式，只是在这之中，抽象的产品接口是现成的，也就是Servlet接口。

虽说这些个elseif并不是好代码的征兆，不过这个简单工厂最起码帮我们解决了恶心的xml配置，说起来也算功不可没。

现在我们可以请求/contextPath/servlet/login来访问LoginServlet，而不再需要添加web.xml的配置，虽说这么做，我们对修改是开放的，因为每增加一个servlet，我们都需要修改工厂类，去添加一个if判断，但是LZ个人还是觉得我宁可写if，也不想去copy那个当初让我痛不欲生的xml标签，虽说我刚才还说让你忘掉elseif，我说过吗？好吧。。我说过，但是这只是我们暂时的做法，我们可以有很多种做法去消除掉这些elseif。

简单工厂是设计模式当中相对比较简单的模式，它甚至都没资格进入GOF的二十三种设计模式，所以可见它多么卑微了，但就是这么卑微的一个设计模式，也能真正的帮我们解决实际当中的问题，虽说这种解决一般只能针对规模较小的项目。

写到这里，简单工厂模式当中出现的角色，已经很清晰了。我们上述简单工厂当中设计到的类就是Servlet接口，ServletFactory以

及各种具体的LoginServlet，RegisterServlet等等。

总结起来就是一个工厂类，一个产品接口（其实也可以是一个抽象类，甚至一个普通的父类，但通常我们觉得接口是最稳定的，所以基本不需要考虑普通父类的情况），和一群实现了产品接口的具体产品，而这个工厂类，根据传入的参数去创造一个具体的实现类，并向上转型为接口作为结果返回。

我们在这里将上述穿插的简单工厂模式抽离出来，注释中有LZ个人的见解，帮助各位理解。

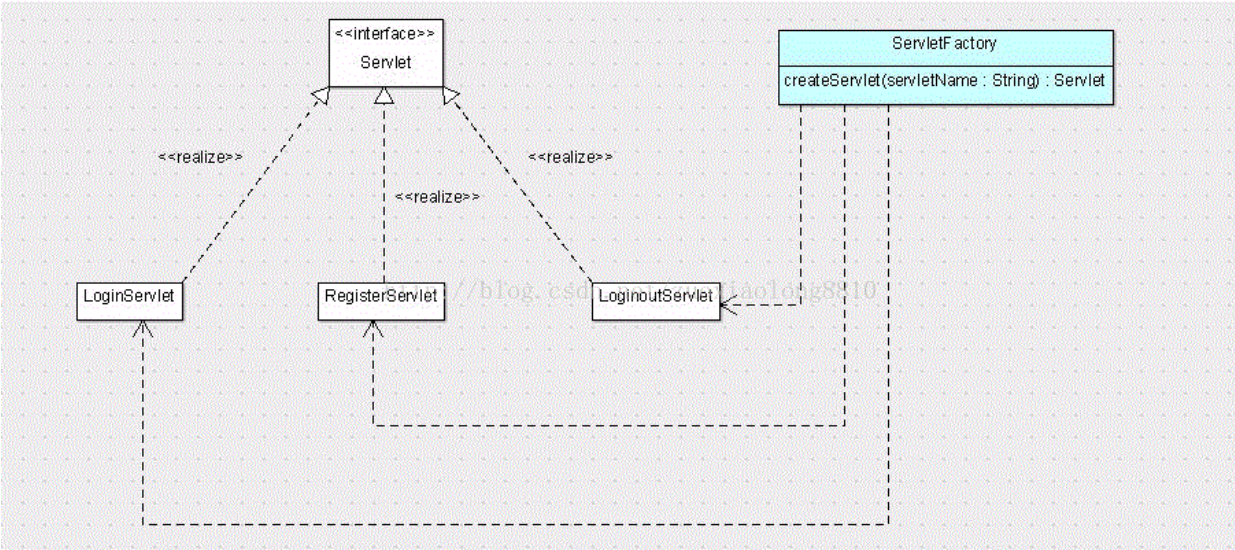
```
//相当于简单工厂模式中的产品接口

interface Servlet{}

//相当于简单工厂模式中的抽象父类产品。

//注意，简单工厂在网络上的资料大部分为了简单容易理解都是只规划了一个产品接口，但这不代表就只能有一个，设计模式的使用要灵活多变。

class HttpServlet implements Servlet{}

//具体的产品

class LoginServlet extends HttpServlet{}

class RegisterServlet extends HttpServlet{}

class LoginoutServlet extends HttpServlet{}

//产品工厂

public class ServletFactory {



	private ServletFactory(){}
```

//典型的创造产品的方法，一般是静态的，因为工厂不需要有状态。

```
	public static Servlet createServlet(String servletName)
{
		if (servletName.equals("login")) {
			return new LoginServlet();
		}else if (servletName.equals("register")) {
			return new RegisterServlet();
		}else if (servletName.equals("loginout")) {
			return new LoginoutServlet();
		}else {
			throw new RuntimeException();
		}
	}
}
```

上面LZ已经将刚才的过程给抽离出来，各位可以对比下标准的简单工厂代码，就会发现它们其实是一模一样的设计方式。

为了更加方便各位的对比，LZ这里给出上面JAVA代码的类图。



上面便是我们例子当中有关Servlet创建时，出现的简单工厂模式类图，各位可以和第一次的标准类图对比一下，它们的设计都是一样的。

其实我们针对创建Servlet实例这一部分逻辑的控制依旧有很多很多的优化余地，但是限于本章介绍的内容，所以我们就适可而止。

LZ觉得想简单工厂这种没有什么技术上的难度，纯粹是依照一些业务场景而出现的设计模式，LZ就必须要创造出一个比较真实的业务场景或者现实中的例子，才能更好的诠释。所以本次LZ先拿出了我们经常做的WEB项目，以后LZ也会尽量举一些实际应用的例子。

好了，本期的简单工厂模式就到这里吧，感谢各位的收看！

（四）工厂方法模式详解（另附简单工厂的死亡之路）

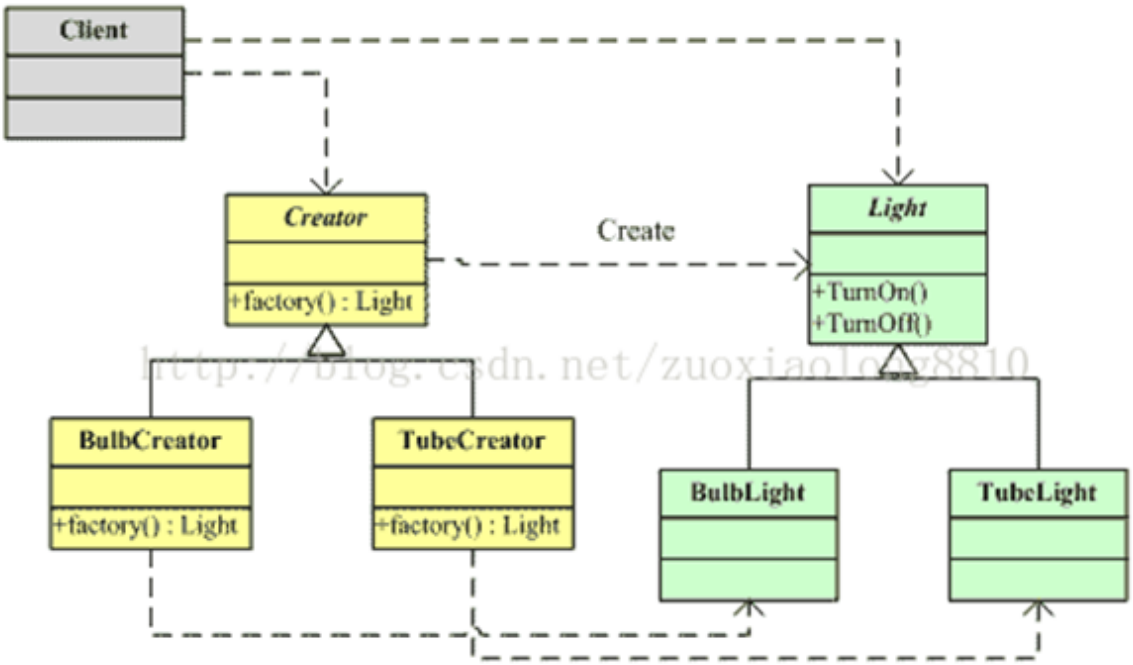
文章开头首先非常感谢各位的支持，代理模式中提到了class文件（即字节码文件）的相关知识，有一位读者说想要看有关class文件的相关内容，我也意识到了这一点，所以以后如果有在讲解那个模式的过程当中用到了其它的技术，我会留一些篇幅去介绍这个技术，有关class文件的内容我也会看以后的模式当中有没有用到的地方顺便简单介绍一下，如果没有的话，我会在设计模式介绍完以后专门写一篇有关java字节码文件的相关内容。

本章我们继续讨论新的设计模式，工厂方式模式，在这之前，LZ决定先给出引自其它地方的标准定义以及类图。

定义：工厂方法(Factory Method)模式的意义是定义一个创建产品对象的工厂接口，将实际创建工作推迟到子类当中。核心工厂类不再负责产品的创建，这样核心类成为一个抽象工厂角色，仅负责具体工厂子类必须实现的接口，这样进一步抽象化的好处是使得工厂方法模式可以使系统在不修改具体工厂角色的情况下引进新的产品。

可以看到工厂方法模式中定义了一个工厂接口，而具体的创建工作推迟到具体的工厂类，它是对简单工厂模式中的工厂类进一步抽象化，从而产生一个工厂类的抽象和实现体系，从而弥补简单工厂模式对修改开放的诟病。

下面LZ给出工厂方法模式的类图，该类图和定义引自百度百科。



可以看到，上面右半部分是产品抽象和实现体系，左半部分是工厂抽象和实现体系，其中工厂体系依赖于产品体系，每一个工厂负责创造一种产品，这就省去了简单工厂中的elseif判断，又客户端决定实例化一个特定的工厂去创建相应的产品。

下面LZ简单的使用JAVA代码诠释上述标准的工厂方法模式的类图。

首先是抽象产品接口。

```
public interface Light {

    public void turnOn();

    public void turnOff();

}
```

下面是具体的产品。

```
public class BuldLight implements Light{
```

```
        public void turnOn() {
                System.out.println("BuldLight On");
        }

        public void turnOff() {
                System.out.println("BuldLight Off");
        }

}

public class TubeLight implements Light{

        public void turnOn() {
                System.out.println("TubeLight On");
        }

        public void turnOff() {
                System.out.println("TubeLight Off");
        }
```

```
}
```

下面是抽象的工厂接口。

```
public interface Creator {
	public Light createLight();
}
```

下面是创建指定产品的具体工厂。

```
public class BuldCreator implements Creator{
	public Light createLight() {
		return new BuldLight();
	}

}

public class TubeCreator implements Creator{
	public Light createLight() {
		return new TubeLight();
	}

}
```

下面我们写个测试类去实验一下这个工厂方法模式的实例代码。

```
public class Client {
	public static void main(String[] args) {
		Creator creator = new BuldCreator();
		Light light = creator.createLight();
		light.turnOn();
		light.turnOff();


		creator = new TubeCreator();
		light = creator.createLight();
		light.turnOn();
		light.turnOff();
	}
}
```

运行结果如下。

可以看到，我们使用可以随意的在具体的工厂和产品之间切换，并且不需要修改任何代码，就可以让原来的程序正常运行，这也是工厂方法模式对扩展开放的表现，另外工厂方法模式弥补了简单工厂模式不满足开闭原则的诟病，当我们需要增加产品时，只需要增加相应的产品和工厂类，而不需要修改现有的代码。

上面的示例可以比较清楚的展示各个类之间的关系，但是始终缺乏说服力，因为它完全没有什么实际意义，下面LZ就给出一些我们接触过的例子来说明工厂方法模式的好处。

关于能够说明工厂方法模式的实例，LZ翻遍了所有能找到的源码，想寻找一个让各位读者既能学习到新的东西，又能对工厂方法理解更深的现有的优秀框架的设计。经过跋山涉水，LZ决定还是拿数据库连接来说事，我知道你想说，我去，又是数据库连接。LZ只想说，我们每天做的最多的就是增删改查好吗，其它的咱也不认识啊，囧。

众所周知，为了统一各个数据库操作的标准，于是有了JDBC的API，它用于给我们这种被称作只会使用现成的东西的程序猿，提供一系列统一的，标准化的操作数据库的接口。其实JDBC的各个类或接口，就是我们操作数据库的过程中各个协助者的抽象，这样的设计是为了让我们对数据库的操作依赖于抽象，还记得我们在设计模式总纲中提到的一句话吗，用抽象构建框架，用细节扩展实现。

JDBC API（即抽象的接口或类）就是整个数据库操作的框架，而各个数据库的驱动就是那些细节。而我们的操作依赖于JDBC API，而不是任何一个具体数据库的细节。

JDBC是如何统一了数据库世界的呢？其实最主要的就是靠两个接口，就统一了世界。

来看第一个接口Driver，附上源码。

```
package java.sql;



import java.sql.DriverPropertyInfo;

import java.sql.SQLException;



/**

  * The interface that every driver class must implement.
```

```
  */
public interface Driver {


      Connection connect(String url, java.util.Properties
info)
            throws SQLException;


      boolean acceptsURL(String url) throws SQLException;


      DriverPropertyInfo[] getPropertyInfo(String url,
java.util.Properties info)
                  throws SQLException;


      int getMajorVersion();


      int getMinorVersion();


      boolean jdbcCompliant();
}
```

由于篇幅，LZ删掉了很多注释，只保留了这个类注释的第一句话，翻译过来是这是一个任何驱动类都必须实现的接口。多么霸气啊。也就是每个数据库厂商都必须实现这个接口来提供JDBC服务，即java数据库连接服务，来方便程序猿对数据库应用编程。

我们先忽略掉下面的五个方法，第一个方法毫无疑问是这个接口中相对而讲最重要的方法了，即创造一个数据库连接，虽然方法名称是connect，但是我觉得这个方法完全可以改为createConnection。

提到Connction，这个接口我们一定不陌生，它的源码也已经在代理模式一章出现过，这里我们再次让它出场，我依旧会删掉它的大部分方法，限于篇幅。

```
package java.sql;



import java.sql.PreparedStatement;

import java.sql.SQLException;



/**

  * <P>A connection (session) with a specific

  * database. SQL statements are executed and results are
returned

  * within the context of a connection.

  * <P>

  */

public interface Connectionextends Wrapper {



      Statement createStatement() throws SQLException;
```

```
    PreparedStatement prepareStatement(String sql) throws
SQLException;



}
```

以上便是Connection接口，这里只留下了两个方法，这两个方法相信各位读者都非常熟悉，它们都是我们最经常用的方法之二。

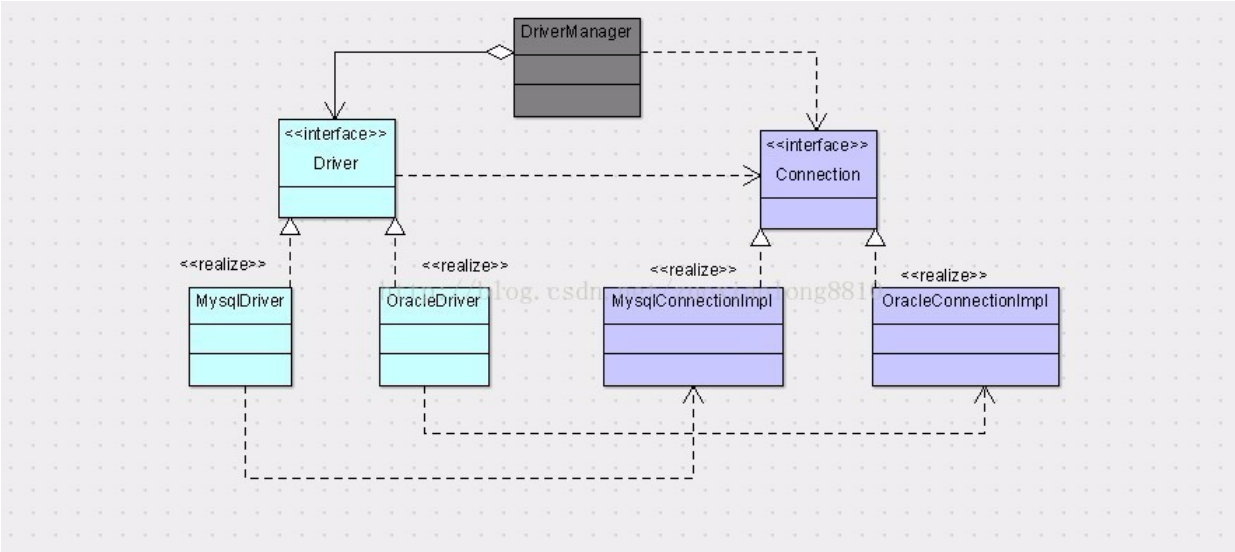
以上两个接口作为JDBC API的一部分，它们相当于告诉了数据库生产厂商两个要求。

第一，数据库厂商要提供一个数据库驱动类，它的作用可以是可以创造数据库连接，而这个数据库连接向上转型为我们JDBC的Connection。

第二，数据库厂商要提供一个数据库连接的实现类，这个实现类可以执行具体数据库的各个操作，比如帮我们执行SQL，返回执行结果，关闭连接等等。

我们都知道mysql的驱动类位于com.mysql.jdbc.Driver，而mysql的connection实现类也在这个包中，名称是ConnectionImpl，而相应的oracle也有驱动类，位于oracle.jdbc.driver.OracleDriver，相应的oracle也有connection实现类，位于oracle.jdbc.OracleConnectionWrapper。一般每个数据库都会有一个Connection的扩展接口，这个接口的作用是提供使用者针对当前数据库特殊的操作。

这里我们忽略掉这些中间接口以及抽象类，我给出上述六个类的UML图，如果各位以前知道工厂方法模式的话，各位看一下，它们的关系是否很熟悉。



我们对比上面标准的工厂方法模式，就会发现它们的关系不正是工厂方法模式吗？

工厂方法模式就是提供一个抽象的工厂，一个抽象的产品，在上述当中相当于Driver（数据库连接工厂）和Connection（抽象产品），实现的一方需要提供一个具体的工厂类（比如mysql驱动）和一个具体的产品（比如mysql数据库连接）。

客户端调用时不依赖于具体工厂和产品（即到底是mysql驱动，mysql数据库连接还是oracle驱动，oracle连接，我们程序猿不需要管的，我们只管使用抽象的driver和connection，对吧？），而是依赖于抽象工厂和抽象产品完成工作。

各位可以看到我在类图里面加入了一个DriverManager，这个类相信各位也不陌生，这是我们天天打交道的类，虽说因为hibernate和ibatis的封装，或许我们不能经常看到，但LZ相信它活在每个程序猿的心中。

DriverMananger在这个设计当中扮演者一个管理者的角色，它帮我们管理数据库驱动，让我们不需要直接接触驱动接口，我们获取连接只需要和DriverManager打交道就可以，也就是说客户端依赖于DriverManager和Connection就可以完成工作，不再需要与Driver关联，所以上述说我们依赖于Driver和Connection，现在DriverManager

帮我们管理Driver，那我们只需要依赖于DriverManager和Connection就可以了。

LZ在类图中拉出了DriverManager的方法，其中的registerDriver方法正是我们注册数据库驱动的入口。来看看mysql的Driver中做了什么，oracle类似。

```
public class Driver extends NonRegisteringDriver

   implements java.sql.Driver

{

   public Driver()

      throws SQLException

   {

   }



   static

   {

      try

      {

         DriverManager.registerDriver(new Driver());

      } catch (SQLException E) {

         throw new RuntimeException("Can't register
driver!");

      }
```

```
    }

}
```

可以看到，在类构造方法中，加入了registerDriver这个方法，所以当我们使用class.forName加载驱动的时候，将会把mysql驱动注册到DriverManager，这时DriverManager中就会持有Mysql驱动所必要的信息，我们就可以使用DriverManager来获得具体的mysql连接了，当然，你要提供url，用户名和密码。

原来我们都是活在温室里的花朵，都被这些设计者细心呵护着，生怕我们知道一点底层的东西。记得LZ当初第一次看到Class.forName时，还觉得真是个神奇的东西，没想到只是这些设计者给我们的糖外衣。

工厂方法模式的好处和适用的场景都相对比较好理解。

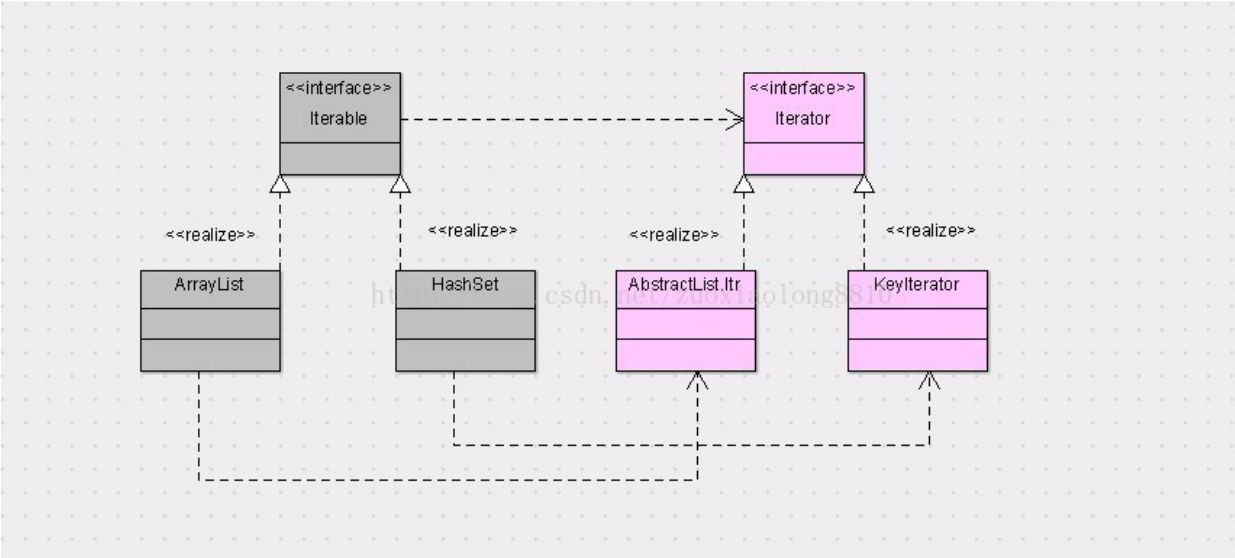
好处就是，从类关系上来说，它可以让客户端与具体的工厂与产品解耦，从业务角度来说，它让客户端与具体的产品解耦。

适用的场景就是我们需要一个产品帮我们完成一项任务，但是这个产品有可能有很多品牌（像这里的mysql，oracle），为了保持我们对产品操作的一致性，我们就可能要用到工厂方法模式。

工厂方法模式也有它所不足的地方，可能你会说，这多好啊，我们操纵数据库不再需要关心具体是哪个数据库。是的，你很爽啊，那是因为这些产品的实现都不用你写啊，都是数据库厂商给你写的。

假设产品数量巨多，而且需要我们亲手去逐个实现的时候，工厂方法模式就会增加系统的复杂性，到处都是工厂类和产品类，而且这里所说的工厂类和产品类只是概念上的，真正的产品可能不是一两个类就能搞定，否则mysql和oracle的驱动包为啥要那么多类，而不是就一个Driver和一个Connection。

当然这也不是绝对，比如我们经常使用的HashSet和ArrayList，也是使用的工厂方法模式，各位看下他们的类图就看出来了。



各位可能会说，不对啊，这和我们刚才理解的不太一样啊，按照刚才的说法，我们不是应该直接使用iterable和iterator吗？这样多牛X，我们不依赖于具体产品了。对于这个LZ表示三条黑线垂下，sun或者说oracle为了集合框架给你提供了这么多具备各个特性的集合，你只用iterator和iterable，估计当初参与设计集合框架的人都要气的去shi了。。

上述这便是工厂方法模式另外一种用法了，刚才因为我们不关心真正的产品是什么，所以我们直接使用抽象接口操作。但是我们使用iterable和iterator的时候，我们是关心真正产品的特性的，所以为了使用产品的特性，我们就需要使用产品特有的接口了，比如特殊的SortedSet可排序，比如ArrayList可以有重复元素，可以根据索引获取元素等等。当然你依然是可以使用iterable和iterator的，但是不管你用什么，在这种场景下，产品是你自己选的，一句话，你随便。。。

两种使用方式一种是对使用者透明的，一种是不透明的，一种是使用者对具体的产品不关心，这种情况下，一般产品提供的功能是类似的。一种是使用者非常了解产品的特性，并想使用产品的特性，这种情况下，一般产品只提供最基本的一致的功能，但每个产品都会有自己独特的一面。

但是LZ个人觉得真正做项目的过程当中很少用到工厂方法模式，这个模式更多的是帮助我们理解现有的开源项目，就像现在，你

是不是对JDBC的大体框架有了一定认识了呢，如果你不知道这个模式，可能看源码会觉得一头雾水呢。

另外，文章最后插播一段内容，如果各位看过上一章（简单工厂模式）的话，一定还记得那个恶心的elseif结构，这是简单工厂的诟病，它对扩展开放，对修改也开放。

简单工厂模式在项目规模相对较小或者说具体的产品类相对不多的情况下（针对上章的描述，特指的servlet数量不多的情况下），其实这种设计还是可以接受的，因为少量的elseif可以换来我们开发上的便利。

所以LZ建议各位永远不要忘记，规则只是用来指导你的，不是用来限制你的，只要设计合理，你的设计就是规则！

不过针对简单工厂模式，你可以认为它给我们提供了一个思路，就是我们其实可以省掉那些让人痛恨的xml配置，对于我们后续的优化有着一定指导意义。

就像上一章中的处理方式，很明显存在着隐患，那就是在servlet数量急剧上升的时候，工厂类就会变得非常臃肿和复杂，变得难以维护和阅读。本章LZ给各位读者介绍一种优化方式，可以采取一项JDK当中在1.5版本引入的技术，即注解，去消除那些elseif的逻辑判断。

我们可以参考struts2的做法，即每一个Servlet我们都可以采用注解去设置它的名称，或者叫url，然后我们让我们的简单工厂依据这个去实例化我们的servlet。

根据以上方案，我们需要按照以下步骤让我们的简单工厂彻底死翘翘。

1.需要声明一个注解，它可以用来给servlet标识它的名称。

2.需要声明一个注解的处理器，用来处理我们的注解，主要作用是通过一个CLASS文件，去获得它的注解信息。

3.基于性能，我们需要将servlet与名称的映射与应用的生命周期绑定，并且这份映射在整个应用当中有且仅有一份，且不可更改。

4.让我们用于分派请求的过滤器，使用映射信息将客户请求对应到相应的servlet去处理，并且将分派逻辑移回过滤器，从而彻底删除简单工厂，即ServletFactory。

特别说一下，这四步当中，其中第三步是可选的，但也是必须的，因为如果不做这种处理，那么你就等着你的项目N长时间打开一个网页吧。

以上是简单工厂给我们的启示，具体如何实现这样一个基于注解的请求分配的架构，LZ不再给各位一一演示，因为这已经只剩下一个堆积代码的过程，具体的实现方案已经有了，如果各位读者有兴趣，可以私底下尝试一下这种方式。

好了，工厂方法模式就给各位分享到这吧，感谢各位的欣赏。

（五）抽象工厂模式详解

前两章我们已经讨论了两种有关工厂的模式，今天我们来看最后一种与工厂相关的模式，抽象工厂模式。

抽象工厂模式算是工厂相关模式的终极形态，如果各位完全理解了上一章的工厂方法模式，那么抽象工厂模式就很好理解了。它与工厂方法唯一的区别就是工厂的接口里是一系列创造抽象产品的方法，而不再是一个，而相应的，抽象产品也不再是一个了，而是一系列相关的产品。这其实是工厂方法模式的一种扩展不是吗?

通常意义来我们谈到扩展，通常有两种方式可以扩展一个接口或者类，就是继承和组合。

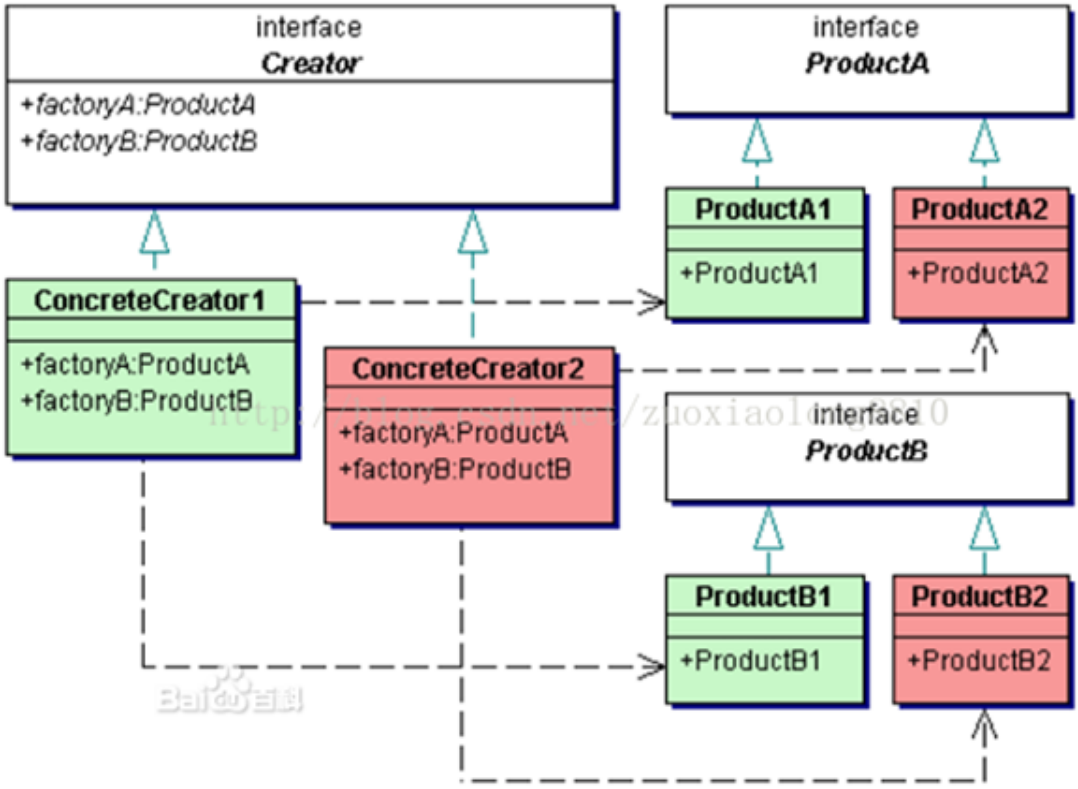
通常情况下，我们推荐使用组合扩展一个现有的类或接口，但这并非绝对，如果你扩展的子类或子接口与现有的类或接口明显是“是一个（is a）”的关系，也就是继承的关系，那么使用继承可以获得更多的好处。

下面我们就首先来看一下抽象工厂模式的定义以及类图，全部引自百度百科。

定义：为创建一组相关或相互依赖的对象提供一个接口，而且无需指定他们的具体类。

定义中说了，我们是要创建一个接口， 而这个接口是干嘛的呢，前面说了，是为了创建一组相关或者相互依赖的对象，而且还有一点就是，我们创建的对象不是具体的类，也就是说我们创建的是一个接口或者一个抽象类。

下面我们来看看抽象工厂模式的类图。



我们对比下刚才的定义，LZ给各位分析下上面的类图，首先刚才说了，我们要创建一个接口，这个接口就是指的Creator，而一组相关或者相互依赖的对象，就是指的ProductA和ProductB以及它们具体的实现类，而上面又提到说不是返回的具体的类，所以我们返回的应该是接口或者抽象类，那么在上述类图当中，则是指的ProductA和ProductB接口。

下面LZ将上述类图诠释成容易理解的JAVA代码，供各位参考。

首先给出我们的产品族，也就是类图中右半部分。

```
package net;



interface ProductA {
```

```
        void methodA();
}

interface ProductB {

        void methodB();
}

class ProductA1 implements ProductA{

        public void methodA() {
                System.out.println("产品A系列中1型号产品的方法");
        }

}

class ProductA2 implements ProductA{

        public void methodA() {
                System.out.println("产品A系列中2型号产品的方法");
```

```
        }

}

class ProductB1 implements ProductB{

        public void methodB() {
                System.out.println("产品B系列中1型号产品的方法");
        }

}

class ProductB2 implements ProductB{

        public void methodB() {
                System.out.println("产品B系列中2型号产品的方法");
        }

}
```

结构比较清晰，下面是类图中左半部分，首先给出工厂接口。

```
package net;

public interface Creator {
	ProductA createProductA();
	ProductB createProductB();
}
```

下面是两个具体的工厂实现类。

```
package net;

public class ConcreteCreator1 implements Creator{

	public ProductA createProductA() {
		return new ProductA1();
	}

	public ProductB createProductB() {
		return new ProductB1();
	}

}
```

```
package net;


public class ConcreteCreator2 implements Creator{


	public ProductA createProductA() {

		return new ProductA2();

	}


	public ProductB createProductB() {

		return new ProductB2();

	}


}
```

这样我们的类图代码就实现完毕，下面我们写一个测试类，去调用一下，感受一下抽象工厂模式的客户端调用方式。

```
package net;


public class Client {


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

```
            Creator creator = new ConcreteCreator1();
            ProductA productA = creator.createProductA();
            ProductB productB = creator.createProductB();
            productA.methodA();
            productB.methodB();

            creator = new ConcreteCreator2();
            productA = creator.createProductA();
            productB = creator.createProductB();
            productA.methodA();
            productB.methodB();
        }
}
```

在过程当中，我们切换过一次工厂实现类，而下面的代码是一模一样的，但是我们使用的就是另一套产品实现体系了，我们看运行结果。

上面的代码比较简单，结构很清晰但不太容易理解，因为它全部是抽象的表示，与实际联系不上，所以也会对各位的理解造成阻碍，下面我们就一起讨论一个现有的例子，去加深去抽象工厂模式的理解。

上一章我们介绍了iterable接口，它可以制作iterator，iterator方法是一个工厂方法，用于让子类制作一系列的iterator，不过java集合框架一般都将iterator的实现作为内部类出现，所以我们从未见过LZ上章提到的ListIterator和KeyIterator的实现类，但它们确实存在

于JAVA的集合框架，并且它们的实现类被封装在相应的抽象类或者具体的容器实现类中。

oracle公司为何不让我们看到这些iterator的实现类呢？其实原因很简单，一是怕我们在写程序的时候依赖于这些iterator的实现类，二是这些迭代器的实现都要依赖于当前的容器实现，我们假设有一天JDK中的集合框架要升级，要替换掉某个iterator的实现，换做一种更快的迭代方式（假设存在这种方式），那么以前使用特定迭代器的程序可能就无法正常运行了。当然大部分的情况下，oracle不会将现有的类剔除，但是会加上@Deprecated注解，来标识这是一个过时的东西，不再推荐你使用。但就算是这样，还是有缺点，就是JDK升级以后，你享受不到JDK集合框架速度上的提升，除非你将所有你使用过具体的Iterator的地方全部手动替换掉。

上述大致描述了下集合框架设计时对iterator处理方式的初衷，从中可以看出抽象工厂模式就是为了解决抽象产品不再是一个的时候的问题。因为不管是简单工厂，还是工厂方法，都有一个缺陷，那就是整个模式当中只能有一个抽象产品，所以直观的，你在工厂方法模式中再添加一个创造抽象产品的方法就是抽象工厂模式了，相应的当然还有添加一个抽象产品，还有一系列具体的该抽象产品的实现。

在集合框架里，有一个不太明显的抽象工厂模式，就是List接口，它在iterable的基础上，扩展了一个创建产品的方法，本次以List接口为例，我们来看看List接口的源码。

```
package java.util;



public interface List<E> extends Collection<E> {

	Iterator<E> iterator();//一种产品

	Object[] toArray();

	<T> T[] toArray(T[] a);

	ListIterator<E> listIterator();//另外一种产品
```

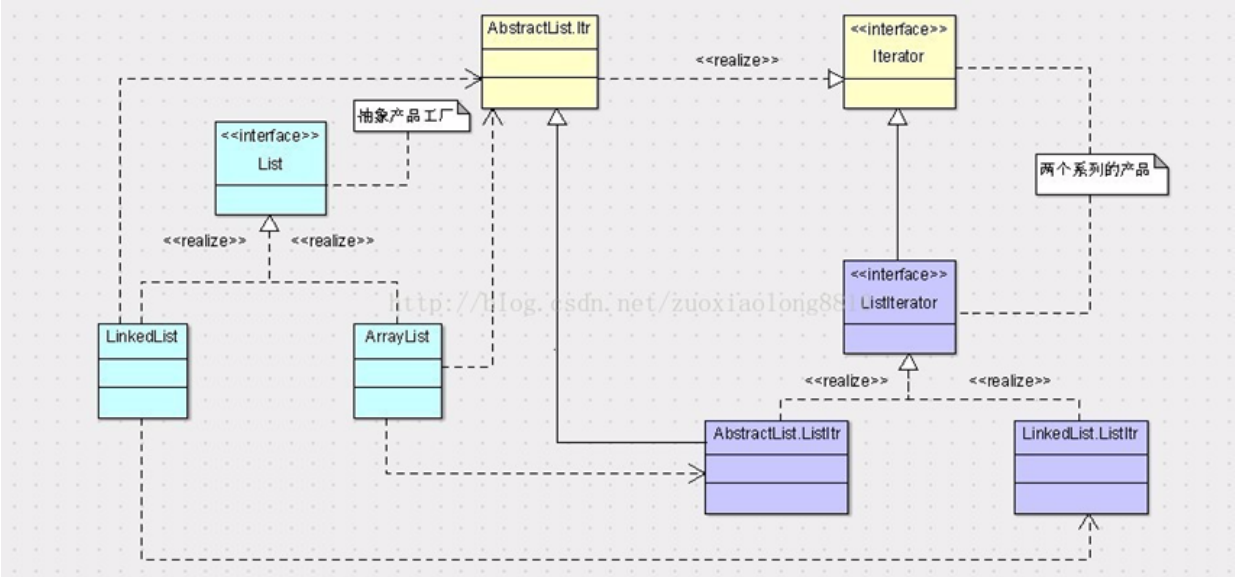
```
    ListIterator<E> listIterator(int index);

}
```

LZ去掉了List接口中的很多方法，一是为了节省版面，另外是为了更清晰，我们主要关注iterator和listIterator方法，LZ在上面加了标注。

其中ListIterator是Iterator的子接口，但归根到底，它其实属于另外一种产品，为什么这么说呢，ListIterator不是Iterator的子接口吗，怎么能算是另外一种产品呢？这是因为我们listIterator方法的返回类型是ListIterator，而不是Iterator，所以两者的功能是不同的，比如ListIterator还可以向前移动。

我们可以认为这两个方法产生的一个是只能向后移动的迭代器，一个是可以前后移动的迭代器，这算是两种产品，相当于上面的ProductA和ProductB。

这个设计可以看做是一个抽象工厂模式，List接口定义了两种生产不同产品的方法，这属于两个系列的产品，不过由于产品接口本身的继承关系，两者的实现类也会被做成继承的关系。下面给出上面提到的接口的UML图。



这个图看起来有点复杂，各位可以和上面标准的抽象工厂模式类图对比一下，下面LZ来解释一下在抽象工厂模式当中，上述几个类都代表的什么角色。

1.List，是抽象工厂的角色，它有两个制造产品的方法，iterator和listIterator，相当于Creator。

2.ListIterator和Iterator都是抽象产品，相当于ProductA和ProductB。其中ListIterator有两个实现类，分别是AbstractList.ListItr和LinkedList.ListItr，相当于ProductA1和ProductA2。Iterator的实现类为AbstractList.Itr，相当于ProductB1，但是没有B2。

3.LinkedList是其中一个具体的工厂类，相当于ConcreteCreator1，实现抽象工厂List，它制造的两个具体产品分别是LinkedList.ListItr和AbstractList.Itr。

4.同样的，ArrayList也是一个具体的工厂类，相当于ConcreteCreator2，实现抽象工厂List，它制造的两个具体产品分别是AbstractList.ListItr和AbstractList.Itr。

结合上一章工厂方法模式，我们来分析一下工厂方法模式和抽象工厂模式二者的关系。

Iterable接口是List的父接口，所以它只负责一个产品Iterator的制造，所以是工厂方法模式，而List接口扩展了Iterable接口，又添加了一个制造产品的方法，即又添加了一个系列的产品，所以就成为了抽象工厂模式。

LZ下面给出上述两个类图的对应关系,会让各位看的更加清晰：

1.Creator=List

2.ConcreteCreator1=ArrayList

3.ConcreteCreator2=LinkedList

4.ProductA=Iterator

5.ProductB=ListIterator

6.ProductA1=AbstractList.Itr

7.ProductA2=无（具体的A产品2在第一个类图中是没有的，但这并不影响整个体系）

8.ProductB1=AbstractList.ListItr

9.ProductB2=LinkedList.ListItr

ArrayList和LinkedList分别是List接口的两种实现，前者是基于数组操作，后者是基于链表。两者都可以产生Iterator和ListIterator，而Iterator的实现都是在AbstractList中实现的，是一样的处理方式，而对于ListIterator的实现却不相同，AbstractList.ListItr是基于数组的操作，LinkedList.ListItr是基于链表的操作方式。

所以抽象工厂模式一般是为了处理抽象产品多于一个的问题，而且这些产品多数情况下是有关系的，像上述JAVA集合框架的例子当中，Iterator和ListIterator就是继承的关系，大部分情况下，很少会使用抽象工厂模式去创造一批毫无关系的产品。

基于抽象工厂一旦定义，抽象产品的个数就已经固定，所以最好在抽象产品的个数不太会变化的情况下使用抽象工厂模式，当然，我们可以使用继承去弥补抽象工厂模式的这一不足，创造另外一个继承体系去扩展现有的框架。

下面LZ给出简单工厂模式，工厂方法模式一直到抽象工厂模式的演变过程，三者是由简到繁的关系。由于三者都已经详细的解释过，所以此处不再多做解释，留给各位读者自己思考它们的进化过程，首先LZ给出简单工厂的具体代码。

```
//抽象产品

interface Product{}



//具体产品
```

```
class ProductA implements Product{}

class ProductB implements Product{}


//产品工厂（下一步就是它的进化，就变成了工厂方法模式）

public class ProductFactory {


      private ProductFactory(){}


      public static Product getProduct(String productName){

            if (productName.equals("A")) {

                  return new ProductA();

            }else if (productName.equals("B")) {

  return new ProductB();

            }else {

                  return null;

            }

      }

}
```

LZ在上面加了简单的注释，下面LZ给出工厂方法模式的代码，注意，前面有关产品的类和接口是不变的。

```
//抽象产品
```

```
interface Product{}

//具体产品
class ProductA implements Product{}
class ProductB implements Product{}

//将简单工厂中的工厂给抽象成接口
interface Factory{
	Product getProduct();
}
//具体的工厂A，创造产品A
class FactoryA implements Factory{

	public Product getProduct() {
		return new ProductA();
	}

}
//具体的工厂B，创造产品B
class FactoryB implements Factory{
```

```
    public Product getProduct() {
        return new ProductB();
    }

}
```

可以看到，产品部分并没有变化，只是将简单工厂中的工厂类抽象成接口，并给相应产品添加相应的工厂类，就进化成了工厂方法模式。下面我们再看工厂方法如何进化成抽象工厂模式。

```
//抽象产品
interface Product{}

//具体产品
class ProductA implements Product{}
class ProductB implements Product{}

//多了一个抽象产品1
interface Product1{}

//具体产品1
class Product1A implements Product1{}
```

```
class Product1B implements Product1{}

//原有的工厂方法模式的工厂里添加一个方法
interface Factory{
	Product getProduct();
	//添加另外一个产品族的创造方法
	Product1 getProduct1();
}
//具体的工厂A，创造产品A
class FactoryA implements Factory{

	public Product getProduct() {
		return new ProductA();
	}
	//添加相应的实现
	public Product1 getProduct1() {
		return new Product1A();
	}

}
```

```
//具体的工厂B，创造产品B
class FactoryB implements Factory{


	public Product getProduct() {
		return new ProductB();
	}
	//添加相应的实现
	public Product1 getProduct1() {
		return new Product1B();
	}


}
```

与工厂方法对比下就发现，多了一个产品系列叫Product1，工厂接口里多了一个方法，叫getProduct1，所以抽象工厂模式就是工厂方法模式添加了抽象产品所演变而来的。

有关工厂的三个模式到这里就全部介绍完了，三者有着很大的关联和明显的关系，要想灵活运用这三种设计模式，还是要彻底理解它们所针对的问题以及三者的关系。下面罗列下这三种设计模式依次进化的原因。

1，首先从简单工厂进化到工厂方法，是因为工厂方法弥补了简单工厂对修改开放的弊端，即简单工厂违背了开闭原则。

2，从工厂方法进化到抽象工厂，是因为抽象工厂弥补了工厂方法只能创造一个系列的产品的弊端。

各位可以思考下，假设我们不使用抽象工厂模式，改用工厂方法去处理抽象工厂中多产品的问题，如何处理呢？其实很简单，就是有几个产品系列，我们就造几个工厂方法模式就可以了，只不过这样处理未免太不优雅，就像下面这样。

```
//抽象产品

interface Product{}



//具体产品

class ProductA implements Product{}

class ProductB implements Product{}



//工厂接口

interface Factory{

	Product getProduct();

}



//具体的工厂A，创造产品A

class FactoryA implements Factory{



	public Product getProduct() {

		return new ProductA();

	}
```

```
}
//具体的工厂B，创造产品B
class FactoryB implements Factory{

	public Product getProduct() {
		return new ProductB();
	}

}

/*	以上是一个产品的工厂方法	*/

//抽象产品1
interface Product1{}

//具体产品1
class Product1A implements Product1{}
class Product1B implements Product1{}
```

```
//工厂接口1
interface Factory1{
	Product1 getProduct1();
}

//具体的工厂1A，创造产品1A
class Factory1A implements Factory1{

	public Product1 getProduct1() {
		return new Product1A();
	}

}
//具体的工厂1B，创造产品1B
class Factory1B implements Factory1{

	public Product1 getProduct1() {
		return new Product1B();
	}
```

```
}
```

以上用两个工厂方法模式，代替了抽象工厂模式，那么可想而知，假设又多了一个产品Product2，那么我们还需要再建立一套工厂方法模式，这显然会大大增加系统的复杂性，而且也不易于客户端操作。

不过这也不一定就不可以，假设我们上面Product和Factory接口包括它们的一套实现是现有的，并且我们无法改变，比如是一个第三方的jar包提供的。那么为了扩展这个第三方的jar包，我们可以将jar包中的工厂方法模式扩展成为抽象工厂来达到我们扩展现有类功能的目的，就像下面这样。

```
//抽象产品

interface Product{}



//具体产品

class ProductA implements Product{}

class ProductB implements Product{}



//工厂接口

interface Factory{

      Product getProduct();

}



//具体的工厂A，创造产品A

class FactoryA implements Factory{
```

```
	public Product getProduct() {
		return new ProductA();
	}

}
//具体的工厂B，创造产品B
class FactoryB implements Factory{

	public Product getProduct() {
		return new ProductB();
	}

}

/*	假设以上是一个第三方jar包中的工厂方法模式，我们无法改动源码	*/

//我们自己特有的产品
interface MyProduct{}
```

```
//我们自己特有的产品实现

class MyProductA implements MyProduct{}

class MyProductB implements MyProduct{}


//扩展原有的工厂接口

interface MyFactory extends Factory{

	MyProduct getMyProduct();

}


//我们自己特有的工厂A，扩展自原有的工厂A，并且实现获得我们自己特有产品的接口方法

class MyFactoryA extends FactoryA implements MyFactory{


	public MyProduct getMyProduct() {

		return new MyProductA();

	}


}

//同A

class MyFactoryB extends FactoryB implements MyFactory{
```

```
        public MyProduct getMyProduct() {
                return new MyProductB();
        }


}
```

这样我们就可以得到我们自己特有的抽象工厂和使用我们自己特有的产品了，并且我们自己的抽象工厂还兼并了第三方jar包中的产品，例如，我们可以使用MyFactoryA获得jar包中的ProductA产品等。

上面的做法相当于我们从现有的体系当中，扩展出一套我们自己的继承体系，这样做的好处是我们可以完整的复用jar包中的各个类功能，缺点是继承会导致系统的复杂性增加，耦合度相对较高。

所以我们还可以有另外一种做法，就是创造我们自己的一套独有的工厂方法模式，这套体系与jar包中的类和接口毫无关系，我们再使用一个组合工厂将二者结合起来，就像下面这样。

```
//抽象产品
interface Product{}


//具体产品
class ProductA implements Product{}
class ProductB implements Product{}


//工厂接口
interface Factory{
```

```
	Product getProduct();
}

//具体的工厂A，创造产品A
class FactoryA implements Factory{

	public Product getProduct() {
		return new ProductA();
	}

}
//具体的工厂B，创造产品B
class FactoryB implements Factory{

	public Product getProduct() {
		return new ProductB();
	}

}
```

```
/*     假设以上是一个第三方jar包中的工厂方法模式，我们无法改动源码     */


//我们自己特有的产品

interface MyProduct{}


//我们自己特有的产品实现

class MyProductA implements MyProduct{}

class MyProductB implements MyProduct{}


//我们自己的工厂接口

interface MyFactory{

      MyProduct getMyProduct();

}


//我们自己特有的工厂A，产生产品A

class MyFactoryA implements MyFactory{


      public MyProduct getMyProduct() {

            return new MyProductA();

      }
```

```
}

//我们自己特有的工厂B，产生产品B
class MyFactoryB implements MyFactory{

	public MyProduct getMyProduct() {
		return new MyProductB();
	}

}

/*	到这里是我们自己的一套工厂方法模式，去创造我们自己的产品，以下我们将以上二者组合	*/

//我们使用组合的方式将我们的产品系列和jar包中的产品组合起来
class AssortedFactory implements MyFactory,Factory{

	MyFactory myFactory;
	Factory factory;
```

```
	public AssortedFactory(MyFactory myFactory, Factory factory) {
		super();
		this.myFactory = myFactory;
		this.factory = factory;
	}

	public Product getProduct() {
		return factory.getProduct();
	}

	public MyProduct getMyProduct() {
		return myFactory.getMyProduct();
	}

}
```

可以看到，组合的工厂AssortedFactory集成了我们自己的工厂和jar包中的工厂两者的功能。这样做则会非常灵活，因为我们的一套体系不再依赖于jar包中的类或接口而存在，哪怕是jar包中的类改变或者不在了，我们自己的这一套依旧可以独立存在。

从上面就可以看出，我们在处理很多问题的时候其实是有很多种方式的，而且每一种方式可能都有各自的好处和坏处，很难去判断说那一种方式是最好的，而且也根本就没有这个说法，所以我们能做的，就

是根据实际的情况去掂量各个方式的利弊，从而选择出一种更适合当前情况的处理方式。

本期抽象工厂模式就到这里了，希望各位看到这里对三个与工厂相关的模式有一定自己的理解，并且可以灵活使用，达到无模式就是最好的模式的境界，甚至，我们可以把我们自己写的东西起个名字，冒充设计模式，比如最后一种我们叫它组合工厂模式。当然，LZ本人并未达到无模式的境界，正在与各位一起努力。

感谢各位的收看。

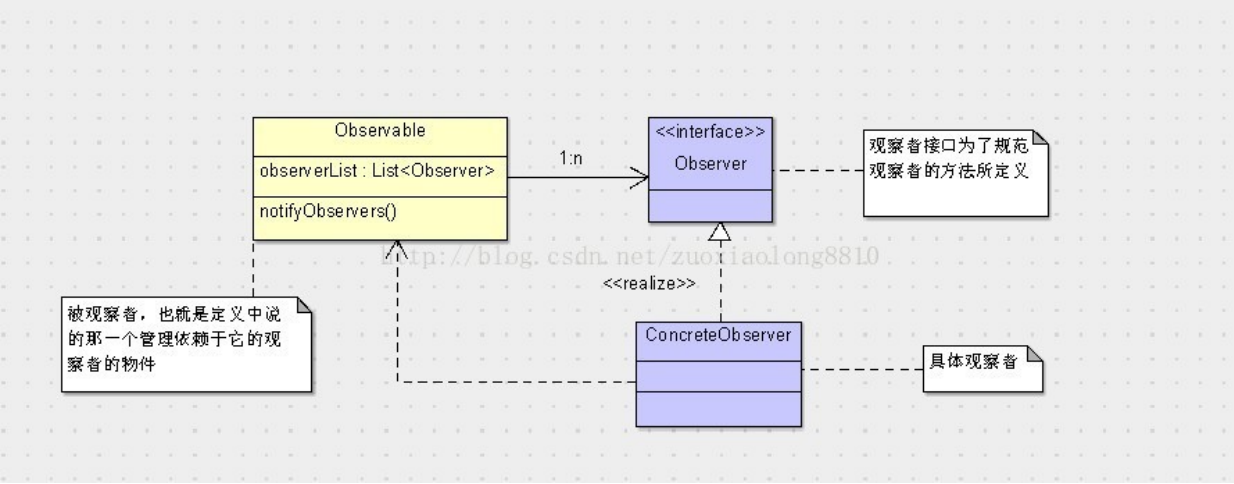
（六）观察者模式详解（包含观察者模式JDK的漏洞以及事件驱动模型）

本章我们讨论一个除前面的单例以及代理模式之外，一个WEB项目中有可能用到的设计模式，即观察者模式。

说起观察者模式，LZ还是非常激动的，当初这算是第一个让LZ感受到设计模式强大的家伙。当初LZ要做一个小型WEB项目，要上传给服务器文件，一个需求就是要显示上传进度，LZ就是用这个模式解决了当时的问题，那时LZ接触JAVA刚几个月，比葫芦画瓢的用了观察者模式。

现在谈及观察者模式，能用到的地方就相对较多了，通常意义上如果一个对象状态的改变需要通知很多对这个对象关注的一系列对象，就可以使用观察者模式。

下面LZ先给出观察者模式标准版的定义，引自百度百科。



定义：观察者模式（有时又被称为发布-订阅模式、模型-视图模式、源-收听者模式或从属者模式）是软件设计模式的一种。在此种模式中，一个目标物件管理所有相依于它的观察者物件，并且在它本身的状态改变时主动发出通知。这通常透过呼叫各观察者所提供的方法来实现。此种模式通常被用来实作事件处理系统。

上面的定义当中，主要有这样几个意思，首先是有一个目标的物件，通俗点讲就是一个类，它管理了所有依赖于它的观察者物件，或者通俗点说是观察者类，并在它自己状态发生变化时，主动发出通知。

简单点概括成通俗的话来说，就是一个类管理着所有依赖于它的观察者类，并且它状态变化时会主动给这些依赖它的类发出通知。

那么我们针对上面的描述给出观察者模式的类图，百度百科没有给出观察者模式的类图，这里LZ自己使用工具给各位画一个。

可以看到，我们的被观察者类Observable只关联了一个Observer的列表，然后在自己状态变化时，使用notifyObservers方法通知这些Observer，具体这些Observer都是什么，被观察者是不关心也不需要知道的。

上面就将观察者和被观察者二者的耦合度降到很低了，而我们具体的观察者是必须要知道自己观察的是谁，所以它依赖于被观察者。

下面LZ给写出一个很简单的观察者模式，来使用JAVA代码简单诠释一下上面的类图。

首先是观察者接口。

```
package net;



//这个接口是为了提供一个统一的观察者做出相应行为的方法

public interface Observer {



void update(Observable o);
```

```
}
```

再者是具体的观察者。

```
package net;


public class ConcreteObserver1 implements Observer{


public void update(Observable o) {

System.out.println("观察者1观察到" +
o.getClass().getSimpleName() + "发生变化");

System.out.println("观察者1做出相应");

}


}


package net;


public class ConcreteObserver2 implements Observer{


public void update(Observable o) {

System.out.println("观察者2观察到" +
o.getClass().getSimpleName() + "发生变化");
```

```
System.out.println("观察者2做出相应");

}



}
```

下面是被观察者，它有一个观察者的列表，并且有一个通知所有观察者的方法，通知的方式就是调用观察者通用的接口行为update方法。下面我们看它的代码。

```
package net;



import java.util.ArrayList;

import java.util.List;



public class Observable {



List<Observer> observers = new ArrayList<Observer>();



public void addObserver(Observer o){

observers.add(o);

}



public void changed(){
```

```
System.out.println("我是被观察者，我已经发生变化了");

notifyObservers();//通知观察自己的所有观察者

}


public void notifyObservers(){

for (Observer observer : observers) {

observer.update(this);

}

}

}
```

这里面很简单，新增两个方法，一个是为了改变自己的同时通知观察者们，一个是为了给客户端一个添加观察者的公共接口。

下面我们使用客户端调用一下，看一下客户端如何操作。

```
package net;



public class Client {


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

Observable observable = new Observable();

observable.addObserver(new ConcreteObserver1());
```

```
observable.addObserver(new ConcreteObserver2());

observable.changed();

}

}
```

运行结果如下。

可以看到我们在操作被观察者时，只要调用changed方法，观察者们就会做出相应的动作，而添加观察者这个行为算是准备阶段，将具体的观察者关联到被观察者上面去。

下面LZ给出一个有实际意义的例子，比如我们经常看的小说网站，都有这样的功能，就是读者可以订阅作者，这当中就有明显的观察者模式案例，就是作者和读者。他们的关系是一旦读者关注了一个作者，那么这个作者一旦有什么新书，就都要通知读者们，这明显是一个观察者模式的案例，所以我们可以使用观察者模式解决。

由于JDK中为了方便开发人员，已经写好了现成的观察者接口和被观察者类，下面LZ先给出JDK中现成的观察者和被观察者代码，外加自己的一点解释，来帮助一些读者对JDK中对观察者模式的支持熟悉一下。

先来观察者接口。

```
//观察者接口，每一个观察者都必须实现这个接口

public interface Observer {

//这个方法是观察者在观察对象产生变化时所做的响应动作，从中传入了观察的对象和一个预留参数

void update(Observable o, Object arg);
```

```
}
```

下面是被观察者类。

```
import java.util.Vector;

//被观察者类
public class Observable {
//这是一个改变标识，来标记该被观察者有没有改变
private boolean changed = false;
//持有一个观察者列表
private Vector obs;

public Observable() {
obs = new Vector();
}
//添加观察者，添加时会去重
public synchronized void addObserver(Observer o) {
if (o == null)
throw new NullPointerException();
if (!obs.contains(o)) {
```

```
obs.addElement(o);

}

}

//删除观察者

public synchronized void deleteObserver(Observer o) {

obs.removeElement(o);

}

//notifyObservers(Object arg)的重载方法

public void notifyObservers() {

notifyObservers(null);

}

//通知所有观察者，被观察者改变了，你可以执行你的update方法了。

public void notifyObservers(Object arg) {

//一个临时的数组，用于并发访问被观察者时，留住观察者列表的当前状态，这种处理方式其实也算是一种设计模式，即备忘录模式。

Object[] arrLocal;

//注意这个同步块，它表示在获取观察者列表时，该对象是被锁定的

//也就是说，在我获取到观察者列表之前，不允许其他线程改变观察者列表

synchronized (this) {

//如果没变化直接返回
```

```
if (!changed)

return;

//这里将当前的观察者列表放入临时数组

arrLocal = obs.toArray();

//将改变标识重新置回未改变

clearChanged();

}

//注意这个for循环没有在同步块，此时已经释放了被观察者的锁，其他线程可以改变观察者列表

//但是这并不影响我们当前进行的操作，因为我们已经将观察者列表复制到临时数组

//在通知时我们只通知数组中的观察者，当前删除和添加观察者，都不会影响我们通知的对象

for (int i = arrLocal.length-1; i>=0; i--)

((Observer)arrLocal[i]).update(this, arg);

}



//删除所有观察者

public synchronized void deleteObservers() {

obs.removeAllElements();

}
```

```
//标识被观察者被改变过了

protected synchronized void setChanged() {

changed = true;

}

//标识被观察者没改变

protected synchronized void clearChanged() {

changed = false;

}

//返回被观察者是否改变

public synchronized boolean hasChanged() {

return changed;

}

//返回观察者数量

public synchronized int countObservers() {

return obs.size();

}

}
```

被观察者除了一点同步的地方需要特殊解释一下，其余的相信各位都能看明白各个方法的用途。其实上述JDK的类是有漏洞的，或者说，在我们使用观察者模式时要注意一个问题，就是notifyObservers这个方法中的这一段代码。

```
for (int i = arrLocal.length-1; i>=0; i--)
```

```
((Observer)arrLocal[i]).update(this, arg);
```

在循环遍历观察者让观察者做出响应时，JDK没有去抓取update方法中的异常，所以假设在这过程中有一个update方法抛出了异常，那么剩下还未通知的观察者就全都通知不到了，所以LZ个人比较疑惑这样的用意（LZ无法想象JAVA类库的制造者没考虑到这个问题），是sun当时真的忘了考虑这一点，还是另有它意？当然各位读者如果有自己的见解可以告知LZ，不过LZ认为，不管是sun如此做是别有用意，还是真的欠考虑，我们都要注意在update方法里一定要处理好异常，个人觉得JDK中比较保险的做法还是如下这样。

```
for (int i = arrLocal.length-1; i>=0; i--){

   try {

      ((Observer)arrLocal[i]).update(this, arg);

   } catch (Throwable e) {e.printStackTrace();}

}
```

这样无论其中任何一个update是否成功都不会影响其余的观察者进行更新状态，我们自己比较保险的做法就是给update方法整个加上try块，或者确认不会发生运行时异常。

上面LZ和各位一起分析了JDK中观察者模式的源码，下面我们就拿上述小说网的例子，做一个DEMO。

首先要搞清楚在读者和作者之间是谁观察谁，很明显，应该是读者观察作者。所以作者是被观察者，读者是观察者，除了这两个类之外，我们还需要额外添加一个管理器帮我们管理下作者的列表便于读者关注，于是一个观察者模式的DEMO就出现了。如下，首先是读者类，LZ在各个类都加了点注释。

```
//读者类，要实现观察者接口

public class Reader implements Observer{
```

```
private String name;

public Reader(String name) {
super();
this.name = name;
}

public String getName() {
return name;
}

//读者可以关注某一位作者，关注则代表把自己加到作者的观察者列表里
public void subscribe(String writerName){
WriterManager.getInstance().getWriter(writerName).addObserver(this);
}

//读者可以取消关注某一位作者，取消关注则代表把自己从作者的观察者列表里删除
public void unsubscribe(String writerName){
```

```
WriterManager.getInstance().getWriter(writerName).deleteObserver(this);

}



//当关注的作者发表新小说时，会通知读者去看

public void update(Observable o, Object obj) {

if (o instanceof Writer) {

Writer writer = (Writer) o;

System.out.println(name+"知道" + writer.getName() + "发布了新书《" + writer.getLastNovel() + "》，非要去看！");

}

}



}
```

下面是作者类。

```
//作者类，要继承自被观察者类

public class Writer extends Observable{



private String name;//作者的名称



private String lastNovel;//记录作者最新发布的小说
```

```
public Writer(String name) {
super();
this.name = name;
WriterManager.getInstance().add(this);
}

//作者发布新小说了，要通知所有关注自己的读者
public void addNovel(String novel) {
System.out.println(name + "发布了新书《" + novel + "》！");
lastNovel = novel;
setChanged();
notifyObservers();
}

public String getLastNovel() {
return lastNovel;
}

public String getName() {
```

```
return name;
}


}
```

然后我们还需要一个管理器帮我们管理这些作者。如下。

```
import java.util.HashMap;
import java.util.Map;

//管理器，保持一份独有的作者列表
public class WriterManager{

private Map<String, Writer> writerMap = new HashMap<String,
Writer>();

//添加作者
public void add(Writer writer){
writerMap.put(writer.getName(), writer);
}
//根据作者姓名获取作者
public Writer getWriter(String name){
return writerMap.get(name);
```

```
}

//单例
private WriterManager(){}

public static WriterManager getInstance(){
return WriterManagerInstance.instance;
}
private static class WriterManagerInstance{
private static WriterManager instance = new WriterManager();
}
}
```

好了，这下我们的观察者模式就做好了，这个简单的DEMO可以支持读者关注作者，当作者发布新书时，读者会观察到这个事情，会产生相应的动作。下面我们写个客户端调用一下。

```
//客户端调用
public class Client {

public static void main(String[] args) {
//假设四个读者，两个作者
Reader r1 = new Reader("谢广坤");
```

```
Reader r2 = new Reader("赵四");
Reader r3 = new Reader("七哥");
Reader r4 = new Reader("刘能");
Writer w1 = new Writer("谢大脚");
Writer w2 = new Writer("王小蒙");
//四人关注了谢大脚
r1.subscribe("谢大脚");
r2.subscribe("谢大脚");
r3.subscribe("谢大脚");
r4.subscribe("谢大脚");
//七哥和刘能还关注了王小蒙
r3.subscribe("王小蒙");
r4.subscribe("王小蒙");
//作者发布新书就会通知关注的读者
//谢大脚写了设计模式
w1.addNovel("设计模式");
//王小蒙写了JAVA编程思想
w2.addNovel("JAVA编程思想");
//谢广坤取消关注谢大脚
r1.unsubscribe("谢大脚");
```

```
//谢大脚再写书将不会通知谢广坤

w1.addNovel("观察者模式");

}



}
```

看下我们得到的结果，就会发现，我们确实通知了读者它所关注的作者的动态，而且读者取消关注以后，作者的动态将不再通知该读者。下面是运行结果。

我们使用观察者模式的用意是为了作者不再需要关心他发布新书时都要去通知谁，更重要的是他不需要关心他通知的是读者还是其它什么人，他只知道这个人是实现了观察者接口的，即我们的被观察者依赖的只是一个抽象的接口观察者接口，而不关心具体的观察者都有谁都是什么，比如以后要是游客也可以关注作者了，那么只要游客类实现观察者接口，那么一样可以将游客列入到作者的观察者列表中。

另外，我们让读者自己来选择自己关注的对象，这相当于被观察者将维护通知对象的职能转化给了观察者，这样做的好处是由于一个被观察者可能有N多观察者，所以让被观察者自己维护这个列表会很艰难，这就像一个老师被许多学生认识，那么是所有的学生都记住老师的名字简单，还是让老师记住N多学生的名字简单？答案显而易见，让学生们都记住一个老师的名字是最简单的。

另外，观察者模式分离了观察者和被观察者二者的责任，这样让类之间各自维护自己的功能，专注于自己的功能，会提高系统的可维护性和可重用性。

观察者模式其实还有另外一种形态，就是事件驱动模型，LZ个人觉得这两种方式大体上其实是非常相似的，所以LZ决定一起引入事件驱动模型。不过观察者更多的强调的是发布-订阅式的问题处理，而事件驱动则更多的注重于界面与数据模型之间的问题，两者还是有很多适用

场景上的区别的，虽不能一概而论，但放在一起讨论还是很方便各位理解二者。

说到事件驱动，由于JAVA在桌面应用程序方面有很多欠缺，所以swing的使用其实并不是特别广泛，因为你不可能要求大多数人的机子上都安装了JDK，除非你是给特殊用户人群开发的应用程序，这些用户在你的可控范围内，那么swing或许可以派上用场。

考虑到学习JAVA或者使用JAVA的人群大部分都是在进行web开发，所以本次讨论事件驱动，采用web开发当中所用到的示例。

相信各位都知道tomcat，这是一个app服务器，在使用的过程中，或许经常会有人用到listener，即监听器这个概念。那么其实这个就是一个事件驱动模型的应用。比如我们的spring，我们在应用启动的时候要初始化我们的IOC容器，那么我们的做法就是加入一个listener，这样伴随着tomcat服务器的启动，spring的IOC容器就会跟着启动。

那么这个listener其实就是事件驱动模型中的监听器，它用来监听它所感兴趣的事，比如我们springIOC容器启动的监听器，就是实现的ServletContextListener这个接口，说明它对servletContext感兴趣，会监听servletContext的启动和销毁。

LZ不打算使用这个例子作为讲解，因为它的内部运作比较复杂，需要搬上来tomcat的源码，对于新手来说，这是个噩耗，所以我们将上述的例子改为事件驱动来实现。也好让各位针对性的对比观察者模式和事件驱动模型。

首先事件驱动模型与观察者模式勉强的对应关系可以看成是，被观察者相当于事件源，观察者相当于监听器，事件源会产生事件，监听器监听事件。所以这其中就搀和到四个类，事件源，事件，监听器以及具体的监听器。

JDK当中依然有现成的一套事件模型类库，其中监听器只是一个标识接口，因为它没有表达对具体对象感兴趣的意思，所以也无法定义监听的事件，只是为了统一，用来给特定的监听器继承。它的源代码如下。

```
package java.util;
```

```
/**

  * A tagging interface that all event listener interfaces
must extend.

  * @since JDK1.1

  */

public interface EventListener {

}
```

由于代码很短，所以LZ没有删减，当中标注了，所有的事件监听器都必须继承，这是一个标识接口。上述的事件，JDK当中也有一个现成的类供继承，就是EventObject，这个类的源代码如下。

```
public class EventObject implements java.io.Serializable {



private static final long serialVersionUID =
5516075349620653480L;



/**

  * The object on which the Event initially occurred.

  */

protected transient Objectsource;



/**
```

```
  * Constructs a prototypical Event.

  *

  * @param      sourceThe object on which the Event initially
occurred.

  * @exception   IllegalArgumentException   if source is
null.

  */

public EventObject(Object source) {

if (source == null)

throw new IllegalArgumentException("null source");



this.source = source;

}



/**

  * The object on which the Event initially occurred.

  *

  * @return    The object on which the Event initially
occurred.

  */

public Object getSource() {

return source;
```

```
}

/**
  * Returns a String representation of this EventObject.
  *
  * @return   A a String representation of this EventObject.
  */
public String toString() {
return getClass().getName() + "[source=" + source + "]";
}
}
```

这个类并不复杂，它只是想表明，所有的事件都应该带有一个事件源，大部分情况下，这个事件源就是我们被监听的对象。

如果我们采用事件驱动模型去分析上面的例子，那么作者就是事件源，而读者就是监听器，依据这个思想，我们把上述例子改一下，首先我们需要自定义我们自己的监听器和事件。所以我们定义如下作者事件。

```
import java.util.EventObject;


public class WriterEvent extends EventObject{


private static final long serialVersionUID =
8546459078247503692L;
```

```
public WriterEvent(Writer writer) {

super(writer);

}



public Writer getWriter(){

return (Writer) super.getSource();

}



}
```

这代表了一个作者事件，这个事件当中一般就是包含一个事件源，在这里就是作者，当然有的时候你可以让它带有更多的信息，以方便监听器做出更加细致的动作。下面我们定义如下监听器。

```
import java.util.EventListener;



public interface WriterListener extends EventListener{



void addNovel(WriterEvent writerEvent);



}
```

这个监听器猛地一看，特别像观察者接口，它们承担的功能是类似的，都是提供观察者或者监听者实现自己响应的行为规定，其中

addNovel方法代表的是作者发布新书时的响应。加入了这两个类以后，我们原有的作者和读者类就要发生点变化了，我们先来看作者类的变化。

```
import java.util.HashSet;

import java.util.Set;



//作者类

public class Writer{



private String name;//作者的名称



private String lastNovel;//记录作者最新发布的小说



private Set<WriterListener> writerListenerList = new
HashSet<WriterListener>();//作者类要包含一个自己监听器的列表



public Writer(String name) {

super();

this.name = name;

WriterManager.getInstance().add(this);

}
```

```
//作者发布新小说了，要通知所有关注自己的读者
public void addNovel(String novel) {
System.out.println(name + "发布了新书《" + novel + "》！");
lastNovel = novel;
fireEvent();
}
//触发发布新书的事件，通知所有监听这件事的监听器
private void fireEvent(){
WriterEvent writerEvent = new WriterEvent(this);
for (WriterListener writerListener : writerListenerList) {
writerListener.addNovel(writerEvent);
}
}
//提供给外部注册成为自己的监听器的方法
public void registerListener(WriterListener writerListener){
writerListenerList.add(writerListener);
}
//提供给外部注销的方法
public void unregisterListener(WriterListener writerListener)
{
writerListenerList.remove(writerListener);
```

```
}

public String getLastNovel() {
return lastNovel;
}

public String getName() {
return name;
}

}
```

可以看到，作者类的主要变化是添加了一个自己的监听器列表，我们使用set是为了它的天然去重效果，并且提供给外部注册和注销的方法，与观察者模式相比，这个功能本身是由基类Observable提供的，不过观察者模式中有统一的观察者Observer接口，但是监听器没有，虽说有EventListener这个超级接口，但它毕竟没有任何行为。所以我们一般需要维持一个自己特有的监听器列表。

下面我们看读者类的变化，如下。

```
public class Reader implements WriterListener{

private String name;

public Reader(String name) {
```

```
super();

this.name = name;

}



public String getName() {

return name;

}



//读者可以关注某一位作者，关注则代表把自己加到作者的监听器列表里

public void subscribe(String writerName){

WriterManager.getInstance().getWriter(writerName).registerListener(this);

}



//读者可以取消关注某一位作者，取消关注则代表把自己从作者的监听器列表里注销

public void unsubscribe(String writerName){

WriterManager.getInstance().getWriter(writerName).unregisterListener(this);

}
```

```
public void addNovel(WriterEvent writerEvent) {

Writer writer = writerEvent.getWriter();

System.out.println(name+"知道" + writer.getName() + "发布了新书《" + writer.getLastNovel() + "》，非要去看！");

}



}
```

读者类的变化，首先本来是实现Observer接口，现在要实现WriterListener接口，响应的update方法就改为我们定义的addNovel方法，当中的响应基本没变。另外就是关注和取消关注的方法中，原来是给作者类添加观察者和删除观察者，现在是注册监听器和注销监听器，几乎是没什么变化的。

我们彻底将刚才的观察者模式改成了事件驱动，现在我们使用事件驱动的类再运行一下客户端，其中客户端代码和WriterManager类的代码是完全不需要改动的，直接运行客户端即可。我们会发现得到的结果与观察者模式一模一样。

走到这里我们发现二者可以达到的效果一模一样，那么两者是不是一样呢？

答案当然是否定的，首先我们从实现方式上就能看出，事件驱动可以解决观察者模式的问题，但反过来则不一定，另外二者所表达的业务场景也不一样，比如上述例子，使用观察者模式更贴近业务场景的描述，而使用事件驱动，从业务上讲，则有点勉强。

二者除了业务场景的区别以外，在功能上主要有以下区别。

1，观察者模式中观察者的响应理论上讲针对特定的被观察者是唯一的（说理论上唯一的原因是，如果你愿意，你完全可以在update方法里添加一系列的elseif去产生不同的响应，但LZ早就说过，你应该忘掉elseif），而事件驱动则不是，因为我们可以定义自己感兴趣的事情，比如刚才，我们可以监听作者发布新书，我们还可以在监听器接

口中定义其它的行为。再比如tomcat中，我们可以监听servletcontext的init动作，也可以监听它的destroy动作。

2，虽然事件驱动模型更加灵活，但也是付出了系统的复杂性作为代价的，因为我们要为每一个事件源定制一个监听器以及事件，这会增加系统的负担，各位看看tomcat中有多少个监听器和事件类就知道了。

3，另外观察者模式要求被观察者继承Observable类，这就意味着如果被观察者原来有父类的话，就需要自己实现被观察者的功能，当然，这一尴尬事情，我们可以使用适配器模式弥补，但也不可避免的造成了观察者模式的局限性。事件驱动中事件源则不需要，因为事件源所维护的监听器列表是给自己定制的，所以无法去制作一个通用的父类去完成这个工作。

4，被观察者传送给观察者的信息是模糊的，比如update中第二个参数，类型是Object，这需要观察者和被观察者之间有约定才可以使用这个参数。而在事件驱动模型中，这些信息是被封装在Event当中的，可以更清楚的告诉监听器，每个信息都是代表的什么。

由于上述使用事件驱动有点勉强，所以LZ给各位模拟一个我们js当中的一个事件驱动模型，就是按钮的点击事件。

在这个模型当中，按钮自然就是事件源，而事件的种类有很多，比如点击（click），双击（dblclick），鼠标移动事件（mousemove）。我们的监听器与事件个数是一样的，所以这也是事件驱动的弊端，我们需要一堆事件和监听器，下面LZ一次性给出这三种事件和监听器，其余还有很多事件，类似，LZ这里省略。

```
import java.util.EventObject;

//按钮事件基类

public abstract class ButtonEvent extends EventObject{



public ButtonEvent(Object source) {

    super(source);
```

```
}

public Button getButton(){
return (Button) super.getSource();
}
}
//点击事件
class ClickEvent extends ButtonEvent{

public ClickEvent(Object source) {
super(source);
}

}
//双击事件
class DblClickEvent extends ButtonEvent{

public DblClickEvent(Object source) {
super(source);
}
```

```
}

//鼠标移动事件

class MouseMoveEvent extends ButtonEvent{

//鼠标移动事件比较特殊，因为它需要告诉监听器鼠标当前的坐标是在哪，我们记录为x,y

private int x;

private int y;


public MouseMoveEvent(Object source, int x, int y) {

super(source);

this.x = x;

this.y = y;

}


public int getX() {

return x;

}


public int getY() {

return y;
```

```
}


}
```

以上是三种事件，都非常简单，只有鼠标移动需要额外的坐标，下面给出三种监听器。

```
import java.util.EventListener;

//点击监听器

interface ClickListener extends EventListener{


void click(ClickEvent clickEvent);


}


//双击监听器

interface DblClickListener extends EventListener{


void dblClick(DblClickEvent dblClickEvent);


}


//鼠标移动监听器
```

```
interface MouseMoveListener extends EventListener{

void mouseMove(MouseMoveEvent mouseMoveEvent);

}
```

三种监听器分别监听点击，双击和鼠标移动。下面给出我们最重要的类，Button。

```
//我们模拟一个html页面的button元素，LZ只添加个别属性，其余属性同理

public class Button {

private String id;//这相当于id属性

private String value;//这相当于value属性

private ClickListener onclick;//我们完全模拟原有的模型，这个其实相当于onclick属性

private DblClickListener onDblClick;//同理，这个相当于双击属性

private MouseMoveListener onMouseMove;//同理

//按钮的单击行为

public void click(){

onclick.click(new ClickEvent(this));
```

```
}
//按钮的双击行为
public void dblClick(){
onDblClick.dblClick(new DblClickEvent(this));
}
//按钮的鼠标移动行为
public void mouseMove(int x,int y){
onMouseMove.mouseMove(new MouseMoveEvent(this,x,y));
}
//相当于给id赋值
public void setId(String id) {
this.id = id;
}
//类似
public void setValue(String value) {
this.value = value;
}
//这个相当于我们在给onclick添加函数，即设置onclick属性
public void setOnclick(ClickListener onclick) {
this.onclick = onclick;
```

```
}
//同理
public void setOnDblClick(DblClickListener onDblClick) {
this.onDblClick = onDblClick;
}
//同理
public void setOnMouseMove(MouseMoveListener onMouseMove) {
this.onMouseMove = onMouseMove;
}
//以下get方法
public String getId() {
return id;
}

public String getValue() {
return value;
}

public ClickListener getOnclick() {
return onclick;
```

```
}

public DblClickListener getOnDblClick() {
return onDblClick;
}

public MouseMoveListener getOnMouseMove() {
return onMouseMove;
}
}
```

可以看到，按钮Button类有很多属性，都是我们经常看到的，id，value，onclick等等。下面我们模拟编写一个页面，这个页面可以当做是一个JSP页面，我们只有一个按钮，我们用JAVA语言把它描述出来，如下。

```
//假设这个是我们写的某一个特定的jsp页面，里面可能有很多元素，input，form，table，等等
//我们假设只有一个按钮
public class ButtonJsp {

private Button button;

public ButtonJsp() {
```

```
super();

button = new Button();//这个可以当做我们在页面写了一个button元素

button.setId("submitButton");//取submitButton为id

button.setValue("提交");//提交按钮

button.setOnclick(new ClickListener() {//我们给按钮注册点击监听器

//按钮被点，我们就验证后提交

public void click(ClickEvent clickEvent) {

System.out.println("--------单击事件代码---------");

System.out.println("if('表单合法'){");

System.out.println("\t表单提交");

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

System.out.println("\treturn false");

System.out.println("}");

}

});

button.setOnDblClick(new DblClickListener() {

//双击的话我们提示用户不能双击“提交”按钮

public void dblClick(DblClickEvent dblClickEvent) {

System.out.println("--------双击事件代码---------");
```

```
System.out.println("alert('您不能双
击"+dblClickEvent.getButton().getValue()+"按钮')");

}

});

button.setOnMouseMove(new MouseMoveListener() {

//这个我们只简单提示用户鼠标当前位置，示例中加入这个事件

//目的只是为了说明事件驱动中，可以包含一些特有的信息，比如坐
标

public void mouseMove(MouseMoveEvent mouseMoveEvent) {

System.out.println("--------鼠标移动代码---------");

System.out.println("alert('您当前鼠标的位置，x坐标
为："+mouseMoveEvent.getX()+"，y坐标
为："+mouseMoveEvent.getY()+"')");

}

});

}



public Button getButton() {

return button;

}



}
```

以上可以认为我们给web服务中写了一个简单的页面，下面我们看客户在访问我们的页面时，我们的页面在做什么。

```
public class Client {


public static void main(String[] args) {

ButtonJsp jsp = new ButtonJsp();//客户访问了我们的这个JSP页面

//以下客户开始在按钮上操作

jsp.getButton().dblClick();//双击按钮

jsp.getButton().mouseMove(10, 100);//移动到10，100

jsp.getButton().mouseMove(15, 90);//又移动到15,90

jsp.getButton().click();//接着客户点了提交

}

}
```

我们看运行结果可以看到，我们的三个事件都起了作用，最终提交了表单。

以上就是模拟整个JSP页面中，我们的按钮响应用户事件的过程，我相信通过这两个例子，各位应该对观察者模式和事件驱动都有了自己的理解和认识，二者都是用来处理变化与响应的问题，其中观察者更多的是发布-订阅，也就是类似读者和作者的关系，而事件驱动更多的是为了响应客户的请求，从而制定一系列的事件和监听器，去处理客户的请求与操作。

二者其实都是有自己的弱项的，只有掌握了模式的弱项才能更好的使用，不是有句话叫“真正了解一个东西，不是知道它能干什么，而是知道它不能干什么。”吗？

观察者模式所欠缺的是设计上的问题，即观察者和被观察者是多对一的关系，那么反过来的话，就无法支持了。

各位可以尝试将二者位置互换达到这个效果，这算是设计模式的活用，很简单，就是让被观察者做成一个接口，提供是否改变的方法，让观察者维护一个被观察者的列表，另外开启一个线程去不断的测试各个被观察者是否改变。由于本篇已经够长，所以LZ不再详细编写，如果有哪位读者有需要，可以在下方留言，LZ看到的话，如果有时间，会写出来放到资源里供各位下载。

观察者模式还有一个缺点就是，每一个观察者都要实现观察者接口，才能添加到被观察者的列表当中，假设一个观察者已经存在，而且我们无法改变其代码，那么就无法让它成为一个观察者了，不过这个我们依然可以使用适配器模式解决。但是还有一个问题就不好解决了，就是假如我们很多类都是现成的，当被观察者发生变化时，每一个观察者都需要调用不同的方法，那么观察者模式就有点捉襟见肘的感觉了，我们必须适配每一个类去统一他们变化的方法名称为update，这是一个很可怕的事情。

对于事件驱动就没有这样的问题，我们可以实现多个监听器来达到监听多个事件源的目的，但是它的缺点刚才已经说过了，在事件源或者事件增加时，监听器和事件类通常情况下会成对增加，造成系统的复杂性增加，不过目前看来，事件驱动模型一般都比较稳定，所以这个问题并不太明显，因为很少见到无限增加事件的情况发生。

还有一个缺点就是我们的事件源需要看准时机触发自己的各个监听器，这也从某种意义上增加了事件源的负担，造成了类一定程度上的臃肿。

最后，LZ再总结下二者针对的业务场景概述。

观察者模式：发布（release）--订阅（subscibe），变化（change）--更新（update）

事件驱动模型：请求（request）--响应（response），事件发生（occur）--事件处理（handle）

（七）策略模式详解

上章我们着重讲解了观察者模式和事件驱动，那么本章来讨论一个个人认为在开发过程中出场率极高的设计模式，策略模式。

策略模式在LZ第一次接触到的时候，LZ是这么理解的，就是如果我们想往一个方法当中插入随便一段代码的话，就是策略模式。即如下形式。

```
public class MyClass {

	public void myMethod(){

		System.out.println("方法里的代码");

		//LZ想在这插入一段代码，而且这个代码是可以改变的，想怎么变就怎么变

		System.out.println("方法里的代码");

	}

}
```

在JAVA中，接口可以满足LZ的这一过分要求，我们可以设计一个接口，并当做参数传进去，就能达到这个效果了。我们来看，先定义一个接口。

```
public interface MyInterface {

	//我想插入的代码

	void insertCode();

}
```

将原来的类改成这样，传递一个接口进去。

```
public class MyClass {

	public void myMethod(MyInterface myInterface){

		System.out.println("方法里的代码");

		//你看我是不是插进来一段代码？而且这段代码是可以随便改变的

		myInterface.insertCode();

		System.out.println("方法里的代码");

	}

}
```

我们只要实现了MyInterface这个接口，在insertCode方法中写入我们想要插进去的代码，再将这个类传递给myMethod方法，就可以将我们随手写的代码插到这个方法当中。比如这样。

```
class InsertCode1 implements MyInterface{

	public void insertCode() {

		System.out.println("我想插进去的代码，第一种");

	}

}



class InsertCode2 implements MyInterface{

	public void insertCode() {

		System.out.println("我想插进去的代码，第二种");
```

```
        }

}
```

这样我们在调用myMethod方法时就可以随意往里面插入代码了，比如。

```
//客户端调用

public class Client {

        public static void main(String[] args) {

                MyClass myClass = new MyClass();

                myClass.myMethod(new InsertCode1());

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

                myClass.myMethod(new InsertCode2());

        }



}
```

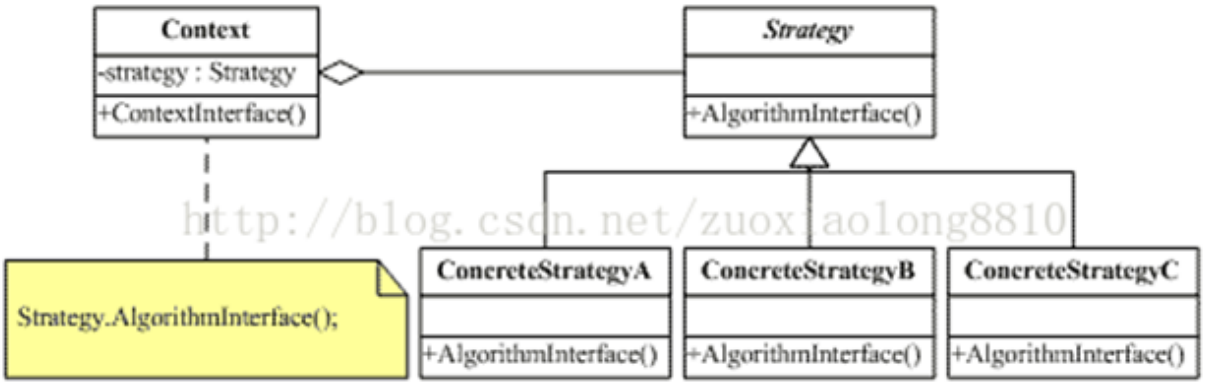
那么运行出来的结果就是我们成功的将两端代码插入到了myMethod方法中，以上所讲的算是JAVA中一种技术层面的实现，就是传入一个接口，封装代码。那么既然谈到设计模式，就要有设计模式的应用场景，有关策略模式，所产生的形式就和上述是一模一样的，只是我们适当的给予模式的应用场景，就会让它变的更有价值。

下面LZ给出策略模式的标准定义，引自百度百科。

定义：策略模式定义了一系列的算法，并将每一个算法封装起来，而且使它们还可以相互替换。策略模式让算法独立于使用它的客户而独立变化。

分析下定义，策略模式定义和封装了一系列的算法，它们是可以相互替换的，也就是说它们具有共性，而它们的共性就体现在策略接口的行为上，另外为了达到最后一句话的目的，也就是说让算法独立于使用它的客户而独立变化，我们需要让客户端依赖于策略接口。

下面给出策略模式的类图，引自百度百科。



http://blog.csdn.net/zuoxiaolong8810

这个类图并不复杂，右边是策略接口以及它的实现类，左边会有一个上下文，这个上下文会拥有一个策略，而具体这个策略是哪一种，我们是可以随意替换的。

LZ下面使用JAVA代码诠释上面的类图，方便各位理解各个类之间的关系。

首先是策略接口以及它的实现类。

```
package net;



public interface Strategy {

      void algorithm();

}
```

```
class ConcreteStrategyA implements Strategy{
	public void algorithm() {
		System.out.println("采用策略A计算");
	}
}
class ConcreteStrategyB implements Strategy{
	public void algorithm() {
		System.out.println("采用策略B计算");
	}
}
class ConcreteStrategyC implements Strategy{
	public void algorithm() {
		System.out.println("采用策略C计算");
	}
}
```

下面是我们的上下文，它会拥有一个策略接口。

```
package net;

public class Context {
	Strategy strategy;
```

```
	public void method(){
		strategy.algorithm();
	}

	public void setStrategy(Strategy strategy) {
		this.strategy = strategy;
	}
}
```

method方法是上下文类的一个公开方法，实际当中一般会和业务相关，这里就暂且取名为method方法。下面我们使用客户端调用一下。

```
package net;

public class Client {
	public static void main(String[] args) throws Exception
{
		Context context = new Context();
		context.setStrategy(new ConcreteStrategyA());
		context.method();

		context.setStrategy(new ConcreteStrategyB());
```

```
            context.method();


            context.setStrategy(new ConcreteStrategyC());

            context.method();

        }

}
```

上面我们替换了两次策略，但是调用方式不变，下面我们看下运行结果。

上面的例子代码清晰但却理解起来很生硬，下面LZ举一个具有实际意义的例子。

就比如我们要做一个商店的收银系统，这个商店有普通顾客，会员，超级会员以及金牌会员的区别，针对各个顾客，有不同的打折方式，并且一个顾客每在商店消费1000就增加一个级别，那么我们就可以使用策略模式，因为**策略模式描述的就是算法的不同**，而且这个算法往往非常繁多，并且可能需要经常性的互相替换。

这里我们举例就采用最简单的，以上四种顾客分别采用原价，八折，七折和半价的收钱方式。

那么我们首先要有一个计算价格的策略接口，如下。

```
public interface CalPrice {

        //根据原价返回一个最终的价格

        Double calPrice(Double originalPrice);

}
```

下面我们给出四个计算方式。

```
class Common implements CalPrice{
	public Double calPrice(Double originalPrice) {
		return originalPrice;
	}
}
class Vip implements CalPrice{
	public Double calPrice(Double originalPrice) {
		return originalPrice * 0.8;
	}
}
class SuperVip implements CalPrice{
	public Double calPrice(Double originalPrice) {
		return originalPrice * 0.7;
	}
}
class GoldVip implements CalPrice{
	public Double calPrice(Double originalPrice) {
		return originalPrice * 0.5;
	}
}
```

以上四种计算方式非常清晰，分别是原价，八折，七折和半价。下面我们看客户类，我们需要客户类帮我们完成客户升级的功能。

```
//客户类

public class Customer {

	private Double totalAmount = 0D;//客户在本商店消费的总额

	private Double amount = 0D;//客户单次消费金额

	private CalPrice calPrice = new Common();//每个客户都有一个计算价格的策略，初始都是普通计算，即原价



	//客户购买商品，就会增加它的总额

	public void buy(Double amount){

		this.amount = amount;

		totalAmount += amount;

		if (totalAmount > 3000) {//3000则改为金牌会员计算方式

			calPrice = new GoldVip();

		}else if (totalAmount > 2000) {//类似

			calPrice = new SuperVip();

		}else if (totalAmount > 1000) {//类似

			calPrice = new Vip();

		}
```

```
    }
    //计算客户最终要付的钱
    public Double calLastAmount(){
        return calPrice.calPrice(amount);
    }
}
```

下面我们看客户端调用，系统会帮我们自动调整收费策略。

```
//客户端调用
public class Client {
    public static void main(String[] args) {
        Customer customer = new Customer();
        customer.buy(500D);
        System.out.println("客户需要付钱：" +
customer.calLastAmount());
        customer.buy(1200D);
        System.out.println("客户需要付钱：" +
customer.calLastAmount());
        customer.buy(1200D);
        System.out.println("客户需要付钱：" +
customer.calLastAmount());
        customer.buy(1200D);
```

```
            System.out.println("客户需要付钱：" +
customer.calLastAmount());

      }

}
```

运行以后会发现，第一次是原价，第二次是八折，第三次是七折，最后一次则是半价。我们这样设计的好处是，客户不再依赖于具体的收费策略，依赖于抽象永远是正确的。不过上述的客户类实在有点难看，尤其是buy方法，我们可以使用简单工厂来稍微改进一下它。我们建立如下策略工厂。

```
//我们使用一个标准的简单工厂来改进一下策略模式

public class CalPriceFactory {

      private CalPriceFactory(){}

      //根据客户的总金额产生相应的策略

      public static CalPrice createCalPrice(Customer
customer){

            if (customer.getTotalAmount() > 3000) {//3000则改
为金牌会员计算方式

                  return new GoldVip();

            }else if (customer.getTotalAmount() > 2000) {//类
似

                  return new SuperVip();

            }else if (customer.getTotalAmount() > 1000) {//类
似

                  return new Vip();
```

```
            }else {
                return new Common();
            }
        }
}
```

这样我们就将制定策略的功能从客户类分离了出来，我们的客户类可以变成这样。

```
//客户类
public class Customer {
        private Double totalAmount = 0D;//客户在本商店消费的总额
        private Double amount = 0D;//客户单次消费金额
        private CalPrice calPrice = new Common();//每个客户都有一个计算价格的策略，初始都是普通计算，即原价


        //客户购买商品，就会增加它的总额
        public void buy(Double amount){
            this.amount = amount;
            totalAmount += amount;
            /* 变化点，我们将策略的制定转移给了策略工厂，将这部分责任分离出去 */
            calPrice = CalPriceFactory.createCalPrice(this);
```

```
    }
    //计算客户最终要付的钱
    public Double calLastAmount(){
        return calPrice.calPrice(amount);
    }

    public Double getTotalAmount() {
        return totalAmount;
    }

    public Double getAmount() {
        return amount;
    }
}
```

现在比之前来讲，我们的策略模式更加灵活一点，但是相信看过LZ博文的都知道，LZ最不喜欢elseif，所以策略模式也是有缺点的，就是当策略改变时，我们需要使用elseif去判断到底使用哪一个策略，哪怕使用简单工厂，也避免不了这一点。比如我们又添加一类会员，那么你需要去添加elseif。再比如我们的会员现在打九折了，那么你需要添加一个九折的策略，这没问题，我们对扩展开放，但是你需要修改elseif的分支，将会员的策略从八折替换为九折，这是简单工厂的诟病，在之前已经提到过，对修改开放。

在简单工厂模式一章中，LZ就遗留了一堆的elseif，并在工厂方法一章提供了解决方案，但是LZ没有写上来，这次LZ将整个解决方案搬上

来，之前的简单工厂可以使用相同的方法改进。

LZ在工厂方法一章中已经指明可以使用注解来处理这一问题，但是LZ只给出了实现的思路，没有给出具体实现的方式，本次刚好将之前的实现方式补上，如果各位掌握了这里的实现方式，那么对于简单工厂模式那一章的问题也自然而然可以轻松解决。

不过简单工厂那一章的问题相对这里更简单一点，因为那里我们只需要做一个servlet名称与Class引用的映射关系，就可以消除掉elseif，使用现有的映射直接创造servlet实例，但是在这里就不行了，因为我们涉及到了客户总金额的判断，这不再是一个简单的名称与策略的对应关系。

所以我们需要给注解加入属性上限和下限，用来表示策略生效的区间，用来解决总金额判断的问题。

下面我们一步一步来，首先我们做一个注解，这个注解是用来给策略添加的，当中可以设置它的上下限，我们来看。

```
package com.calprice;

import java.lang.annotation.ElementType;

import java.lang.annotation.Retention;

import java.lang.annotation.RetentionPolicy;

import java.lang.annotation.Target;



//这是我们的有效区间注解，可以给策略添加有效区间的设置

@Target(ElementType.TYPE)//表示只能给类添加该注解

@Retention(RetentionPolicy.RUNTIME)//这个必须要将注解保留在运行时

public @interface TotalValidRegion {
```

```
        //为了简单，我们让区间只支持整数
      int max() default Integer.MAX_VALUE;
      int min() default Integer.MIN_VALUE;
}
```

这个注解很简单，我们只是用它来记录每一个策略的生效区间，下面我们就可以在我们的各个策略类里去设置我们的生效区间了，我们将策略类全部改成如下形式。

```
package com.calprice;


@TotalValidRegion(max=1000)//设置普通的在0-1000生效，以下类似，不再注释
class Common implements CalPrice{


      public Double calPrice(Double originalPrice) {
            return originalPrice;
      }


}
@TotalValidRegion(min=1000,max=2000)
class Vip implements CalPrice{


      public Double calPrice(Double originalPrice) {
```

```
            return originalPrice * 0.8;
      }

}
@TotalValidRegion(min=2000,max=3000)
class SuperVip implements CalPrice{

      public Double calPrice(Double originalPrice) {
            return originalPrice * 0.7;
      }

}
@TotalValidRegion(min=3000)
class GoldVip implements CalPrice{

      public Double calPrice(Double originalPrice) {
            return originalPrice * 0.5;
      }

}
```

好了，我们使用注解表示每一个策略生效的区间，下面我们要做最重要的工作，就是处理注解。我们在策略工厂处理注解。这个类会变的比较复杂一点，所以LZ直接加注释。我们来看策略工厂，即改善后的简单工厂。

```
package com.calprice1;



import java.io.File;

import java.io.FileFilter;

import java.lang.annotation.Annotation;

import java.net.URISyntaxException;

import java.util.ArrayList;

import java.util.List;


//我们使用一个标准的简单工厂来改进一下策略模式

public class CalPriceFactory {



	private static final String CAL_PRICE_PACKAGE =
"com.calprice";//这里是一个常量，表示我们扫描策略的包，这是LZ的包名



	private ClassLoader classLoader =
getClass().getClassLoader();//我们加载策略时的类加载器，我们任何类运行时信息必须来自该类加载器
```

```
    private List<Class<? extends CalPrice>> calPriceList;//策略列表


    //根据客户的总金额产生相应的策略

    public CalPrice createCalPrice(Customer customer){

        //在策略列表查找策略

        for (Class<? extends CalPrice> clazz : calPriceList) {

            TotalValidRegion validRegion = handleAnnotation(clazz);//获取该策略的注解

            //判断金额是否在注解的区间

            if (customer.getTotalAmount() > validRegion.min() && customer.getTotalAmount() < validRegion.max()) {

                try {

                        //是的话我们返回一个当前策略的实例

                        return clazz.newInstance();

                    } catch (Exception e) {

                        throw new RuntimeException("策略获得失败");

                    }

            }

        }
```

```
            throw new RuntimeException("策略获得失败");
        }

        //处理注解，我们传入一个策略类，返回它的注解
        private TotalValidRegion handleAnnotation(Class<? extends CalPrice> clazz){
            Annotation[] annotations = clazz.getDeclaredAnnotations();
            if (annotations == null || annotations.length == 0) {
                return null;
            }
            for (int i = 0; i < annotations.length; i++) {
                if (annotations[i] instanceof TotalValidRegion) {
                    return (TotalValidRegion) annotations[i];
                }
            }
            return null;
        }

        //单例
```

```
	private CalPriceFactory(){
		init();
	}


	//在工厂初始化时要初始化策略列表
	private void init(){
		calPriceList = new ArrayList<Class<? extends CalPrice>>();
		File[] resources = getResources();//获取到包下所有的class文件
		Class<CalPrice> calPriceClazz = null;
		try {
			calPriceClazz = (Class<CalPrice>) classLoader.loadClass(CalPrice.class.getName());//使用相同的加载器加载策略接口
		} catch (ClassNotFoundException e1) {
			throw new RuntimeException("未找到策略接口");
		}
		for (int i = 0; i < resources.length; i++) {
			try {
				//载入包下的类
```

```
                        Class<?> clazz =
classLoader.loadClass(CAL_PRICE_PACKAGE +
"."+resources[i].getName().replace(".class", ""));

                        //判断是否是CalPrice的实现类并且不是
CalPrice它本身，满足的话加入到策略列表

                        if
(CalPrice.class.isAssignableFrom(clazz) && clazz !=
calPriceClazz) {

calPriceList.add((Class<? extends CalPrice>) clazz);

                        }

                  } catch (ClassNotFoundException e) {

                        e.printStackTrace();

                  }

            }

      }

      //获取扫描的包下面所有的class文件

      private File[] getResources(){

            try {

                  File file = new
File(classLoader.getResource(CAL_PRICE_PACKAGE.replace(".",
"/")).toURI());

                  return file.listFiles(new FileFilter() {

                        public boolean accept(File pathname)
{
```

```
                                if
(pathname.getName().endsWith(".class")) {//我们只扫描class文
件

                                      return true;

                                }

                                return false;

                            }

                        });

                } catch (URISyntaxException e) {

                        throw new RuntimeException("未找到策略资
源");

                }

        }



        public static CalPriceFactory getInstance(){

                return CalPriceFactoryInstance.instance;

        }



        private static class CalPriceFactoryInstance{



                private static CalPriceFactory instance = new
CalPriceFactory();
```

```
        }

}
```

上述便是我们改善后的简单工厂，虽然比刚开始的简单工厂复杂了很多，但是我们的收益是很明显的，现在我们随便加入一个策略，并设置好它的生效区间，策略工厂就可以帮我们自动找到适应的策略。LZ在上面已经加了详细的注释，相信可以帮助各位看懂了。在这里再特别说明一点，我们的策略实现类最好放在一个包中，这样我们可以扫描特定的包，可以加快初始化速度。

有了这个基于注解的简单工厂，我们还需要稍微改变下客户类，因为客户类本来是调用的工厂的静态方法，现在我们将工厂做成了单例，所以应该改成如下形式。

```
calPrice =
CalPriceFactory.getInstance().createCalPrice(this);
```

好了，现在各位直接再调用客户端代码，会产生与之前一样的结果，说明我们的策略正确选择了。各位可以自己再试一下，比如到4000就变成四折，然后修改下客户端，测试一下看是否能起效。

现在看来，我们已经使用简单工厂，注解，反射等技术将策略模式优化的非常完美了，我们可以随意新增策略，并且不需要修改原有的任何代码。

但是，其实我们的设计还是有些不完美的，因为它无法支持策略的重叠，这是什么意思呢？

就是说我们同一时间只能采用一种策略，假设我们商店现在有这么一个需求，假设到端午节了，我们商店要采取满1000返200，满2000返400的方式，并且原有的打折还要继续，这就相当于将返现金的活动与打折重叠计算了。

比如我是个金牌会员，假设我买了2000的东西，那么计算方式应该是先减去400为1600，再打五折，为800。最后这个会员只需要付800（靠，这减价有点狠，不过我们只是举个例子，各位不要太在意数字）。

这就相当于将两个策略重叠使用了，我们现在的设计无法支持这种方式。那怎么办？你可能会想，可恶的老板就爱改需求。不过请永远记得，优秀的程序猿面对各种刁难的需求，都可以轻松解决，而不是抱怨需求变的太频繁或者太不合理，因为现实会告诉你，抱怨是没用的，而且我们很多时候可以使用一些编程的技巧去容纳这种变化。

刚才的需求，也是在提醒我们在设计一个系统时要考虑全面，我们虽然不应该考虑一些本不存在或者发生概率很小的需求，但像商店或者商场这种灵活的促销方式，却是我们刚开始就应该考虑到的。

现在我们的需求变了，即我们任意的策略都可以随意组合，并且我们要求工厂帮我们自动判断，并将策略叠加返回给我们。那么针对上面的设计我们还需要改善，如果要改善一个设计，我们就需要考虑现有的设计不能支持什么需求。我们考虑上述设计不能支持什么。LZ列出以下两条。

1，我们只能根据客户消费的总金额去处理，而不能根据客户当次消费的金额去处理。

2，我们的设计只能支持单一策略，不能支持策略叠加。

为了满足这两个要求，我们需要添加一个类型的注解，去针对单次消费产生计费策略，另外，我们需要让策略工厂能够产生叠加的策略接口，那么冲着这个目标，我们首先定义如下三个注解，采用嵌套注解。

```
package com.calprice;



import java.lang.annotation.ElementType;

import java.lang.annotation.Retention;

import java.lang.annotation.RetentionPolicy;

import java.lang.annotation.Target;
```

```
//我们定义一个嵌套注解

@Target(ElementType.ANNOTATION_TYPE)

@Retention(RetentionPolicy.RUNTIME)

public @interface ValidRegion {

      int max() default Integer.MAX_VALUE;

      int min() default Integer.MIN_VALUE;



      //既然可以任意组合，我们就需要给策略定义下顺序，就比如刚才说的2000那个例子，按先返后打折的顺序是800，反过来就是600了。

      //所以我们必须支持这一特性，默认0，为最优先

      int order() default 0;

}
```

我们定义上面这个嵌套注解是为了避免代码的重复，因为这三个属性我们在总额消费的策略注解和单次消费的策略注解中都要包括。下面给出另外两个注解。

```
package com.calprice;

import java.lang.annotation.ElementType;

import java.lang.annotation.Retention;

import java.lang.annotation.RetentionPolicy;

import java.lang.annotation.Target;
```

```
//这是我们的总额有效区间注解，可以给策略添加有效区间的设置

@Target(ElementType.TYPE)//表示只能给类添加该注解

@Retention(RetentionPolicy.RUNTIME)//这个必须要将注解保留在运行时

public @interface TotalValidRegion {

	//我们引用有效区间注解

	ValidRegion value() default @ValidRegion;

}
```

上述这个总额注解与之前的注解基本一样，直接换成了嵌套注解。下面还有一个一次性消费的注解。如下。

```
package com.calprice;



import java.lang.annotation.ElementType;

import java.lang.annotation.Retention;

import java.lang.annotation.RetentionPolicy;

import java.lang.annotation.Target;



//这是我们针对单次消费的有效区间注解，可以给策略添加有效区间的设置

@Target(ElementType.TYPE)//表示只能给类添加该注解

@Retention(RetentionPolicy.RUNTIME)//这个必须要将注解保留在运行时
```

```
public @interface OnceValidRegion{

	//我们引用有效区间注解

	ValidRegion value() default @ValidRegion;

}
```

以上三个注解我们就可以支持刚才的第一个要求了，我们可以针对一次消费进行策略判断，接下来我们需要修改策略工厂，去支持单次消费判断，并且还要支持策略重叠。如下。

```
package com.calprice;



import java.io.File;

import java.io.FileFilter;

import java.lang.annotation.Annotation;

import java.net.URISyntaxException;

import java.util.ArrayList;

import java.util.List;

import java.util.SortedMap;

import java.util.TreeMap;


//我们使用一个标准的简单工厂来改进一下策略模式

public class CalPriceFactory {
```

private static final String CAL_PRICE_PACKAGE = "com.calprice";//这里是一个常量，表示我们扫描策略的包，这是LZ的包名

private ClassLoader classLoader = getClass().getClassLoader();//我们加载策略时的类加载器，我们任何类运行时信息必须来自该类加载器

private List<Class<? extends CalPrice>> calPriceList;//策略列表

//根据客户的总金额产生相应的策略

public CalPrice createCalPrice(Customer customer){

//变化点:为了支持优先级排序，我们采用可排序的MAP支持,这个Map是为了储存我们当前策略的运行时类信息

SortedMap<Integer, Class<? extends CalPrice>> clazzMap = new TreeMap<Integer, Class<? extends CalPrice>>();

//在策略列表查找策略

for (Class<? extends CalPrice> clazz : calPriceList) {

Annotation validRegion = handleAnnotation(clazz);//获取该策略的注解

//变化点：根据注解类型进行不同的判断

if (validRegion instanceof TotalValidRegion) {

```
                        TotalValidRegion totalValidRegion =
(TotalValidRegion) validRegion;

                        //判断总金额是否在注解的区间

                        if (customer.getTotalAmount() >
totalValidRegion.value().min() && customer.getTotalAmount() <
totalValidRegion.value().max()) {


clazzMap.put(totalValidRegion.value().order(), clazz);//将采
用的策略放入MAP

                        }

                }

                else if (validRegion instanceof
OnceValidRegion) {

                        OnceValidRegion onceValidRegion =
(OnceValidRegion) validRegion;

                        //判断单次金额是否在注解的区间，注意
这次判断的是客户当次消费的金额

                        if (customer.getAmount() >
onceValidRegion.value().min() && customer.getAmount() <
onceValidRegion.value().max()) {

clazzMap.put(onceValidRegion.value().order(), clazz);//将采用
的策略放入MAP

                        }

                }

            }
```

```
            try {
                //我们采用动态代理处理策略重叠的问题，相信看过LZ的代理模式的同学应该都对代理模式的原理很熟悉了，那么下面出现的代理类LZ将不再解释，留给各位自己琢磨。
                return CalPriceProxy.getProxy(clazzMap);
            } catch (Exception e) {
                throw new RuntimeException("策略获得失败");
            }
        }

        //处理注解，我们传入一个策略类，返回它的注解
        private Annotation handleAnnotation(Class<? extends CalPrice> clazz){
            Annotation[] annotations = clazz.getDeclaredAnnotations();
            if (annotations == null || annotations.length == 0) {
                return null;
            }
            for (int i = 0; i < annotations.length; i++) {
                //变化点：这里稍微改动了下,如果是TotalValidRegion,OnceValidRegion这两种注解则返回
                if (annotations[i] instanceof TotalValidRegion || annotations[i] instanceof
```

```
OnceValidRegion) {
                        return annotations[i];
                }
            }
            return null;
        }

        /*   以下不需要改变   */

        //单例，并且我们需要在工厂初始化的时候
        private CalPriceFactory(){
            init();
        }

        //在工厂初始化时要初始化策略列表
        private void init(){
            calPriceList = new ArrayList<Class<? extends
CalPrice>>();
            File[] resources = getResources();//获取到包下所
有的class文件
            Class<CalPrice> calPriceClazz = null;
```

```
            try {
                  calPriceClazz = (Class<CalPrice>)
classLoader.loadClass(CalPrice.class.getName());//使用相同的
加载器加载策略接口
            } catch (ClassNotFoundException e1) {
                  throw new RuntimeException("未找到策略接
口");
            }
            for (int i = 0; i < resources.length; i++) {
                  try {
  //载入包下的类
                        Class<?> clazz =
classLoader.loadClass(CAL_PRICE_PACKAGE +
"."+resources[i].getName().replace(".class", ""));
                        //判断是否是CalPrice的实现类并且不是
CalPrice它本身，满足的话加入到策略列表
                        if
(CalPrice.class.isAssignableFrom(clazz) && clazz !=
calPriceClazz) {
calPriceList.add((Class<? extends CalPrice>) clazz);
                        }
                  } catch (ClassNotFoundException e) {
                        e.printStackTrace();
                  }
```

```
        }

    }

    //获取扫描的包下面所有的class文件

    private File[] getResources(){

        try {

            File file = new
File(classLoader.getResource(CAL_PRICE_PACKAGE.replace(".",
"/")).toURI());

            return file.listFiles(new FileFilter() {

                public boolean accept(File pathname)
{

                    if
(pathname.getName().endsWith(".class")) {//我们只扫描class文
件

                        return true;

                    }

                    return false;

                }

            });

        } catch (URISyntaxException e) {

            throw new RuntimeException("未找到策略资
源");

        }
```

```
	}

	public static CalPriceFactory getInstance(){
		return CalPriceFactoryInstance.instance;
	}

	private static class CalPriceFactoryInstance{

		private static CalPriceFactory instance = new CalPriceFactory();
	}
}
```

上面我们改动的地方并不多，主要是添加了一个单次消费的判断，另外就是没有直接返回策略实例，而是将满足条件的策略类信息传递给代理，产生一个代理，从而满足我们第二个要求，即策略可以重叠，下面LZ给出代理类，相信如果各位看过LZ的代理模式，并完全理解了，那么看懂这里面的道理是非常简单的，为此，LZ不在多做解释。代理类如下。

```
package com.calprice;

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

```
import java.util.SortedMap;


public class CalPriceProxy implements InvocationHandler{


	private SortedMap<Integer, Class<? extends CalPrice>>
clazzMap;


	private CalPriceProxy(SortedMap<Integer, Class<?
extends CalPrice>> clazzMap) {

		super();

		this.clazzMap = clazzMap;

	}


	public Object invoke(Object proxy, Method method,
Object[] args)

			throws Throwable {

		Double result = 0D;

		if (method.getName().equals("calPrice")) {

			for (Class<? extends CalPrice> clazz :
clazzMap.values()) {

				if (result == 0) {

					result = (Double)
method.invoke(clazz.newInstance(), args);
```

```
                        }else {

                                result = (Double)
method.invoke(clazz.newInstance(), result);

                        }

                }

                return result;

            }

            return null;

        }



        public static CalPrice getProxy(SortedMap<Integer,
Class<? extends CalPrice>> clazzMap){

            return (CalPrice)
Proxy.newProxyInstance(CalPriceProxy.class.getClassLoader(),
new Class<?>[]{CalPrice.class}, new CalPriceProxy(clazzMap));

        }



}
```

好了，这下我们可以支持策略重叠了，我给各位一个指定好的一系列策略，如下。

```
package com.calprice;

//我们使用嵌套注解，并且制定我们打折的各个策略顺序是99，这算
是很靠后的
```

//因为我们最后打折算出来钱是最多的，这个一算就很清楚，LZ不再解释数学问题

```
@TotalValidRegion(@ValidRegion(max=1000,order=99))
class Common implements CalPrice{

	public Double calPrice(Double originalPrice) {
		return originalPrice;
	}

}
@TotalValidRegion(@ValidRegion(min=1000,max=2000,order=99))
class Vip implements CalPrice{

	public Double calPrice(Double originalPrice) {
		return originalPrice * 0.8;
	}

}
@TotalValidRegion(@ValidRegion(min=2000,max=3000,order=99))
class SuperVip implements CalPrice{
```

```
	public Double calPrice(Double originalPrice) {
		return originalPrice * 0.7;
	}

}
@TotalValidRegion(@ValidRegion(min=3000,order=99))
class GoldVip implements CalPrice{

	public Double calPrice(Double originalPrice) {
		return originalPrice * 0.5;
	}

}
@OnceValidRegion(@ValidRegion(min=1000,max=2000,order=40))
class OneTDTwoH implements CalPrice{

	public Double calPrice(Double originalPrice) {
		return originalPrice - 200;
	}
```

```
}

@OnceValidRegion(@ValidRegion(min=2000,order=40))
class TwotDFourH implements CalPrice{

	public Double calPrice(Double originalPrice) {
		return originalPrice - 400;
	}

}
```

这里面相比上面，又添了两种策略，即满1000返200和满2000返400，并且优先级高于打折，也就是说会先计算现金返回，再打折。各位可以使用如下客户端测试一下。

```
package com.calprice;

//客户端调用
public class Client {

	public static void main(String[] args) {
		Customer customer = new Customer();
		customer.buy(500D);
```

```
            System.out.println("客户需要付钱：" +
customer.calLastAmount());

            customer.buy(1200D);

            System.out.println("客户需要付钱：" +
customer.calLastAmount());

            customer.buy(1200D);

            System.out.println("客户需要付钱：" +
customer.calLastAmount());

            customer.buy(1200D);

            System.out.println("客户需要付钱：" +
customer.calLastAmount());

            customer.buy(2600D);

            System.out.println("客户需要付钱：" +
customer.calLastAmount());

      }



}
```

当然你还可以多写几个buy方法，看下我们的策略是否在发挥作用。这下我们的策略模式就更加灵活了，不仅可以支持策略的随意增加，而且还可以重叠。相信这下老板再要搞什么促销活动，都可以轻松应付了吧。当然，没有最好的设计只有更适合的设计，只要可以满足大部分需求，容纳大部分变化就算是很好的设计了。实在容纳和满足不了，我们还可以重构，而且重构往往会比预先设计更加凑效。

本次讲解策略模式算是由浅及深的方式，刚开始给出的是LZ作为一个JAVA新人的时候对策略模式的理解，后面是一个代入了业务场景的例子，以及后面的逐渐使用简单工厂，注解，反射，代理等方式改善我

们的策略工厂的过程，算是LZ和各位读者一起进行一个设计思想的锻炼吧。

策略模式本身并不太复杂，实现也比较简单，但是我们却花费了大量的篇幅去完善它，这是因为完善策略模式往往比使用更加复杂。

最后总结一下策略模式的使用场景，就是有一系列的可相互替换的算法的时候，我们就可以使用策略模式将这些算法做成接口的实现，并让我们依赖于算法的类依赖于抽象的算法接口，这样可以彻底消除类与具体算法之间的耦合。

比如我们现在客户类，它就知道有个CalPrice接口可以计算最终价格，其它的它什么都不知道了，这不正是我们之前总纲中提到的最小知道原则吗？

当然策略模式也有缺点，就是我们不停的在各个算法间切换，造成很多逻辑判断，不过我们本章已经给各位提供了思路，我们是可以使用一些其他的模式或者JAVA的技术去消除这种逻辑判断的，对吧？

（八）适配器模式详解

各位好，我们本次接着讨论第八个设计模式，适配器模式。

适配器模式从实现方式上分为两种，类适配器和对象适配器，这两种的区别在于实现方式上的不同，一种采用继承，一种采用组合的方式。

另外从使用目的上来说，也可以分为两种，特殊适配器和缺省适配器，这两种的区别在于使用目的上的不同，一种为了复用原有的代码并适配当前的接口，一种为了提供缺省的实现，避免子类需要实现不该实现的方法。

首先应该明白一点，适配器模式是补救措施，所以在系统设计过程中请忘掉这个设计模式，这个模式只是在你无可奈何时的补救方式。

那么我们什么时候使用这个模式呢？场景通常情况下是，系统中有一套完整的类结构，而我们需要利用其中某一个类的功能（通俗点说可以说是方法），但是我们的客户端只认识另外一个和这个类结构不相关的接口，这时候就是适配器模式发挥的时候了，我们可以将这个现有的类与我们的目标接口进行适配，最终获得一个符合需要的接口并且包含待复用的类的功能的类。

接下来我们举一个例子，比如我们在观察者一章中就提到一个问题，就是说观察者模式的一个缺点，即如果一个现有的类没有实现Observer接口，那么我们就无法将这个类作为观察者加入到被观察者的观察者列表中了，这实在太遗憾了。

在这个问题中，我们需要得到一个Observer接口的类，但是又想用原有的类的功能，但是我们又改不了这个原来的类的代码，或者原来的类有一个完整的类体系，我们不希望破坏它，那么适配器模式就是你的不二之选了。

我们举个具体的例子，比如我们希望将HashMap这个类加到观察者列表里，在被观察者产生变化时，假设我们要清空整个MAP。但是现在加不进去啊，为什么呢？

因为Observable的观察者列表只认识Observer这个接口，它不认识HashMap，怎么办呢？

这种情况下，我们就可以使用类适配器的方式将我们的HashMap做点手脚，刚才已经说了，类适配器采用继承的方式，那么我们写出如下适配器。

```
public class HashMapObserverAdapter<K, V> extends HashMap<K, V> implements Observer{



	public void update(Observable o, Object arg) {

		//被观察者变化时，清空Map

		super.clear();

	}



}
```

即我们继承我们希望复用其功能的类，并且实现我们想适配的接口，在这里就是Observer，那么就会产生一个适配器，这个适配器具有原有类（即HashMap）的功能，又具有观察者接口，所以这个适配器现在可以加入到观察者列表了。

看，类适配器很简单吧？那么下面我们来看看对象适配器，刚才说了对象适配器是采用组合的方式实现。

为什么要采用组合呢？上面的方式不是很好吗？

究其根本，是因为JAVA单继承的原因，一个JAVA类只能有一个父类，所以当我们要适配的对象是两个类的时候，你怎么办呢？你难道要将两个类全部写到extends后面吗，如果你这么做了，那么编译器会表示它的不满的。

我们还是拿观察者模式那一章的例子来说（观察者模式比较惨，老要适配器模式擦屁股），比如我们现在有一个写好的类，假设就是个实体类吧。如下。

```
public class User extends BaseEntity{

      private Integer id;

      private String name;

      public Integer getId() {

            return id;

      }

      public void setId(Integer id) {

            this.id = id;

      }

      public String getName() {

            return name;

      }

      public void setName(String name) {

            this.name = name;

      }
```

}

看到了吧，我们的实体类大部分都是继承自BaseEntity的，那现在你怎么办吧，你要想具有被观察者的功能还要继承Observable类，你说你怎么继承吧。

你是不是想说，那我的User不继承BaseEntity不就完事了，我把BaseEntity里面的东西全部挪动到User类，或者我不继承Observable了，把Observable里面的东西全部挪到User类里面。

这并不是不行，但是这是个很大的隐患，比如我们项目到时候要针对BaseEntity的子类进行扫描，用来做一些事情，这时候如果User没继承BaseEntity，那么你就会遗漏掉这个类，这会破坏你的继承体系，付出太大了。

相反，如果你不继承Observable，那么你的User类看起来会非常杂乱，而且假设我现在不仅User类可以被观察了，我的Person类，Employee都能被观察了，你难道要把Observable的代码COPY三次到这三个类里面吗？

不要忘了刚才说的，适配器模式就是为了帮助我们复用代码的，这里使用适配器模式就可以帮我们复用Observable的代码或者说功能。

基于上面LZ的讨论，我们做出如下适配器，这里采用的对象适配器。

```
//我们继承User,组合Observable.

public class ObservableUser extends User{



        private Observable observable = new Observable();



        public synchronized void addObserver(Observer o) {

                observable.addObserver(o);
```

```
}

public synchronized void deleteObserver(Observer o) {
      observable.deleteObserver(o);
}

public void notifyObservers() {
      observable.notifyObservers();
}

public void notifyObservers(Object arg) {
      observable.notifyObservers(arg);
}

public synchronized void deleteObservers() {
      observable.deleteObservers();
}

protected synchronized void setChanged() {
      observable.setChanged();
```

```
    }

    protected synchronized void clearChanged() {
        observable.clearChanged();
    }

    public synchronized boolean hasChanged() {
        return observable.hasChanged();
    }

    public synchronized int countObservers() {
        return observable.countObservers();
    }

}
```

我们继承User，而不是继承Observable，这个原因刚才已经说过了，我们不能破坏项目中的继承体系，所以现在可观察的User（ObservableUser）依然处于我们实体的继承体系中，另外如果想让ObservableUser具有User的属性，则需要将User的属性改为protected。

这下好了，我们有了可观察的User了。不过LZ早就说过，设计模式要活用，这里明显不是最好的解决方案。因为我们要是还有Person,Employee类都要具有可观察的功能的话，那其实也相当惨，因为下面那些Observable的方法我们还要再复制一遍。

提示到这里，不知各位想到更好的解决方案了吗？尤其是新手可以好好思考下。

LZ这里给出最终相对来说比较好的解决方案，那就是我们定义如下可观察的基类。

```
//我们扩展BaseEntity,适配出来一个可观察的实体基类

public class BaseObservableEntity extends BaseEntity{



	private Observable observable = new Observable();



	public synchronized void addObserver(Observer o) {

		observable.addObserver(o);

	}



	public synchronized void deleteObserver(Observer o) {

		observable.deleteObserver(o);

	}



	public void notifyObservers() {
```

```
        observable.notifyObservers();
    }

    public void notifyObservers(Object arg) {
        observable.notifyObservers(arg);
    }

    public synchronized void deleteObservers() {
        observable.deleteObservers();
    }

    protected synchronized void setChanged() {
        observable.setChanged();
    }

    protected synchronized void clearChanged() {
        observable.clearChanged();
    }

    public synchronized boolean hasChanged() {
```

```
        return observable.hasChanged();
    }

    public synchronized int countObservers() {
        return observable.countObservers();
    }

}
```

这下好了，现在我们的User，Person，Employee要是想具有可被观察的功能，那就改去继承我们适配好的BaseObservableEntity就好了，而且由于BaseObservableEntity继承了BaseEntity，所以他们三个依然处于我们实体的继承体系中，而且由于我们的BaseObservableEntity是新增的扩展基类，所以不会对原来的继承体系造成破坏。

适配器模式的用法还是比较清晰的，我们以上两种方式都是为了复用现有的代码而采用的适配器模式，LZ刚才说了，根据目的的不同，适配器模式也可以分为两种，那么上述便是第一种，可称为定制适配器，还有另外一种称为缺省适配器。

首先我们得先说下缺省适配器为什么要出现，因为适配器模式大部分情况下是为了补救，所以既然补救，那么肯定是历史原因造成的我们需要使用这个模式。

我们来看看缺省适配器的历史来由，不知各位还是否记得在第一章总纲中，LZ曾经提到过一个原则，最小接口原则。

这个原则所表达的思想是说接口的行为应该尽量的少，那么还记得LZ当时说如果你没做到的话会产生什么情况吗？

结果就是实现这个接口的子类，很可能出现很多方法是空着的情况，因为你的接口设计的过大，导致接口中原本不该出现的方法出现了，结果现在子类根本用不上这个方法，但由于JAVA语言规则的原因，实现一个接口必须实现它的全部方法，所以我们的子类不得不被迫写一堆空方法在那，只为了编译通过。

所以为了解决这一问题，缺省适配器就出现了。比如我们有如下接口。

```
public interface Person {



      void speak();



      void listen();



      void work();



}
```

这是一个人的接口，这个接口表示了人可以说话，听和工作，假设是两年前的LZ，还在家待业呢，LZ没工作啊，但是LZ也是个人啊，所以LZ要实现这个接口，所以LZ只能把work方法抄下来空着放在那了，假设LZ是个聋哑人，好吧，三个方法都要空着了，但是LZ表示，LZ是人，LZ一定要实现Person接口。

当然，上述只是举个例子，但是真实项目当中也会出现类似的情况，那么怎么办呢？

这下来了，我们的缺省适配器来了，如下。

```
public class DefaultPerson implements Person{
```

```
    public void speak() {}

    public void listen() {}

    public void work() {}

}
```

我们创造一个Person接口的默认实现，它里面都是一些默认的方法，当然这里因为没什么可写的就空着了，实际当中可能会加入一些默认情况下的操作，比如如果方法返回结果整数，那么我们在缺省适配器中可以默认返回个0。

这下好了，LZ只要继承这个默认的适配器（DefaultPerson），然后覆盖掉LZ感兴趣的方法就行了，比如speak和listen，至于work，由于适配器帮我们提供了默认的实现，所以就不需要再写了。

这种情况其实蛮多的，因为接口设计的最小化只是理想状态，难免会有一些实现类，对其中某些方法不感兴趣，这时候，如果方法过多，子类也很多，并且子类的大部分方法都是空着的，那么就可以采取这种方式了。

当然，这样做违背了里氏替换原则，但是上面的做法原本就违背了接口的最小化原则，所以我们在真正使用时要权衡二者的利弊，到底我们需要的是什么。所以从此也可以看出来，原则只是指导，并不一定也不可能全部满足，所以我们一定要学会取舍。

总结下两种实现方式的适配器所使用的场景，两者都是为了将已有类的代码复用并且适配到客户端需要的接口上去。

1，第一种类适配器，一般是针对适配目标是接口的情况下使用。

2，第二种对象适配器，一般是针对适配目标是类或者是需要复用的对象多于一个的时候使用，这里再专门提示一下，对象适配器有时候是为了将多个类一起适配，所以才不得不使用组合的方式，而且我们采用对象适配器的时候，继承也不是必须的，而是根据实际的类之间的关系来进行处理，上述例子当中一定要直接或间接的继承自

BaseEntity是为了不破坏我们原来的继承体系，但有些情况下这并不是必须的。

对于第三个缺省适配器，一般是为了弥补接口过大所犯下的过错，但是也请注意衡量利弊，权衡好以后再考虑是否要使用缺省适配器。

好了，本次适配器模式的分享就到此结束了，希望各位可以从中得到点收获。

（九）模板方法模式详解（包含与类加载器不得不说的故事）

模板方法模式，这是一个在许多优秀的开源项目中LZ见的最多的一个设计模式，也是LZ觉得最为优秀的一个设计模式，所以这一章LZ会尽自己所能的去尽量将这个设计模式解释清楚。

模板方法模式，一般是为了统一子类的算法实现步骤，所使用的一种手段或者说是方式。它在父类中定义一系列算法的步骤，而将具体的实现都推迟到子类。

最典型的形式就是一个接口，一个抽象父类，父类中会有一系列的抽象方法，而在子类中去一一实现这些方法。

下面LZ给举一个例子，比如我们有一个接口，里面就一个方法，是用来制造一个HTML页面，如下。

```
public interface PageBuilder {

      String bulidHtml();

}
```

这个接口很简单，就是直接制造一个Html页面的内容，假设我们不使用模板方法模式，直接让各个子类去直接实现这个接口，那么肯定实现的方式千奇百怪，而且步骤也乱七八糟的，这样实在不利于维护和扩展。所以我们可以使用模板方法模式，将这个过程给制定好，然后把具体的内容填充交给子类就好，这样这些子类生成的HTML页面就会非常一致。

基于这个目的，我们定义如下抽象类，去实现这个接口，并且我们定义好步骤。

```
public abstract class AbstractPageBuilder implements
PageBuilder{



      private StringBuffer stringBuffer = new StringBuffer();



      public String bulidHtml() {

  //首先加入doctype,因为都是html页面,所以我们父类不需要推迟给
子类实现,直接在父类实现

  stringBuffer.append("<!DOCTYPE html PUBLIC \"-//W3C//DTD
XHTML 1.0 Transitional//EN\"
\"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd\">");

            //页面下面就是成对的一个HTML标签，我们也在父类加
入,不需要给子类实现

            stringBuffer.append("<html
xmlns=\"http://www.w3.org/1999/xhtml\">");

            //下面就应该是head标签里的内容了,这个我们父类做不
了主了,推迟到子类实现,所以我们定义一个抽象方法,让子类必须实现

            appendHead(stringBuffer);

            //下面是body的内容了，我们父类依然无法做主，仍然
推迟到子类实现

            appendBody(stringBuffer);

            //html标签的关闭

stringBuffer.append("</html>");

            return stringBuffer.toString();
```

```
    }

    //第一个模板方法
    protected abstract void appendHead(StringBuffer stringBuffer);

    //第二个模板方法
    protected abstract void appendBody(StringBuffer stringBuffer);

}
```

上面LZ已经加了注释，这下我们如果要制作一个html页面，就直接继承我们的抽象父类就可以了，而我们的子类只需要实现两个模板方法，就可以成功完成html页面的创建，下面LZ给出一个子类，我们随意制造一个html页面。

```
public class MyPageBuilder extends AbstractPageBuilder{

    @Override
    protected void appendHead(StringBuffer stringBuffer) {
stringBuffer.append("<head><title>你好</title></head>");
    }
```

```
    @Override
    protected void appendBody(StringBuffer stringBuffer) {
stringBuffer.append("<body><h1>你好,世界！</h1></body>");
    }

    public static void main(String[] args) {
        PageBuilder pageBuilder = new MyPageBuilder();
        System.out.println(pageBuilder.bulidHtml());
    }

}
```

我们简单的加入一个head和body标签，然后创建测试类运行一下，就会发现，我们按照父类给的标准模板，生成了一个html页面。

这样做的方式的好处是，父类可以规范子类的创建过程，便于我们维护，而且子类也更省事，因为像doctype包括html标签都是一样的，所以子类不再需要关心这些。当然上述LZ写的有点粗糙，其实我们可以定义的更仔细一点，比如head标签里，第一个是title，然后是meta等等。但作为例子，我们还是遵循简单的原则，主要还是想给各位传达模板方法模式的思想。

模板方法模式是所有设计模式当中，LZ觉得最无侵入性的模式，因为它的好处实在是太明显了。模板方法模式并不强制接口的实现类必须继承，所以不会对子类造成任何影响，而如果子类的实现可以配得上模板类的模板，那么就可以享受模板方法模式带来的好处。

通常情况下，模板方法模式用于定义构建某个对象的步骤与顺序，或者定义一个算法的骨架。

我们刚才的示例明显就是构建一个String对象的过程，在这里要声明一点，对于模板方法模式，父类提供的构建步骤和顺序或者算法骨架，通常是不希望甚至是不允许子类去覆盖的，所以在某些场景中，可以直接将父类中提供骨架的方法声明为final类型。

模板方法模式还有一种使用的方式，为了给子类足够的自由度，可以提供一些方法供子类覆盖，去实现一些骨架中不是必须但却可以有自定义实现的步骤。

比如上述的例子当中，我们应该都知道，HTML页面中有一些标签是可有可无的。比如meta标签，link标签，script标签等。那么我们可以将刚才的例子细化一下，去看一下上面说的供子类覆盖的方法是什么。我们将刚才的抽象父类细化成如下形式。

```
public abstract class AbstractPageBuilder implements
PageBuilder{



      private static final String DEFAULT_DOCTYPE = "
<!DOCTYPE html PUBLIC \"-//W3C//DTD XHTML 1.0
Transitional//EN\" \"http://www.w3.org/TR/xhtml1/DTD/xhtml1-
strict.dtd\">";



      private static final String DEFAULT_XMLNS =
"http://www.w3.org/1999/xhtml";



      private StringBuffer stringBuffer = new StringBuffer();



      public String bulidHtml() {

            stringBuffer.append(DEFAULT_DOCTYPE);
```

```
            stringBuffer.append("<html xmlns=\"" +
DEFAULT_XMLNS + "\">");

stringBuffer.append("<head>");

            appendTitle(stringBuffer);

            appendMeta(stringBuffer);

            appendLink(stringBuffer);

            appendScript(stringBuffer);

stringBuffer.append("</head>");

            appendBody(stringBuffer);

stringBuffer.append("</html>");

            return stringBuffer.toString();

      }


      protected void appendMeta(StringBuffer stringBuffer){

      }


      protected void appendLink(StringBuffer stringBuffer){

      }


      protected void appendScript(StringBuffer stringBuffer){

      }
```

```
	protected abstract void appendTitle(StringBuffer stringBuffer);



	protected abstract void appendBody(StringBuffer stringBuffer);



}
```

可以看到，我们将head标签的生成过程更加细化了，分成四个方法，title，meta，link和script。但是这四个里面appendTitle是模板方法，子类必须实现，而其它三个则是普通的空方法。

那么上述三个方法，就是留给子类覆盖的，当然子类可以选择不覆盖，那么生成的HTML就没有meta，link和script这三种标签，如果想有的话，就可以覆盖其中任意一个，比如下面这样。

```
public class MyPageBuilder extends AbstractPageBuilder{



	protected void appendMeta(StringBuffer stringBuffer) {

		stringBuffer.append("<meta http-equiv=\"Content-Type\" content=\"text/html; charset=utf-8\" />");

	}



	protected void appendTitle(StringBuffer stringBuffer) {

		stringBuffer.append("<title>你好</title>");
```

```
	}

	protected void appendBody(StringBuffer stringBuffer) {
		stringBuffer.append("<body>你好，世界！</body>");
	}

	public static void main(String[] args) {
		PageBuilder pageBuilder = new MyPageBuilder();
System.out.println(pageBuilder.bulidHtml());
	}

}
```

我们覆盖了appendMeta方法，所以我们就可以在head标签中生成一个meta标签。如果各位看过上章的适配器模式，其实这里和缺省适配很像，目的都是一样的，因为如果把appendMeta也写成抽象方法，那么子类就必须实现，但是meta标签又不是必须的，所以子类就有可能把appendMeta，appendLink，appendScript方法全空着了。

所以为了不强制子类实现不必要的抽象方法，但又不剥夺子类自由选择的权利，我们在父类提供一个默认的空实现，来让子类自由选择是否要覆盖掉这些方法。

说到模板方法模式，我们JDK当中有一个类与它还有一个不得不说的故事，那就是类加载器。

JDK类加载器可以大致分为三类，分别是启动类加载器，扩展类加载器，以及应用程序加载器。

这三者加载类的路径分别为如下：

启动类加载器：JAVA_HOME/lib目录下，以及被-Xbootcalsspath参数设定的路径，不过启动类加载器加载的类是有限制的，如果JVM不认识的话，你放在这些目录下也没用。

扩展类加载器：JAVA_HOME/lib/ext目录下，以及被java.ext.dirs系统变量指定的路径。

应用程序类加载器：用户自己的类路径（classpath），这个类加载器就是我们经常使用的系统类加载器，并且JDK中的抽象类ClassLoader的默认父类加载器就是它。

在这里为什么说类加载器和模板方法模式有关呢，是因为ClassLoader类就使用了模板模式，去保证类加载过程中的唯一性。LZ先给各位看下这个类当中的模板模式的应用。

```
public abstract class ClassLoader {

	//这是一个重载方法

	public Class<?> loadClass(String name) throws
ClassNotFoundException {

	return loadClass(name, false);

	}



	//这里就是父类算法的定义

	protected synchronized Class<?> loadClass(String name,
boolean resolve)

	throws ClassNotFoundException

	{
```

```
Class c = findLoadedClass(name);
if (c == null) {
      try {
      if (parent != null) {
            c = parent.loadClass(name, false);
      } else {
            c = findBootstrapClass0(name);
      }
      } catch (ClassNotFoundException e) {
            c = findClass(name);
      }
}
if (resolve) {
      resolveClass(c);
}
return c;
}
//这里留了一个方法给子类选择性覆盖
protected Class<?> findClass(String name) throws
ClassNotFoundException {
throw new ClassNotFoundException(name);
```

```
    }


}
```

LZ截取了主要的部分，为了突出这三个方法。在上面LZ加了简单的注释，相信经过刚才的介绍，各位应该能看出来这是一个模板方法模式，只是它没有定义抽象方法，因为findClass这个方法，并不是必须实现的，所以JDK选择留给程序员们自己选择是否要覆盖。

从代码上我们可以看出，在ClassLoader中定义的算法顺序是。

1，首先看是否有已经加载好的类。

2，如果父类加载器不为空，则首先从父类类加载器加载。

3，如果父类加载器为空，则尝试从启动加载器加载。

4，如果两者都失败，才尝试从findClass方法加载。

这是JDK类加载器的双亲委派模型，即先从父类加载器加载，直到继承体系的顶层，否则才会采用当前的类加载器加载。这样做的目的刚才已经说了，是为了JVM中类的一致性。

如果有读者第一次接触这方面的知识，估计会比较迷茫，下面LZ给出一个例子。各位猜测下下面程序的运行结果会是什么？

```
package com.classloader;


public class ClassLoaderTest {


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

```
            Class<?> clazz =
ClassLoader.getSystemClassLoader().loadClass("com.classloader
.ClassLoaderTest");

            Object entity = clazz.newInstance();

            System.out.println(entity instanceof
ClassLoaderTest);

      }

}
```

相信各位都可以毫无疑问的猜测出来，结果应该是true，这是因为entity是ClassLoaderTest类的一个实例，instanceof关键字用来判断一个实例是否属于一个特定的类型，所以结果就是true。

那么各位再来猜猜下面这段代码的运行结果会是什么？

```
package com.classloader;



import java.io.IOException;

import java.io.InputStream;



class MyClassLoader extends ClassLoader{



      public Class<?> loadClass(String name) throws
ClassNotFoundException {

            String fileName =
name.substring(name.lastIndexOf(".")+1) + ".class";
```

```
            InputStream is =
getClass().getResourceAsStream(fileName);

            if (is == null) {

                  return super.loadClass(name);

            }

            try {

                  byte[] b = new byte[is.available()];

                  is.read(b);

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

            } catch (IOException e) {

                  throw new ClassNotFoundException();

            }

      }



}



public class ClassLoaderTest {



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

```
            ClassLoader classLoader = new MyClassLoader();

            Class<?> clazz =
classLoader.loadClass("com.classloader.ClassLoaderTest");

            Object entity = clazz.newInstance();

            System.out.println(entity instanceof
ClassLoaderTest);

      }



}
```

对于类加载器比较熟悉的读者们可能觉得这个结果并不出乎意料，可是或许还是有人会比较意外，为什么结果会是false呢？

这是因为如果没有按照ClassLoader中提供的骨架算法去加载类的话，可能会造成JVM中有两个一模一样的类信息，他们是来自一个类文件，但却不是一个加载器加载的，所以这两个类不相等。

这也是类加载器为何要使用模板模式给我们定义好查找的算法，是为了保证我们加载的每一个类在虚拟机当中都有且仅有一个。

不过你可能会想，既然如此，为何不把loadClass方法写成final类型的，这样不是更安全吗？

这是因为有的时候我们希望JVM当中每一个类有且仅有一个，但有的时候我们希望有两个，甚至N个，就比如我们的tomcat，你可以想象下，你每一个项目假设都有com.xxx.xxxx.BaseDao等等，如果这些类都是一个的话，你的tomcat还能同时启动多个WEB服务吗？虽说tomcat也是遵循的双亲委派模型，但是从此也可以看出来，我们并不是在所有时候都希望同一个全限定名的类在整个JVM里面只有一个。

这里提到类加载器，是为了给模板方法一个现有的现实中的例子，以便于有些看多了自己制造的例子的读者可以换个口味，如果有机会，

LZ会在这个系列完结以后，专门开一个系列来和各位分享学习虚拟机过程中的感悟，本次不再过多介绍类加载器的相关内容。

另外，如果多掌握一些类加载器的知识，还是对平时的工作和学习有很大帮助的，各位也可以私下去搜索下相关资料。

好了，模板方法模式就介绍到这里吧，希望各位都有自己的收获。

（十）装饰器模式详解（与IO不解的情缘）

LZ到目前已经写了九个设计模式，回过去看看，貌似写的有点凌乱，LZ后面会尽量改进。

那么本章LZ和各位读友讨论一个与JAVA中IO有着不解情缘的设计模式，装饰器模式。

定义：装饰模式是在不必改变原类文件和使用继承的情况下，动态的扩展一个对象的功能。它是通过创建一个包装对象，也就是装饰来包裹真实的对象。

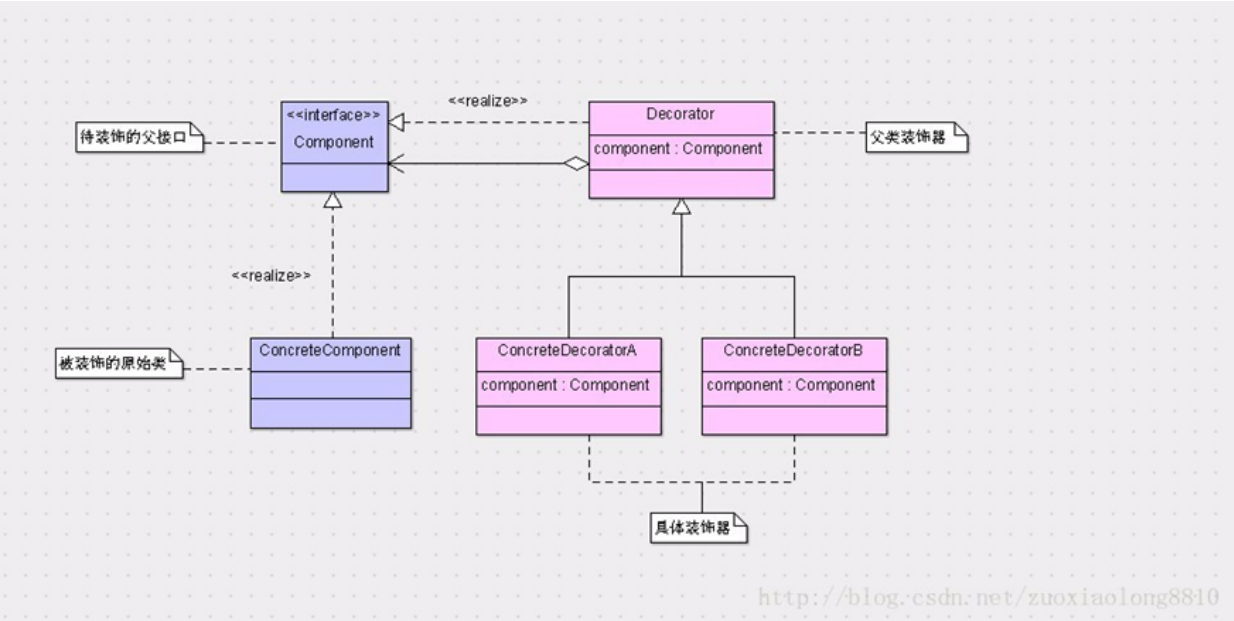
这一个解释，引自百度百科，我们注意其中的几点。

1，不改变原类文件。

2，不使用继承。

3，动态扩展。

上述三句话一语道出了装饰器模式的特点，下面LZ给出装饰器模式的类图，先上图再解释。



从图中可以看到，我们装饰的是一个接口的任何实现类，而这些实现类也包括了装饰器本身，装饰器本身也可以再被装饰。

另外，这个类图只是装饰器模式的完整结构，但其实里面有很多可以变化的地方，LZ给出如下两条。

1，Component接口可以是接口也可以是抽象类，甚至是一个普通的父类（这个强烈不推荐，普通的类作为继承体系的超级父类不易于维护）。

2，装饰器的抽象父类Decorator并不是必须的。

那么我们将上述标准的装饰器模式，用我们熟悉的JAVA代码给诠释一下。首先是待装饰的接口Component。

```
package com.decorator;



public interface Component {

      void method();
```

```
}
```

接下来便是我们的一个具体的接口实现类，也就是俗称的原始对象，或者说待装饰对象。

```
package com.decorator;


public class ConcreteComponent implements Component{


	public void method() {

		System.out.println("原来的方法");

	}


}
```

下面便是我们的抽象装饰器父类，它主要是为装饰器定义了我们需要装饰的目标是什么，并对Component进行了基础的装饰。

```
Package com.decorator;


public abstract class Decorator implements Component{


	protected Component component;


	public Decorator(Component component) {
```

```
        super();
        this.component = component;
    }

    public void method() {
        component.method();
    }

}
```

再来便是我们具体的装饰器A和装饰器B。

```
package com.decorator;

public class ConcreteDecoratorA extends Decorator{

    public ConcreteDecoratorA(Component component) {
        super(component);
    }

    public void methodA(){
        System.out.println("被装饰器A扩展的功能");
```

```
    }

    public void method(){
        System.out.println("针对该方法加一层A包装");
        super.method();
        System.out.println("A包装结束");
    }
}

package com.decorator;

public class ConcreteDecoratorB extends Decorator{

    public ConcreteDecoratorB(Component component) {
        super(component);
    }

    public void methodB(){
        System.out.println("被装饰器B扩展的功能");
    }
```

```
	public void method(){
		System.out.println("针对该方法加一层B包装");
		super.method();
		System.out.println("B包装结束");
	}
}
```

下面给出我们的测试类。我们针对多种情况进行包装。

```
package com.decorator;


public class Main {


	public static void main(String[] args) {
		Component component =new ConcreteComponent();//原来的对象
System.out.println("------------------------------");
		component.method();//原来的方法
		ConcreteDecoratorA concreteDecoratorA = new ConcreteDecoratorA(component);//装饰成A
System.out.println("------------------------------");
		concreteDecoratorA.method();//原来的方法
```

```
            concreteDecoratorA.methodA();//装饰成A以后新增的
方法

            ConcreteDecoratorB concreteDecoratorB = new
ConcreteDecoratorB(component);//装饰成B

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

            concreteDecoratorB.method();//原来的方法

            concreteDecoratorB.methodB();//装饰成B以后新增的
方法

            concreteDecoratorB = new
ConcreteDecoratorB(concreteDecoratorA);//装饰成A以后再装饰成B

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

            concreteDecoratorB.method();//原来的方法

            concreteDecoratorB.methodB();//装饰成B以后新增的
方法

      }

}
```

下面看下我们运行的结果，到底是产生了什么效果。

从此可以看到，我们首先是使用的原始的类的方法，然后分别让A和B装饰完以后再调用，最后我们将两个装饰器一起使用，再调用该接口定义的方法。

上述当中，我们分别对待装饰类进行了原方法的装饰和新功能的增加，methodA和methodB就是新增加的功能，这些都是装饰器可以做

的，当然两者并不一定兼有，但一般至少会有一种，否则也就失去了装饰的意义。

另外，文章开篇就说道了IO与装饰器的情缘，相信各位就算不太清楚，也都大致听说过JAVA的IO是装饰器模式实现的，所以LZ也不再废话，在给出一个标准的模板示例以后，直接拿出IO的示例，我们真枪实弹的来。

下面LZ直接给出IO包中的部分装饰过程，上面LZ加了详细的注释以及各个装饰器的功能演示，各位可以和上面标准的装饰器模式对比一下，LZ不得不感叹，IO与装饰器的孽缘。

```
package com.decorator;



import java.io.BufferedInputStream;

import java.io.BufferedReader;

import java.io.DataInputStream;

import java.io.FileInputStream;

import java.io.IOException;

import java.io.InputStream;

import java.io.InputStreamReader;

import java.io.LineNumberReader;

import java.io.PushbackInputStream;

import java.io.PushbackReader;



public class IOTest {
```

```
    /* test.txt内容：

     * hello world!

     */

    public static void main(String[] args) throws
IOException, ClassNotFoundException {

        //文件路径可自行更换

        final String filePath = "E:/myeclipse
project/POITest/src/com/decorator/test.txt";



        //InputStream相当于被装饰的接口或者抽象类，
FileInputStream相当于原始的待装饰的对象，FileInputStream无法
装饰InputStream

        //另外FileInputStream是以只读方式打开了一个文件，
并打开了一个文件的句柄存放在FileDescriptor对象的handle属性

        //所以下面有关回退和重新标记等操作，都是在堆中建
立缓冲区所造成的假象,并不是真正的文件流在回退或者重新标记

        InputStream inputStream = new
FileInputStream(filePath);

        final int len = inputStream.available();//记录一
下流的长度

        System.out.println("FileInputStream不支持mark和
reset：" + inputStream.markSupported());
```

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


            /* 下面分别展示三种装饰器的作用 BufferedInputStream,DataInputStream,PushbackInputStream,LZ下面做了三个装饰器的功能演示    */


            //首先装饰成BufferedInputStream，它提供我们mark，reset的功能

            BufferedInputStream bufferedInputStream = new BufferedInputStream(inputStream);//装饰成 BufferedInputStream

            System.out.println("BufferedInputStream支持mark和reset：" + bufferedInputStream.markSupported());

            bufferedInputStream.mark(0);//标记一下

            char c = (char) bufferedInputStream.read();

            System.out.println("LZ文件的第一个字符：" + c);

            bufferedInputStream.reset();//重置

            c = (char) bufferedInputStream.read();//再读

            System.out.println("重置以后再读一个字符，依然会是第一个字符:" + c);

            bufferedInputStream.reset();


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

//装饰成 DataInputStream,我们为了又使用DataInputStream,又使用BufferedInputStream的mark reset功能，所以我们再进行一层包装

//注意，这里如果不使用BufferedInputStream，而使用原始的InputStream，read方法返回的结果会是-1，即已经读取结束

//因为BufferedInputStream已经将文本的内容读取完毕，并缓冲到堆上，默认的初始缓冲区大小是8192B

DataInputStream dataInputStream = new DataInputStream(bufferedInputStream);

dataInputStream.reset();//这是BufferedInputStream提供的功能，如果不在这个基础上包装会出错

System.out.println("DataInputStream现在具有readInt，readChar,readUTF等功能");

int value = dataInputStream.readInt();//读出来一个int,包含四个字节

//我们转换成字符依次显示出来，可以看到LZ文件的前四个字符

String binary = Integer.toBinaryString(value);

int first = binary.length() % 8;

System.out.print("使用readInt读取的前四个字符：");

for (int i = 0; i < 4; i++) {

if (i == 0) {

```
System.out.print(((char)Integer.valueOf(binary.substring(0, first), 2).intValue()));

                }else {


System.out.print(((char)Integer.valueOf(binary.substring(( i - 1 ) * 8 + first, i * 8 + first), 2).intValue()));

                }

            }

            System.out.println();



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



            //PushbackInputStream无法包装BufferedInputStream支持mark reset，因为它覆盖了reset和mark方法

            //因为流已经被读取到末尾，所以我们必须重新打开一个文件的句柄，即FileInputStream

            inputStream = new FileInputStream(filePath);

            PushbackInputStream pushbackInputStream = new PushbackInputStream(inputStream,len);//装饰成PushbackInputStream

System.out.println("PushbackInputStream装饰以后支持退回操作unread");

            byte[] bytes = new byte[len];
```

```
            pushbackInputStream.read(bytes);//读完了整个流

            System.out.println("unread回退前的内容：" + new
String(bytes));

            pushbackInputStream.unread(bytes);//再退回去

            bytes = new byte[len];//清空byte数组

            pushbackInputStream.read(bytes);//再读

            System.out.println("unread回退后的内容：" + new
String(bytes));



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



            /*   以上有两个一层装饰和一个两层装饰,下面我们先
装饰成Reader，再进行其它装饰     */



            //由于之前被PushbackInputStream将流读取到末尾，我
们需要再次重新打开文件句柄

            inputStream = new FileInputStream(filePath);

            InputStreamReader inputStreamReader = new
InputStreamReader(inputStream,"utf-8");//先装饰成
InputStreamReader

System.out.println("InputStreamReader有reader的功能，比如转
码：" + inputStreamReader.getEncoding());
```

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


            BufferedReader bufferedReader = new
BufferedReader(inputStreamReader);//我们进一步在reader的基础
上装饰成BufferedReader

            System.out.println("BufferedReader有readLine等功
能：" + bufferedReader.readLine());


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


            LineNumberReader lineNumberReader = new
LineNumberReader(inputStreamReader);//我们进一步在reader的基
础上装饰成LineNumberReader

System.out.println("LineNumberReader有设置行号，获取行号等功
能（行号从0开始）,当前行号：" +
lineNumberReader.getLineNumber());


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


            //此处由于刚才被readLine方法将流读取到末尾,所以我
们再次重新打开文件句柄,并需要将inputstream再次包装成reader

            inputStreamReader = new InputStreamReader(new
FileInputStream(filePath));
```

```
            PushbackReader pushbackReader = new
PushbackReader(inputStreamReader,len);//我们进一步在reader的
基础上装饰成PushbackReader

            System.out.println("PushbackReader是拥有退回操作
的reader对象");

            char[] chars = new char[len];

            pushbackReader.read(chars);

            System.out.println("unread回退前的内容：" + new
String(chars));

            pushbackReader.unread(chars);//再退回去

            chars = new char[len];//清空char数组

            pushbackReader.read(chars);//再读

            System.out.println("unread回退后的内容：" + new
String(chars));

      }

}
```

上述便是IO的装饰器使用，其中InputStream就相当于上述的Component接口，只不过这里是一个抽象类，这是我们装饰的目标抽象类。FileInputstream就是一个ConcreteComponent，即待装饰的具体对象，它并不是JAVA的IO结构中的一个装饰器，因为它无法装饰InputStream。剩下BufferedInputStream，DataInputstream等等就是各种装饰器了，对比上述的标准装饰器样板，JAVA的IO中也有抽象的装饰器基类的存在，只是上述没有体现出来，就是FilterInputStream，它是很多装饰器最基础的装饰基类。

在上述过程中，其中dataInputStream是经过两次装饰后得到的，它具有了dataInputStream和bufferedInputStream的双重功能，另外，InputStreamReader是一个特殊的装饰器，它提供了字节流到字符流的

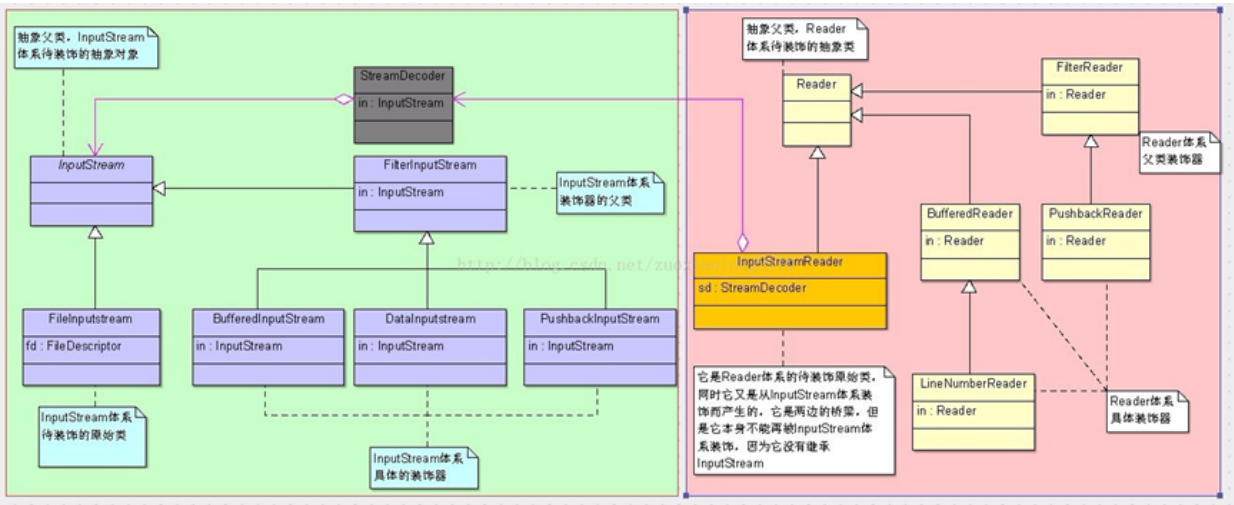
桥梁，其实它除了具有装饰器的特点以外，也有点像一个适配器，但LZ还是觉得它应当算是一个装饰器。

其它的IO装饰器各位可以自行尝试或者和上述的标准的装饰器模式代码比对一下，下面另附LZ的IO装饰器程序运行后结果。

从上面的展示中，已经可以充分体会到装饰器模式的灵活了，我们创建的一个FileInputstream对象，我们可以使用各种装饰器让它具有不同的特别的功能，这正是动态扩展一个类的功能的最佳体现，而装饰器模式的灵活性正是JAVA中IO所需要的，不得不赞一下JAVA类库的建造者实在是强悍。

上述的XXXXInputStream的各个类都继承了InputStream，这样做不仅是为了复用InputStream的父类功能（InputStream也是一种模板方法模式，它定义了read(byte[])方法的简单算法，并将read()方法交给具体的InputStream去实现），也是为了可以重叠装饰，即装饰器也可以再次被装饰，而过渡到Reader以后，Reader的装饰器体系则是类似的。

下面LZ给出上面IO包中所涉及的类的类图，各位可以自行和上面的标准装饰器模式对比一下。



LZ在类图上标注了各个类负责的角色，并且使用背景颜色将InputStream和Reader体系分开，其中左半部分就是InputStream的装饰体系，右半部分就是Reader的装饰体系，并且他们之间的桥梁是InputStreamReader，他们每一个装饰体系都与上面标准的装饰器模式类图极其相似，各位可以自己看一下，感受一下，尤其是InputStreamReader，它的位置比较特殊。

总之呢，装饰器模式就是一个可以非常灵活的动态扩展类功能的设计模式，它采用组合的方式取代继承，使得各个功能的扩展更加独立和灵活。

本次装饰器模式就到此结束了，感谢各位的收看，下期再见。

（十一）外观模式详解（Service第三者插足，让action与dao分手）

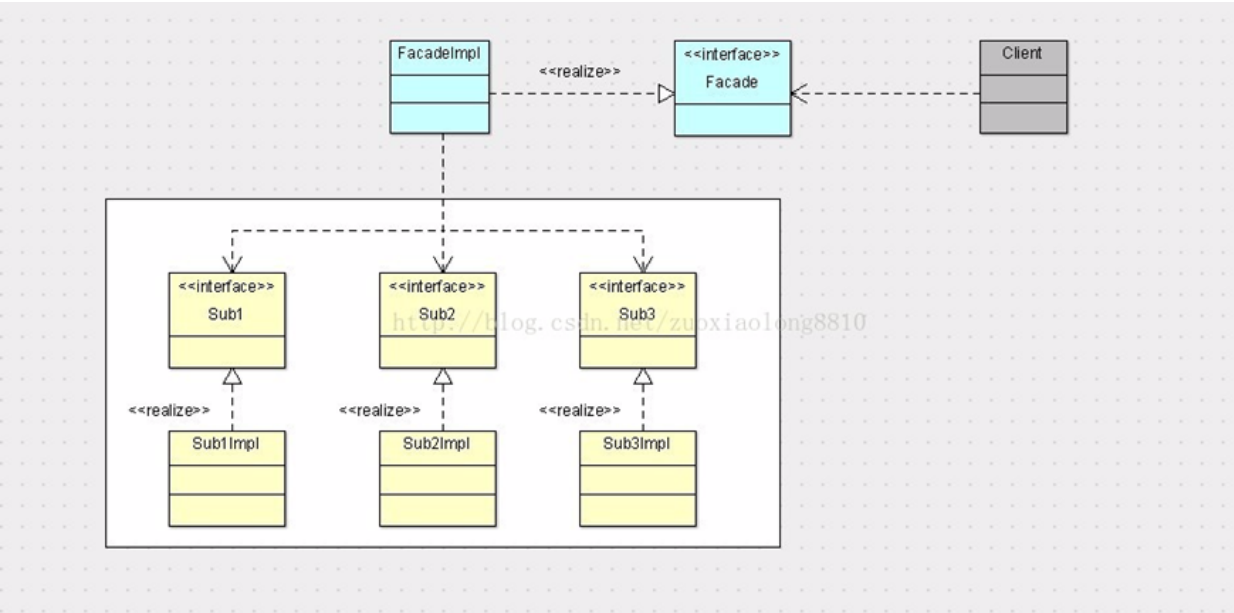
各位好，LZ今天给各位分享一个不太熟悉的面孔，但却是我们最经常使用的设计模式，外观模式。

定义：外观模式是软件工程中常用的一种软件设计模式。它为子系统中的一组接口提供一个统一的高层接口。这一接口使得子系统更加容易使用。

该定义引自百度百科，它的表现很简单，将一系列子接口的功能进行整理，从而产生一个更高层的接口。

相信做JAVA的各位大部分是WEB开发，那么肯定都对XXXDao，XXXService非常熟悉了，这显然和外观模式有一腿。当然，还有一大部分是android开发，LZ没接触过android开发，但是LZ大胆的想象，在移动领域的JAVA开发，应该也有类似的情况发生。

接下来，我们来看看外观模式的标准类图。



上述便是外观模式的类图，它主要由两部分组成，一部分是子系统（包括接口，实现类，等等），一部分是外观接口和实现类，外观接口负责提供客户端定制的服务，外观实现则负责组合子系统中的各个类和接口完成这些服务，外观接口则是提供给客户端使用的，这样就解除了客户端与子系统的依赖，而让客户端只依赖于外观接口，这是一个优秀的解耦实践。

下面LZ依然使用JAVA代码将上述的类图诠释出来，我们来直观的看看外观模式的实现方式。首先是我们的子系统，它包括三个接口，三个实现，LZ这里一并给出。

```
package com.facade;


public interface Sub1 {


      void function1();


}


package com.facade;


public interface Sub2 {


      void function2();
```

```
}

package com.facade;

public interface Sub3 {

	void function3();

}

package com.facade;

public class Sub1Impl implements Sub1{

	public void function1() {
		System.out.println("子系统中Sub1接口的功能");
	}

}
```

```
package com.facade;

public class Sub2Impl implements Sub2{

	public void function2() {
		System.out.println("子系统中Sub2接口的功能");
	}

}

package com.facade;

public class Sub3Impl implements Sub3{

	public void function3() {
		System.out.println("子系统中Sub3接口的功能");
	}

}
```

以上便是我们模拟出的一个子系统，那么现在便是我们最重要的接口出场的时候了，LZ给出Facade以及它的简单实现。

```
package com.facade;

public interface Facade {

	/*	下面随便组装几个功能	*/

	void function12();

	void function23();

	void function123();

}

package com.facade;

public class FacadeImpl implements Facade{

	private Sub1 sub1;
```

```
private Sub2 sub2;

private Sub3 sub3;

public FacadeImpl() {
      super();
      this.sub1 = new Sub1Impl();
      this.sub2 = new Sub2Impl();
      this.sub3 = new Sub3Impl();
}

public FacadeImpl(Sub1 sub1, Sub2 sub2, Sub3 sub3) {
      super();
      this.sub1 = sub1;
      this.sub2 = sub2;
      this.sub3 = sub3;
}

public void function12() {
      sub1.function1();
```

```
        sub2.function2();
    }

    public void function23() {
        sub2.function2();
        sub3.function3();
    }

    public void function123() {
        sub1.function1();
        sub2.function2();
        sub3.function3();
    }
}
```

以上便是我们的外观接口和实现类，它当中的功能一般是根据是客户端的需要定制的，将客户端的一个完整功能作为一个行为，然后调用子系统完成。下面我们看看客户端的调用。

```
package com.facade;

public class Client {
```

```
        public static void main(String[] args) {
                Facade facade = new FacadeImpl();
                facade.function12();
System.out.println("------------------------");
                facade.function23();
System.out.println("------------------------");
                facade.function123();

                /*   以上为使用了外观模式的调用方式，以下为原始方式    */

                System.out.println("-------------以下原始方式----------");
                Sub1 sub1 = new Sub1Impl();
                Sub2 sub2 = new Sub2Impl();
                Sub3 sub3 = new Sub3Impl();
                sub1.function1();
                sub2.function2();
System.out.println("------------------------");
                sub2.function2();
                sub3.function3();
```

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

            sub1.function1();

            sub2.function2();

            sub3.function3();

        }


}
```

LZ在下面还给出了原始的调用方式，可以看出在外观模式的作用下，我们客户端只依赖外观一个接口，而在原始的方式下，我们的客户端依赖于整个子系统，所以外观模式主要解决的是类之间的耦合过于复杂。

附上LZ运行结果。

以上便是标准的外观模式展现，LZ下面再给出需要知晓的几点。

1，实际使用当中，接口并不是必须的，虽说根据依赖倒置原则，无论是处于高层的外观层，还是处于底层的子系统，都应该依赖于抽象，但是这会倒置子系统的每一个实现都要对应一个接口，从而导致系统的复杂性增加，所以这样做并不是必须的。

2，外观接口当中并不一定是子系统中某几个功能的组合，也可以是将子系统中某一个接口的某一功能单独暴露给客户端。

3，外观接口如果需要暴露给客户端很多的功能的话，可以将外观接口拆分为若干个外观接口，如此便会形成一层外观层。

上述LZ给出的第三点，便是为了引出我们标题当中的service，相信各位做过web开发的都见过我们项目中很多的service和dao（注：小型项目或许不需要service这一层），这一层service层，有一个非常重要

的作用，就是为了方便我们管理项目中与业务逻辑相关的事物，而service层，其实是给我们的事务管理器提供了一个可以方便的配置切入点的事物管理层。

除了上述这个重要的功能外，service层同时也是组合dao层暴露给action的功能，dao层的各个类只是简单的数据操作对象，它们不具有业务逻辑，而赋予了它们业务逻辑方便action调用的功臣，正是service这一层。各位可以想象一下，假设没有service这一层，你的action当中有很多功能需要依赖多少个dao才可以完成工作。

同时在WEB项目中，有的项目会抽象出一层service接口和一层dao接口，这是为了降低客户端（这里的客户端可以认为是action）与业务实现细节以及service外观层与数据操作实现细节的耦合，而有的项目则没有抽象层，这也并非就是不合适的。

首先添加抽象层会大大的加剧项目的类文件数量，从而使项目的复杂性增加，而且在项目刚进入开发的时候，往往接口是不稳定的，因为我们经常会需要要给某一个service添加一个方法，而为了将方法暴露给客户端（即action），我们必须将该方法添加到对应的接口当中。

所以针对这一情况，我们更好的做法是等到接口行为相对稳定时，再考虑是否要重构去添加抽象的接口，而且现在的IDE工具都在一定程度上对重构进行了支持，比如eclipse就可以直接导出一个类的接口，所以我们完全可以在需要时快速的给项目添加抽象的接口层。

相比起观察者模式，适配器模式等适合小规模使用的设计模式，外观模式更多的是大范围的使用，它会是很多时候支撑我们整个架构的设计思路。

鉴于此，LZ此处不再给出具体的service和dao的示例，各位的项目中到处都充斥着这种例子。

如果形象的去形容外观模式在WEB中的应用，可以说它让action和dao分了手，而插入了一个第三者service，断开了action与dao的耦合，转而使用更高层的service。

这里需要提醒各位的是，外观模式并不是简单的使用组合将功能组合起来，也就是说它的重点不在组合功能，而在于制作一个对外暴露的外观。它一般是用来将一个子系统（注意，是一个子系统，也就是说

外观并不是简单的几个类的组合就是外观模式了）的功能进行调配，暴露给客户端一个外部的表象，使得客户端与子系统断开依赖关系。

由于外观模式属于一种“大”模式，所以我们平时很少会接触到，但是有很多技术的应用，其实都有着外观模式的影子。

比如webservice，它是给一个WEB应用提供一个外观，让客户端可以调用一些接口去使用WEB应用当中的一些功能或者说服务。再比如API，中文名称应用程序接口，它其实也可以看做是给底层的操作系统做了一层外观，使程序猿在编程的时候可以直接使用外观提供的接口，从而间接的指挥操作系统完成一些事情。

本次外观模式的分享，LZ没有像之前一样写一堆示例代码给各位看，更多的是在讨论外观模式的应用场景和应用范围，希望各位看完之后对外观模式有一个宏观的认识，而不是仅限于代码层次的理解。

好了，本次外观模式的分享就到此结束了，谢谢各位的收看，下期再见。

（十二）命令模式详解（故事版）

背景：小左是魔都某公司技术部的一名屌丝程序猿，每天的工作就是维护一个20世纪的古董级项目，由于公司不大，所以公司很多制度不太完善，导致小左每天都郁闷异常，只是靠偶尔的在刚毕业的小小美女程序媛旁边露一手，来丰富自己的精神生活。

某一天下午一点半，阴。

小左正趴在桌子上，迷迷糊糊想着某些个不切实际的美梦，突然QQ上“滴滴滴”的声音，将本就只是半梦半醒的小左给吵醒了。无奈之下，小左只好懒懒散散的抬起头，打开了QQ消息。

业务人员A：“小左啊，我们的XX列表需要加一个显示项和一个搜索项，你看你啥时候能做完啊，这个很急。”

小左眯缝着眼一看，心里小小一盘算，不算太复杂，于是就匆匆的回复了。

小左：“这个啊，很快，下午就搞定。”

于是小左摩拳擦掌的就开始准备下手，结果eclipse还没打开，QQ就又响起来了。

业务人员B：“小左啊，我们的XX页面需要改一下啊，得多添几个保存的项，你看你啥时候能做完，给我个时间？”

心里嘀嘀咕咕的盘算一下，小左就回复到：“大概需要大半天吧，明天搞定成不？”

业务人员B：“不行啊，这个很急啊，最好今天就做完，你看你加加班，搞定它，没问题吧？”

小左心里一边骂娘，一边回复到：“好吧，我尽快。”

关了QQ窗口，小左心里开始犯嘀咕：“这下活多了，得赶紧做，争取一下，一下午做两个功能差不多，幸亏自己在给完成时间时多加了一点。”

刚要下手，小左的QQ又响了起来。

业务人员C：“小左啊，我刚带一个客户来做业务，我不小心把东西填错了，你看你能不能给我改一下，客户在这等着呢，着急啊。”

小左心里已经开始抓狂了，但还是淡定的回复到：“好吧，我看看。”

不过刚连上生产环境的数据库，select语句还没写完，小左的QQ又响起来了。

业务人员D：“小左啊，我们的系统为什么XX列表点搜索不好用啊，你赶紧把这个问题解决啊，这么简单的系统怎么这么多问题啊。”

小左此时已经彻底被激怒，一个程序猿最忍受不了的就是被不懂程序的人指指点点，于是没好气的回复到：“尽量吧，这会太忙，没空。”

这一下，估计是刚毕业的小美女听到了不断的QQ消息声，竟然主动和小左搭话。

小甜甜：“噢爸（韩语），你好像很忙啊，要不要小妹帮帮你啊。”

小左：“没事，你刚来公司没多久，业务还不太熟悉，这点事我分分钟就搞定。”

说完，小左还扬起嘴角，摆了个超自信的表情，简直是猥琐到极点。

不过这一招对付刚毕业的单纯小姑娘还真好使，只见小甜甜一脸的崇拜相，笑着回道：“你好棒哦，谢谢噢爸的照顾啊。”

这不吹牛不要紧，苦逼的小左愣是独自加班到夜里十一点半，把今天接到的任务都给搞定了，为的就是在小甜甜面前证明自己的能力。这血淋淋的事实说明了，屌丝的生活中，随便一个有点姿色的美女都足以秒杀屌丝，并且让其卖命。

托着疲惫的身体回到家中，小左到了家里就往床上一躺，衣服袜子扔的满屋子都是，也懒得洗脸刷牙，就准备睡觉了。

忽然小左猛地坐起来，自言自语到："对啊，今晚该写博客了，好久没写了，最近都忙的不行了。"

想到这些，小左觉得刚才的困意一瞬间就消失无踪，这也算是屌丝的一大气质了吧---夜里欢。

随着电脑缓慢的启动，小左的脑子中就开始飘荡着今天的事，"由于公司的制度不完善，人员缺乏，导致业务人员一有什么事情，全部都来找自己。本来自己就已经身兼数职，现在还要每天面临业务员们的各种炮轰，实在是有点受不了啊。"

于是，小左想着想着，就打算将这个问题写出来，看能不能从代码里找到现实中解决问题的答案。

在键盘前犹豫片刻后，小左先把自己用JAVA代码解释了一遍。

```
package com.command;



public class Programmer {



	private String name;



	public Programmer(String name) {
```

```
        super();
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void handleDemand(){
        System.out.println( name + "处理新需求");
    }

    public void handleBug(){
        System.out.println( name + "处理bug");
    }

    public void handleProblem(){
        System.out.println( name + "处理线上问题");
    }
```

```
}
```

看着自己的功能，小左不禁感叹，“自己就是一个机器啊”。感叹之余，小左又继续摸着下巴思考了片刻，写出了业务人员的类。

```
package com.command;


public class Salesman {


	private String name;


	public Salesman(String name) {
		super();
		this.name = name;
	}


	public String getName() {
		return name;
	}


	public void putDemand(Programmer programmer){
		System.out.println( "业务员" + name + "提出新需求");
```

```
        programmer.handleDemand();
    }

    public void putBug(Programmer programmer){
        System.out.println( "业务员" + name + "提出bug");
        programmer.handleBug();
    }

    public void putProblem(Programmer programmer){
        System.out.println( "业务员" + name + "提出线上问题");
        programmer.handleProblem();
    }
}
```

最后，小左就干脆拿今天的工作为例，写了一下一天的工作。

```
package com.command;

public class Work{

    public static void main(String[] args) {
        Programmer xiaozuo = new Programmer("小左");
```

```
        Salesman salesmanA = new Salesman("A");
        salesmanA.putDemand(xiaozuo);

        Salesman salesmanB = new Salesman("B");
        salesmanB.putDemand(xiaozuo);

        Salesman salesmanC = new Salesman("C");
        salesmanC.putBug(xiaozuo);

        Salesman salesmanD = new Salesman("D");
        salesmanD.putProblem(xiaozuo);
    }

}
```

看着运行的结果，小左不禁发出一声冷笑，暗自为自己感到悲哀。仔细的看着代码，小左开始一一分析自己的悲哀。

1，业务员和自己的耦合度太高，导致每个业务人员都可以直接命令自己，这导致自己心里很不爽。

2，由于自己本身只是个程序猿，所以自己对公司的业务并不是特别擅长，很难给任务制定优先级，给任务排序，自己所擅长的还是码代

码，结果造成的后果就是经常按时完成了一个不重要的任务，但不小心却将很重要的任务向后推迟了，如果业务人员告状，自己可能就要挨批。可是这么多业务人员提问题，到底谁轻谁重，我哪知道。

3，如果任务堆积过多的时候，自己有时会忘记其中的一个甚至几个，人毕竟精力有限，如果任务多了，难免会出现这种情况，结果有的业务人员就直接告状到老板那里，实在是憋屈。

4，由于任务是随即产生的，业务人员什么时候想到任务就随时给自己，所以没有一个整体的规划，很容易导致自己加班。

“看来，说来说去，还是业务人员跟我太紧密了啊。”小左叹气道。

由于小左最近在研究设计模式，所以就想到了是不是可以用设计模式来解决自己和业务人员之间的问题。于是小左开始在度娘上面找寻适合的设计模式。

“有了，命令模式应该是可以解决的，看它的定义好像挺符合的”。电脑前的小左突然一拍大腿，大叫道。

定义：在软件系统中，“行为请求者”与“行为实现者”通常呈现一种“紧耦合”。但在某些场合，比如要对行为进行“记录、撤销/重做、事务”等处理，这种无法抵御变化的紧耦合是不合适的。在这种情况下，如何将“行为请求者”与“行为实现者”解耦？将一组行为抽象为对象，实现二者之间的松耦合。这就是命令模式（Command Pattern）。

看着这个定义，小左心中想道：“命令模式中所说的两个角色，不正是我和业务人员吗。业务人员是行为请求者，他们请求我，噢，不，应该说命令我产生编码，修改bug和处理线上问题的行为，而我就去实现或者说执行这些行为。况且，看命令模式的定义，还可以支持记录，我正需要这个记录啊，否则每次任务太多，忘了哪个任务，都要我挨批。”

于是小左迫不及待的开始研究命令模式的类图，试图从中找到解决的办法。

“不对啊，这个模式当中好像有一个invoker（调用者），这会是什么样的人呢？”小左看着命令模式的类图，不禁有些疑惑。

小左看到类图的时候，一下就看出自己便是Receiver，而业务人员便是Client，可是这个Invoker是什么呢？从类图上看，小左知道，这应该是一个特别的人，它是专门用来记录业务人员提出的需求，bug以及线上问题等等，并且还要负责通知我完成各个任务。

“我擦，这不就是产品经理（ProductManager）吗。”小左脑子中忽然闪出这个名词。

“这下有了，产品经理负责接收业务人员的各个任务，然后过滤和排好优先级以后再交代给我完成，并且他可以记住所有接受过的任务，这样我也不会再忘记了，因为有他提醒着我呢。”

说干就干，小左立马就开始尝试用命令模式去处理自己和业务人员的紧耦合问题，而解决这个问题最直接的手段就是添加了一个产品经理，并且将各个任务都抽象成一个对象，这样产品经理就可以管理这些对象了，或者也可以说是管理这些任务。

小左觉得自己的类是不需要变化的，主要变化的是要添加一组抽象的行为对象，还有改变客户端调用的方式，也就是业务人员的类。

那么首先是添加命令接口和具体的命令，不过这里似乎叫任务（Task）更合适，下面就写。

```
package com.command;



public interface Task {



      void handle();



}
```

快速的写出这个接口，小左脑子里开始思考具体的命令应该是哪几个，而且具体的命令是要关联一个接受者的，而接受者正是自己，也就是Programmer。

“有了，应该有这几个具体的命令。”有了思路，小左就开始疯狂的敲打键盘码代码，写出了下面几个具体的任务。

```
package com.command;


public class Demand implements Task{


	private Programmer programmer;


	public Demand(Programmer programmer) {
		super();
		this.programmer = programmer;
	}


	public void handle() {
		programmer.handleDemand();
	}


	public String toString() {
```

```
            return "Demand [programmer=" +
programmer.getName() + "]";
      }

}

package com.command;

public class Bug implements Task{

      private Programmer programmer;

      public Bug(Programmer programmer) {
            super();
            this.programmer = programmer;
      }

      public void handle() {
            programmer.handleBug();
      }
```

```
	public String toString() {
		return "Bug [programmer=" + programmer.getName() + "]";
	}

}

package com.command;

public class Problem implements Task{

	private Programmer programmer;

	public Problem(Programmer programmer) {
		super();
		this.programmer = programmer;
	}

	public void handle() {
		programmer.handleProblem();
	}
```

```
	public String toString() {

		return "Problem [programmer=" + programmer.getName() + "]";

	}



}
```

“这下妥了，这些任务包括了新的需求，bug和线上问题等等这些种类，而接受者正是我自己，或者说是程序猿，所以handle方法里，应该使用程序猿处理这个任务。”顿了一下，小左继续自言自语道：“这下就差一个最主要的角色，产品经理了，我来试着写一下。”

```
package com.command;



import java.util.ArrayList;

import java.util.Arrays;

import java.util.List;



public class ProductManager {



	private static final int TASK_NUMBER_IN_DAY = 4;//一天最多分派掉四个任务，多了推到第二天



	private List<Task> taskList;
```

```
	public ProductManager() {
		super();
		taskList = new ArrayList<Task>();
	}

	//接受一个任务
	public void receive(Task task){
		taskList.add(task);
	}
	//分配给程序猿任务，督促程序猿完成
	public void assign(){
		Task[] copy = new Task[taskList.size() > TASK_NUMBER_IN_DAY ? taskList.size() - TASK_NUMBER_IN_DAY : 0];
		for (int i = 0; i < TASK_NUMBER_IN_DAY && i < taskList.size(); i++) {
			taskList.get(i).handle();
		}
		System.arraycopy(taskList.toArray(), TASK_NUMBER_IN_DAY > taskList.size() ? taskList.size() : TASK_NUMBER_IN_DAY, copy, 0, copy.length);
		taskList = Arrays.asList(copy);
```

```
    }
    //打印剩下的任务
    public void printTaskList(){
        if (taskList == null || taskList.size() == 0) {
            System.out.println("----------当前无任务----------");
            return;
        }
        System.out.println("---------当前剩下的任务列表---------");
        for (Task task : taskList) {
            System.out.println(task);
        }
System.out.println("----------------------------------");
    }
}
```

“仔细想想，产品经理负责的任务还挺多呢，比如优先级排序，任务过滤，需求转化等等，怪不得产品经理要比程序猿工资高呢。不过貌似全部写出来有点复杂啊，就让他负责记录下任务，然后分配下任务吧。”小左皱着眉头自言自语道。

“不过这样的话，业务员也应该变一下了，他是要认识产品经理的。”

```
package com.command;
```

```
public class Salesman {

	private String name;

	public Salesman(String name) {
		super();
		this.name = name;
	}

	public String getName() {
		return name;
	}

	public void putDemand(ProductManager
productManager,Programmer programmer){
		System.out.println( "业务员" + name + "提出新需
求");
		productManager.receive(new Demand(programmer));
	}
```

```
	public void putBug(ProductManager productManager,Programmer programmer){

		System.out.println( "业务员" + name + "提出bug");

		productManager.receive(new Bug(programmer));

	}



	public void putProblem(ProductManager productManager,Programmer programmer){

		System.out.println( "业务员" + name + "提出线上问题");

		productManager.receive(new Problem(programmer));

	}

}
```

“这下好了，业务人员再也不能让我直接去写代码了，必须得发给产品经理，再让产品经理分配给我。这下我看下我的工作应该是什么样子了。”

```
package com.command;



public class Work {



	public static void main(String[] args) {

		Programmer xiaozuo = new Programmer("小左");
```

```
        ProductManager productManager = new
ProductManager();


        Salesman salesmanA = new Salesman("A");
        Salesman salesmanB = new Salesman("B");
        Salesman salesmanC = new Salesman("C");
        Salesman salesmanD = new Salesman("D");

        salesmanA.putDemand(productManager, xiaozuo);
        salesmanB.putDemand(productManager, xiaozuo);
        salesmanB.putBug(productManager, xiaozuo);
        salesmanC.putDemand(productManager, xiaozuo);
        salesmanC.putProblem(productManager, xiaozuo);
        salesmanD.putDemand(productManager, xiaozuo);

        System.out.println("第一天产品经理分配任务");
        productManager.assign();
        productManager.printTaskList();
        System.out.println("第二天产品经理分配任务");
        productManager.assign();
        productManager.printTaskList();
```

```
        }


}
```

小左看着运行结果和代码开始思考，“这样和标准的命令模式几乎已经一样了，只是设计模式还是不能死搬硬套啊，有些地方还是有些别扭。比如业务人员不应该再依赖于程序猿，业务人员应该只依赖产品经理就可以了，他不应该负责把任务对应给我，而应该是产品经理决定某一个任务由谁去处理的。我还得把这个地方再改一下，要不按照现在的样子，业务人员还是能把任务分配到我身上。不过目前的好处是，我不需要当时就做了，而是等着产品经理分配。从结果就看出来了，刚才是业务人员发个任务我就做一个，现在是产品经理分配了一天的任务，然后一起做。”

“按照刚才想的思路，产品经理应该要认识所有的程序猿，而且还要提供一个可以选择程序猿的方法。”小左拖着下巴喃喃的说道，说完以后就开始修改刚才的代码。

```
package com.command;


import java.util.ArrayList;

import java.util.Arrays;

import java.util.List;


public class ProductManager {
```

```
    private static final int TASK_NUMBER_IN_DAY = 4;//一天最多分派掉四个任务，多了推到第二天


    private List<Task> taskList;


    private List<Programmer> programmerList;//产品经理应该认识所有的程序猿


    private int currentIndex;


    public ProductManager(Programmer... programmers) {
        super();
        if (programmers == null || programmers.length == 0) {
            throw new RuntimeException("产品经理手下没有程序员，任务分配不出去，无法正常工作！");
        }
        taskList = new ArrayList<Task>();
        programmerList = Arrays.asList(programmers);
    }


    //接受一个任务
```

```
	public void receive(Task task){

		taskList.add(task);

	}



	public void assign(){

		Task[] copy = new Task[taskList.size() > TASK_NUMBER_IN_DAY ? taskList.size() - TASK_NUMBER_IN_DAY : 0];

		for (int i = 0; i < TASK_NUMBER_IN_DAY && i < taskList.size(); i++) {

			taskList.get(i).handle();

		}

		System.arraycopy(taskList.toArray(), TASK_NUMBER_IN_DAY > taskList.size() ? taskList.size() : TASK_NUMBER_IN_DAY, copy, 0, copy.length);

		taskList = Arrays.asList(copy);

	}

	//产品经理可以选择程序猿，简单的循环选取。

	public Programmer chooseProgrammer(){

		return programmerList.get(currentIndex == programmerList.size() ? 0 : currentIndex++);

	}
```

```
	public void printTaskList(){
		if (taskList == null || taskList.size() == 0) {
			System.out.println("----------当前无任务----------");
			return;
		}
		System.out.println("---------当前剩下的任务列表---------");
		for (Task task : taskList) {
			System.out.println(task);
		}
		System.out.println("------------------------------------");
	}
}
```

“这下业务人员就不需要认识我了，哈哈。”小左边YY着有了产品经理的美好生活，一边将业务人员的类改成了下面的样子。

```
package com.command;


public class Salesman {


	private String name;
```

```
    private ProductManager productManager;


    public Salesman(String name) {
        super();
        this.name = name;
    }


    public Salesman(String name, ProductManager
productManager) {
        super();
        this.name = name;
        this.productManager = productManager;
    }


    public void putDemand(){
        System.out.println( "业务员" + name + "提出新需
求");
        productManager.receive(new
Demand(productManager.chooseProgrammer()));
    }
```

```
	public void putBug(){

		System.out.println( "业务员" + name + "提出bug");

		productManager.receive(new
Bug(productManager.chooseProgrammer()));

	}



	public void putProblem(){

		System.out.println( "业务员" + name + "提出线上问
题");

		productManager.receive(new
Problem(productManager.chooseProgrammer()));

	}



	public String getName() {

		return name;

	}



	public ProductManager getProductManager() {

		return productManager;

	}
```

```
	public void setProductManager(ProductManager
productManager) {

		this.productManager = productManager;

	}

}
```

这下一天的工作形式就变化了，小左将一天的工作写成了下面这个样子。

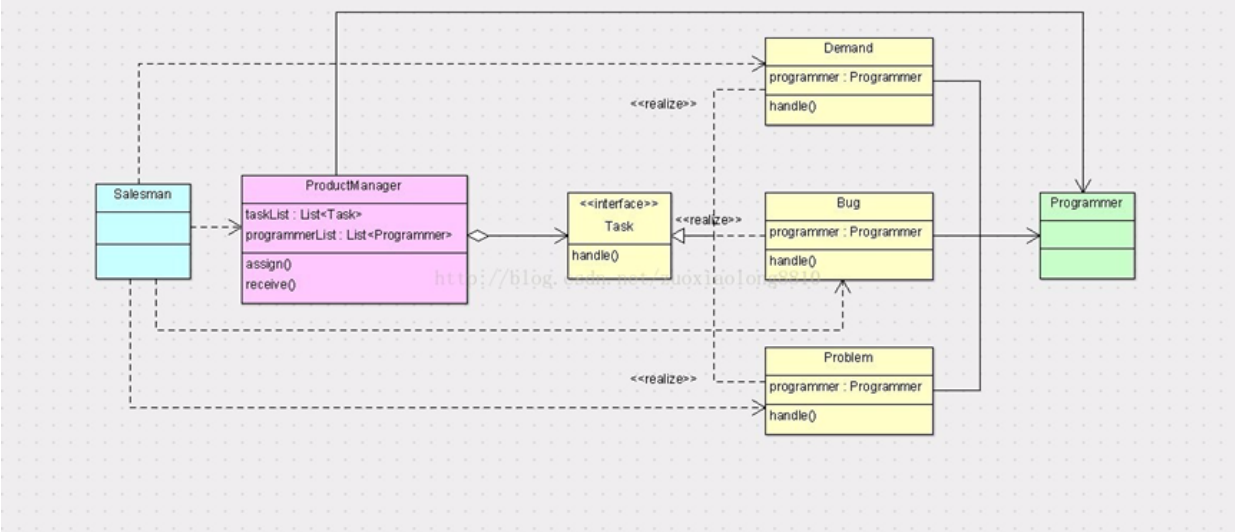
```
package com.command;



public class Work {



	public static void main(String[] args) {

		Programmer xiaozuo = new Programmer("小左");

		ProductManager productManager = new
ProductManager(xiaozuo);



		Salesman salesmanA = new
Salesman("A",productManager);

		Salesman salesmanB = new
Salesman("B",productManager);

		Salesman salesmanC = new
Salesman("C",productManager);

		Salesman salesmanD = new
Salesman("D",productManager);
```

```
        salesmanA.putDemand();
        salesmanB.putDemand();
        salesmanB.putBug();
        salesmanC.putDemand();
        salesmanC.putProblem();
        salesmanD.putDemand();

        System.out.println("第一天产品经理分配任务");
        productManager.assign();
        productManager.printTaskList();
        System.out.println("第二天产品经理分配任务");
        productManager.assign();
        productManager.printTaskList();
    }

}
```

“哈哈，这下业务人员彻底不认识我了，只认识产品经理，只不过由于公司就我一个程序员，所以产品经理那个平均的循环分配算法，分来分去还是分给我一个人了。不过这个也没办法，公司不招人，我也说了不算啊，而且小甜甜还没熟悉好公司业务呢，也帮不了我。”

虽然最终这些任务都还是小左去做，但是这下小左不用加班了，一天最多四个任务，就算完不成临时需要加班，那也是小左编码速度的问题，心里相对会舒服很多，不至于太憋屈。毕竟自己没完成，那加会儿班也会心甘情愿。

这会小左心情好了，心情一好，小左竟然开始动手画起刚才的类图来了。



小左看着自己的作品，自言自语道：“虽然这个实际的类图比命令模式的原版类图稍微多了一层关系，就是ProductManager和Programmer的关联关系，但是其它的可是和命令模式一模一样啊，而且很明显，业务人员（Salesman）和我们程序猿（Programmer）之间没有关联的关系了。”

“看来我们公司应该招一个产品经理了啊，还得多招几个程序猿，这样我的工作就简单的多了。公司和程序倒是挺相似的，程序是需要多个类之间相互协作，公司也是一样啊，需要每个员工都分工协作，这样就不需要一些人身兼数职了，简直是累死个人啊！”

“博客今天就先不写了，稍微总结一下，明天好去给小甜甜说一下我今晚的发现，顺便给她讲讲命令模式。这下又可以在她面前露一手了，估计这样下去，她早晚会爱上我啊。啊，哈哈。。”小左又开始YY了。

不过为了能给小甜甜留一个好印象，小左可是煞费心思啊，这不，他又开始总结起来了。

“为了减轻自己的负担，我添加了一个产品经理，并且还将任务抽象成类，这样确实解决了我的很多问题，我得稍微总结下。”

1，程序猿和业务员解耦，不直接打交道。

2，产品经理分担了程序猿的很多潜在任务，比如制定任务优先级，先做哪个后做哪个。

3，程序猿不至于忘掉其中一个任务，因为产品经理那有任务列表的。

4，任务有规划的完成，不至于加班或者说加班太频繁。

“不过要是给小甜甜解释，这样还不够啊，得专业一点，才好忽悠啊。”小左暗暗的点头道。

用编程的语言来解释命令模式的使用场景或者说可以解决的问题，就是下面几点。

1，希望将行为请求者和行为实现者解耦，不直接打交道。

2，希望分离掉行为请求者一部分的责任，行为请求者只需要将命令发给调用者，不再主动的去让行为实现者产生行为，符合单一职责原则。

3，希望可以控制执行的命令列表，方便记录，撤销/重做以及事务等功能。

4，期待可以将请求排队，有序执行。

5，希望可以将请求组合使用。

“不过上面我总结的第五点在例子当中没有体现，不过这个也好办，其实就相当于是任务可以组合，比如一个线上的问题（Problem）很可能伴随着一个bug（Bug），这样的话可以做一个组合的任务，既要处理线上的问题同时也要处理Bug。”

“装X就要装到底啊，命令模式的优点和缺点都有哪些，我也得给总结出来，别到时候在小甜甜面前掉链子。”小左一甩自己的中分头说道。

1，最大的优点，就是将行为请求者和行为实现者解耦。

2，命令的添加特别方便，并且可以方便的制定各种命令和利用现有命令组合出新的命令。

3，如果针对每一类具有共同接口的接受者制作一个调用者，可以控制命令的执行情况。

“命令模式的缺点应该和大部分设计模式一样，会增加系统的复杂性，这里的复杂性应该主要指的是类的数量，这个倒是好理解，看一下上面的例子就知道多了很多类。”

“这下可以睡个安稳觉了，希望明天一举拿下啊。啊哈。”说完，小左就躺床上进入了梦乡。

（十三）中介者模式详解（玄幻版）

自盘古开天辟地以来，模式大陆之上，逐渐孕育出一种具有高级智慧的生物----人类。

多年以来，人类凭借着强大的繁衍能力，使得人类的数量急剧增长。伴随着数量的增加，各种争执与战争开始不可避免的产生，这一度使得人类的数量急剧减少，模式大陆也岌岌可危。

不过自古乱世出英雄，就在大陆上各个势力纷杂排布，战争不断时，一位名为GOF的绝世强者凭空而出，他一生修行，最终领悟天道，掌握了单一、迪米特、接口隔离、里氏替换、依赖倒置、开闭六大天道，也正因如此，后人都称其为六道天人。

当时凭借着六大天道，GOF在模式大陆以一敌万并非虚言，进而一举统一天下，制止了人类无尽的战争。随后，GOF为了方便控制模式大陆，创造出二十三个修行分支，设立门派和宗主，并亲自传授各宗主修炼之道，让其帮助掌管一方土地。

随后二十三门派渐渐稳定了模式大陆的局势，而六道天人本人，自那以后，则再未出现过。

要说这二十三门派当中，实力最强的一派便是抽象工厂宗门，由于其门派绝学《抽象工厂模式》高度的抽象化，使得抽象工厂宗主的实力要稍稍领先于各个宗门。但总的来说，各个门派的宗主，其实力都在伯仲之间。值得一提的是策略宫，相较策略宫宫主的实力，可以说任何一个宗门的宗主都在其之上，但是其层出不穷的策略一向阴狠毒辣，令人防不胜防，所以各大门派都对其十分厌烦。

中介者神阁，阁主大厅。

“参见左阁主！”大厅内，一群人正全部单膝跪地，恭敬的向正位上端坐的一人行礼。

这位左阁主看起来年龄不大，中分头，身材也属于矮小一列，一脸的青春美丽痘象征着他还属于青年阶段，猛一看去，实在是不像一位高高在上的阁主。但这并没有影响到这位阁主在阁中的威望，因为所有神阁中的人都知道，左阁主仅用了两年的时间，潜心修炼阁中绝学《中介者模式》，就以绝对的优势，打败了原阁主，进而登上了阁主宝座。

“起来吧！各位长老齐聚本座大厅，想必是有什么要事禀报吧。”左阁主的左手轻轻一摆，众人便自觉一股推力凭空而出，将众人全部托起。

“阁主，最近由于六道天人大人久未现身，各个宗门都蠢蠢欲动，尤其是与我神阁相距不远的命令门和单例宗，近期总是隔三差五的骚扰我神阁，请阁主下令，命属下带领阁中将士铲平异族。”众人之中，一位约莫七十岁的白发老人说道，尤其是说到最后时，老者明显情绪波动很大，显然是对于命令门和单例宗的做法十分愤怒。

“戚长老莫要冲动，最近碰见这种事的可不止我神阁，这件事并不该是各个宗门自行处理，本阁主要想一个万全之策。”

左阁主在听完戚长老的话之后，神色之间并没有任何慌乱与愤怒，而是十分淡然的挥了挥手，表示让众人先退下，由他思量之后再做定夺。不过在神阁中，戚长老是出了名的冲动派，尽管已经年迈七旬，但却比一些个二十多岁的青年还要冲动。

“阁主，我们阁中已经损失了不少人手，难道阁主就如此轻易的便宜那帮无耻之徒吗？”戚长老并未立即退出，而是向前一步继续说道。

“放肆，你想违抗本阁主的决定吗！”听到戚长老的话，左阁主猛然释放出自身的气势，愤怒的说道。

被左阁主的气势所压迫，戚长老知道，此时再多言的话，恐怕今日就真的要被阁主收拾一番了，于是，心中虽然不情愿，但还是信誓旦旦的离开了。

中介者神阁，阁主卧房。

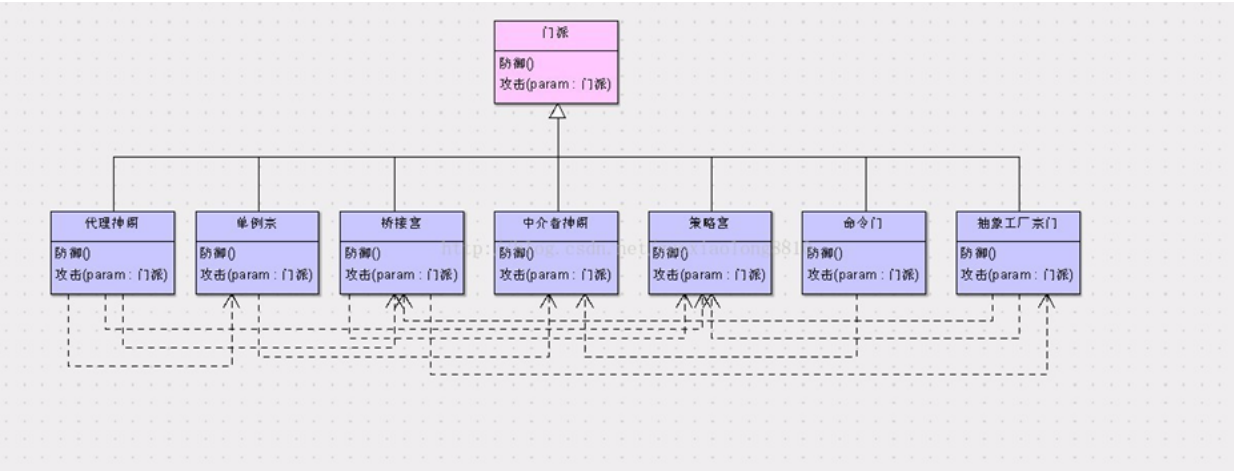
“当年六道天人曾预言，模式大陆将会产生第二次动荡，想必就是指的此事了吧。”左阁主眉头微皱，喃喃的说道。

顿了一下，左阁主继续自言自语道：“老阁主在传位时曾告诉我，六道天人当年预测出大陆的第二次动荡时，曾专门夜访中介者神阁，并告知老阁主，此次动荡若想彻底压住，则需要依靠中介者神阁的力量，而其中的关键，正是藏于本阁的秘籍《中介者模式》当中。看来此次，我是要仔细研究一下本阁的震阁之宝了。”

说着，左阁主便盘膝而坐，灵魂当中开始参透《中介者模式》与此次动荡的关联，而《中介者模式》秘籍的内容也一一浮现于左阁主脑海。

中介者模式：用一个中介对象来封装一系列的对象交互，中介者使各对象不需要显式地相互引用，从而使其耦合松散，而且可以独立地改变它们之间的交互。中介者模式又称为调停者模式，它是一种对象行为型模式。

“按照当年六道天人所说，这秘籍当中所说的一系列的对象，应该就是指的各个门派了吧。现在各个门派之间相互单独交往，关系复杂，很难让大陆安静。”左阁主说着说着，虚空之中便出现一幅当前大陆格局的地图。



“虽然有了抽象的父类门派，但是各个具体门派之间的关系，看起来实在是有点混乱，如果要用秘籍暗语来诠释的话，我神阁和命令门以

及单例宗的关系应该是这样的。”左阁主说着，双手则开始在虚空之中画出了一个个奇异的符号。

“首先是一个抽象的门派父类，它代表了我们各个门派的共性。这个好像有点模板宗秘籍《模板方法模式》的味道，像是一个模板类。”

```
package com.mediator;

//门派抽象类

public abstract class School {



	public void defense() {

		System.out.println(getName() + "防御");

	}



	public void attack(School school) {

		System.out.println(getName() + "攻击" +
school.getName());

	}



	public abstract String getName();



}
```

“下面就是我神阁以及命令门还有单例宗具体的门派了。”左阁主说着，一个个奇异的符号不断的从其手中涌出，飘散在前方的虚空之中。

```
package com.mediator;

public class MediatorSchool extends School{

	public String getName() {
		return "中介者神阁";
	}

}

package com.mediator;

public class CommandSchool extends School{

	public String getName() {
		return "命令门";
	}

}
```

```
package com.mediator;


public class SingletonSchool extends School{


	public String getName() {
		return "单例宗";
	}


}
```

“有了这些的话，现在大陆的格局就好描述了，应该就是现在这样了。”

```
package com.mediator;
//大陆格局
public class ContinentalPattern {


	public static void main(String[] args) {
		School singletonSchool = new SingletonSchool();
		School commandSchool = new CommandSchool();
		School mediatorSchool = new MediatorSchool();


		singletonSchool.attack(mediatorSchool);
```

```
            commandSchool.attack(mediatorSchool);


            mediatorSchool.defense();
        }


}
```

“这样果然是不行啊，各个门派可以随意进攻，混乱不堪，没有任何节制力量，而且各个门派交往起来非常繁琐，每一个门派都要应付与其余二十二个门派的关系，怪不得现在模式大陆逐渐开始混乱呢。况且我神阁都已经被单例宗和命令门攻击一段时间了，最后才迟迟反应过来，慌忙的采取防御策略，难免损失惨重。”看着虚空之中漂浮的文字结果，左阁主暗暗的摇头道。

顿了一下，左阁主继续自言自语道：“按照我神阁秘籍所述，大陆之中应该出现一个中介者对象，它可以负责各个门派之间的交往以及控制各个门派的动作。这样的话，确实可以解决大陆的动荡，去除掉各个门派之间复杂的关系，让所有的门派都听命于这个中介者。看来六道天人果然料事如神啊。”

双眼紧闭思考片刻之后，左阁主便猛地睁开双眼，随即一字一顿的说出了中介者的名称，“模式联盟”。

随后，左阁主的双手在虚空中一阵虚画，一个个秘籍暗语在虚空之中散发着夺目的光芒，这些光芒赫然组成了模式联盟接口。

```
package com.mediator;

//模式联盟，中介者接口

public interface IPatternAlliance {
```

```
	//加入联盟

	public abstract void add(School school);



	//联盟攻击

	public abstract void resolveAttack(School activeSide,
School passiveSide);



	//联盟防御

	public abstract void resolveDefense(School
passiveSide);



}
```

“这便是模式联盟接口了，它算是模式联盟的联盟宗旨，指导着联盟所要做的事情。”左阁主点头道。

“下面便是模式联盟的真正面目了。”

```
package com.mediator;



import java.util.ArrayList;

import java.util.List;

//模式联盟，中介者

public class PatternAlliance implements IPatternAlliance {
```

```
    private List<School> schoolList;


    public PatternAlliance() {
        super();
        schoolList = new ArrayList<School>();
    }


    public void add(School school){
        schoolList.add(school);
    }


    public void resolveAttack(School activeSide,School
passiveSide){
        if (schoolList.contains(activeSide) &&
schoolList.contains(passiveSide)) {
            System.out.println("主攻方" +
activeSide.getName() + "以及被攻方" + passiveSide.getName() +
"都已加入联盟，不允许内讧！");
        }else if (schoolList.contains(activeSide) &&
!schoolList.contains(passiveSide)) {
            System.out.println("主攻方" +
activeSide.getName() + "已加入联盟，被攻方" +
passiveSide.getName() + "不在联盟之中，将集体攻打该门派！");
            for (School school : schoolList) {
```

```
                    school.attack(passiveSide);
                }
            }else {
                System.out.println("主攻方" +
activeSide.getName() + "未加入联盟，联盟无权干预此事！");
                activeSide.attack(passiveSide);
            }
        }


        public void resolveDefense(School passiveSide){
            if (schoolList.contains(passiveSide)) {
                System.out.println("被攻方" +
passiveSide.getName() + "已加入联盟，联盟将齐力防御！");
                for (School school : schoolList) {
                    school.defense();
                }
            }else {
                System.out.println("被攻方" +
passiveSide.getName() + "未加入联盟，联盟不给予防御帮助！");
                passiveSide.defense();
            }
        }
```

```
}
```

“这应该便是联盟的大致蓝图了，它可以负责各个门派之间的交往，去除掉门派之间直接的关联，并且这样做的话，还可以控制联盟中的门派。剩下的就是各个门派的布局需要稍微改变一下。”左阁主看着自己一手策划的联盟，轻声说道。

“首先应该就是抽象父类需要变化一下了，因为现在每一个门派的动作都需要委托给联盟去做。”

```
package com.mediator;

//门派抽象类

public abstract class School {


	protected IPatternAlliance patternAlliance;


	public School(IPatternAlliance patternAlliance) {

		super();

		this.patternAlliance = patternAlliance;

	}


	public void defense() {

		System.out.println(getName() + "防御");

	}
```

```
    public void attack(School school) {
        System.out.println(getName() + "攻击" + school.getName());
    }

    //有了中介者，将由中介者处理
    public void attackByPatternAlliance(School school){
        patternAlliance.resolveAttack(this, school);
    }

    public void defenseByPatternAlliance(){
        patternAlliance.resolveDefense(this);
    }

    public abstract String getName();

}
```

“这一下各个门派的布局只需要稍微变化一下就好，因为每一个门派都要认识这个中介者，也就是模式联盟。那么策略宫，这个众矢之的，也是要认识的了，只是联盟要不要收留这个家伙，那就另说了，嘿嘿！”左阁主阴笑着说道。

```
package com.mediator;

public class CommandSchool extends School{

	public CommandSchool(IPatternAlliance patternAlliance)
{
		super(patternAlliance);
	}

	public String getName() {
		return "命令门";
	}

}

package com.mediator;

public class MediatorSchool extends School{

	public MediatorSchool(IPatternAlliance patternAlliance)
{
```

```
            super(patternAlliance);
        }

        public String getName() {
            return "中介者神阁";
        }

}

package com.mediator;

public class SingletonSchool extends School{

        public SingletonSchool(IPatternAlliance
patternAlliance) {
            super(patternAlliance);
        }

        public String getName() {
            return "单例宗";
        }
```

```
}

package com.mediator;

public class StrategySchool extends School{

	public StrategySchool(IPatternAlliance patternAlliance)
{
		super(patternAlliance);
	}

	public String getName() {
		return "策略宫";
	}

}
```

“这下大陆的格局就要变化一下了，本阁主就先来模拟一个吧。不过策略宫的下场好像不怎么好啊。嘿嘿。”阴笑一声，左阁主便开始在虚空中画起秘籍暗语。

```
package com.mediator;

//大陆格局
```

```
public class ContinentalPattern {


      public static void main(String[] args) {

            IPatternAlliance patternAlliance = new
PatternAlliance();



            School singletonSchool = new
SingletonSchool(patternAlliance);

            School commandSchool = new
CommandSchool(patternAlliance);

            School mediatorSchool = new
MediatorSchool(patternAlliance);



            School strategySchool = new
StrategySchool(patternAlliance);



            //策略宫没有被联盟收留

            patternAlliance.add(mediatorSchool);

            patternAlliance.add(commandSchool);

            patternAlliance.add(singletonSchool);



singletonSchool.attackByPatternAlliance(mediatorSchool);
```

```
commandSchool.attackByPatternAlliance(mediatorSchool);

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

mediatorSchool.attackByPatternAlliance(strategySchool);

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

strategySchool.attackByPatternAlliance(mediatorSchool);

mediatorSchool.defenseByPatternAlliance();

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

      }



}
```

“嘿嘿，如果有此格局，本阁主相信策略宫应该马上就要从大陆上消失了。”左阁主看着虚空中漂浮的文字结果，阴笑声更加猖狂了。

收起脸上的阴笑，左阁主忽然表情严肃的自言自语道：“想必这就是当初六道天人的预言了，看来模式联盟是必须要建立的了，这样可以避免各个门派之间的混乱场面，而且也可以集齐力量，一举消灭一些黑暗势力，实在是妙招。”

“接下来本阁主就要总结一下自己对《中介者模式》的参悟了，想来这应该对本阁主功力的提升有很大好处。”说完，左阁主便开始在虚空之中勾画自己的参悟。

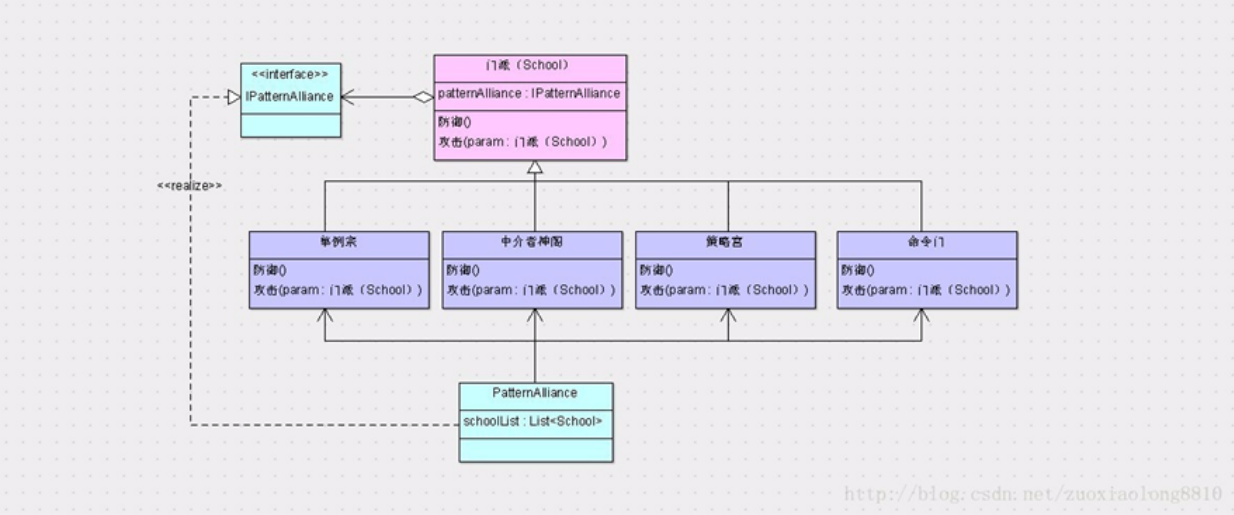
《中介者模式》针对的问题是，解决一系列对象之间复杂的耦合关系，这一系列对象往往是“多对多”的耦合关系，《中介者模式》采用一个中介者对象将这一系列对象集中管理，而各个对象也将自己与其它对象的交互行为委托给中介者处理，从而减少这一系列对象之间的耦合。

它的优点倒是非常明显，清除了一系列对象之间复杂的耦合关系，并且中介者可以控制这一系列对象的行为，统一管理。

“这算是《中介者模式》秘籍的优势，不过它很明显有一个缺点，如果被抓住的话，则很容易被破招。那就是模式联盟一旦成立，就必须可以正常运行，否则模式联盟一旦崩溃，那么下面的门派都要危险了。”左阁主暗暗的说道。

《中介者模式》的缺点是，由于中介者负责着一系列对象的交互与控制，所以中介者的类会非常复杂，而且一旦中介者类无法正常工作，那么所有将行为委托给中介者的类都将会出现问题，所以在使用的时候还是要特别小心。

“下面，本阁主就要看下现在大陆的格局地图了。”说完，左阁主便开始在虚空中勾画格局地图。



“这下门派之间那些乱七八糟的关系全部都消失了，模式联盟关联着所有的门派，进而简化了各个门派之间的交往。看来本阁主得尽快联系抽象工厂宗门的宗主，他算是二十三门派中最有声望的一位修

者，想必要是说服他的话，模式联盟的建立会顺利很多，至于策略宫的话，嘿嘿。”想到以后策略宫可能要面临二十二门派的讨伐，左阁主就不禁阴笑连连。

魔都某出租屋里。

小左正呼呼的做着美梦，脸上挂着阴险的笑容，嘴里还不停的说着：“该死的策略宫。”

忽然，小左猛的睁开双眼，慌忙的看了下手机，随即便立刻穿上衣服，向门外冲去，边跑着嘴中还不停的说道：“我擦，原来只是个梦，都十点半了，这下完蛋了，该死的策略宫。”

（十四）桥接模式详解（都市异能版）

魔都国贸附近，某天夜里十一点半。

那一晚，魔都出奇的没有做只打雷不下雨的勾当，老天似乎是要将魔都淹没一般，倾盆大雨像不要命似的拍打着地面。

漆黑的夜幕中，一道黑影从一个十字路口狂奔而出，而就在此时，一辆红色宝马急速穿过，一瞬间，黑影就犹如被巨力抛飞一般腾空而起，直到几秒钟后，才重重的摔落在车前足足二十多米处。

“吱...兹....”

一声凄厉的刹车声划破了魔都的长空。

之后，一个身材高挑的女子，慌忙的从车中奔跑到黑影落下的地方，或许是由于看到倒下之人满身鲜血的模样，女子吓的不禁发出一声惊叫后退了几步，愣是咬着牙迟迟不敢上前。

或许是由于慌乱之下失了神，女子后退的时候，差点被一个物体绊倒。女子低头一看，竟然是一只拖鞋，不知为何，看到这一幕，一时间女子的心绪稳定了不少。

待得情绪稳定以后，女子这才再次意识到自己撞到人了，而此人到现在还不知死活。于是女子急忙再次跑到黑影身旁，开始试图叫醒这位被自己撞伤的路人，而由于此时一直是大雨倾盆，被撞之人身上的鲜血已经被冲刷干净，女子也渐渐的看清了黑影的面貌。

此人看起来年纪不大，穿着一条大裤衩和一条普通的黑色T恤，光着两个脚丫子，五官倒还算端正，只是满脸的青春痘将其仅有的一点英俊全部淹没，不过从此人光着的脚丫子就可以看出，刚才地上的拖鞋应该就是此人被撞飞的鞋子了。

久久尝试叫醒伤者未果之后，女子无奈之下，只好使出浑身的力气才堪堪拖动此人，等到将受伤之人拖入车中之后，女子其实已经累的满

身大汗，只是在这风雨交加的夜晚，看不出来罢了。

XX医院急救室。

“你说什么？死了？”女子瞪大着一双晶莹的眼珠，不可思议的凝视着眼前宣布结果的医生。

此时，在灯光的照耀下，才看清女子的面貌，女子穿着一身白色蕾丝连衣裙，一双眼睛简直像浸在水中的水晶一样澄澈，肤色晶莹如玉，薄薄的唇，色淡如水，清秀的五官里透露出一种说不出的乖巧。而且由于刚才被大雨淋湿，女子的全身已经湿透，仔细看去，隐约可见其包裹在着装之内的嫩肤，更是给原本就美到极点的女子添加了一份诱人的姿色。

“是的，本来接上输氧管后是有微弱的呼吸的，结果过了几秒钟之后就彻底断气了，对不起，我们已经尽力了。”医生叹息一声说道，紧接着，医生继续说道：“这是他临死前让我给你的，请节哀吧”。说着，医生便将手中的一张纸条递给了女子，而快接触到女子玉手的时候，医生忍不住吞了口口水，显然是被女子的美色惊到了。

女子在听完医生的宣判之后，差点晕倒在地，幸好离座位很近，所以才一屁股做到了座位上，不至于摔倒。呆呆的望着正在被缓缓的推往太平间的急救车，女子有点不知所措，她到现在甚至都不知道自己撞死之人姓谁名谁。

想起刚才医生说的话，女子这才意识到手中握着的纸条，打开一看，女子的表情瞬间凝固了，只见纸条上歪歪曲曲的写着几个字：你TM踩着我输氧管了。

XX医院太平间。

“尼玛，二货医生，老子用尽了生前的力量，才勉强写了那么几个字。这蠢货也不打开看看，竟然以为我是临死前写给女友的小情书，也不想想，老子命都快没了， 还写毛毛情书啊。再说，看我这屌丝样，外面那女神能是我女友吗。我真是去年买了个表。” 骂骂咧咧的

声音从太平间中传来，这要是被他人听到了，估计当场就要被吓昏过去。

只见一个人影从急救床上缓缓的坐起来，令人惊奇的是，此人竟然生生的穿过了铺在身上的白布，看上去就像是此人不存在一般。坐起的一人，正是被刚才的貌美女子撞死之人，而此人，正是加班到深夜的小左。

“貌似老子转生了啊，就是这身体咋跟个魂儿似的，完全没有实体感啊。难道让我就这样每天飘来飘去的，我擦。”看着自己几乎透明的身体，小左没有时间为自己的死后转生所兴奋，而是紧皱着眉头开始琢磨自己的身体。

“啊哈，我拥有异能了啊，这简直是太TM逆天了。”小左研究了将近三个小时，终于发现了自己的异常。然而在发现自己竟然神奇的拥有了异能之后，小左已然压抑不住内心的兴奋，在太平间里大喊大叫，完全不顾身边沉睡的众人们。

不过兴奋只持续了片刻，小左便眉头紧皱，一脸闷闷不乐的样子，因为他的面前站着八个小左，是的，八个。

小左在研究以后发现，他是拥有异能没错，而且可以转换成屌丝和高富帅的形象，但是让人抓狂的是，他必须先分离出屌丝和高富帅的分身，然后才能由这两个分身，再次分离出带有异能的屌丝和高富帅形象的分身，这些异能分别都有隐身，飞行以及读心术，而且一个分身同时只能有一种能力。所以这样两两组合，最后就出现了八个小左，分别是纯屌丝和高富帅分身，以及拥有一种异能的三个屌丝分身，以及分别拥有一种异能的三个高富帅分身。

不过令人欣喜的是，这八个分身都是真真实实的实体，或者说他们除了有特殊能力之外，其实和现实中的人类是一模一样的，尤其是具有屌丝模样的分身，完全和身前的小左一模一样。不过小左变成高富帅以后的样子，倒是真的很帅，长眉若柳，身如玉树，原本坑坑洼洼的脸庞一时间也光滑嫩白，比之女生也丝毫不差。想必就这个样子，除非是与小左特别熟悉的人，才能仔细的从这分身当中看出一丝似曾相识的感觉。

只是目前这种情况虽说有意外的惊喜，但小左仍然觉得十分糟糕。

原因就是，他要是想使用带有异能的分身，则至少需要创造两次分身，一次无技能的屌丝或者高富帅分身，一次则是在此基础上分离出带有异能的分身。还有更致命的一点，就是如果以后正在使用某一个技能的分身，而恰巧在关键时刻需要使用当前异能之外的两个的话，则需要临时替换掉分身，而替换一个分身，则至少需要两次分身的分离。要知道，分身的产生可是相当耗时的，虽说也不长，但十分八分的，还是要的。

“不知道这种情况，能不能用设计模式解决呢？”哪怕是死后转生，思维方式依旧没有变化。前世身为程序猿的小左，在转生后遇到问题时，第一时间想到的，竟然是设计模式。

说干就干，小左收起八个分身，便控制着自己飘忽的身体开始寻找电脑，因为必须要有电脑，才能激发自己的设计灵感。

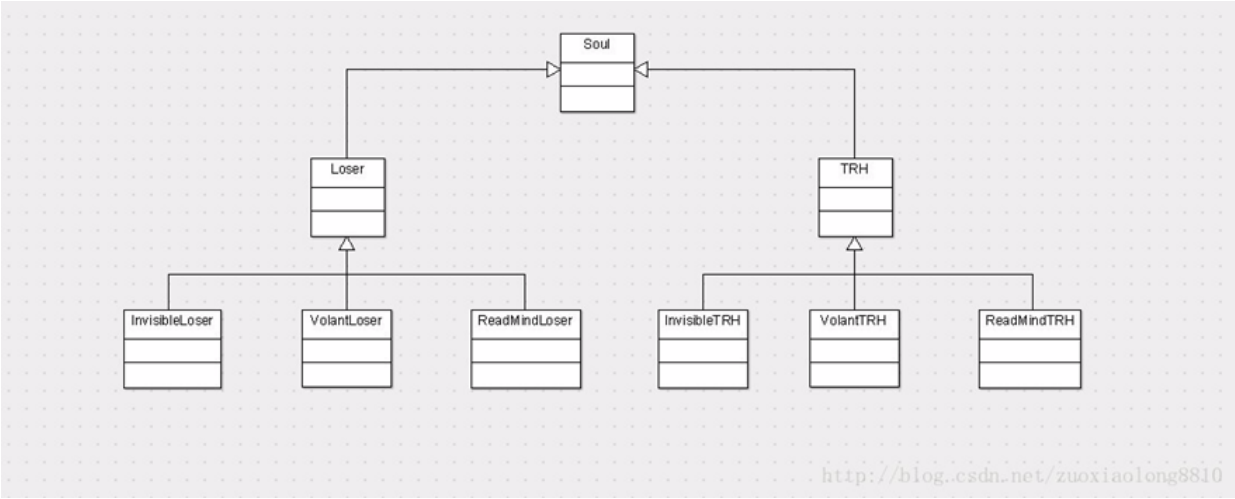
寻找的途中，小左也发现，别人竟然完全看不到自己，不过这其实早在小左的意料之中。至于原因，用脚趾头都能想的明白，都已经变成魂魄了，还能看见个毛啊。

由于此时已至深夜，所以医院所剩的工作人员寥寥无几，凭借着可以穿透一切实物的身体，小左很轻易的就找到了一个无人的办公室。不过当小左按下电脑开机键的时候，竟然发现自己的手直接穿过了机箱。

“靠！”愤怒的爆了声粗口，无奈之下，小左只好暂时将自己不带异能的屌丝分身给招了出来，用来帮自己操作电脑。

在经过半个多小时左右的捯饬之后，小左已经在机子上安装好了JDK以及eclipse。于是没有任何犹豫，小左就开始模拟刚才自己的情况。

“刚才的情况来看，我自己的灵魂体以及八个分身，很明显应该是一个继承体系。而我的灵魂体，就是这个继承体系的超级父类。不过由于灵魂体没有实体，所以做成抽象的可能更合适点。不过我可以先来个简单的类图看一下他们的关系。”小左一边自言自语，一边已经开始了在电脑上勾画类图。



“画出类图果然更加清晰一点，这是一个标准的简单的继承体系，下面我就用代码来尝试着写一下吧。”说着，小左便开始在键盘上敲打着代码。

```
package com.bridge;

//抽象灵魂类

public abstract class Soul {


	public void createDoppelganger(){

		System.out.println("制作分身");

	}


}

package com.bridge;

//屌丝分身
```

```
class Loser extends Soul{

	public Loser(){
		System.out.println("制造一个无技能的屌丝分身");
	}

	public void doAllLikePeople(){
		System.out.println("分身可以像正常人一样做任何事");
	}

	public void show(){
		System.out.println("展示屌丝形象");
	}

}
//高富帅分身
class TRH extends Soul {

	public TRH(){
		System.out.println("制造一个无技能的高富帅分身");
```

```
	}

	public void doAllLikePeople(){
		System.out.println("分身可以像正常人一样做任何事");
	}

	public void show(){
		System.out.println("展示高富帅形象");
	}

}


package com.bridge;
//带有隐身技能的屌丝分身
class InvisibleLoser extends Loser{

	public InvisibleLoser(){
		System.out.println("制造一个带隐身技能的屌丝分身");
```

```
        }

        public void releaseSkills(){
                System.out.println("屌丝分身释放隐身技能");
        }
}
//带有飞行技能的屌丝分身
class VolantLoser extends Loser{

        public VolantLoser(){
                System.out.println("制造一个带飞行技能的屌丝分
身");
        }

        public void releaseSkills(){
                System.out.println("屌丝分身释放飞行技能");
        }
}
//带有读心术技能的屌丝分身
class ReadMindLoser extends Loser{
```

```
	public ReadMindLoser(){
		System.out.println("制造一个带读心术技能的屌丝分身");
	}

	public void releaseSkills(){
		System.out.println("屌丝分身释放读心术技能");
	}
}
//带有隐身技能的高富帅分身
class InvisibleTRH extends TRH{

	public InvisibleTRH(){
		System.out.println("制造一个带隐身技能的高富帅分身");
	}

	public void releaseSkills(){
		System.out.println("高富帅分身释放隐身技能");
	}
}
```

```
//带有飞行技能的高富帅分身
class VolantTRH extends TRH{

	public VolantTRH(){
		System.out.println("制造一个带飞行技能的高富帅分身");
	}

	public void releaseSkills(){
		System.out.println("高富帅分身释放飞行技能");
	}
}
//带有读心术技能的高富帅分身
class ReadMindTRH extends TRH{

	public ReadMindTRH(){
		System.out.println("制造一个带读心术技能的高富帅分身");
	}

	public void releaseSkills(){
```

```
            System.out.println("高富帅分身释放读心术技能");
        }
}
```

“恩，我的灵魂体以及这几个分身的关系倒是挺简单粗暴，就是继承，那么现在这种情况下我在使用的时候，应该就像下面这样了。”

```
package com.bridge;


public class Use {


        public static void main(String[] args) {
            InvisibleLoser invisibleLoser = new InvisibleLoser();
            invisibleLoser.releaseSkills();
System.out.println("------------------------------------");
            ReadMindTRH readMindTRH = new ReadMindTRH();
            readMindTRH.releaseSkills();
System.out.println("------------------------------------");
            InvisibleTRH invisibleTRH = new InvisibleTRH();
            invisibleTRH.releaseSkills();
        }
```

}

看着输出的结果，小左摇了摇头，自言自语道：“果然与我刚才的分析一模一样，每次切换技能的时候，都至少需要制造两次分身，才可以切换成功，这样的话，切换技能非常不方便。而且万一以后我又领悟一个技能，又或者增加了一种外貌，那么分身的数量会急剧增加。”

“那么到底什么设计模式可以解决这个问题呢？”小左喃喃的说道。

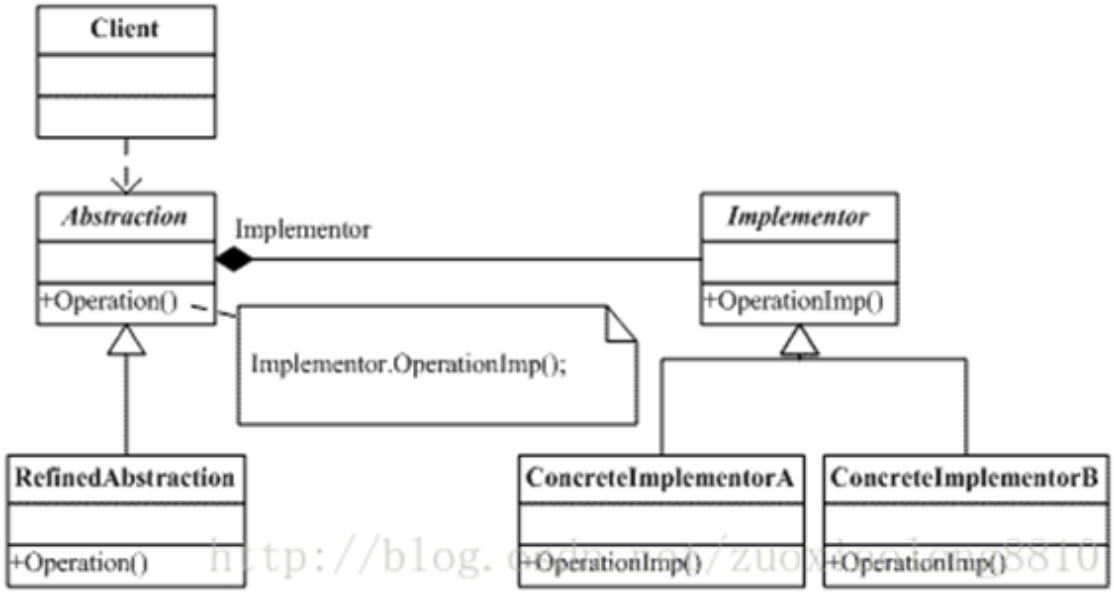
在说话的同时，小左已经开始在度娘上搜索合适的设计模式，试图解决自己的分身问题。

“有了，啊哈。桥接模式正是解决这个问题的利器啊。”小左犹如发现了新大陆一般兴奋的叫道，而双眼则死死的盯着桥接模式的定义。

定义：在软件系统中，某些类型由于自身的逻辑，它具有两个或多个维度的变化，那么如何应对这种“多维度的变化”？如何利用面向对象的技术来使得该类型能够轻松的沿着多个方向进行变化，而又不引入额外的复杂度？这就要使用Bridge模式。而具体使用的方式，则是将抽象部分与他们的实现部分分离，使得它们都可以独立的变化。

“桥接模式中说到了某些类型，会有两个或多个维度的变化，这里指的不正是我的分身吗。我的分身既可以按外貌分类，即屌丝和高富帅，又可以按技能分类，即隐身、飞行和读心术。不过这使用方式倒是说的有些模糊。”小左看着桥接模式的定义，喃喃的说道。

“还是看下类图吧，说不定能找到一些端倪。”说着，小左就在度娘上打开了桥接模式的类图。



“从类图上看，抽象部分的具体行为交给了实现部分，而就我目前的情况来说，我的不同实现有两种，一种是外貌，一种是技能。这么说的话，抽象部分和实现部分，通常意义下，应该指的是继承体系中，接口相同而实现也相同的部分则为抽象部分，而接口相同但是实现不同的部分则为实现部分。”

“这样的话，按照我目前的情况，制作分身以及像正常人一样做任何事的方法则是抽象部分，而形象的展示和技能的释放则是实现部分了。下面我就尝试着来分离一下吧，哈哈！”有了解决方案，小左不禁心情大好，慌里慌张的就开始尝试着分离抽象和实现部分。

“首先，我要将实现部分给写出来，这里有两个维度，一个是技能，一个是外貌。”

```
package com.bridge;



//技能接口

public interface Skills {
```

```
	public void releaseSkills();

}

//隐身技能
class Invisible implements Skills{

	public void releaseSkills(){
		System.out.println("释放隐身技能");
	}
}
//飞行技能
class Volant implements Skills{

	public void releaseSkills(){
		System.out.println("释放飞行技能");
	}
}
//读心术技能
class ReadMind implements Skills{
```

```
	public void releaseSkills(){
		System.out.println("释放读心术技能");
	}
}

package com.bridge;
//外貌接口
public interface Appearance {

	public void show();

}
//屌丝外形
class Loser implements Appearance{

	public void show() {
		System.out.println("展示屌丝形象");
	}
```

```
}
//高富帅外形
class TRH implements Appearance{

	public void show() {
		System.out.println("展示高富帅形象");
	}

}
```

“这便是两个实现化的部分了，现在我已经将它分离出来，剩下的就是要将他们加入到灵魂基类中了。”

```
package com.bridge;
//抽象灵魂类
public abstract class Soul {

	protected Appearance appearance;//外貌

	protected Skills skills;//技能

	//具体的实现交给实现部分处理
	public void show(){
```

```
        appearance.show();
    }
    //具体的实现交给实现部分处理
    public void releaseSkills(){
        skills.releaseSkills();
    }

    public void createDoppelganger(){
        System.out.println("制作分身");
    }
    //抽象的方法，留给分身实现
    public abstract void doAllLikePeople();

    public Soul() {
        super();
    }

    public Soul(Skills skills) {
        super();
        this.skills = skills;
```

```
}

public Soul(Appearance appearance) {
      super();
      this.appearance = appearance;
}

public Soul(Appearance appearance, Skills skills) {
      super();
      this.appearance = appearance;
      this.skills = skills;
}

public Appearance getAppearance() {
      return appearance;
}

public void setAppearance(Appearance appearance) {
      this.appearance = appearance;
}
```

```
	public Skills getSkills() {
		return skills;
	}

	public void setSkills(Skills skills) {
		this.skills = skills;
	}

}
```

“这下已经基本搞定了，最后一步则是我要修正一下灵魂类，因为灵魂类只是定义了抽象化的角色，它是需要分身类来修正以后，才可以拥有真正的像人类一样活动的能力。”

```
package com.bridge;

//分身类
public class Doppelganger extends Soul{

	public Doppelganger(){
		System.out.println("制作一个暂无外貌和技能的分身");
	}
```

```
	public void doAllLikePeople(){
		System.out.println("分身可以像正常人一样做任何事");
	}


}
```

“OK了，这下我来看一下现在要使用的话要怎么使用吧。”到了此时，小左明显已经有点迫不及待了。

```
package com.bridge;


public class Use {


	public static void main(String[] args) {
		Soul soul = new Doppelganger();
		soul.setAppearance(new TRH());//设置分身外貌为高富帅
		soul.setSkills(new Invisible());//设置隐身技能
		soul.doAllLikePeople();
System.out.println("----------------------------");
		soul.show();
		soul.releaseSkills();
```

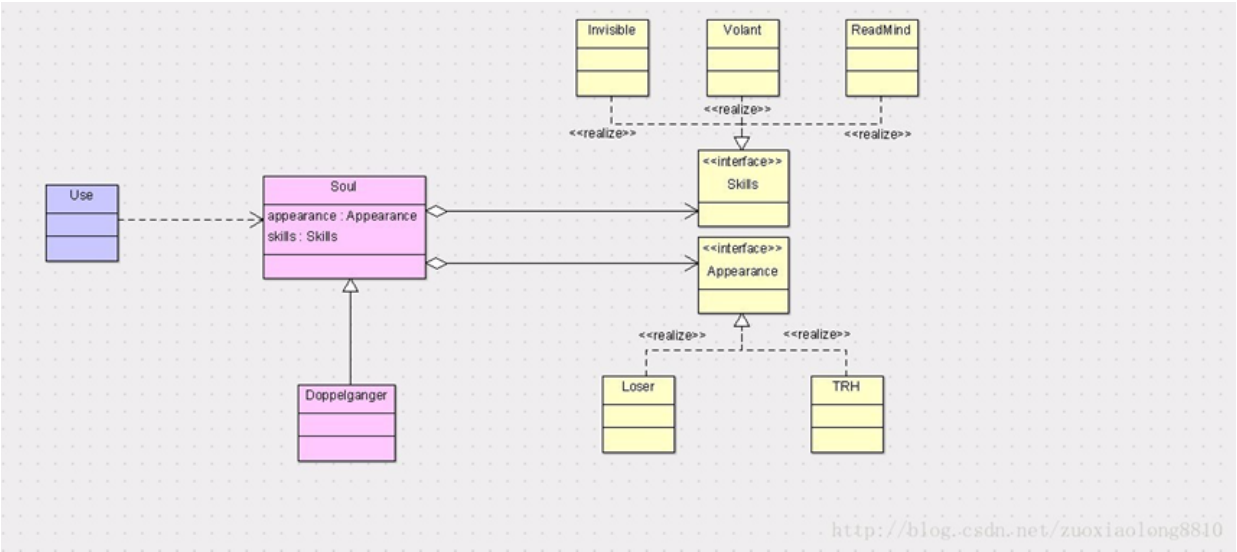
```
System.out.println("----------------------------");
            //切换技能
            soul.setSkills(new ReadMind());
            soul.releaseSkills();
System.out.println("----------------------------");
            //切换外貌
            soul.setAppearance(new Loser());
            soul.show();
System.out.println("----------------------------");
            //切换技能
            soul.setSkills(new Volant());
            soul.releaseSkills();
      }


}
```

“不错不错。这下我可以随意切换外貌和技能了，而且要是以后万一我领悟什么新的技能，按照以前的方式的话，我至少需要添加两个子类，而要是拥有新的外貌的话，则至少需要添加四个子类，而现在的话，不管是那种情况，我都只需要添加一个子类，就可以随意切换。”

“不过说起来的话，这好处应该不止这些，我还是专门总结和对比一下吧，省得以后使用的时候遇见什么问题，自己搞不清楚。”

转生之后，小左还是改不掉前世的习惯，在每次使用设计模式解决完问题之后，都喜欢总结一番。而在世之时，小左的这个习惯，则全是为了在小甜甜面前得瑟所养成的。

“先画个类图来看一下现在的关系。”说着，小左便开始画起类图来了。



“这图当中，红色的则是抽象部分，黄色的则是实现部分，而两者之间被两条线连接，看起来就像是一座桥一样，怪不得这个设计模式叫桥接模式呢。”看着自己画出来的类图，小左忽然觉得，桥接模式这个名字，起的实在是妙。

“下面，我就得总结一下前后的好处和坏处了。”

首先，得先总结一下以前的坏处都是哪些。

第一，分身太多，不容易控制。

第二，不同技能和不同外貌之间，切换起来非常不灵活。

第三，要是想添加技能或者外貌的分类的话，要产生的分身非常多。

要是从专业点的角度来分析的话，则应该是下面这几点了。

第一，子类太多。

第二，由于客户端与具体子类的耦合度很高，导致在子类间切换非常繁琐。

第三，如果以后需要扩展子类功能的话，则子类的数目会随着维度以及每一个维度中子类的个数的增加呈几何倍数增长。

那么现在的好处则是一下几点了。

第一，分身的数目减少了，而且伴随着技能和外貌的种类的增加，这种减少会越来越明显。

第二，在不同技能和不同外貌之间的切换将会非常灵活。

第三，如果以后技能和外貌的种类增加了，又或者分身又可以按其它的属性分类了，那么这种添加会非常容易。

如果把好处说的专业一点的话，则是下面这样。

第一，子类的数目减少了。

第二，实现部分的切换非常容易，主要表现在抽象部分和实现部分的耦合度很低，因为使用聚合取代了继承。

第三，扩展的时候很简单，可以更好的容纳变化，不论是维度的增加还是每一个维度中子类的个数的增加，都会变得非常简单。

疯狂的总结了一番，小左便开始按照桥接模式的设计，尝试着改变对外貌和技能的控制。

果然，在尝试了数十次之后，一个普通的分身终于被小左给分离了出来，而小左的灵魂中则保存着屌丝和高富帅的外貌以及三种异能。

伴随着小左的意念变动，这个普通的分身则会产生响应的变化。忽而是屌丝模样，忽而又变成了高富帅的样子，忽而隐形不见，忽而飞出窗外，又迅速的飞回原地，实在是让小左玩的不亦乐乎。

一年后，魔都某天桥。

“天灵灵，地灵灵，看相算命我最行，上知天文，下知地理，中间明白事理，而且通晓人理，想算命的请找小左先生算命啦。”一个身材高大，长相英俊的小伙子，卖着力气的叫喊着。

此人，正是开启了读心术的小左，而之所以变成现在这样，则是因为隐身和飞行虽然很酷，但却偏偏不能拿来赚钱。无奈之下，小左只好委曲求全，先暂时当起了算命先生。说是这么说，但实际上小左现在每天赚的钱，是以前当程序猿时的数倍不止。

此时的他，更是华丽丽的披了一身高富帅的形象，为的就是多吸引几个女顾客。不过这一招还真的相当凑效，已经有不少穿着暴露，性感火辣的女顾客来光顾了小左的生意。

这不，说着说着，一位穿着火红短裙的性感辣妹已然来到了小左摊前，“帅哥先生，麻烦你帮我看下，我的命中有没有你呢？”

“我要收摊了，不好意思。”感受到对方明显的勾引，小左并没有任何触动。而此时刚好也正值傍晚，于是小左只是淡漠的摆了摆手，就打算收摊回家。

这一年中，前世做了二十多年单身屌丝的小左，在拥有了飞行和隐身的异能之后，自然是好好的享受了一番人间乐趣，故而现在纵然是面对这种程度的勾引，小左依然可以做到无动于衷。而之所以变成这样，不光是因为这一年之中，小左对女人的定力大大增加，最重要的是，面前的女人一看就是只调戏不掏钱的主。

然而就在小左准备转身回家的时候，却是被身后一道轻柔的声音给叫住了。

“您好，可以给我算一下吗？”

听到有人叫，小左就打算回头婉拒之后直接回家，然而就在小左转过身来，看到面前女子的时候，小左却犹如被雷击一般，楞在那里一动不动。片刻之后，才颤抖的说道：“你...你就是...”

（十五）建造者模式详解（都市异能版）

傍晚，魔都某天桥。

“嗯？你认识我吗？”韩雨露疑惑的看着眼前的男子，轻声的问道。

“额...”被对方这么突然一问， 小左顿时有些语塞，一时之间竟然忘记了回答对方的问题。

“怎么了？我们认识吗？”看到对方支支吾吾的样子，韩雨露不禁疑惑更甚，再次追问道。

再次被对方追问，小左这才意识到自己刚才的表现有些失态，随即便轻咳一声，故作淡定的解释道：“不好意思，刚才有些失礼了，让您见笑了，不过我想我们应该不认识。”

“喔...是吗？”听到答案之后，韩雨露不禁有些失望，隐隐之中，她始终觉得面前之人似曾相识，不然一向不迷信的她也不会突然想起算命，但接触到现在，她却仍是想不起来二人曾经在哪里见过。

此时，小左已经确认了眼前之人，正是那一晚开车撞死自己的女子。刚才初见此女子，小左就觉得十分眼熟，之后凭借着读心术特有的能力，小左更是看到了对方心中那零零碎碎的片段，故而现在已然十分确定心中的答案。

不过看到对方将信将疑的表情，想必应该是联想到了什么，想到此，小左不禁感叹女人的直觉实在是逆天。在对方的心中，自己应该是已经死了的，而且如今自己的面貌已经发生了翻天覆地的变化，没想到对方还是能隐隐的感觉到什么，这如何能不让小左感到震惊。

“对了，你刚才不是说想算命吗？”心中震惊的同时，小左还是巧妙的转移了话题。

“是的，不过看你好像要收摊了，不知道会不会打扰到你...”韩雨露轻轻的点了点头，客气的说道。

“没事，相遇便是缘分，我们算命的最相信的便是天意，既然老天让我们相遇，就顺从天意吧。只是不知道你想算哪方面的呢？”

“这个我也不知道，就请你看着算吧。”算命本就不是韩雨露的本意，所以她就索性将主动权交给了对方。

“这个...你等下...我算算...”说完以后，小左双目紧闭，一副思索的表情，看起来还真像个算命的先生，不久一会，小左缓缓的睁开双眼，神色平静的说道：“你是不是心中有一些事无法释怀？”

听到小左的推测，韩雨露并没有答话，而是震惊了片刻之后，继续凝神望着小左等待后话。

“有些事不能跟你说的太直白，否则就违背了天意，但是我可以告诉你的是，你所无法释怀的事，其实在你这内疚的一年已经消失了。”小左面无表情的说着，刻意的停顿了一下之后，小左继续说道：“或者说...让你心怀愧疚的人，其实已经早就不怪你了。”

韩雨露一直死死的望着小左，而听到小左说出最后一句话的时候，柔软的娇躯不禁一颤。不知为何，韩雨露总觉得眼前之人所说的话是真的，不过尽管如此，韩雨露还是心存顾虑的问道：“你说的是真的吗？”

“当然是真的，一切皆为天意。小姐尽可相信。”微微的点了点头，小左悠悠的说道。

“谢谢你。我叫韩雨露，我还会再来的。”韩雨露迷人的一笑之后，随手便扔给小左一沓钱，转眼就消失在人群之中。

然而我们的小左，此时正在回味刚才美好的一幕，愣愣的呆在那里一动不动，嘴上耷拉的哈喇子清晰可见，以至于手中厚厚的钞票都没有注意到，口中还模糊不清的说着：“太美了，实在太TM美了...”

自那一次之后，韩雨露便隔三差五的会光顾小左的生意，而且每次都出手阔绰，弄的小左十分不好意思。小左也多次尝试过把钱还给对方，但无一例外，每一次小左都被韩雨露撅起的小嘴给秒杀了。说到底，屌丝在女神的面前，抵抗力还是太弱了点。

多次尝试无果之后，小左渐渐的也就不再做作，从韩雨露的出手就能看出其家境肯定不凡，而小左之前可是一个名符其实的屌丝与贫困户。再者说，虽然自己幸运的获得重生并且还无意间拥有了异能，但说起来也是九死一生，如果不是看在对方在出事之后的表现，恐怕小左此时早已做出了过激的行为。

如此平淡无奇的生活过了几个月，小左也渐渐习惯了每隔几日的相聚。韩雨露本身就生的一幅可爱模样，再加之心中的顾虑逐渐消除，故而韩雨露已经渐渐的会经常露出甜甜的微笑，这着实让小左的魔都之旅增加了不少光彩。

不过好景不长，几个月后的某一天，韩雨露却突然神奇的消失了，这让已经习惯了每隔几天就能见到韩雨露的小左十分不适应，一时之间心神不宁，而且隐隐约约之中，小左觉得韩雨露一定是出了什么事。

可惜小左此时也无可奈何，唯一能做的只有等待，伴随着时间一天一天的过去，小左不禁有些后悔，当初没有使用隐身跟着韩雨露，看看她到底住在何处，如此一来，现在也不会束手无策了。

转眼之间，已然又过去了数月，而韩雨露却仍然迟迟不见踪影，小左心中也越来越着急，而小左能做的，也只有每日按时摆摊子，期待韩雨露能尽快出现。

“卖报啦，卖报啦，韩氏集团独女入狱，引起数千民众围堵警察局。”一道稚嫩的叫卖声在天桥响起。

“小弟弟，报纸多少钱，给我一份。”说话的正是小左，当听到韩氏集团独女的时候，小左一个箭步便出现在卖报小童跟前，急切的说道。

“哥哥，一块钱一份。”卖报小童高兴的说道。

“喏，给你，不用找了。”匆匆的扔给卖报小童一张毛爷爷之后，小左便回到原地迫不及待的打开了报纸。

看到报纸上头条新闻的图片，小左的双手不禁有些颤抖，心中暗暗想道：“没想到韩雨露还真是一个大家之女，好一个韩氏集团啊。”

忽然看到韩雨露的真实身份，小左不禁觉得有点失落，纵然自己现在容貌尽变，还拥有异能，但这依然没有弥补小左心中的落差。好一会之后，小左才渐渐的调整好心态，眼神望着前方，坚定的说道：“不管怎样，我也要先把她救出来！”

魔都某出租屋，深夜。

“到底要怎么样才能把她救出来，而且也不暴露我的身份呢？”小左坐在电脑前，抓着头发眉头紧皱的喃喃道。

“如果我暴露自己的身份与异能，倒是很简单，直接隐身进去把门打开，抓住韩雨露飞出去就行。不过这样实在有点太惊世骇俗了，这样一弄，我基本上应该没有安生日子了。”想到自己暴露以后要面临的处境，小左暗自摇了摇头。

苦思冥想了好大一会，小左暗暗的道：“看来要是想悄无声息的把她放出来，只能将整个看守所变成我的了。”想到这里，小左自己也不禁被自己震撼了一把。

为了不引起注意，小左思来想去，还是觉得将看守所的人全部偷梁换柱才是最好的方案，只有如此，才能确保万无一失。原本，小左是打算冒充看守所其中一人，直接将韩雨露放出来，不过这个主意很快就被小左否定了。这样一来，万一还有其他人稍微阻拦一下，或者向上级请示一下，就彻底暴露了。

除此之外，小左也想过别的办法，就是冒充上级直接去看守所下达命令，但是这样一来，小左就得去追查具有这个权利的人，这样难免会浪费不少时间，而且这个方法也有同样的风险，因为他也不确定看守所的人是否认识这些领导，到时候能不能顺利放人，也不得而知。

“看来今晚需要去看守所一趟了，我得先去探探情况，顺便看下这些人的样子。”说干就干，既然想到了办法，小左也便不再犹豫，当即就甩门而去。

手机开着导航功能，凭借着飞行和隐身的异能，小左只用了不到一个小时，便摸清了看守所的底细。凭借着灵魂体，小左还专门去牢房确

认了一下韩雨露的下落，不过当看到韩雨露的时候，小左着实吃了一惊。

不为其他，只是因为韩雨露的牢房里，不止干净的一尘不染，而且装饰的比小左的出租屋只好不差，最让小左震惊的是，牢房里竟然还放着一台电脑。这样的待遇，比起其余惨不忍睹的牢房，实在是不得不让人感叹钱的无上神力。

“尼玛，我为什么这么想去坐牢呢。”回到家中，之前的担心一扫而空，小左现在竟然只是一心想坐牢。不过说归说，那种暗无天日的生活，纵然是金碧辉煌，想必时间久了，人也会憋坏的吧。

“既然有了看守所人的模样，想必我的灵魂应该可以分离出来相似的分身吧。”自从小左拥有异能，至今已经近两年了，这两年里，小左已经渐渐的彻底摸清了自己的能力。

小左发现自己的灵魂不仅可以分离出自己屌丝和高富帅的分身，也可以分离出其它模样的分身。只是相对于分离出与自己相似的分身，后者不应该说是分离，说是创造更加准确点，因为整个过程是要完全凭借小左的灵魂力去控制并创造的，所以制造起来非常费时费力，最重要的是，还没什么作用。所以刚开始发现时，小左还是挺兴奋的，但是新鲜感一过之后，他便发现，这个能力实在是个鸡肋。

略微尝试了一下，小左才发现制造一个全新的分身是多么困难。就在刚才，仅仅是制造一个分身的手，就花费了小左整整半个小时的时间。

“看来鸡肋技能也有用到的时候啊，真是后悔当初没有好好练习一下，照我现在的速度，恐怕连一个分身都还没制造完，天就亮了，还如何制造整个看守所人员的分身。”

沉吟了片刻，小左自言自语道：“现在是没有时间来练习了，短时间内，也只能依靠设计模式想想办法了。不过分析之前，还是先来看看我现在的状况吧。”

说着，小左便开始在电脑上敲起了代码，写的过程中，小左还时不时的摸着下巴沉思一会，然后再接着一阵噼里啪啦的敲打键盘。

“首先应该有一个分身的类，这个分身可跟以前不同了，由于不是我的灵魂直接分离的，所以其实它和灵魂已不再是继承关系，而且里面的属性也会很多，因为每一个部分我都要制造的。”

```
package com.builder;

//分身类

public class Doppelganger {

	private String name;

	private String body;

	private String head;

	private String leftArm;

	private String leftHand;

	private String rightArm;

	private String rightHand;

	private String leftFoot;

	private String leftLeg;

	private String rightFoot;

	private String rightLeg;


	public Doppelganger(String name) {

		super();

		this.name = name;
```

```
}

public String getName() {
      return name;
}

public void setName(String name) {
      this.name = name;
}

public String getBody() {
      return body;
}

public void setBody(String body) {
      this.body = body;
}

public String getHead() {
      return head;
```

```
}

public void setHead(String head) {
      this.head = head;
}

public String getLeftArm() {
      return leftArm;
}

public void setLeftArm(String leftArm) {
      this.leftArm = leftArm;
}

public String getLeftHand() {
      return leftHand;
}

public void setLeftHand(String leftHand) {
      this.leftHand = leftHand;
```

```
}

public String getRightArm() {
      return rightArm;
}

public void setRightArm(String rightArm) {
      this.rightArm = rightArm;
}

public String getRightHand() {
      return rightHand;
}

public void setRightHand(String rightHand) {
      this.rightHand = rightHand;
}

public String getLeftFoot() {
      return leftFoot;
```

```
}

public void setLeftFoot(String leftFoot) {
      this.leftFoot = leftFoot;
}

public String getLeftLeg() {
      return leftLeg;
}

public void setLeftLeg(String leftLeg) {
      this.leftLeg = leftLeg;
}

public String getRightFoot() {
      return rightFoot;
}

public void setRightFoot(String rightFoot) {
      this.rightFoot = rightFoot;
```

```
    }

    public String getRightLeg() {
        return rightLeg;
    }

    public void setRightLeg(String rightLeg) {
        this.rightLeg = rightLeg;
    }

    public String toString() {
        return name;
    }

}
```

“人要这么多部位干嘛，真是烦躁，下面应该是我的灵魂类了，它可以制造分身。”

```
package com.builder;
//灵魂类
public class Soul {
```

```
	private Doppelganger doppelganger;

	public void createDoppelganger(String name){
		doppelganger = new Doppelganger(name);
	}

	public Doppelganger getDoppelganger(){
		try {
			return doppelganger;
		}finally{
			doppelganger = null;
		}
	}

	public void createBody(String body) {
		System.out.println("设置" +
doppelganger.getName() + "的" + body);
		doppelganger.setBody(body);
	}

	public void createHead(String head) {
```

```
            System.out.println("设置" +
doppelganger.getName() + "的" + head);

            doppelganger.setHead(head);

      }



      public void createLeftArm(String leftArm) {

            System.out.println("设置" +
doppelganger.getName() + "的" + leftArm);

  doppelganger.setLeftArm(leftArm);

      }



      public void createLeftHand(String leftHand) {

            System.out.println("设置" +
doppelganger.getName() + "的" + leftHand);

            doppelganger.setLeftHand(leftHand);

      }



      public void createRightArm(String rightArm) {

            System.out.println("设置" +
doppelganger.getName() + "的" + rightArm);

            doppelganger.setRightArm(rightArm);

      }
```

```
	public void createRightHand(String rightHand) {

		System.out.println("设置" +
doppelganger.getName() + "的" + rightHand);

		doppelganger.setRightHand(rightHand);

	}



	public void createLeftLeg(String leftLeg) {

		System.out.println("设置" +
doppelganger.getName() + "的" + leftLeg);

		doppelganger.setLeftLeg(leftLeg);

	}



	public void createLeftFoot(String leftFoot) {

		System.out.println("设置" +
doppelganger.getName() + "的" + leftFoot);

		doppelganger.setLeftFoot(leftFoot);

	}



	public void createRightLeg(String rightLeg) {

		System.out.println("设置" +
doppelganger.getName() + "的" + rightLeg);
```

```
            doppelganger.setRightLeg(rightLeg);
        }


        public void createRightFoot(String rightFoot) {
            System.out.println("设置" +
doppelganger.getName() + "的" + rightFoot);
            doppelganger.setRightFoot(rightFoot);
        }


}
```

“这下就可以开始制造了，我来试着制造两个分身吧，一个瘦小的，一个胖胖的。制造的过程应该是下面这样的。”

```
package com.builder;
//制造过程
public class Creator {


        public static void main(String[] args) {
            Soul soul = new Soul();
            //制造一个瘦小的看守所的人员
            soul.createDoppelganger("看守所人员1");
            soul.createBody("瘦瘦的身体");
```

```
            soul.createHead("小小的头");
            soul.createLeftArm("瘦瘦的左胳膊");
            soul.createLeftHand("瘦瘦的左手");
            soul.createRightArm("瘦瘦的右胳膊");
            soul.createRightHand("瘦瘦的右手");
            soul.createLeftLeg("瘦瘦的左腿");
            soul.createLeftFoot("小小的左脚");
            soul.createRightLeg("瘦瘦的右腿");
            soul.createRightFoot("小小的右脚");
            System.out.println("获得了" +
soul.getDoppelganger());


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


            //制造一个胖胖的看守所的人员
            soul.createDoppelganger("看守所人员2");
            soul.createBody("胖胖的身体");
            soul.createHead("大大的头");
            soul.createLeftArm("胖胖的左胳膊");
            soul.createLeftHand("胖胖的左手");
```

```
            soul.createRightArm("胖胖的右胳膊");

            soul.createRightHand("胖胖的右手");

            soul.createLeftLeg("胖胖的左腿");

            soul.createLeftFoot("大大的左脚");

            soul.createRightLeg("胖胖的右腿");

            soul.createRightFoot("大大的右脚");

            System.out.println("获得了" +
soul.getDoppelganger());

      }

}
```

“仅仅是制造了两个人，就费了这么大的事，现在这样果然不行啊。仅仅是制造了两个分身，就已经如此繁琐了，而且万一我不小心，再缺胳膊少腿的，这要到时候充当看守所人员的时候，被韩雨露看到，还不得吓个魂飞魄散的。”不过想到韩雨露惊恐无助的样子，小左竟然不自觉的露出了暗暗的坏笑。

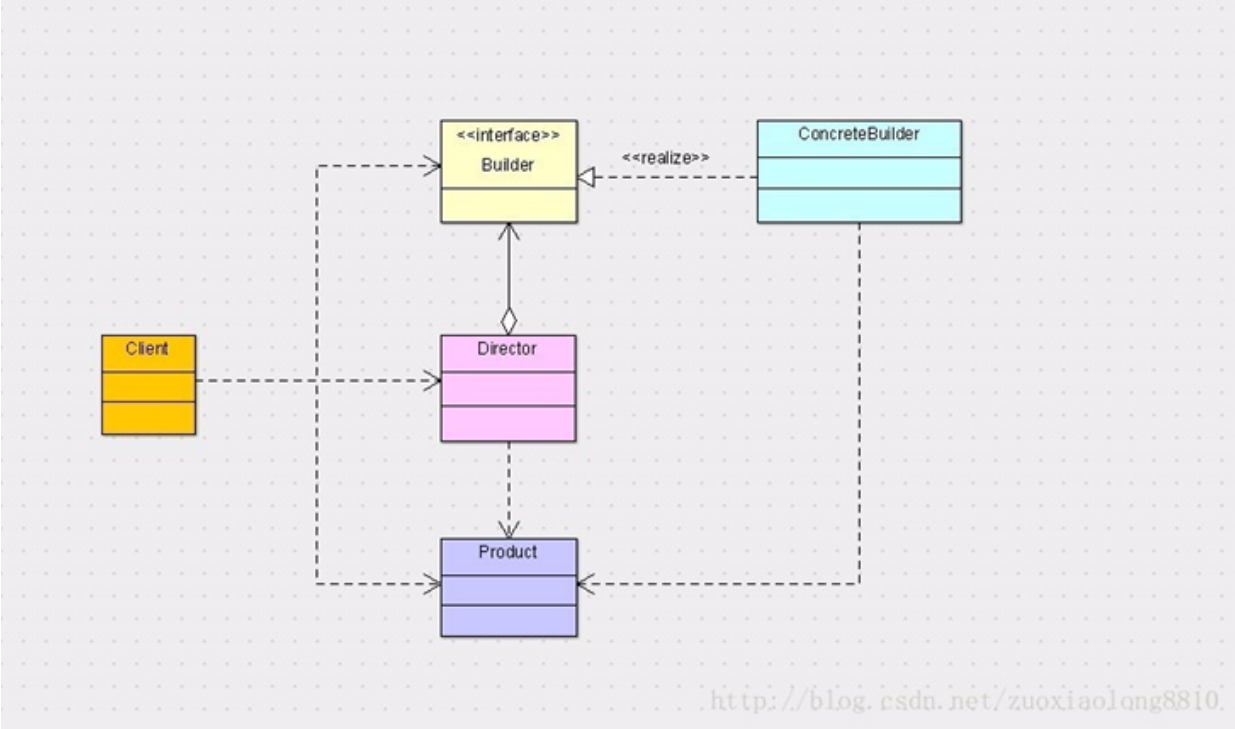
“看来还得求助于设计模式啊，还是让我来搜搜吧，度娘，老爹来啦。”

说着，小左便开始在度娘上专注的寻找起来，忽的，小左猛地一拍大腿，“有了，就它了。建造者模式，这名字太贴切了，来看看它的定义吧，哈哈！”

定义：将一个复杂对象的构造与它的表示分离，使同样的构建过程可以创建不同的表示，这样的设计模式被称为建造者模式。

“定义中说复杂的对象，不正是指的我的分身吗，那是相当复杂啊。看定义中的意思是，复用同样的构建过程，但是使用不同的表示。”说完小左露出一副思索的表情，随后缓缓的说道：“就我目前的情况来看，创建一个分身的过程就是构建过程，表示则应该指的是分身的形状了，比如瘦的，胖的。”

“不过这样还不够清晰，还是来看看建造者模式的类图吧。”



“建造者模式的类图并不复杂啊，不过看图中是有着四种角色，首先建造者接口（Builder）和具体的建造者（ConcreteBuilder）应该是要新建的，而指挥者（Director）毫无疑问就是我的灵魂了，产品（Product）则是分身。”

“哈哈，看着类图就是清晰了许多啊，分身类应该是不需要变的，不过作为指挥者的灵魂就要变一下了。现在还是先把建造者接口和具体的建造者写出来吧。”

```
package com.builder;

//分身制造者接口
```

```
public interface DoppelgangerBuilder {


      public void createDoppelganger(String name);

      public void buildBody();

      public void buildHead();

      public void buildLeftArm();

      public void buildLeftHand();

      public void buildRightArm();

      public void buildRightHand();

      public void buildLeftLeg();

      public void buildLeftFoot();

      public void buildRightLeg();

      public void buildRightFoot();

      public Doppelganger getDoppelganger();

}
```

“下面便是具体的建造者了，按照现在的情况，应该是两个具体的建造者，一个瘦的，一个胖的。”

```
package com.builder;

//瘦的分身的建造者

public class ThinBuilder implements DoppelgangerBuilder{
```

```
private Doppelganger doppelganger;

public void createDoppelganger(String name){
      doppelganger = new Doppelganger(name);
}

public Doppelganger getDoppelganger(){
      try {
            return doppelganger;
      }finally{
            doppelganger = null;
      }
}

public void buildBody() {
      System.out.println("设置" +
doppelganger.getName() + "的瘦瘦的身体");
      doppelganger.setBody("瘦瘦的身体");
}

public void buildHead() {
```

```
            System.out.println("设置" +
doppelganger.getName() + "的小小的头");

            doppelganger.setHead("小小的头");

      }



      public void buildLeftArm() {

            System.out.println("设置" +
doppelganger.getName() + "的瘦瘦的左胳膊");

            doppelganger.setLeftArm("瘦瘦的左胳膊");

      }



      public void buildLeftHand() {

            System.out.println("设置" +
doppelganger.getName() + "的瘦瘦的左手");

            doppelganger.setLeftHand("瘦瘦的左手");

      }



      public void buildRightArm() {

            System.out.println("设置" +
doppelganger.getName() + "的瘦瘦的右胳膊");

            doppelganger.setRightArm("瘦瘦的右胳膊");

      }
```

```
	public void buildRightHand() {

		System.out.println("设置" +
doppelganger.getName() + "的瘦瘦的右手");

		doppelganger.setRightHand("瘦瘦的右手");

	}



	public void buildLeftLeg() {

		System.out.println("设置" +
doppelganger.getName() + "的瘦瘦的左腿");

		doppelganger.setLeftLeg("瘦瘦的左腿");

	}



	public void buildLeftFoot() {

		System.out.println("设置" +
doppelganger.getName() + "的小小的左脚");

		doppelganger.setLeftFoot("小小的左脚");

	}



	public void buildRightLeg() {

		System.out.println("设置" +
doppelganger.getName() + "的瘦瘦的右腿");
```

```
		doppelganger.setRightLeg("瘦瘦的右腿");
	}


	public void buildRightFoot() {
		System.out.println("设置" +
doppelganger.getName() + "的小小的右脚");
		doppelganger.setRightFoot("小小的右脚");
	}



}


package com.builder;
//胖的分身的建造者
public class FatBuilder implements DoppelgangerBuilder{

	private Doppelganger doppelganger;

	public void createDoppelganger(String name){
		doppelganger = new Doppelganger(name);
	}
```

```
	public Doppelganger getDoppelganger(){
		try {
			return doppelganger;
		}finally{
			doppelganger = null;
		}
	}

	public void buildBody() {
		System.out.println("设置" + doppelganger.getName() + "的胖胖的身体");
		doppelganger.setBody("胖胖的身体");
	}

	public void buildHead() {
		System.out.println("设置" + doppelganger.getName() + "的大大的头");
		doppelganger.setHead("大大的头");
	}
```

```
	public void buildLeftArm() {

		System.out.println("设置" +
doppelganger.getName() + "的胖胖的左胳膊");

		doppelganger.setLeftArm("胖胖的左胳膊");

	}



	public void buildLeftHand() {

		System.out.println("设置" +
doppelganger.getName() + "的胖胖的左手");

		doppelganger.setLeftHand("胖胖的左手");

	}



	public void buildRightArm() {

		System.out.println("设置" +
doppelganger.getName() + "的胖胖的右胳膊");

		doppelganger.setRightArm("胖胖的右胳膊");

	}



	public void buildRightHand() {

		System.out.println("设置" +
doppelganger.getName() + "的胖胖的右手");

		doppelganger.setRightHand("胖胖的右手");
```

```
        }

        public void buildLeftLeg() {
                System.out.println("设置" +
doppelganger.getName() + "的胖胖的左腿");
                doppelganger.setLeftLeg("胖胖的左腿");
        }

        public void buildLeftFoot() {
                System.out.println("设置" +
doppelganger.getName() + "的大大的左脚");
                doppelganger.setLeftFoot("大大的左脚");
        }

        public void buildRightLeg() {
                System.out.println("设置" +
doppelganger.getName() + "的胖胖的右腿");
                doppelganger.setRightLeg("胖胖的右腿");
        }

        public void buildRightFoot() {
```

```
            System.out.println("设置" +
doppelganger.getName() + "的大大的右脚");

            doppelganger.setRightFoot("大大的右脚");

      }





}
```

“由于分身类不需要变化，下面就只剩一个指挥者了，也就是我的灵魂。”

```
package com.builder;

//灵魂类，指挥者

public class Soul {



      public Doppelganger
createDoppelganger(DoppelgangerBuilder
doppelgangerBuilder,String name){

            doppelgangerBuilder.createDoppelganger(name);

            doppelgangerBuilder.buildBody();

            doppelgangerBuilder.buildHead();

            doppelgangerBuilder.buildLeftArm();

            doppelgangerBuilder.buildLeftHand();

            doppelgangerBuilder.buildRightArm();
```

```
            doppelgangerBuilder.buildRightHand();

            doppelgangerBuilder.buildLeftLeg();

            doppelgangerBuilder.buildLeftFoot();

            doppelgangerBuilder.buildRightLeg();

            doppelgangerBuilder.buildRightFoot();

            return doppelgangerBuilder.getDoppelganger();

      }



}
```

“这下我的灵魂只管指挥就可以了，不过看样子，我是要牢牢记住分身建造的过程了，看来这指挥者果然不是好当的啊。下面让我看看制造分身的过程吧。”

```
package com.builder;

//制造过程

public class Creator {



      public static void main(String[] args) {

            Soul soul = new Soul();

            //制造一个瘦小的看守所的人员

            System.out.println("获得了" +
soul.createDoppelganger(new ThinBuilder(),"看守所人员1"));
```

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



            //制造一个胖胖的看守所的人员

            System.out.println("获得了" +
soul.createDoppelganger(new FatBuilder(),"看守所人员2"));

      }

}
```

“哈哈，这下明显快多了，不过两个具体的建造者有重复的代码，倒是可以抽出一个抽象的模板类去表示这些共性。不过现在还是先算了吧。”

“还是老样子，让我来对比下前后的利弊吧，这样好让我以后方便的造人。”

首先便是先前的坏处了，应该有以下两点。

1、制造分身的非常复杂，每次都要现场制造，容易丢三落四，缺胳膊少腿的。

2、分身的样子是可以变化的，所以假设我要重复制造几个瘦瘦的家伙的话，那就不得不每次都将整个建造过程和整个身体的部分全部重来一遍了。

如果换成专业点的语言来总结的话，则是下面两点。

1、对象的构建过程非常复杂，每次客户端调用的时候，都要自己控制构建过程，这会导致构建过程的重复，而且非常容易忘记某一个步骤。

2、对象的表示是可以变化的，所以如果要重复制造几个表示相同的对象的时候，只能每次都将过程和表示重复一遍。

那么现在改变了方式之后的好处则是下面两点。

1、分身制造的过程被灵魂专门记忆和控制，所以不会再出现丢三落四的情况。

2、由于分身的样子由固定的分身制造者提供，所以如果要重复制造相同样子的分身，只需要让灵魂控制过程再重来一次即可。

而好处的话，如果换成专业点的语言来总结的话，则是下面两点。

1、最主要的好处是，使得对象的构建与表示分离，可以让表示独立于过程而独立的变化，并且客户端不再需要关心具体的构建过程和表示。

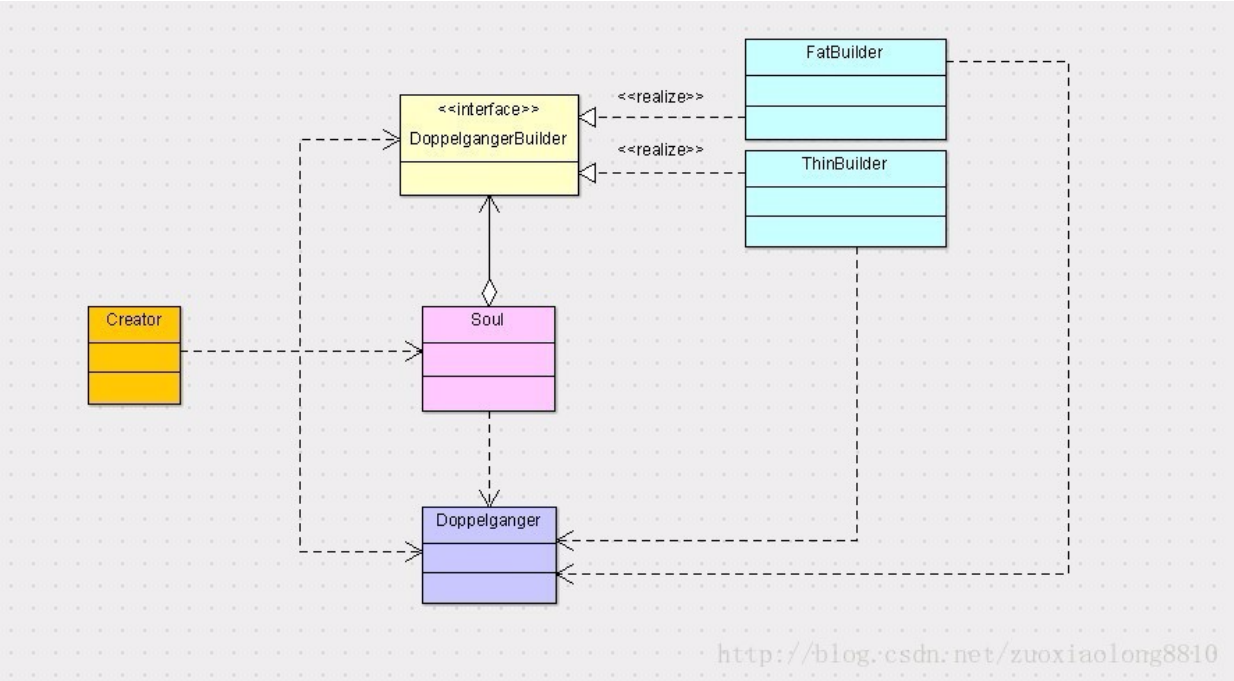
2、由于固定的表示由具体的制造者提供，所以要重复制造几个表示相同的对象的时候，只需要让指挥者控制制造者再进行一次即可。

“看来这个建造者模式很适合我制造分身的情况啊，那么总结下它适用的场景吧。它适用的场景应该是这样的。”

1、当对象的构建过程十分复杂，需要与表示分离的时候。

2、当对象的表示有不同种类，需要加以区分的时候。

“最后还是让我看看现在的类图吧，哈哈。”解决了问题的小左此时心情大好，大笑着说道。



“看起来和建造者模式的类图倒是几乎一模一样，这下我得好好的练习一下建造者模式的方法了。”

说着，小左便将自己的高富帅和屌丝分身再次分离了出来，如今看来，分离这两个分身实在比制造分身要快多了。接下来，小左的灵魂便开始一门心思的记忆组装过程，而两个分身，则一个练习制造各种瘦分身的零件，另外一个练习制造各种胖分身的零件。

伴随着一魂两分身不停的练习，时间也是悄然而逝，转眼间，已是第二天深夜。

魔都某看守所。

“李刚，里面那小妞那么好看，你说我们俩大老爷们这么一直干看着，也太折磨人了吧。”

“你这死双江，做了人民警察还是改不了你那副德行，小心被头听到把你也送进去。”李刚听完忽的一拍李双江后脑壳，奸笑的说道。

“靠！你敢袭警，你小子才是想进局子了吧...”李双江笑骂的同时，右手顺便也在李刚的脑壳上拍了一记。

“你敢打我？咦？头好晕...”不停的摇着头，李刚的眼神开始渐渐迷离。

正当李双江准备表示自己不会上当时，只见李刚嘭的一声便直直的摔倒在地，而李双江刚到嘴边的话也咽了回去，战战兢兢的拍了拍李刚，然而李刚此时却毫无反应，李双江这下彻底吓尿了，颤颤巍巍的说道：“嘿...你吓我呢吧...死了？”

（十六）备忘录模式详解（都市异能版）

凌晨两点，魔都某看守所六号牢房。

“叮咣...”

一道开门声陡然响起，牢房中的韩雨露心中一紧，明亮的眼眸看着来人，不过眼神之中的一丝惊恐却是掩饰不去。

来人正是李刚和李双江，只是此时二人眼中无神，如果仔细观察，可以看出这两个人的动作十分蹩脚，只是韩雨露此时并未注意这些细节，而是犹如等待宣判一般，惶恐不安的等待着二人说话。

“韩小姐，恭喜你，你可以出去了。”脸上勉强挤出一丝僵硬的笑容，李刚向前一步说道。

好消息从天而降，韩雨露不禁愣了一瞬间，不过很快就反应过来，诱人的小嘴没有发出声音，只是微笑的点了点头。

“肯定是爸爸救了我。”没有任何怀疑，韩雨露心中暗暗的想到。

两男一女走在看守所的道路上，途中偶尔碰到的工作人员，像是商量好了似的，全部低头不语，仿佛什么事都未发生一般，如此奇怪的场景，实在让韩雨露一时之间摸不着头脑。

按照常理来说，这些人看到有人被释放，应该不会如此安静，而且韩雨露清楚的记得当初自己被带到看守所时，眼前的二人全部都是一脸猥琐的表情盯着自己的两个山丘，而此时，二人一直走在自己前方，完全对自己视若无睹。

这倒不是韩雨露天性开放，而是换做任何人，想必都会对身边的人陡然的变化产生疑虑，况且一路上又是如此诡异，这更是让韩雨露慌乱之中，已然在不知不觉间香汗淋漓。

一路从牢房到看守所门口，徒步也仅有不到十分钟的路程，而韩雨露却觉得走的特别慢，好不容易来到了看守所门口，却是空无一人。

“爸爸为什么没来接我？”看到眼前的场景，韩雨露心中不自觉的升起一丝疑问。

“韩小姐，我们给你拦一辆出租车，你打车回去吧。”此时说话的已经换成了李双江，二人皆是面无表情的看着眼前的伊人。

魔都的出租车深夜并不好拦，不过好在运气还不错，没过多久，一辆出租车便被李双江拦下，待得韩雨露坐进去以后，李双江便扔给了司机两张毛爷爷，僵硬的说了句：“将这位小姐送到家里。”

李双江说话的同时，还专门钻到车窗里看了看司机的职照。李刚也没闲着，专门走到出租车后面，看着车牌号轻声默念了几遍，才慢慢的走回李双江身边，到此李双江才朝司机摆了摆手，示意司机出发。

韩雨露一直看着二人对自己似乎特别关心的样子，心中已经疑问满天，却是不敢多问。不过韩雨露不知道的是，当出租车刚驶出几十米之后，整个看守所的工作人员瞬间全部消失不见，犹如人间蒸发了一般，场面极其诡异。

魔都某出租屋。

“人算不如天算，天算不如不算。老子计划的这么周密，竟然还是被人给看到了。”说话之人正是小左，只不过此时的小左脸色十分不好。

经过一系列的计划之后，原本小左是打算放出韩雨露之后，自己隐身跟着韩雨露，一来可以确保韩雨露的安全，二来也可以看看韩雨露的家在何处。可是计划实施的过程中出现了一些变故，导致小左不得不让韩雨露自己做出租车回去。

原来小左在行动之前，花费了整整两个小时，才找到了一家出售蒙汗药的隐秘药店，随后小左便借助着飞行和隐身的异能，偷偷在看守所

所有的饮水机里放入了大量的蒙汗药。待得所有工作人员都昏睡之后，小左才使用飞行将众人一一带回自己的出租屋。

意外就在小左带走最后两个工作人员的时候发生了，三人正在空中飞行，小左意外的发现其中一人竟然已经醒了，出乎意料的是这人醒来之后，便被高空的刺激吓得大喊大叫，导致另外一人也在途中醒来，最后还是小左在空中威胁二人，让二人自己服下剩余的蒙汗药，才制止了二人凄厉的叫喊声。

无奈之下，小左放下二人之后，便匆匆忙忙的回到看守所，凭借着之前的分身制造之法，快速的制造了一堆看守所人员的分身之后，小左就立即马不停蹄的回到了住处。小左此时已经打算在住处控制分身，他必须看着昏睡的众人，以免这坑爹的假药再闹出什么意外。

虽说距离较远的时候，控制诸多分身有些勉强，但小左还是将韩雨露顺利的放了出来，而现在小左要处理的，则是最棘手的问题，那就是自己的能力已经暴露了。

尽管小左此时完全可以干掉知情的二人，但小左毕竟不是心狠手辣之辈，如此狠毒的事情，小左自问还是做不到的。

平复了一下烦躁的心情，遇到问题的小左还是不自觉的想到了设计模式，仿佛设计模式就是现实中的问题大全，可以一解千愁一般。事实也确实如此，凭借着设计模式的理论，小左已经发掘了很多异能的特殊用法，甚至领悟新的异能，那种感觉，就犹如修炼一本内功心法一般。

“度娘啊，GOF啊，你们可要帮助下老爹啊，实在不行，我只能使出隐身恐吓一下这两个家伙了，只是这样始终难以让人安心啊。”说话的同时，小左已经开始在度娘上快速的寻找起来。

“咦？这个备忘录模式似乎和我遇到的问题有一些关系，只是为啥没有遗忘模式。我擦！”尽管有些郁闷，小左还是死死的盯着备忘录模式的定义，试图找到一点灵感。

定义：在不破坏封闭的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态。这样以后就可将该对象恢复到原先保存的状

态。

“定义里说可以捕获一个对象的内部状态，在对象之外保存之后，就可以恢复该对象保存的状态。看着这个定义，总觉得似乎是和我现在的问题有点关联，可是到底应该如何使用呢？”似有所悟的看着备忘录模式的定义，小左抓着头发喃喃的说道。

“哎！管他呢，先把我现在的问题试图描述一下再说，这样比较好找出解决办法。”说着，小左便开始敲打键盘。

“如果是现在来说的话，应该只有一个类，就是表示人类，而人类应该有一个故事列表，都是从小到大的记忆。”边敲打着键盘，小左的口中依然是喋喋不休。

```
package com.memento;


import java.util.ArrayList;

import java.util.Collections;

import java.util.List;


public class Person {


	private String name;


	private List<String> storyList;//故事列表


	public Person(String name){
```

```
        this.name = name;
        storyList = new ArrayList<String>();
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
          //故事列表不能修改
    public List<String> getStoryList() {
        return Collections.unmodifiableList(storyList);
    }
          //可以添加，但不可以删除
    public void addStory(String history) {
        this.storyList.add(history);
    }
```

```
	public String toString(){
		StringBuffer stringBuffer = new StringBuffer(name + "的记忆：\n");
		for(String memory : storyList){
stringBuffer.append(memory).append("\n");
		}
		return stringBuffer.toString();
	}
}
```

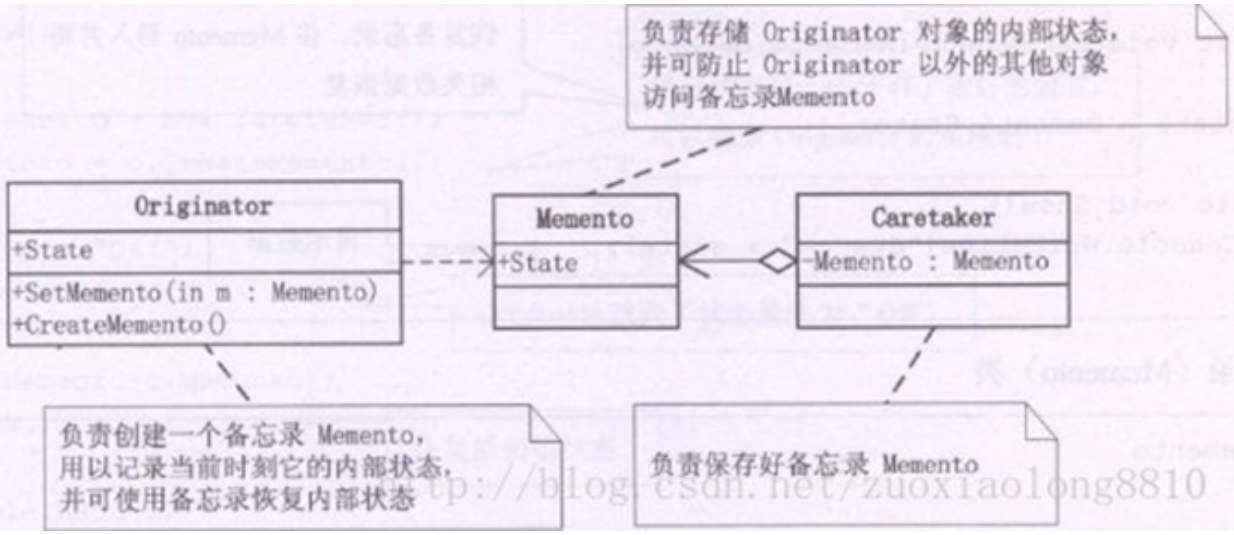
“这样的话，现在的情况就是两个看守所的人员都记住了我可以飞行这件事，而且还都看到了我的长相。”

```
package com.memento;


public class Client {


	public static void main(String[] args) {
		Person personA = new Person("看守所人员A");
		Person personB = new Person("看守所人员B");
		//看守所人员A的记忆
		personA.addStory("小时候偷换男女厕所的牌子");
		personA.addStory("用裤衩上的猴皮筋做成弹弓打别人家玻璃，结果被弹了小JJ");
```

```
            //看守所人员B的记忆

            personB.addStory("小时候比谁尿的更高更远，我得了第一，只是不小心方向变成了直上直下，尿到了自己脸上");

            personB.addStory("有一次正上课，不小心放了个屁，结果带出了点杂物，满教室都是臭味熏天");



            //两个看守所人员一起经历的事

            personA.addStory("有一个人竟然可以飞行，记忆中还记得他的样貌");

            personB.addStory("有一个人竟然可以飞行，记忆中还记得他的样貌");



            System.out.println(personA);

            System.out.println(personB);

        }



}
```

“这个便是目前的现状了，这两个人全部记住了我的能力和长相。如果想想刚才备忘录模式的定义，说是可以将一个对象恢复到保存的状态，那么放在这里，应该是要将一个人的记忆恢复到某一个时刻把。原来如此啊，哈哈。”瞬间顿悟的小左一时间像打了鸡血似的，完全没有了刚才的一脸愁容。

“让我来看看备忘录模式的类图把，估计看完类图，解决问题的办法自然就浮现出来了。”说着小左便开始寻找度娘上备忘录模式的类图。



“看这个类图中，有着三个角色，一个便是发起人（Originator），它不仅有自己的状态，而且还负责创建一个备忘录（Memento）以及使用备忘录恢复自己的状态，而备忘录则记录了发起人的状态，至于最后一个角色管理者（Caretaker），则是用来管理备忘录的。”看着备忘录模式的类图，小左已经开始分析了起来。

“这里发起人自然便是人类（Person）了，只不过现在人类需要改变一下了，它需要提供创建一个备忘录以及使用备忘录恢复状态的操作，而备忘录和管理者都是需要新建的。只不过这里的管理者，应该就是我的灵魂了。”说道这里，小左的思路已经渐渐清晰起来，手下已经开始了代码的编写。

“先写一个备忘录的类吧，这个类如果形象的来称呼的话，应该叫记忆类。”

```
package com.memento;



import java.util.List;
```

```
//记忆类（备忘录）
public class Memory {

	private List<String> storyList;

	public List<String> getStoryList() {
		return storyList;
	}

	public void setStoryList(List<String> storyList) {
		this.storyList = storyList;
	}

}
```

“再来便是人（Person）类的更改了，应该是这样的。”

```
package com.memento;

import java.util.ArrayList;
import java.util.Collections;
```

```
import java.util.List;

public class Person {

	private String name;

	private List<String> storyList;

	public Person(String name){
		this.name = name;
		storyList = new ArrayList<String>();
	}

	public String getName() {
		return name;
	}

	public void setName(String name) {
		this.name = name;
	}
```

```
	public List<String> getStoryList() {
		return Collections.unmodifiableList(storyList);
	}

	public void addStory(String history) {
		this.storyList.add(history);
	}

	/* 新增 */

	//获取记忆
	public Memory getMemory(){
		Memory memory = new Memory();
		memory.setStoryList(new ArrayList<String>
(storyList));
		return memory;
	}

	//恢复记忆
	public void setMemory(Memory memory){
```

```
            storyList = memory.getStoryList();
        }

        /* 新增 */

        public String toString(){
            StringBuffer stringBuffer = new StringBuffer(name
+ "的记忆：\n");
            for(String memory : storyList){
stringBuffer.append(memory).append("\n");
            }
            return stringBuffer.toString();
        }
}
```

“下面便是我的灵魂类了，它是一个记忆的管理者，可以搜集人的记忆，也可以将人的记忆强行固定或者说恢复到某一个阶段。”

```
package com.memento;

public class Soul {

        private Memory memory;
```

```
        //抽离或者说搜集一个人的记忆
        public void pullAwayMemory(Person person){
                memory = person.getMemory();
        }

        //强行将一个人的记忆固定在某一刻
        public void forceFixMemory(Person person){
                person.setMemory(memory);
        }
}
```

“现在就让我试试备忘录模式的威力吧，哈哈。”小左得瑟的说道。

```
package com.memento;

public class Client {

        public static void main(String[] args) {
                Soul soulA = new Soul();
                Soul soulB = new Soul();
                Person personA = new Person("看守所人员A");
```

```
Person personB = new Person("看守所人员B");

//看守所人员A的记忆

personA.addStory("小时候偷换男女厕所的牌子");

personA.addStory("用裤衩上的猴皮筋做成弹弓打别人家玻璃，结果被弹了小JJ");


//看守所人员B的记忆

personB.addStory("小时候比谁尿的更高更远，我得了第一，只是不小心方向变成了直上直下，尿到了自己脸上");

personB.addStory("有一次正上课，不小心放了个屁，结果带出了点杂物，满教室都是臭味熏天");


System.out.println(personA);

System.out.println(personB);

//两个灵魂（管理者）抽离两个人之前的记忆

soulA.pullAwayMemory(personA);

soulB.pullAwayMemory(personB);

//两个人看到了小左的技能和面貌

personA.addStory("有一个人竟然可以飞行，记忆中还记得他的样貌");

personB.addStory("有一个人竟然可以飞行，记忆中还记得他的样貌");
```

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

            System.out.println(personB);

            //强行恢复记忆

            soulA.forceFixMemory(personA);

            soulB.forceFixMemory(personB);


            System.out.println(personA);

            System.out.println(personB);

        }


}
```

“恩，备忘录模式应该叫遗忘录模式了，哈哈。这么一来，这两个人的记忆就恢复到原先没看到我之前的记忆了。”看着输出的结果，小左不禁更加得意，不过瞬间小左就脸色一变，“咦？上面使用的时候怎么出来两个灵魂？”

顿了一下，小左继续自言自语道：“这不行啊，与现实不符啊。”挠了挠头，小左继续说道：“在备忘录模式的标准实现里，一个管理者只管理了一个备忘录，所以要备份两个人的记忆的时候，就有些麻烦了。看来我还需要再稍微改一下。”说着，小左便开始匆忙的修改起来。

```
package com.memento;


import java.util.HashMap;
```

```
import java.util.Map;


public class Soul {


	private Map<Person, Memory> memoryMap = new
HashMap<Person, Memory>();


	//抽离或者说搜集一个人的记忆

	public void pullAwayMemory(Person person){

		memoryMap.put(person, person.getMemory());

	}


	//强行将一个人的记忆固定在某一刻

	public void forceFixMemory(Person person){

		if (memoryMap.containsKey(person)) {

person.setMemory(memoryMap.get(person));

		}

	}

}
```

“让我的灵魂可以搜集多个人的记忆，应该就可以解决问题了。这下再试一下吧。”

```
package com.memento;


public class Client {


	public static void main(String[] args) {
		Soul soul = new Soul();
		Person personA = new Person("看守所人员A");
		Person personB = new Person("看守所人员B");
		//看守所人员A的记忆
		personA.addStory("小时候偷换男女厕所的牌子");
		personA.addStory("用裤衩上的猴皮筋做成弹弓打别人家玻璃，结果被弹了小JJ");


		//看守所人员B的记忆
		personB.addStory("小时候比谁尿的更高更远，我得了第一，只是不小心方向变成了直上直下，尿到了自己脸上");
		personB.addStory("有一次正上课，不小心放了个屁，结果带出了点杂物，满教室都是臭味熏天");


		System.out.println(personA);
		System.out.println(personB);
		//灵魂（管理者）抽离两个人之前的记忆
```

```
        soul.pullAwayMemory(personA);

        soul.pullAwayMemory(personB);

        //两个人看到了小左的技能和面貌

        personA.addStory("有一个人竟然可以飞行，记忆中还记得他的样貌");

        personB.addStory("有一个人竟然可以飞行，记忆中还记得他的样貌");


        System.out.println(personA);

        System.out.println(personB);

        //强行恢复记忆

        soul.forceFixMemory(personA);

        soul.forceFixMemory(personB);


        System.out.println(personA);

        System.out.println(personB);

    }


}
```

“现在的效果应该和刚才一样，不过是灵魂稍微改变了一下而已。不过看起来仍然是有些问题啊，这样的使用方式，我必须得在两个人没有看到我的时候抽离两个人的记忆，然后再恢复回去，可惜现在已经晚了。”突然意识到了另外一个问题，小左已然有些郁闷了。

“还是让我先来总结一下备忘录模式的特点吧，看能不能找到其它的办法。”一时之间想不起来对策，小左决定先总结一下备忘录模式的特点。

对于现在的情况来说，备忘录模式应该有以下几点好处。

1、人类可以将自己的记忆保存在外部，使得备份的功能不会影响人类本身，而人类则只需要提供获取记忆和恢复记忆的方法。

2、对于人类记忆的备份与恢复，人类自身不需要关心，而是使用灵魂的人关心，在这里，就是指小左本人了。

3、如果一个人的记忆丧失或出现问题，可以很轻松的恢复到原来的记忆。

如果换成专业点的语言来说的话，则应该是以下几点了。

1、发起人备份自己的状态不需要自己管理，可以备份到外部，这样可以很好的保持封装的边界，这样做的意义在于可以给外部提供一个简单的操作该对象内部状态的接口。保持封装的边界这应该算是最重要的优点了。

2、发起人状态的备份与恢复，发起人自身不需要管理与操作，而是由客户端自行按需要处理。

3、如果发起人的状态出现问题，可以很轻松的恢复。

参照刚才备忘录模式的使用，备忘录模式显然是有缺点的，应该有以下几点。

1、如果一个人岁数比较大，记忆比较多的话，全部备份会让储存记忆的灵魂压力很大。

2、由于记忆是由人类自己提供给灵魂的，所以灵魂在储存记忆的时候可能一股脑储存，无法预测一个人的记忆是否庞大。

3、如果恢复人的记忆成功率较低的话，使用备忘录模式可能会导致恢复失败，这样的话可能会使人的记忆处于不正确的状态，比如缺失或者记忆混乱。

这些缺点如果换成专业点的语言来说的话，则是以下几点。

1、如果全部备份发起人的状态，或者其中有占用内存较大的属性（比如一个长数组），则会让备忘录模式的使用代价昂贵。

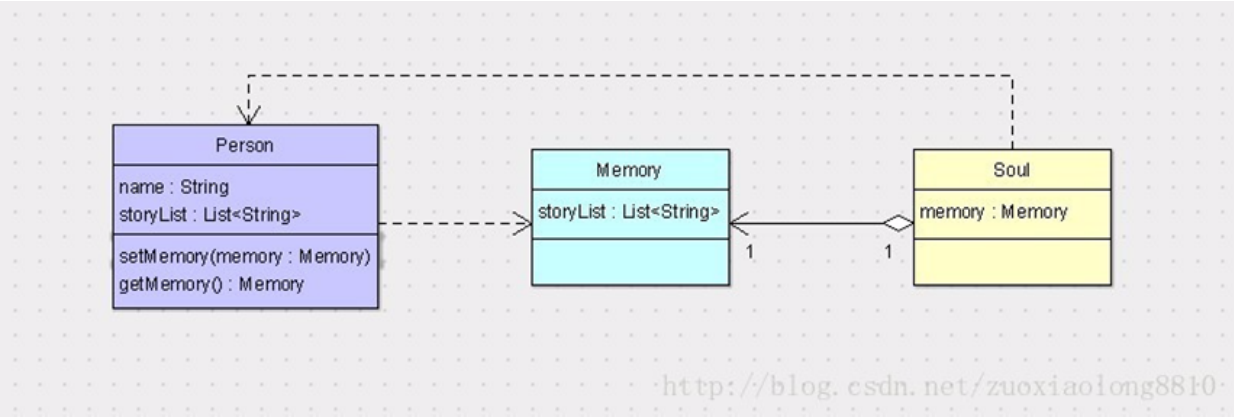
2、由于备份的信息是由发起人自己提供的，所以管理者无法预知备份的信息的大小，所以在客户端使用时，可能一个操作占用了很大的内存，但客户端并不知晓。

3、当发起人的状态改变的时候，有可能这个状态无效。如果状态改变的成功率不高的话，可以采取“假如”协议模式。“假如”的意思是指，我们将一直假如状态的改变会失败，从而对此做出一系列准备的工作。不过很明显，如果状态改变的成功率很高，则这样做的收益甚微。

“感觉备忘录模式最大的缺点就是备份的信息所占用的资源可能会很庞大，这直接导致的就是我的灵魂压力颇大啊。不过还有一点，就是在使用备忘录模式的时候，必须要能关注一个对象整个状态历史，才可以使用。就是因为这个，所以现在无法将两个人的记忆恢复到见到我之前了，因为我根本没有备份他们之前的记忆啊。”

想到这个，小左不禁郁闷非常，不过片刻之后，小左还是迅速的调整好了心态，似有所悟的说道：“不过这倒算不上缺点，只能说是使用备忘录模式的前提。”

“还是看一下类图把，看看有没有什么变通的方式。”



“现在的类图与标准的备忘录模式有些出入，一个是其中一对一的聚合关系已经被我改成了一对多，还有一个就是灵魂类对人类的依赖关系，这都是我针对现实状况对设计模式做出的调整。尽管已经对现实情况做出了相应的调整，不过关于两个人记忆备份的问题依旧是看不出来什么解决方案啊。”

略微一顿，小左叹了口气，继续暗暗的摇头道：“看来是没有办法了，只能退而求其次。虽然你们会忘掉所有的过往，不过我会想办法弥补你们的。”

魔都某看守所，所长办公室。

“你们两个怎么回事，不想活了是吧。”一个满脸胡茬的中年人，此时正冲着面前的二人大发雷霆。

“所长，我们没有私自释放韩小姐啊。实在是冤枉啊...”说话的正是李刚，只不过此时，李刚与身边的李双江都是一脸憋屈的表情。

“放NMD狗臭屁！这监控录像里就是你们两个把人给放走了，你现在跟我说你们没有？”听到李刚的无力辩解，中年人更加怒气冲天，大爆粗口的骂道。

就在此时，一个身材瘦小的青年突然闯了进来，中年所长一看便认出了此人，正是看守所里的工作人员小赵，看到小赵不打招呼就硬闯进来，正在气头上的中年所长自然是怒气更甚，愤怒的吼道：“你这个家伙，什么事不打招呼就进来了。先给老子滚出去。”

“你说话声音这么大干嘛，吵着人家了。傻X，嘿嘿...”小赵像是不知道中年所长的愤怒一般，傻笑着说道。

没有料到会出现这样的情况，中年所长顿时一脸铁青，指着李刚和李双江二人，面目狰狞的说道：“去！给我拖出去狠狠的打，一会回来再跟你们俩算账。NND，我看你们是都不想活了。”

（十七）迭代器模式详解（foreach的精髓）

各位好，很久没以LZ的身份和各位对话了，前段时间为了更加逼真的解释设计模式，LZ费尽心思给设计模式加入了故事情节，本意是为了让各位在看小说的过程中就可以接触到设计模式，不过写到现在，LZ最深的感触就是，构思故事的时间远远超过了LZ对设计模式本身的研究。

本章介绍迭代器模式，不再采用故事嵌入的讲解方式，主要原因是因为迭代器模式本身有更多需要介绍的东西，如果嵌入到小说当中，会不太方便去阐述这些内容。

另外，由于LZ的大部分设计模式文章主要针对的人群是对设计模式已经有一定了解，希望更加深入理解的程序猿们。所以LZ希望各位在看本篇文章时，可以打开一个初步介绍迭代器模式的文章，对比观看。下面进入正题，我们先来看看百度百科或者说GOF对迭代器模式的定义。

定义：提供一种方法顺序访问一个聚合对象中各个元素，而又不需暴露该对象的内部表示。

从定义中可以看出，迭代器模式是为了在不暴露该对象内部表示的情况下，提供一种顺序访问聚合对象中元素的方法。这种思想在JAVA集合框架中已经体现的淋漓尽致，而且LZ相信每一个接触JAVA的同学都难免要去触碰。

所以LZ这次先不给出迭代器的类图与标准实现，我们先来看看迭代器模式解决了JAVA集合框架中的哪些问题。

为了更加清晰，LZ斗胆写了几个简单的集合类（向JDK类库的缔造者致敬），我们从这几个简单的集合类出发，去仔细体会下定义的意思，下面是LZ分别写的缩小版的ArrayList、LinkedList和HashSet。

```
package com.iterator;

public class ArrayList<E> {

	private static final int INCREMENT = 10;

	private E[] array = (E[]) new Object[10];

	private int size;

	public void add(E e){
		if (size < array.length) {
			array[size++] = e;
		}else {
			E[] copy = (E[]) new Object[array.length +
INCREMENT];
			System.arraycopy(array, 0, copy, 0, size);
			copy[size++] = e;
			array = copy;
		}
	}
```

```
	public Object[] toArray(){
		Object[] copy = new Object[size];
		System.arraycopy(array, 0, copy, 0, size);
		return copy;
	}

	public int size(){
		return size;
	}
}

package com.iterator;

public class LinkedList<E> {

	private Entry<E> header = new Entry<E>(null, null,
null);
	private int size;

	public LinkedList() {
```

```
        header.next = header.previous = header;
    }

    public void add(E e){
        Entry<E> newEntry = new Entry<E>(e, header,
header.next);
        newEntry.previous.next = newEntry;
        newEntry.next.previous = newEntry;
        size++;
    }

    public int size(){
        return size;
    }

    public Object[] toArray(){
        Object[] result = new Object[size];
        int i = size - 1;
        for (Entry<E> e = header.next; e != header; e =
e.next)
            result[i--] = e.value;
```

```
            return result;
        }

        private static class Entry<E>{
            E value;
            Entry<E> previous;
            Entry<E> next;
            public Entry(E value, Entry<E> previous, Entry<E>
next) {
                super();
                this.value = value;
                this.previous = previous;
                this.next = next;
            }
        }

}
```

```
package com.iterator;
```

```
import java.util.HashMap;
import java.util.Map;

public class HashSet<E> {

	private static final Object NULL = new Object();

	private Map<E, Object> map = new HashMap<E, Object>();

	public void add(E e){
		map.put(e, NULL);
	}

	public int size(){
		return map.size();
	}

  public Object[] toArray(){
		return map.keySet().toArray();
	}
```

```
}
```

下面我们看看三个类的遍历方式。

```
package com.iterator;



public class Main {



      public static void main(String[] args) {

            ArrayList<Integer> arrayList = new
ArrayList<Integer>();

            for (int i = 1; i <= 11; i++) {

                  arrayList.add(i);

            }

            System.out.println("arrayList size:" +
arrayList.size());

            Object[] arrayListArray = arrayList.toArray();

            for (int i = 0; i < arrayListArray.length; i++) {

                  System.out.println(arrayListArray[i]);

            }



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

```
            HashSet<Integer> hashSet = new HashSet<Integer>
();

            for (int i = 1; i <= 11; i++) {

                  hashSet.add(i);

            }

            System.out.println("hashSet size:" +
hashSet.size());

            Object[] setArray = hashSet.toArray();

            for (int i = 0; i < setArray.length; i++) {

                  System.out.println(setArray[i]);

            }



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



            LinkedList<Integer> linkedList = new
LinkedList<Integer>();

            for (int i = 1; i <= 11; i++) {

                  linkedList.add(i);

            }

            System.out.println("linkedList size:" +
linkedList.size());

            Object[] linkedListArray = linkedList.toArray();
```

```
            for (int i = 0; i < linkedListArray.length; i++)
{
System.out.println(linkedListArray[i]);
            }
      }
}
```

结果在这里LZ就不贴了，只是简单的输出三次1到11。

各位思考一下，我们这里的遍历是如何做到的。很明显，我们是通过一个通用的方法，toArray做到的。当然，为了迎合面向接口的思想，你可以添加一个接口规定toArray的行为，让三个类去实现它。

但是在这里有一个很大的弊端，不知道各位注意到没有，那就是不论我们的集合类是如何实现的（比如链表，数组，散列），在使用数组遍历集合类的时候，我们其实遍历了两次。

在这三个类中，由于System的arraycopy和set的toArray方法是黑箱子，所以最明显的便是LinkedList的实现，它是先遍历了一遍链表，做出来一个数组，然后当客户端获得到这个数组的时候，则需要再来一次循环，去遍历每一个元素。

为何会是这种情况呢？

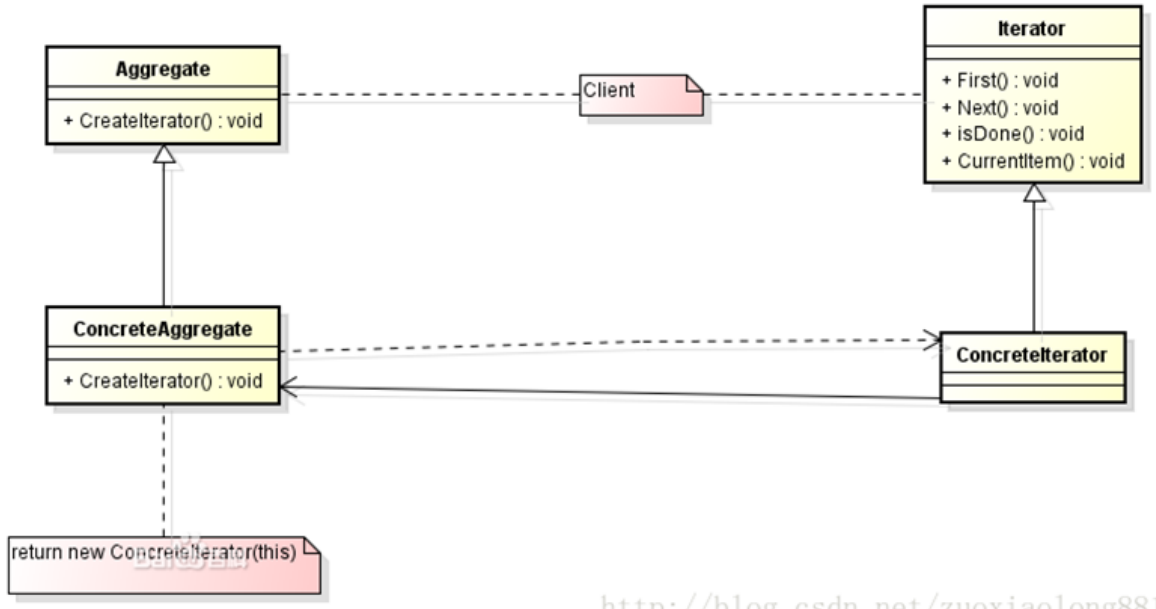
很简单，因为我们的集合类本身就不是一个数组，所以自然要多一步从集合类到数组的过渡。哪怕是本身由数组实现的ArrayList，也避免不了多这一步，各位可以试一下在ArrayList中直接返回array属性，结果中会出现一堆null值，而且这样做的话，对于array的改变会直接影响到ArrayList本身，这并不是我们所希望看到的，所以我们返回的只是一个拷贝。

当然，为了解决这个问题，我们并不是没有办法，比如给LinkedList和ArrayList加入get方法，而这个方法有一个参数index，这是我们常用的遍历方法。如此一来，便解决了二次遍历的问题。

但是问题又来了，那就是我们无法给HashSet提供一个根据索引获取元素的方法，由于散列特性的缘故，set中的元素是无序的，或者说顺序是不被保证的。那么这个get方法，在HashSet中便无法提供，因为这里没有我们通俗意义上的索引的概念。

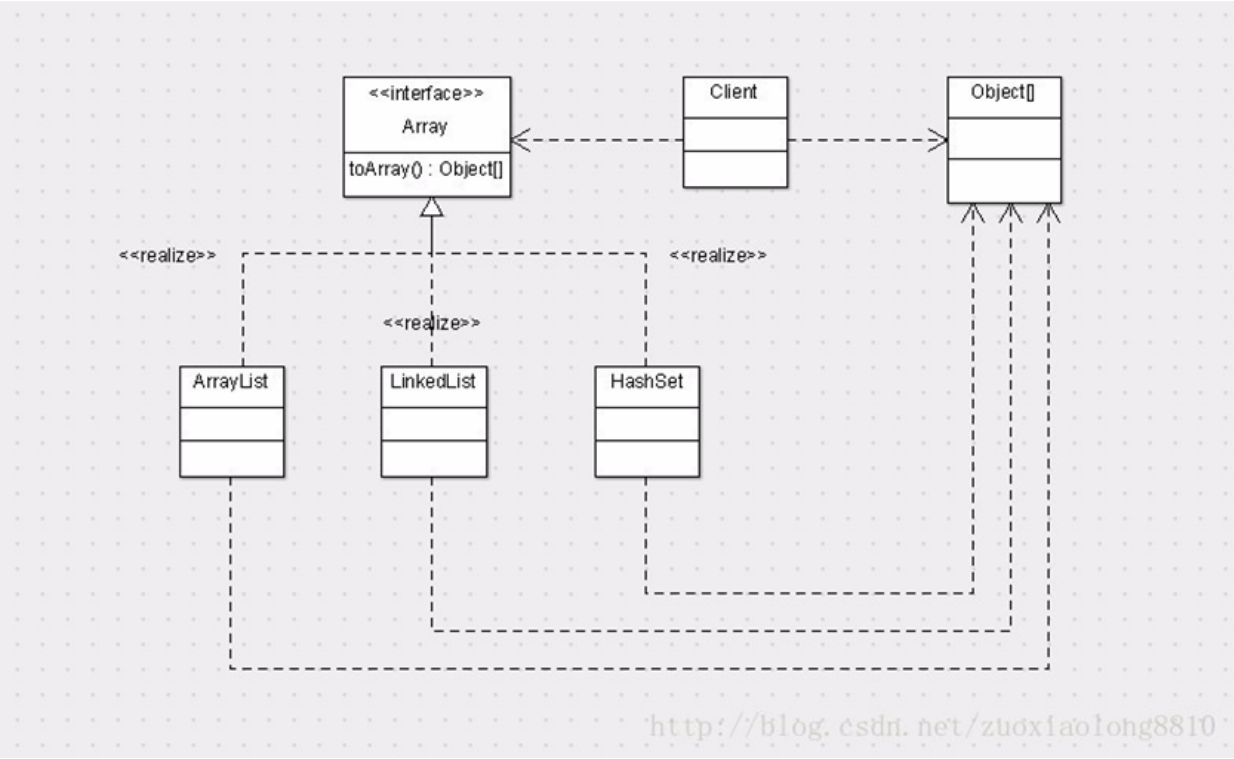
可以看到，上面LZ粗浅的分析，得出一个结论。三个集合类，如果统一提供数组给客户端遍历，那么在遍历过程中会出现重复遍历的现象。而如果消除这种重复遍历，则由于内部数据结构的不同，三个集合类无法做到像提供数组一样，给客户端提供统一的遍历方式。

为了解决上面的问题，迭代器模式就随之出现了。我们先来看看迭代器模式在百度百科中的类图，稍后各位可以自己体会下，迭代器模式是否解决了上面的问题，以及是否提供了额外的一些好处。



http://blog.csdn.net/zuoxiaolong8810

看着上面的类图，我们可以分析出来，上面我们所写的ArrayList等三个类都属于ConcreteAggregate的位置。如果我们刚才设计一个数组接口让三个类去实现的话，其实已经和迭代器模式十分相似了。他们的类图会是下面这样的。



在上述类图中，我们从面向对象的角度思考，将Object[]当做一个对象对待，我们对比下两个类图，他们其实是非常相似的，其中最大的区别在于，第二个类图当中，没有抽象数组接口这个概念，而在迭代器模式的类图中，是有迭代器接口这个概念的。

上述区别最终所造成的结果就是，由于数组是以固定的排列方式存在的，即数组必须是一组连续的内存区域（逻辑上连续），故而以数组为基础的遍历方式只能是按照索引遍历。而迭代器则不限制，我们注意到，在迭代器模式的类图中，具体的迭代器是有一条到具体聚合对象的关联线的，这就意味着迭代器的实现是与具体的聚合对象息息相关的，也就是说迭代器满足了多种迭代方式。

好了，截止到目前，我们前面所讨论的都是为何要使用迭代器模式，或者说迭代器模式解决了哪些问题。我们来稍微总结一下。

1、迭代器模式可以提供统一的迭代方式，这个要归功于Iterator接口。

2、迭代器模式可以在对客户透明的前提下，做出各种不同的迭代方式。

3、在迭代的时候不需要暴露聚合对象的内部表示，我们只需要认识Iterator即可。

4、在第1条的前提下，解决了基于数组的迭代方式中重复遍历的问题。

这里LZ就不再给出迭代器模式的标准代码实现了，如果各位看过LZ的前十几篇设计模式，会发现，LZ其实很多时候是不写标准实现的，一个是因为网上的这种资料很多，很容易找到，LZ不想重复造轮子。还有一个重要的原因是，标准实现总难免给人死板硬套的感觉，很难让人理解，至少LZ个人当时是这种感觉。

这里LZ直接使用迭代器模式，将我们上面的三个集合类稍微优化一下，首先我们应该写一个迭代器接口，它大概会有类图中的那几个方法。为了简单起见，我们直接利用JDK提供的Iterator接口，源码如下。

```
public interface Iterator<E> {



        boolean hasNext();



        E next();



        void remove();

}
```

这里迭代器接口已经有了，我们还需要一个可迭代的类接口，在JDK中相当于Iterable接口，它规定了返回一个迭代器的行为，与我们的类

图中Array接口类似，只不过那里是toArray方法。下面我们引用JDK中的Iterable接口，非常简单，源码如下。

```
public interface Iterable<T> {

	Iterator<T> iterator();

}
```

下面我们就让三个集合类全部提供一个方法，可以返回一个Iterator实例，并且实现Iterable接口。

```
package com.iterator;



import java.util.Iterator;



public class ArrayList<E> implements Iterable<E>{



	private static final int INCREMENT = 10;



	private E[] array = (E[]) new Object[10];



	private int size;



	public void add(E e) {

		if (size < array.length) {
```

```
                array[size++] = e;
            } else {
                E[] copy = (E[]) new Object[array.length +
INCREMENT];
                System.arraycopy(array, 0, copy, 0, size);
                copy[size++] = e;
                array = copy;
            }
        }

        public Object[] toArray() {
            Object[] copy = new Object[size];
            System.arraycopy(array, 0, copy, 0, size);
            return copy;
        }

        public int size() {
            return size;
        }

        public Iterator<E> iterator() {
```

```
        return new Itr();
    }

    private class Itr implements Iterator<E> {

        int cursor = 0;

        public boolean hasNext() {
            return cursor != size();
        }

        public E next() {
            return array[cursor++];
        }

        public void remove() {

        }

    }
```

```
}

package com.iterator;

import java.util.Iterator;

public class LinkedList<E> implements Iterable<E>{

	private Entry<E> header = new Entry<E>(null, null,
null);
	private int size;

	public LinkedList() {
		header.next = header.previous = header;
	}

	public void add(E e){
  Entry<E> newEntry = new Entry<E>(e, header, header.next);
		newEntry.previous.next = newEntry;
		newEntry.next.previous = newEntry;
		size++;
```

```
	}

	public int size(){
		return size;
	}

	public Object[] toArray(){
		Object[] result = new Object[size];
		int i = size - 1;
		for (Entry<E> e = header.next; e != header; e =
e.next)
			result[i--] = e.value;
		return result;
	}

	private static class Entry<E>{
		E value;
		Entry<E> previous;
		Entry<E> next;
		public Entry(E value, Entry<E> previous, Entry<E>
next) {
```

```
            super();
            this.value = value;
            this.previous = previous;
            this.next = next;
        }
    }

    public Iterator<E> iterator() {
        return new Itr();
    }

    private class Itr implements Iterator<E> {

        Entry<E> current = header;

        public boolean hasNext() {
            return current.previous != header;
        }

        public E next() {
```

```
                E e = current.previous.value;
                current = current.previous;
                return e;
            }

            public void remove() {

            }

        }

}

package com.iterator;

import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;

public class HashSet<E> implements Iterable<E>{
```

```
	private static final Object NULL = new Object();


	private Map<E, Object> map = new HashMap<E, Object>
(7,1);


	public void add(E e){
		map.put(e, NULL);
	}


	public int size(){
		return map.size();
	}


	public Object[] toArray(){
		return map.keySet().toArray();
	}


	public Iterator<E> iterator() {
		return map.keySet().iterator();
	}
```

```
}
```

这下我们已经将迭代器模式应用到了我们上面的例子当中，至于remove方法，我们为了简短清晰，就不做实现了，而且它并不影响我们理解迭代器模式。现在我们客户端的迭代方式就可以改变一下了。可以像下面这样迭代。

```
package com.iterator;



import java.util.Iterator;





public class Main {



      public static void main(String[] args) {

            ArrayList<Integer> arrayList = new
ArrayList<Integer>();

            for (int i = 1; i <= 11; i++) {

                  arrayList.add(i);

            }

            System.out.println("arrayList size:" +
arrayList.size());

            Iterator<Integer> arrayListIterator =
arrayList.iterator();
```

```
            while(arrayListIterator.hasNext()) {
System.out.println(arrayListIterator.next());
            }


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


            HashSet<Integer> hashSet = new HashSet<Integer>
();
            for (int i = 1; i <= 11; i++) {
                  hashSet.add(i);
            }
            System.out.println("hashSet size:" +
hashSet.size());
            Iterator<Integer> hashSetIterator =
hashSet.iterator();
            while(hashSetIterator.hasNext()) {
                  System.out.println(hashSetIterator.next());
            }


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

```
            LinkedList<Integer> linkedList = new
LinkedList<Integer>();

            for (int i = 1; i <= 11; i++) {

                  linkedList.add(i);

            }

            System.out.println("linkedList size:" +
linkedList.size());

            Iterator<Integer> LinkedListIterator =
linkedList.iterator();

            while(LinkedListIterator.hasNext()) {


System.out.println(LinkedListIterator.next());

            }

      }

}
```

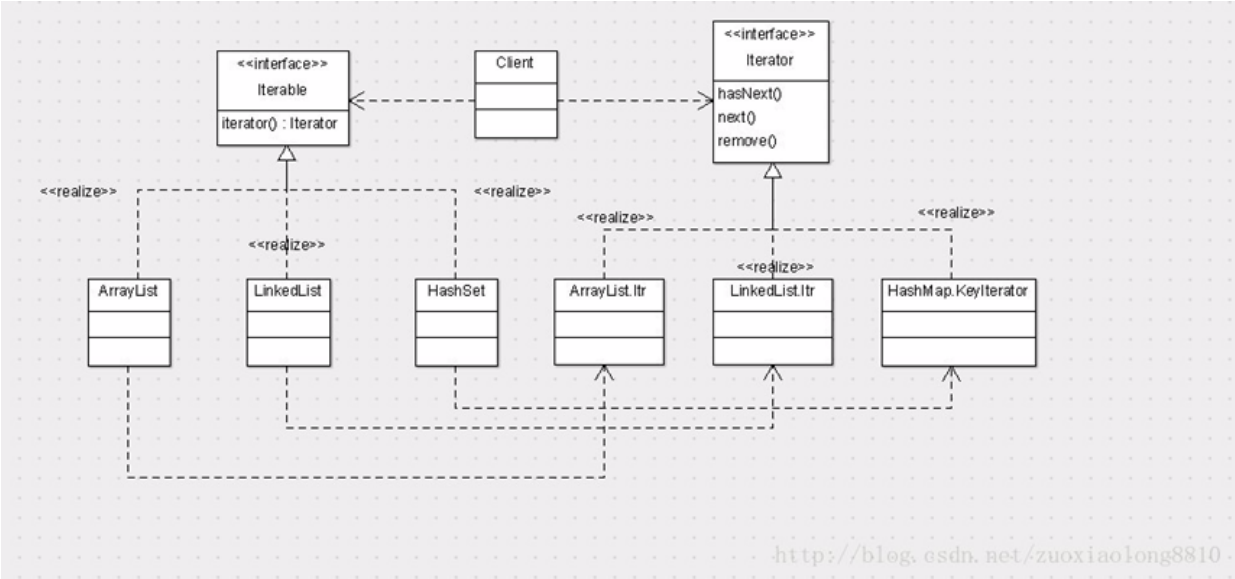
其中输出的结果与第一例是一样的，都是将1到11输出三遍，可以明显的看出，我们刚才的重复遍历问题不见了，而且三个集合类的迭代方法是一样的，而有了这个特点，JAVA在此基础上，给我们提供了foreach语法，所以我们可以写成下面这样。

```
package com.iterator;



public class Main {



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

```
            ArrayList<Integer> arrayList = new
ArrayList<Integer>();

            for (int i = 1; i <= 11; i++) {

                  arrayList.add(i);

            }

            System.out.println("arrayList size:" +
arrayList.size());

            for (Integer i : arrayList) {

                  System.out.println(i);

            }



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



            HashSet<Integer> hashSet = new HashSet<Integer>
();

            for (int i = 1; i <= 11; i++) {

                  hashSet.add(i);

            }

            System.out.println("hashSet size:" +
hashSet.size());

            for (Integer i : hashSet) {

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

```
            }

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

            LinkedList<Integer> linkedList = new
LinkedList<Integer>();

            for (int i = 1; i <= 11; i++) {

                  linkedList.add(i);

            }

            System.out.println("linkedList size:" +
linkedList.size());

            for (Integer i : linkedList) {

                  System.out.println(i);

            }

      }

}
```

这样一来，我们不管你是基于何种数据结构提供的集合类，我们只管foreach遍历，迭代器模式对JAVA集合框架做出的贡献不可谓不大。下面LZ带各位看看现在的类图是如何。



从类图中可以清楚的看出，与迭代器模式的类图是一模一样的，当然，客户端与Iterable的依赖关系有待商议，之前我们已经提到过，JAVA集合框架的工厂方法模式是非透明的处理方式，所以我们很多时候不会使用Iterable，不过这并不影响我们对迭代器模式的理解。

然而迭代器模式所带来的好处已经不言而喻，上面分析的过程中已经提到过，LZ这里不再赘述。

值得注意的是，LZ全部采用的内部类作为各个集合类迭代器的实现，这在LZ之前的文章中已经提到过，当时的解释是说内部类是为了完全杜绝客户端对迭代器实现类的依赖，而进行到现在，我们可以更深一步讨论。

这里我们的理解是，内部类在这里目的是为了隐藏实现细节，并且如此一来，迭代器的实现类可以自由的使用集合类的各个属性，而不需要集合类提供自己属性访问的接口以及建立二者的关联关系，这种感觉十分像C/C++中的友元类。

不过缺点也接踵而至，由于具体的集合类与具体的迭代器是绑定的关系，所以这种实现方式在复用的过程中会有很大的限制甚至是不能复用，这个缺点对于C/C++中的友元类来说，是不存在的。

到此，迭代器模式的介绍就基本结束了，希望各位可以从LZ的分析当中得到一点益处，LZ便十分知足了。下面是对于foreach语法的深入讨论，新手或者已经了解的同学们可以忽略，当然，如果新手们有耐心，也可以继续看下去。

JAVA的foreach语法中，冒号后面的变量可以是两种，一个是数组，一个便是实现了Iterable接口的任何类，数组的迭代方式本次不在讨论之列，我们主要关注实现了Iterable接口的类，是如何使用foreach遍历的。

我们使用JDK中自带的一个工具javap，去获得刚才Main类的class文件描述，去看看遍历的过程。我们在命令行模式下，进入到Main类class文件的目录中，然后执行下面的命令，为了更加清晰，我们只保留ArrayList的代码片段，将其余两部分删掉或者注释掉。

```
javap -verbose Main
```

执行这个命令，我们可以看到在main函数中有关foreach的JVM指令，我们来看下JVM是怎么执行foreach的，生成的有关内容如下。

```
   56:  invokevirtual   #59; //Method com/iterator/ArrayList.iterator:()Ljava/ut
il/Iterator;
   59:  astore_3
   60:  goto    80
   63:  aload_3
   64:  invokeinterface #63,  1; //InterfaceMethod java/util/Iterator.next:()Lja
va/lang/Object;
   69:  checkcast       #20; //class java/lang/Integer
   72:  astore_2
   73:  getstatic       #29; //Field java/lang/System.out:Ljava/io/PrintStream;
   76:  aload_2
   77:  invokevirtual   #69; //Method java/io/PrintStream.println:(Ljava/lang/Ob
ject;)V
   80:  aload_3
   81:  invokeinterface #71,  1; //InterfaceMethod java/util/Iterator.hasNext:()
Z
   86:  ifne    63
   89:  return
```

http://blog.csdn.net/zuoxiaolong881

各位的行数不一定会和LZ一样，不过总能找到和LZ一样的JVM指令片段，下面LZ直接按照上面的行号去解释每一行都做了什么事情。

56：执行arrayList的iterator方法。

59：将返回值从值栈赋给第四个引用变量，这个是在解释执行过程中自动加入的临时变量。

60：跳到80行。

80：将第四个引用变量压入值栈，这个引用变量是个迭代器，也就是iterator方法的返回值。

81：执行接口方法hasNext。

86：当hasNext方法返回的值不等于0时，也就是不为false时，跳到63行。

63：将第四个引用变量压入值栈。

64：执行接口方法next。

69：检查强转，不成功将ClassCastException。

72：将next返回的值强转后赋给第三个引用变量，在代码里相当于i。

73：获取out输出流静态属性，压入值栈。

76：将i压入值栈。

77：执行println方法，打印i。

80：同上面的80行。

到这里，可以看到，JVM对于foreach的执行过程，其实相当于下面的代码。

```
//其中iterator这个变量就是第四个引用变量，i是第三个。

for (Iterator iterator = arrayList.iterator();
iterator.hasNext();) {

        Integer i = (Integer) iterator.next();
```

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

这么做的前提自然是建立在iterable接口和iterator接口的基础之上，由于这两者规定了集合类的行为，所以JAVA才得以利用这一特性，提供给开发者foreach的语法。

对于foreach的语法就分析到这里了，最后我们再对迭代器模式进行一下总结。

迭代器模式是为了在隐藏聚合对象的内部表示的前提下，提供一种遍历聚合对象元素的方法。这在之前的例子当中已经有体现，我们给三个集合类提供了统一的遍历方式，消除了重复遍历，最终还使用内部类将细节包装在集合类内部隐藏起来，使得外部无法访问集合类的任何一个属性。

这当中还使用到了工厂方法模式，这个在工厂方法模式一章中，LZ已经提到过，对于迭代器的产生，是工厂方法模式处理的。两个设计模式互相结合，让JAVA的集合框架更加优美、健壮。

LZ如此诠释迭代器模式，希望对大家有帮助，好了，咱们下期再见吧。

（十八）享元模式详解（都市异能版）

魔都。

自从越狱风波过去以后，小左的生活便又回到了之前的节奏，依旧是每日徘徊在魔都某天桥，继续着自己的算命之旅。

说起这次越狱风波，着实让小左心情沉闷了很久。原因无他，就是因为在施展一次记忆恢复的过程中，发生了一点意外，结果导致一名看守所人员的记忆产生了混乱，如果用通俗的话来讲，就是变成了人们口中常说的精神病。

小左当时选择的是一个生物专业硕士毕业的应届生的记忆，而另外一位被注入记忆的看守所成员，已经成功了迈入了生物界的神奇领域，虽说谈不上一步登天，但相信现在生活的应该要比之前在看守所好上数倍。当然，这番生活条件的改观也并非白来的，这位新生的生物界奇才，所失去的，正是之前数十年的记忆，这之中也当然包括了对家人朋友的记忆。

至于故事当中的主角韩雨露，此时应该已经被其能力通天的父亲给保护了起来。自从韩雨露被小左私自释放，除了刚开始的新闻漫天之外，短短的十数天，此事就像是没有发生过一样烟消云散。尽管这是小左意料之中的，否则小左也不会义无反顾的释放韩雨露，不过之前小左可远远没有想到消息的封锁会如此之快。

值得一说的是李刚和李双江，自从被误认为私自释放了韩雨露，二人的生活就直接陷入了暗无天日的境地，虽说这样的结果对二人有些不公平，不过由于二人对韩雨露语言上的亵渎，导致小左并未对此有所愧疚。

魔都某网吧。

“尼玛，辅助买个鸟啊。”小左坐在烟雾缭绕的网吧里，骂骂咧咧的大声喊道。细一看，就发现小左在叫骂的同时，正拉着自己红血的SF马不停蹄的往泉水跑。

.....

“我考，不带这么玩的吧。”

只见小左的电脑屏幕之上，三道红光一闪即逝，小左的SF瞬间倒地，在释放一记魂之挽歌之后，便不甘的回到了泉水。

“SF，你TM会不会啊，别出门了行不？”

“9494，你这么送下去，我们还怎么玩啊。”

“猪一样的队友啊，真TM倒霉。”

.....

看着屏幕上陆续浮现出的队友们的谩骂，小左终于忍受不了，无奈的看了看自己40分钟依旧是假腿的SF，匆匆的退出了游戏。

魔都某出租屋。

“好不容易想起去体验几把DOTA，结果全跪了。尼玛，太不爽了。不行，我得虐虐电脑，找找快感。”

由于被连续虐了几把，小左回到家里依旧感觉十分不爽，于是便再次打开DOTA，准备虐几把电脑找找感觉。

机器始终不是人类，在连续几把无限超神之后，小左终于感觉心里舒畅了许多。无聊之际，开始漫无目的的在网上冲浪。不过由于小左转生之前毕竟是一名程序猿，所以看着看着，不知不觉之间便打开了享元模式的定义。

定义：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。

“看定义的意思，这个模式主要是为了减少不必要的重复对象，减少内存消耗。而要做到这个的话，那么就需要把一个对象可共享的状态给封装起来，而不可能共享的状态则从外部获取。”

小左指尖不停的敲打着桌面，继续说道：“如果所有DOTA的玩家都在一台服务器上操作，那么DOTA里的英雄应该就可以使用享元模式。每一个英雄应该都是唯一的实例，而不应该是一旦有人选了一个英雄，就要实例化一个。”

“如果是传统意义上的方式，应该是这种写法。假设有一个抽象的英雄基类。”

```
package com.flyweight;



//抽象英雄基类

public abstract class AbstractHero {



	protected final String name;//英雄名称



	protected final String[] skills = new String[4];//每个英雄都有四个技能



	protected long hp;//血量
```

```
protected long mp;//魔法

public AbstractHero() {
	super();
	this.name = getName();
	initSkills();
	checkSkills();
}

private void checkSkills(){
	for (int i = 0; i < skills.length; i++) {
		if (skills[i] == null) {
throw new NullPointerException();
		}
	}
}

//释放技能
public void release(int index){
	if (index < 0) {
```

```
                index = 0;
            }else if (index > 3) {
                index = 3;
            }
            System.out.println(name + "释放" +
skills[index]);
        }

        //物理攻击
        public void commonAttack(){
            System.out.println(name + "进行物理攻击");
        }

        public abstract String getName();

        public abstract void initSkills();

        public long getHp() {
            return hp;
        }
```

```
	public void setHp(long hp) {
		this.hp = hp;
	}

	public long getMp() {
		return mp;
	}

	public void setMp(long mp) {
		this.mp = mp;
	}

}
```

“英雄可以释放技能，物理攻击等等，而且英雄是有血量和魔法量的。下面就写两个具体的英雄看一下。”

```
package com.flyweight;

public class Lion extends AbstractHero{

	public String getName() {
		return "恶魔巫师";
```

```
	}

	public void initSkills() {
		skills[0] = "穿刺";
		skills[1] = "妖术";
		skills[2] = "法力汲取";
		skills[3] = "死亡一指";
	}

}

package com.flyweight;

public class SF extends AbstractHero{

	public String getName() {
		return "影魔";
	}

	public void initSkills() {
```

```
        skills[0] = "毁灭阴影";
        skills[1] = "支配死灵";
        skills[2] = "魔王降临";
        skills[3] = "魂之挽歌";
    }

}
```

“一个恶魔巫师，一个影魔。很显然，假设现在有四个solo局，都是SF对LION，那么现在这种方式则需要实例化四个LION和四个SF。就像下面这样。”

```
package com.flyweight;


public class Main {


    public static void main(String[] args) {
        //假设有四个solo局，则需要创建四个lion和四个sf
        Lion lion1=new Lion();
        SF sf1 = new SF();

        Lion lion2=new Lion();
        SF sf2 = new SF();
```

```
        Lion lion3=new Lion();

        SF sf3 = new SF();


        Lion lion4=new Lion();

        SF sf4 = new SF();


        /* 以下为释放技能，物理攻击等的打架过程    */

    }



}
```

“这样显然是非常浪费资源啊，四个lion和四个sf其实是有很多一样的地方的。这应该就可以使用享元模式了吧。”

“不过享元模式强调内部状态和外部状态，内部状态则是可以共享的状态，外部状态则是随外部环境而变化的状态，是无法共享的状态。那么下面要做的就是将外部状态分离出来，只保留内部状态，这样的话对象的实例就可以共享了。”

“啪！”

小左打了个响指，继续说道：“对于上面DOTA英雄的简单例子来说，血量和魔法量应该是外部状态了，因为四个solo局中，每个lion和每个sf的血量和魔法量不一定是相等的，所以这两个状态是无法共享的。但是技能和名称就不一样了，它们俩应该属于内部状态，是可以共享的，因为不管是哪个solo局里的lion或者sf，它们的英雄名称和技能都是一样的。”

“这下好办了，首先应该把基类里面的血量和魔法量删掉。”

```
package com.flyweight;


//抽象英雄基类
public abstract class AbstractHero {


	protected final String name;//英雄名称


	protected final String[] skills = new String[4];//每个英雄都有四个技能


	public AbstractHero() {
		super();
		this.name = getName();
		initSkills();
		checkSkills();
	}


	private void checkSkills(){
		for (int i = 0; i < skills.length; i++) {
			if (skills[i] == null) {
```

```
            throw new NullPointerException();
                }
        }
    }

    //释放技能
    public void release(int index){
        if (index < 0) {
            index = 0;
        }else if (index > 3) {
            index = 3;
        }
        System.out.println(name + "释放" +
skills[index]);
    }

    //物理攻击
    public void commonAttack(){
        System.out.println(name + "进行物理攻击");
    }
```

```
	public abstract String getName();


	public abstract void initSkills();


}
```

“其余的两个子类应该是不用变的，下面我还需要再写一个类去组合英雄的内部状态和外部状态。”

```
package com.flyweight;

//用于组合英雄内部状态和外部状态的类，假设称为角色

public class Role {


	private AbstractHero hero;//角色选择的英雄


	//我们把血量和魔法量这两个外部状态从英雄里剥离出来，放到外部的角色类中

	private long hp;


	private long mp;


	public Role(AbstractHero hero) {

		super();
```

```
        this.hero = hero;
    }

    //释放技能
    public void release(int index){
        hero.release(index);
    }

    //物理攻击
    public void commonAttack(){
        hero.commonAttack();
    }

    public long getHp() {
        return hp;
    }

    public void setHp(long hp) {
        this.hp = hp;
    }
```

```
	public long getMp() {
		return mp;
	}

	public void setMp(long mp) {
		this.mp = mp;
	}

}
```

“我们用角色这个类去组合英雄的内部和外部状态，下面还需要一个享元模式最重要的类，就是提供共享功能的类。”

```
package com.flyweight;

import java.util.HashMap;
import java.util.Map;
//提供共享功能，单例
public class HeroManager {

	private static HeroManager heroManager = new HeroManager();
```

```
private Map<String, AbstractHero> heroMap;

private HeroManager(){
	heroMap = new HashMap<String, AbstractHero>();
}

public static HeroManager getInstance(){
	return heroManager;
}

//该方法提供共享功能
public AbstractHero getHero(String name){
	AbstractHero hero = heroMap.get(name);
	if (hero == null) {
		if (name.equals("恶魔巫师")) {
			hero = new Lion();
		}else if (name.equals("影魔")) {
			hero = new SF();
		}
```

```
                heroMap.put(name, hero);
            }
            return hero;
        }
}
```

“现在如果再来四个solo局，那么情况就和刚才不太一样了。”

```
package com.flyweight;


public class Main {


    public static void main(String[] args) {
            //假设有四个solo局，则使用了享元模式的情况下，其实恶魔巫师和影魔的实例各自只有一个
            HeroManager heroManager = HeroManager.getInstance();
            Role role1 = new Role(heroManager.getHero("恶魔巫师"));
            Role role2 = new Role(heroManager.getHero("影魔"));


            Role role3 = new Role(heroManager.getHero("恶魔巫师"));
```

```
            Role role4 = new Role(heroManager.getHero("影魔"));


            Role role5 = new Role(heroManager.getHero("恶魔巫师"));

            Role role6 = new Role(heroManager.getHero("影魔"));


            Role role7 = new Role(heroManager.getHero("恶魔巫师"));

            Role role8 = new Role(heroManager.getHero("影魔"));


            /* 以下为释放技能，物理攻击等的打架过程    */

        }


}
```

“四个solo局当中有八个角色，选了八个英雄，四个lion和四个sf，但是很明显，使用了享元模式之后，这里面的八个英雄其实只有一个lion实例和一个sf实例，这样就大大减少了英雄的实例个数。试想一下，如果同时有1000个solo局，按照之前的方式，那么会有1000个lion的实例和1000个sf的实例，而现在使用享元模式的话，依旧是只有一个lion实例和一个sf实例。”

“这么说起来的话，享元模式节省的内存实在是不可估量啊。只不过DOTA这么大型的游戏，它的设计应该不会这么简单，不过用来理解享元模式还是个不错的例子啊，哈哈。”

理解了享元模式的好处，小左像打了鸡血似的，一拍桌子大叫道：“NND，再去网吧搞两把。”

话音刚落，小左便一溜烟跑出了出租屋，直奔网吧而去。

（十九）组合模式详解

定义：(GoF《设计模式》)：将对象组合成树形结构以表示“部分整体”的层次结构。组合模式使得用户对单个对象和组合对象的使用具有一致性。

从定义中可以看出，组合模式用来表示部分与整体的层次结构（类似于树结构），而且也可以使用户对单个对象（叶子节点）以及组合对象（非叶子节点）的使用具有一致性，一致性的意思就是说，这些对象都拥有相同的接口。

很多书中包括文章都喜欢使用公司、子公司以及部门的例子，这其实就是一个典型的树结构。其实生活当中的树结构还有很多，比如书的目录、文件系统、网站的菜单等等，有很多很多。

我们先来看看组合模式的类图，引自百度百科。

类图当中有三个类，一个是Component（节点的统一接口），它的目的是为了统一节点的操作。接下来的两个实现类，一个则是非叶子节点（Composite），它可以有子节点。另外一个则是叶子节点（Leaf），它不能含有子节点。

我们随便挑一个树结构的例子，比如文件系统，我们来分析一下，在文件系统中，如果使用组合模式，各个部分的类都应该是什么样子的。

首先，文件系统中，叶子节点是文件，非叶子节点是文件夹，所以Leaf和Composite实现类就是文件和文件夹。对于Component接口，其实也很简单，就是提取文件和文件夹的共性就可以了。

很显然，二者的共性有很多，比如都可以进行复制、剪切、删除、重命名等操作。但是不同的是，对于文件和文件夹的这些操作是有细微

的区别的，最明显的就是删除操作，如果是文件，那么我们只需要删除当前文件即可，而如果是文件夹，则需要删除文件夹下的所有文件以及文件夹，然后再删除该文件夹。

那么定义当中的一致性就体现在，我们的客户端不需要知道当前操作的是文件还是文件夹，它只知道它要进行删除操作，而我们去针对文件类别的不同去进行相应的处理。

下面我们来模拟一下组合模式，采用文件系统。

首先，我们先给出一个接口，它相当于Component接口，定义了文件与文件夹的公共行为。

```
package com.composite;




//文件系统中的节点接口

public interface IFile {

	//下面两个方法，相当于类图中operation方法

	void delete();

	String getName();



	/* 以上为公共行为，以下为文件夹才有的行为 */



	//创建新文件，相当于add方法

	void createNewFile(String name);
```

```
        //相当于remove方法

        void deleteFile(String name);

        //相当于GetChild方法

        IFile getIFile(int index);

}
```

类图中的operation方法是一个宏观定义，它代表的意思是叶子节点和非叶子节点的公共行为，并不是说只有一个operation方法，本次LZ给出两个共有行为作为代表，即删除操作和获取文件名称的操作。

下面我们来看下非叶子节点，即文件夹的实现类。

```
package com.composite;



import java.util.ArrayList;

import java.util.List;



//文件夹

public class Folder implements IFile{



        private String name;

        private IFile folder;

        private List<IFile> files;
```

```
public Folder(String name) {
        this(name, null);
}

public Folder(String name,IFile folder) {
        super();
        this.name = name;
        this.folder = folder;
        files = new ArrayList<IFile>();
}

public String getName() {
        return name;
}

//与File的删除方法不同，先删除下面的文件以及文件夹后再删除自己
public void delete() {
        List<IFile> copy = new ArrayList<IFile>(files);
```

```
            System.out.println("------------删除子文件-------
------");
            for (IFile file : copy) {
                  file.delete();
            }
            System.out.println("----------删除子文件结束-----
--------");
            if (folder != null) {
                  folder.deleteFile(name);
            }
            System.out.println("---删除[" + name + "]---");
      }

      public void createNewFile(String name) {
            if (name.contains(".")) {
                  files.add(new File(name,this));
            }else {
                  files.add(new Folder(name,this));
            }
      }
```

```
    public void deleteFile(String name) {
        for (IFile file : files) {
            if (file.getName().equals(name)) {
                files.remove(file);
                break;
            }
        }
    }

    public IFile getIFile(int index) {
        return files.get(index);
    }
}
```

我们看到这里面最主要的地方在于它有一个List<IFile>属性，这个属性是树结构的关键点，当我们删除一个文件夹时，即delete方法，我们会首先删除该文件夹下面的所有文件以及文件夹，这与我们平时使用的windows操作系统的文件操作是一致的。

下面三个方法，createNewFile、deleteFile和getIFile，分别对应于类图当中的add、remove以及getChild方法，只不过为了更加形象，此处修改了方法名称。

下面我们看叶子节点的实现，即文件类。

```
package com.composite;
```

```
//文件
public class File implements IFile{

	private String name;
	private IFile folder;

	public File(String name,IFile folder) {
		super();
		this.name = name;
		this.folder = folder;
	}

	public String getName() {
		return name;
	}

	public void delete() {
		folder.deleteFile(name);
		System.out.println("---删除[" + name + "]---");
```

```
    }

    //文件不支持创建新文件
    public void createNewFile(String name) {
        throw new UnsupportedOperationException();
    }
    //文件不支持删除文件
    public void deleteFile(String name) {
        throw new UnsupportedOperationException();
    }
    //文件不支持获取下面的文件列表
    public IFile getIFile(int index) {
        throw new UnsupportedOperationException();
    }
}
```

文件类中的delete方法与文件夹中的不同，一个文件的删除操作，只需要删除它自己即可。我们还会注意到，下面的三个方法，LZ全部抛出了不支持的操作的异常，这也是与我们传统意义上的文件操作是一致的，一个文件当然不能在该文件下进行创新新文件、删除文件以及获取某个文件的操作。

当然，你也可以直接将三个方法放空，或者返回null值，不过LZ觉得这样的方式不易于以后进行调试，所以LZ个人不推荐。

下面我们来简单的模拟下我们的文件系统，我们创建一个简单的文件系统，然后在上面进行删除操作。

```
package com.composite;


public class Main {


	public static void main(String[] args) {
		IFile root = new Folder("我的电脑");
		root.createNewFile("C盘");
		root.createNewFile("D盘");
		root.createNewFile("E盘");
		IFile D = root.getIFile(1);
		D.createNewFile("project");
		D.createNewFile("电影");
		IFile project = D.getIFile(0);
		project.createNewFile("test1.java");
project.createNewFile("test2.java");
project.createNewFile("test3.java");
		IFile movie = D.getIFile(1);
		movie.createNewFile("致青春.avi");
		movie.createNewFile("速度与激情6.avi");
```

```
        /* 以上为当前文件系统的情况，下面我们尝试删除文件和文件夹 */
        display(null, root);
        System.out.println();

        project.delete();
        movie.getIFile(1).delete();

        System.out.println();
        display(null, root);
    }

    //打印文件系统
    public static void display(String prefix,IFile iFile){
        if (prefix == null) {
            prefix = "";
        }
        System.out.println(prefix + iFile.getName());
        if(iFile instanceof Folder){
            for (int i = 0; ; i++) {
```

```
                        try {
                              if (iFile.getIFile(i) != null)
{
                                    display(prefix + "--",
iFile.getIFile(i));
                              }
                        } catch (Exception e) {
                              break;
                        }
                  }
            }
      }
}
```

我们首先模拟了一个简单的文件系统，有C/D/E盘，然后又在D盘下建立了两个文件夹以及一些文件，接下来我们使用统一的操作接口去操作文件和文件夹，进行删除操作。

在删除的前后，LZ分别打印了一遍当前的文件系统，结果如下。

可以看到，我们成功删除了[project]文件夹和[速度与激情6.avi]文件，在删除[project]文件夹时，首先删除了其文件夹下面的三个java文件。

所以结合组合模式的定义，在上面的例子中，我们做了下面两件事，正好是组合模式定义中提到的。

第一、就是我们使用组合模式，描述了一个文件系统的树结构。

第二、就是在组合模式下，我们给客户端提供了统一的删除操作，当然，我们还可以提供统一的复制，剪切，查看文件属性等等操作，只不过作为例子，我们只列出了删除操作。

上面我们针对标准的组合模式，给出了一个例子，下面请各位思考一下，上面的例子当中，是否有不妥的地方。

答案是肯定的，上面的例子当中，我们的叶子节点类（File）中，有三个不支持的方法，而之所以出现这样的情况，是因为我们在IFile接口中，提供的是宽接口，这样做的目的是为了对客户端保持透明，然而相应的却带来了不安全性。

所以有时候我们为了安全性，会相应的牺牲透明性，把IFile接口中叶子节点不支持的三个行为全部删掉，由此可见，在组合模式中，安全性和透明性是互相矛盾的，这是由于叶子节点和非叶子节点行为的不一致以及需要提供一个一致的行为接口所造成的，是不可调和的矛盾。

针对这种情况，我们只能做出相应的取舍，如果我们使用非透明且相对安全的方式去实现上面的例子，那么我们的客户端调用时，会经常出现下面这样的代码。

```
IFile movie = D.getIFile(1);

        if (movie instanceof Folder) {

              Folder folder = (Folder) movie;

              //下面使用folder进行文件夹独有的操作

        }
```

出现上面代码的原因很明显，这是由于我们IFile接口不再提供Folder的行为所造成的。所以使用非透明的组合模式，会相应的增加客户端操作的复杂性。

LZ个人认为大部分情况下，我们应当优先考虑透明的策略，即本文给出的方式。

最后，LZ来总结一下组合模式的应用场景，其实就是定义当中所提到的两点。

1、如果你想表示“部分整体”的层次结构，可以使用组合模式。

2、如果你想让客户端可以忽略复杂的层次结构，使用统一的方式去操作层次结构中的所有对象，也可以使用组合模式。

好了，本次组合模式就到此结束了，感谢各位的收看。

“咦？我订的餐到啦，哈哈。”

（二十）职责链模式详解（都市异能版）

凌晨两点，魔都某出租屋。

"God like."

.....

"Holy shit."

.....

“哈哈。老子终于超神一次啦。”伴随着低音炮中一声怒吼，小左在自己十来平米的出租房里兴奋的尖叫起来。近了一看，原来小左操作的J8脸30分钟已经一身神装，如今正在疯狂的大杀四方。

“今晚杀的真爽，干脆通宵得了。”

依依不舍的退出了超神的一局，小左摸了摸自己像漏气了一样的肚子，喃喃的说道：“肚子有点饿了啊，这大半夜的可怎么办呢。”

“喂，你好，请问现在还送餐吗？”

“不好意思，先生，您的住址离我们太远，请您问一下其它分店。”

“问你妹啊，老子问了好几家了。”心里虽然不爽，小左嘴上还是很客气的说道：“嗯，好的，谢谢啊。”

.....

“喂，你好，请问你们往XXXX小区送餐吗？”

“嗯，是的。先生。”

小左心中一乐，“嗯，那好，我要一个XXX，一个XXX，一个XXX。”

“不好意思，先生，现在太晚了，我们的送餐时间已经过了。”

“我....去....你....妹啊。不送不早说。”心中暗骂了一句，小左直接挂了电话。

.....

“喂，你好，请问你们往XXXX小区送餐吗？”

“嗯，是的。先生。”

“嗯，那你们送餐时间没过吧。”有了上次的经验，小左没有先报菜单，而是先确认对方是否过了送餐时间。

“没有，先生。”

“嗯，那好，我要一个XXX，一个XXX，一个XXX。”

“不好意思，先生，这些都已经卖完了，您看您能换别的吗？”

“我....换你大爷.....”

半个小时过去了...

“哎呀妈呀，累死我了。终于找到一家送餐的了。这麦当劳号称24小时外卖是坑爹呢吧。”经过将近半个小时的折腾，小左终于找到一家比较合适的麦当劳分店，现在就只等着外卖送过来了。

“外卖估计半个小时就到了，这一会也开不了一局，干脆干点别的，等吃完饭再开始。”说着，小左便开始在网上找寻设计模式的内容，这一闲下来就研究设计模式的劲头，当真是不辱转生前程序猿的头衔。

“啊哈，职责连模式。看起来可以解决麦当劳订餐的问题啊，好好研究一下，给麦当劳老总提个建议，说不定赏我个几亿美金。哇哈

哈。”还没开始研究，小左已经开始YY起来。

定义：为了避免请求的发送者和接收者之间的耦合关系，使多个接受对象都有机会处理请求。将这些对象连成一条链，并沿着这条链传递该请求，直到有一个对象处理它为止。

“看这个定义，就是将一堆可以处理请求的对象连成一条链，然后一个一个试着处理请求。这好像是可以解决麦当劳订餐的问题的，我先来看看我刚才苦B的订餐过程是什么样子的。”

“首先应该有一个麦当劳的分店的类，它的主要功能是可以订餐。”

```
package com.chain;



import java.util.Collections;

import java.util.Map;



//麦当劳分店

public class McSubbranch {

	private final static int MIN_DISTANCE = 500;//假设是500米以内送餐

	private static int count;//类计数

	private final int number;//分店号

	private int x;//分店的横坐标，用于判断距离

	private int y;//分店的纵坐标，用于判断距离

	private Map<String, Integer> menu;//菜单
```

```
    public McSubbranch(int x, int y, Map<String, Integer>
menu) {

        super();

        this.x = x;

        this.y = y;

        this.menu = menu;

        number = ++count;

    }



    public boolean order(int x,int y,Map<String, Integer>
order){

        //如果距离小于500米并且订单中的食物不缺货，则订单
成功，否则失败

        if (CommonUtils.getDistance(x, y, this.x, this.y)
< MIN_DISTANCE && !CommonUtils.outOfStock(menu, order)) {

            for (String name : order.keySet()) {

            menu.put(name, menu.get(name) -
    order.get(name));

            }

            return true;

        }else {

            return false;

        }
```

```
	}

	public Map<String, Integer> getMenu() {
		return Collections.unmodifiableMap(menu);
	}

	public String toString() {
		return "麦当劳分店第" + number + "个";
	}
}
```

“这里面用到了一个工具类，主要是用来计算距离和判断是否缺货的，就是这样写的。”

```
package com.chain;

import java.util.Map;
//简单的工具类
public class CommonUtils {
	private CommonUtils(){}
	//计算坐标之间的距离
	public static double getDistance(int x1,int y1,int x2,int y2){
```

```
            return Math.sqrt((x1 - x2) * (x1 - x2) + (y1 -
y2) * (y1 - y2));

      }

      //是否缺货

      public static boolean outOfStock(Map<String, Integer>
menu,Map<String, Integer> order){

            if (order == null || order.size() == 0) {

                  return false;

            }

            if (menu == null || menu.size() == 0) {

                  return true;

            }

            for (String name : order.keySet()) {

                  if (menu.get(name) - order.get(name) < 0) {

                        return true;

                  }

            }

            return false;

      }

}
```

“下面就是我苦B的订餐过程了，假设有五个分店的话，我订餐的过程就是一家一家挨着去订，直到成功为止。”

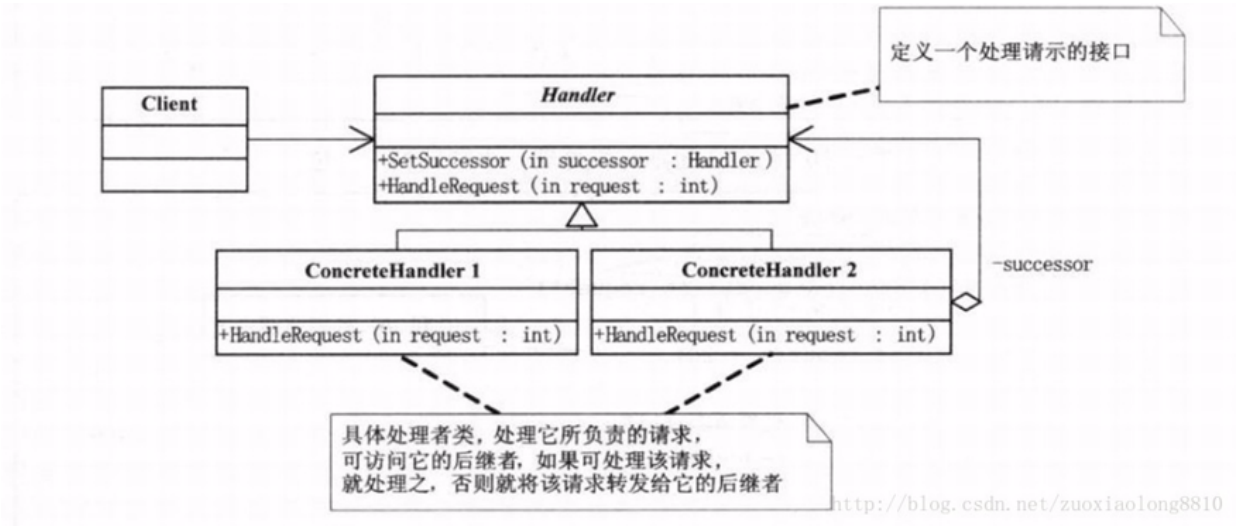
```
package com.chain;


import java.util.Arrays;

import java.util.HashMap;

import java.util.List;

import java.util.Map;


public class Client {

	public static void main(String[] args) {

		//假设初始菜单都是以下这些东西

		Map<String, Integer> menu = new HashMap<String,
Integer>();

		menu.put("汉堡", 5);

		menu.put("薯条", 5);

		menu.put("可乐", 5);

		menu.put("雪碧", 5);

		//假设有5个分店

		McSubbranch mcSubbranch1 = new McSubbranch(0, 0,
new HashMap<String, Integer>(menu));

		McSubbranch mcSubbranch2 = new McSubbranch(100,
120, new HashMap<String, Integer>(menu));
```

```
            McSubbranch mcSubbranch3 = new McSubbranch(-100,
-120, new HashMap<String, Integer>(menu));

            McSubbranch mcSubbranch4 = new McSubbranch(1000,
20, new HashMap<String, Integer>(menu));

            McSubbranch mcSubbranch5 = new McSubbranch(-500,
0, new HashMap<String, Integer>(menu));

            List<McSubbranch> mcSubbranchs =
Arrays.asList(mcSubbranch1,mcSubbranch2,mcSubbranch3,mcSubbra
nch4,mcSubbranch5);



            //小左开始订餐，假设小左的坐标是900,20

            Map<String, Integer> order = new HashMap<String,
Integer>();

            order.put("汉堡", 2);

            order.put("可乐", 1);

            order.put("薯条", 1);



            print(mcSubbranchs);

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

            //小左开始一家一家挨着尝试订餐，直到成功

            for (McSubbranch mcSubbranch : mcSubbranchs) {

                  if (mcSubbranch.order(900, 20, order)) {
```

```
                        System.out.println("订餐成功，接受订单的分店是：" + mcSubbranch);

                        break;

                    }

            }

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

            print(mcSubbranchs);

        }



        public static void print(List<McSubbranch> mcSubbranchs){

            for (McSubbranch mcSubbranch : mcSubbranchs) {

                    System.out.println("[" + mcSubbranch + "]的菜单:" + mcSubbranch.getMenu());

            }

        }

}
```

“这样确实比较悲催啊，我得一家一家打电话问，太麻烦了。麦当劳这么大一个企业，订餐的服务竟然这么烂哦。看我用设计模式给你优化一下吧，哈哈。”

“先来看看职责链模式的类图，这样比较好设计。”



“类图还是比较简单的啊，有一个通用的接口，然后就是若干个具体的处理者。按照现在麦当劳的情况来说，接口里handleRequest方法其实就是order（订餐）方法了，而setSuccessor方法，则是用来设置职责链的下一个处理者。”

“对于麦当劳的问题来说，每一个分店就是具体的处理者了，主要的改动应该是抽象出来一个接口以及职责链的连接过程，而刚才发送订单的时候是拆分成方法参数传递给订餐方法的，现在最好是把订单做成一个数据类。”

```
package com.chain;



import java.util.Map;



//订单类（相当于request，其实就是封装一个请求）

public class Order {

	private int x;

	private int y;
```

```
    private Map<String, Integer> order;

    public Order(int x, int y, Map<String, Integer> order)
{
        super();
        this.x = x;
        this.y = y;
        this.order = order;
    }

    public int getX() {
        return x;
    }

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

    public int getY() {
        return y;
    }
```

```
	public void setY(int y) {
		this.y = y;
	}

	public Map<String, Integer> getOrder() {
		return order;
	}

	public void setOrder(Map<String, Integer> order) {
		this.order = order;
	}

}
```

“下面便应该是分店接口了，它有两个方法，和类图当中的方法类似，只是名称略有改变。”

```
package com.chain;
//分店接口（相当于Hanlder）
public interface Subbranch {
	void setSuccessor(Subbranch subbranch);
	boolean handleOrder(Order order);
```

```
}
```

“下面便是麦当劳分店的实现类了，它主要的改变是添加了一个属性（下一个分店），这应该就是链形成的基石了。”

```
package com.chain;


import java.util.Collections;

import java.util.Map;


//麦当劳分店

public class McSubbranch implements Subbranch{

	private final static int MIN_DISTANCE = 500;//假设是500米以内送餐

	private static int count;//类计数

	private final int number;//分店号

	private int x;//分店的横坐标，用于判断距离

	private int y;//分店的纵坐标，用于判断距离

	 private Map<String, Integer> menu;//菜单

	private Subbranch nextSubbranch;//下一家分店

	public McSubbranch(int x, int y, Map<String, Integer> menu) {

		super();

		this.x = x;
```

```
        this.y = y;

        this.menu = menu;

        number = ++count;

    }

    //设置下一家分店

    public void setSuccessor(Subbranch subbranch) {

        this.nextSubbranch = subbranch;

    }

    //按照职责链处理订单

    public boolean handleOrder(Order order){

        //如果距离小于500米并且订单中的食物不缺货，则订单成功，否则失败

        if (CommonUtils.getDistance(order.getX(), order.getY(), this.x, this.y) < MIN_DISTANCE && !CommonUtils.outOfStock(menu, order.getOrder())) {

            for (String name : order.getOrder().keySet()) {

                menu.put(name, menu.get(name) - order.getOrder().get(name));

            }

            System.out.println("订餐成功，接受订单的分店是：" + this);

            return true;
```

```
        }
        if (nextSubbranch == null) {
            return false;
        }
        return nextSubbranch.handleOrder(order);
    }

    public Map<String, Integer> getMenu() {
        return Collections.unmodifiableMap(menu);
    }

    public Subbranch getNextSubbranch() {
        return nextSubbranch;
    }

    public String toString() {
        return "麦当劳分店第" + number + "个";
    }
}
```

“handleOrder方法中的逻辑就是职责链的精髓了，它会试图处理请求，如果处理不了，则交给链中的下一个分店。刚才用的

CommonUtils应该不用变了。下面就看下有了职责链模式之后，我的订餐方式吧。”

```
package com.chain;


import java.util.HashMap;

import java.util.Map;


public class Client {

      public static void main(String[] args) {

            //假设初始菜单都是以下这些东西

            Map<String, Integer> menu = new HashMap<String,
Integer>();

            menu.put("汉堡", 5);

            menu.put("薯条", 5);

            menu.put("可乐", 5);

            menu.put("雪碧", 5);

            //假设有5个分店

            Subbranch mcSubbranch1 = new McSubbranch(0, 0,
new HashMap<String, Integer>(menu));

            Subbranch mcSubbranch2 = new McSubbranch(100,
120, new HashMap<String, Integer>(menu));

            Subbranch mcSubbranch3 = new McSubbranch(-100,
-120, new HashMap<String, Integer>(menu));
```

```
            Subbranch mcSubbranch4 = new McSubbranch(1000,
20, new HashMap<String, Integer>(menu));

            Subbranch mcSubbranch5 = new McSubbranch(-500, 0,
new HashMap<String, Integer>(menu));

            //以下设置职责链

             mcSubbranch4.setSuccessor(mcSubbranch5);

             mcSubbranch3.setSuccessor(mcSubbranch4);

            mcSubbranch2.setSuccessor(mcSubbranch3);

             mcSubbranch1.setSuccessor(mcSubbranch2);

            //小左开始订餐，假设小左的坐标是900,20

            Map<String, Integer> order = new HashMap<String,
Integer>();

            order.put("汉堡", 2);

            order.put("可乐", 1);

            order.put("薯条", 1);


            print(mcSubbranch1);

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


            //小左开始订餐，直接找mcSubbranch1的这一家分店订
餐即可
```

```
            mcSubbranch1.handleOrder(new Order(900, 20,
order));



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

            print(mcSubbranch1);

      }



      public static void print(Subbranch subbranch){

            if (subbranch == null ) {

                  return;

            }

            do {

                  if (subbranch instanceof McSubbranch) {

                        System.out.println("[" + subbranch
+ "]的菜单:" + ((McSubbranch) subbranch).getMenu());

                  }

            } while ((subbranch = ((McSubbranch)
subbranch).getNextSubbranch()) != null);

      }

}
```

“输出结果和刚才是一样的，不过这下我订餐就好办多了，直接找第一家分店订餐就行，至于到最后谁给我送餐，我就不用管了。”

“职责链模式果然强大啊，哈哈。不过现在还是有一点缺陷，那就是订餐的时候我还得记着应该找哪家分店，如果麦当劳有一个专门的订餐管理部门就好了，这样的话，就更容易找到该找谁订餐了。下面我就再添加一个订餐管理部门，完善一下吧。”

```
package com.chain;

//订餐管理部门，它并不是职责链模式中的角色

//但是在职责链模式中，通常情况下会有一个类专门负责维护职责链

//在本例中，这个类称为订餐管理部门更合适

public class OrderManager {

	private static OrderManager orderManager = new
OrderManager();

	private Subbranch head;

	private Subbranch last;

	private OrderManager(){}

	public static OrderManager getOrderManager(){

		return orderManager;

	}



	//注册分店

	public void registerSubbranch(Subbranch... subbranchs){

		for (Subbranch subbranch : subbranchs) {

			registerSubbranch(subbranch);
```

```
            }
        }
        public void registerSubbranch(Subbranch subbranch){
            if (head == null) {
                last = head = subbranch;
            }else {
                last.setSuccessor(subbranch);
                last = subbranch;
            }
        }

        public boolean handleOrder(Order order){
            return head.handleOrder(order);
        }
}
```

“其余的都不用变，这下我就使用订餐管理部门（可以理解为麦当劳总店或者总服务台都可以）订餐试一下吧。”

```
package com.chain;

import java.util.HashMap;
import java.util.Map;
```

```
public class Client {


      public static void main(String[] args) {

            //假设初始菜单都是以下这些东西

            Map<String, Integer> menu = new HashMap<String,
Integer>();

            menu.put("汉堡", 5);

            menu.put("薯条", 5);

            menu.put("可乐", 5);

            menu.put("雪碧", 5);

            //假设有5个分店

            Subbranch mcSubbranch1 = new McSubbranch(0, 0,
new HashMap<String, Integer>(menu));

            Subbranch mcSubbranch2 = new McSubbranch(100,
120, new HashMap<String, Integer>(menu));

            Subbranch mcSubbranch3 = new McSubbranch(-100,
-120, new HashMap<String, Integer>(menu));

            Subbranch mcSubbranch4 = new McSubbranch(1000,
20, new HashMap<String, Integer>(menu));

            Subbranch mcSubbranch5 = new McSubbranch(-500, 0,
new HashMap<String, Integer>(menu));
```

```
            //注册5个分店
OrderManager.getOrderManager().registerSubbranch(mcSubbranch1
,mcSubbranch2,mcSubbranch3,mcSubbranch4,mcSubbranch5);


            //小左开始订餐，假设小左的坐标是900,20
            Map<String, Integer> order = new HashMap<String,
Integer>();
            order.put("汉堡", 2);
            order.put("可乐", 1);
            order.put("薯条", 1);


            print(mcSubbranch1);
System.out.println("----------------------------------------
-");


            //小左开始订餐，直接找订餐管理部门订餐
OrderManager.getOrderManager().handleOrder(new Order(900, 20,
order));


System.out.println("----------------------------------------
-");
            print(mcSubbranch1);
        }
```

```
    public static void print(Subbranch subbranch){
        if (subbranch == null ) {
            return;
        }
        do {
            if (subbranch instanceof McSubbranch) {
                System.out.println("[" + subbranch + "]的菜单:" + ((McSubbranch) subbranch).getMenu());
            }
        } while ((subbranch = ((McSubbranch) subbranch).getNextSubbranch()) != null);
    }


}
```

“现在有了分店，直接向管理部门注册一下就好了，到时候有订单就会自动由管理部门一次分发下去了。这一次输出的结果和第一次应该也是一样的。这一下订餐就更方便了，我只管记着麦当劳订餐部门的电话就行，至于它到底有几个分店，我就完全不用管啦。”

“下面我就总结一下职责链的好处吧，到时候给麦当劳老总打个电话描述一下，哈哈。”

用了职责链模式之后，主要的好处是下面两点。

1、不再需要记忆所有分店的号码和联系方式，然后一个一个去订餐。

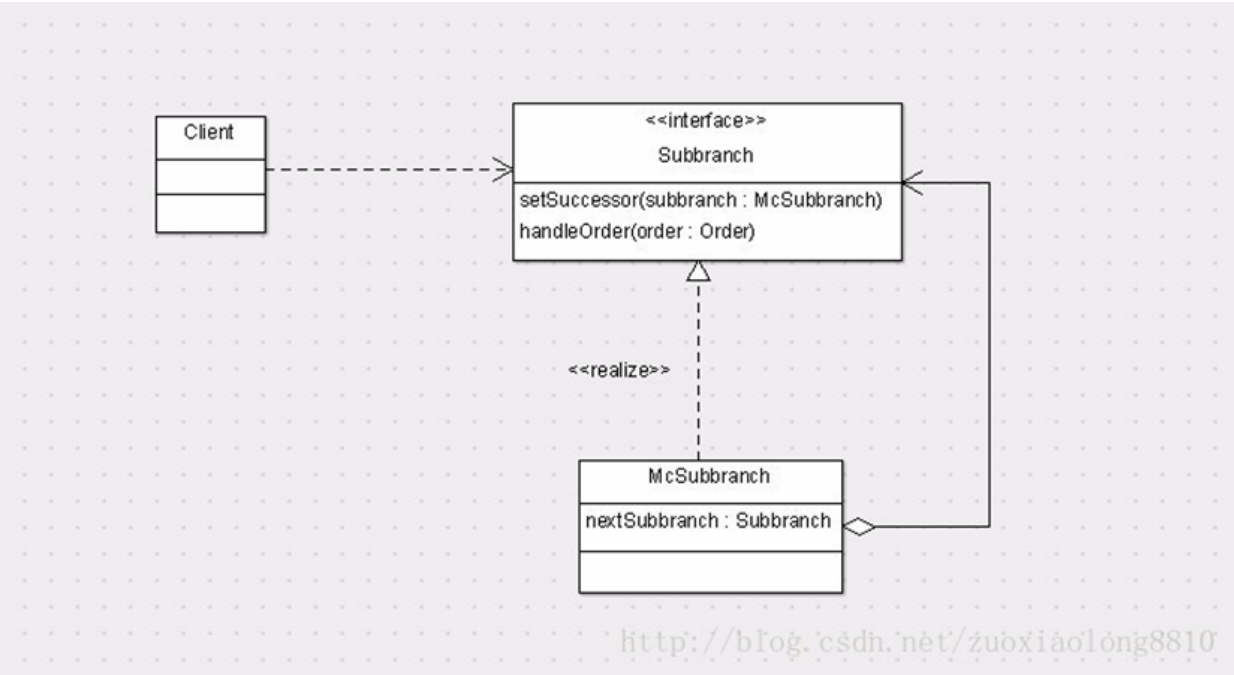
2、不需要知道麦当劳的内部管理结构，正因为这样，麦当劳再开多少分店，订餐的客户都不需要关心，而按照以前的方式，麦当劳每多开一个分店，客户都有可能需要多记忆一个分店的联系方式。

用专业点的语言来说，就是下面两点。

1、客户端与具体的处理者解耦，客户端只认识一个Hanlder接口，降低了客户端（即请求发送者）与处理者的耦合度。

2、客户端和处理者都不关心职责链的具体结构，而是交给职责链的创造者（在上述例子当中则是交给了OrderManager），也正因为如此，当在职责链中添加处理者的时候，这对客户端和处理者来说，都是透明的，二者不知道也不必要知道职责链的变化。

“职责链模式的好处应该就是这些啦，下面我来看看刚才的类图吧。”



“和标准的职责链模式类图几乎一模一样，少了一个具体的处理者，不过这并不影响类图所要传达的意义。不过这个类图也有点太简单了，不过也正因为简单，从类图里一眼就能看出来，职责链模式的精髓就在于那一条神奇的聚合线啊。”

“叮咚. . . ”

（二十一）状态模式详解（DOTA版）

本次LZ给各位介绍状态模式，之前在写设计模式的时候，引入了一些小故事，二十章职责连模式是故事版的最后一篇，之后还剩余四个设计模式，LZ会依照原生的方式去解释这几个设计模式，特别是原型模式和解释器模式，会包含一些其它的内容。

好了，接下来，我们先来看看状态模式的定义吧。

定义：(源于Design Pattern)：当一个对象的内在状态改变时允许改变其行为，这个对象看起来像是改变了其类。

上述是百度百科中对状态模式的定义，定义很简单，只有一句话，请各位形象的去理解这句话，它说当状态改变时，这个对象的行为也会变，而看起来就像是这个类改变了一样。

这正是应验了我们那句话，有些人一旦发生过什么事以后，就像变了个人似的，这句话其实与状态模式有异曲同工之妙。

我们仔细体会一下定义当中的要点。

1、有一个对象，它是有状态的。

2、这个对象在状态不同的时候，行为不一样。

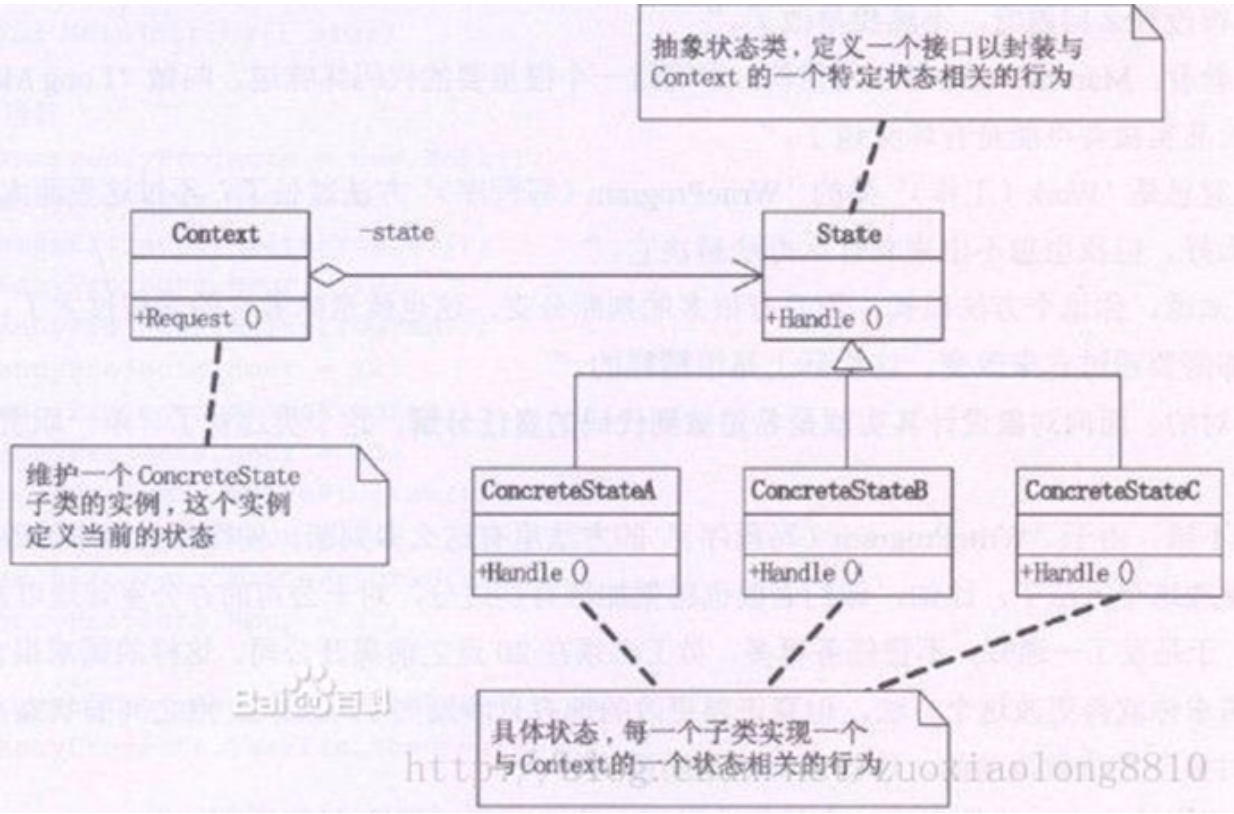
3、这些状态是可以切换的，而非毫无关系。

前两点比较好理解，第3点有时候容易给人比较迷惑的感觉，什么叫这些状态是可以切换的，而非毫无关系？

举个例子，比如一个人的状态，可以有很多，像生病和健康，这是两个状态，这是有关系并且可以转换的两个状态。再比如，睡觉、上班、休息，这也算是一组状态，这三个状态也是有关系的并且可以互相转换。

那如果把生病和休息这两个状态放在一起，就显得毫无意义了。所以这些状态应该是一组相关并且可互相切换的状态。

下面我们来看看状态模式的类图。



类图中包含三个角色。

Context：它就是那个含有状态的对象，它可以处理一些请求，这些请求最终产生的响应会与状态相关。

State：状态接口，它定义了每一个状态的行为集合，这些行为会在Context中得以使用。

ConcreteState：具体状态，实现相关行为的具体状态类。

如果针对刚才对于人的状态的例子来分析，那么人（Person）就是Context，状态接口依然是状态接口，而具体的状态类，则可以是睡觉，上班，休息，这一系列状态。

不过LZ也看过不少状态模式的文章和帖子，包括《大话设计模式》当中，都举的是有关人的状态的例子，所以这里给大家换个口味，我们换一个例子。

我们来试着写一个DOTA的例子，最近貌似跟DOTA干上了，不为其他，就因为DOTA伴随了LZ四年的大学时光。

玩过的朋友都知道，DOTA里的英雄有很多状态，比如正常，眩晕，加速，减速等等。相信就算没有玩过DOTA的朋友们，在其它游戏里也能见到类似的情况。那么假设我们的DOTA没有使用状态模式，则我们的英雄类会非常复杂和难以维护，我们来看下，原始版的英雄类是怎样的。

```
package com.state;



//英雄类

public class Hero {


	public static final int COMMON = 1;//正常状态


	public static final int SPEED_UP = 2;//加速状态


	public static final int SPEED_DOWN = 3;//减速状态


	public static final int SWIM = 4;//眩晕状态
```

```
private int state = COMMON;//默认是正常状态

private Thread runThread;//跑动线程

//设置状态
public void setState(int state) {
	this.state = state;
}
//停止跑动
public void stopRun() {
	if (isRunning()) runThread.interrupt();
	System.out.println("--------------停止跑动----------------");
}
//开始跑动
public void startRun() {
	if (isRunning()) {
		return;
	}
	final Hero hero = this;
```

```
            runThread = new Thread(new Runnable() {
                  public void run() {
                        while (!runThread.isInterrupted()) {
     try {
                              hero.run();
                        } catch (InterruptedException
e) {
                              break;
                        }
                  }
            }
      });
      System.out.println("--------------开始跑动-------
--------");
      runThread.start();
}


private boolean isRunning(){
      return runThread != null &&
!runThread.isInterrupted();
}
//英雄类开始奔跑
```

```
        private void run() throws InterruptedException{
                if (state == SPEED_UP) {
System.out.println("--------------加速跑动---------------");
                        Thread.sleep(4000);//假设加速持续4秒
                        state = COMMON;
                        System.out.println("------加速状态结束，变为正常状态------");
                }else if (state == SPEED_DOWN) {
System.out.println("--------------减速跑动---------------");
                        Thread.sleep(4000);//假设减速持续4秒
                        state = COMMON;
                        System.out.println("------减速状态结束，变为正常状态------");
                }else if (state == SWIM) {
System.out.println("--------------不能跑动---------------");
                        Thread.sleep(2000);//假设眩晕持续2秒
                        state = COMMON;
                        System.out.println("------眩晕状态结束，变为正常状态------");
                }else {
                        //正常跑动则不打印内容，否则会刷屏
                }
```

```
    }


}
```

下面我们写一个客户端类，去试图让英雄在各种状态下奔跑一下。

```
package com.state;


public class Main {


    public static void main(String[] args) throws InterruptedException {
        Hero hero = new Hero();
        hero.startRun();
        hero.setState(Hero.SPEED_UP);
        Thread.sleep(5000);
        hero.setState(Hero.SPEED_DOWN);
        Thread.sleep(5000);
        hero.setState(Hero.SWIM);
        Thread.sleep(5000);
        hero.stopRun();
    }
```

```
}
```

可以看到，我们的英雄在跑动过程中随着状态的改变，会以不同的状态进行跑动。

在上面原始的例子当中，我们的英雄类当中有明显的if else结构，我们再来看看百度百科中状态模式所解决的问题的描述。

状态模式解决的问题：状态模式主要解决的是当控制一个对象状态的条件表达式过于复杂时的情况。把状态的判断逻辑转移到表示不同状态的一系列类中，可以把复杂的判断逻辑简化。

不用说，状态模式是可以解决我们上面的if else结构的，我们采用状态模式，利用多态的特性可以消除掉if else结构。这样所带来的好处就是可以大大的增加程序的可维护性与扩展性。

下面我们就使用状态模式对上面的例子进行改善，首先第一步，就是我们需要定义一个状态接口，这个接口就只有一个方法，就是run。

```
package com.state;


public interface RunState {


	void run(Hero hero);


}
```

与状态模式类图不同的是，我们加入了一个参数Hero（Context），这样做的目的是为了具体的状态类当达到某一个条件的时候可以切换上下文的状态。下面列出四个具体的状态类，其实就是把if else拆掉放到这几个类的run方法中。

```
package com.state;

public class CommonState implements RunState{

	public void run(Hero hero) {
		//正常跑动则不打印内容，否则会刷屏
	}

}

package com.state;

public class SpeedUpState implements RunState{

	public void run(Hero hero) {
		System.out.println("--------------加速跑动----------------");
		try {
			Thread.sleep(4000);//假设加速持续4秒
		} catch (InterruptedException e) {}
		hero.setState(Hero.COMMON);
```

```
            System.out.println("------加速状态结束，变为正常状态------");
      }

}

package com.state;

public class SpeedDownState implements RunState{

      public void run(Hero hero) {
            System.out.println("--------------减速跑动----------------");
            try {
                  Thread.sleep(4000);//假设减速持续4秒
            } catch (InterruptedException e) {}
            hero.setState(Hero.COMMON);
            System.out.println("------减速状态结束，变为正常状态------");
      }

}
```

```
package com.state;


public class SwimState implements RunState{


      public void run(Hero hero) {

            System.out.println("--------------不能跑动--------
--------");

            try {

                  Thread.sleep(2000);//假设眩晕持续2秒

            } catch (InterruptedException e) {}

            hero.setState(Hero.COMMON);

            System.out.println("------眩晕状态结束，变为正常
状态------");

      }


}
```

这下我们的英雄类也要相应的改动一下，最主要的改动就是那些if else可以删掉了，如下。

```
package com.state;


//英雄类
```

```
public class Hero {

	public static final RunState COMMON = new CommonState();//正常状态

	public static final RunState SPEED_UP = new SpeedUpState();//加速状态

	public static final RunState SPEED_DOWN = new SpeedDownState();//减速状态

	public static final RunState SWIM = new SwimState();//眩晕状态

	private RunState state = COMMON;//默认是正常状态

	private Thread runThread;//跑动线程

	//设置状态
	public void setState(RunState state) {
		this.state = state;
	}
```

```
	//停止跑动
	public void stopRun() {
		if (isRunning()) runThread.interrupt();
		System.out.println("--------------停止跑动--------------");
	}
	//开始跑动
	public void startRun() {
		if (isRunning()) {
			return;
		}
		final Hero hero = this;
		runThread = new Thread(new Runnable() {
			public void run() {
				while (!runThread.isInterrupted()) {
					state.run(hero);
				}
			}
		});
		System.out.println("--------------开始跑动--------------");
```

```
            runThread.start();

        }



        private boolean isRunning(){

            return runThread != null &&
!runThread.isInterrupted();

        }



}
```

可以看到，现在我们的英雄类优雅了许多，我们使用刚才同样的客户端运行即可得到同样的结果。

对比我们的原始例子，现在我们使用状态模式之后，有几个明显的优点：

一、我们去掉了if else结构，使得代码的可维护性更强，不易出错，这个优点挺明显，如果试图让你更改跑动的方法，是刚才的一堆if else好改，还是分成了若干个具体的状态类好改呢？答案是显而易见的。

二、使用多态代替了条件判断，这样我们代码的扩展性更强，比如要增加一些状态，假设有加速20%，加速10%，减速10%等等等（这并不是虚构，DOTA当中是真实存在这些状态的），会非常的容易。

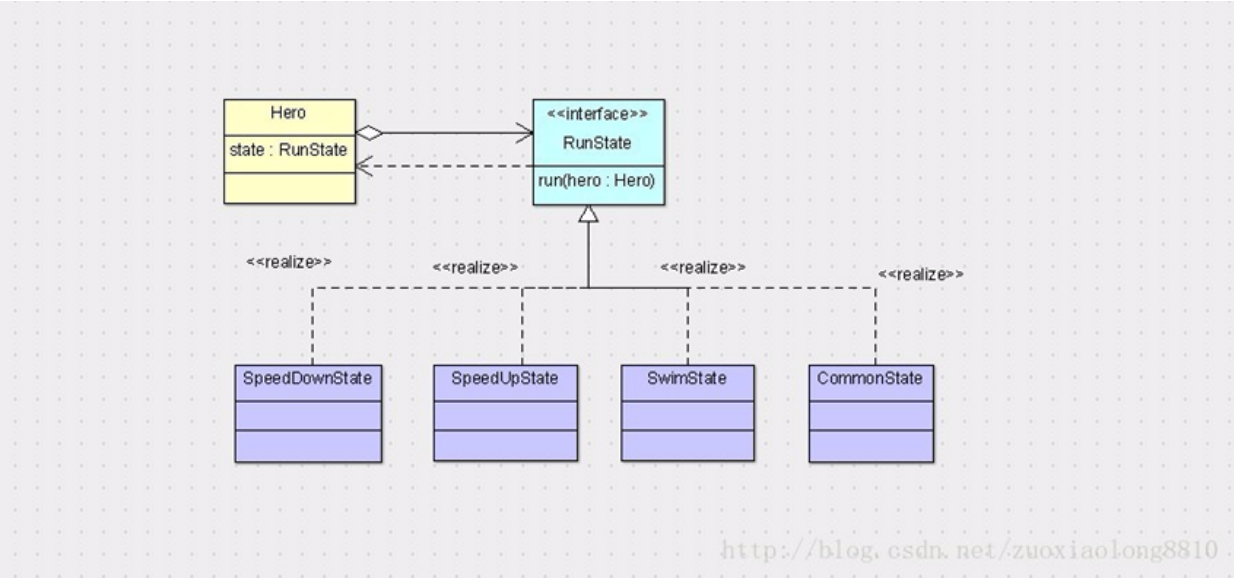
三、状态是可以被共享的，这个在上面的例子当中有体现，看下Hero类当中的四个static final变量就知道了，因为状态类一般是没有自己的内部状态的，所有它只是一个具有行为的对象，因此是可以被共享的。

四、状态的转换更加简单安全，简单体现在状态的分割，因为我们把一堆if else分割成了若干个代码段分别放在几个具体的状态类当中，

所以转换起来当然更简单，而且每次转换的时候我们只需要关注一个固定的状态到其他状态的转换。安全体现在类型安全，我们设置上下文的状态时，必须是状态接口的实现类，而不是原本的一个整数，这可以杜绝魔数以及不正确的状态码。

状态模式适用于某一个对象的行为取决于该对象的状态，并且该对象的状态会在运行时转换，又或者有很多的if else判断，而这些判断只是因为状态不同而不断的切换行为。

上面的适用场景是很多状态模式的介绍中都提到的，下面我们就来看下刚才DOTA中，英雄例子的类图。



可以看到，这个类图与状态模式的标准类图是几乎一模一样的，只是多了一条状态接口到上下文的依赖线，而这个是根据实际需要添加的，而且一般情况下都是需要的。

状态模式也有它的缺点，不过它的缺点和大多数模式相似，有两点。

1、会增加的类的数量。

2、使系统的复杂性增加。

尽管状态模式有着这样的缺点，但是往往我们牺牲复杂性去换取的高可维护性和扩展性是相当值得的，除非增加了复杂性以后，对于后者的提升会乎其微。

状态模式在项目当中也算是较经常会碰到的一个设计模式，但是通常情况下，我们还是在看到if else的情况下，对项目进行重构时使用，又或者你十分确定要做的项目会朝着状态模式发展，一般情况下，还是不建议在项目的初期使用。

好了，本次状态模式的分享就到此结束了，希望各位有所收获。

（二十二）访问者模式详解（伪动态双分派）

本次LZ和各位分享一下访问者模式，从场景、设计初衷以及实现方面来说，访问者模式算是LZ即将写到的24种设计模式当中，最复杂也是最难理解的一个设计模式。

针对这样一个设计模式，LZ到底该如何和各位分享呢？

废话不多说，我们先来看看访问者模式的定义。

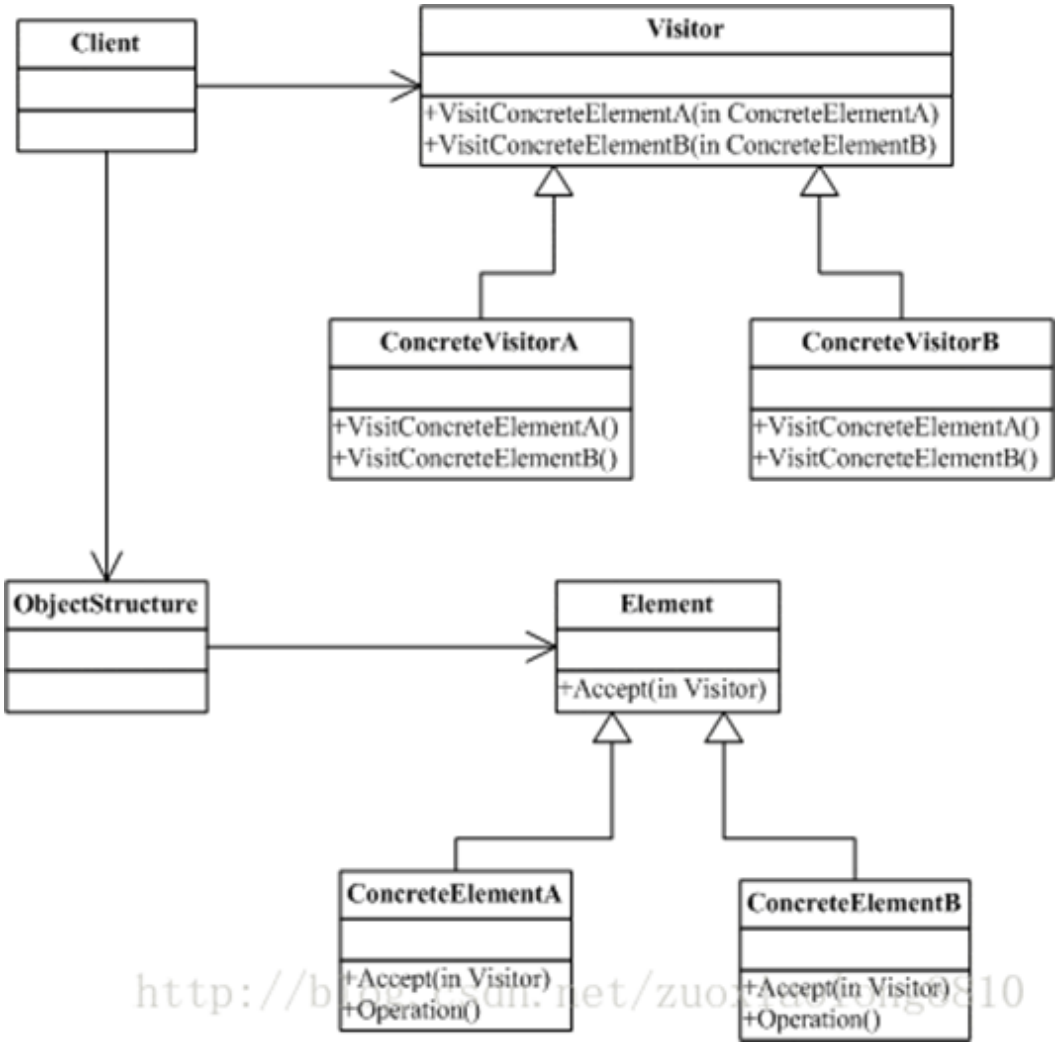
定义（源于GoF《Design Pattern》）：表示一个作用于某对象结构中的各元素的操作。它使你可以在不改变各元素类的前提下定义作用于这些元素的新操作。

初次接触，定义会显得晦涩并且难于理解，没关系，LZ来陪着各位一起一点一点分析定义中所提到的关键点。

先来看第一句话，说是一个作用于某对象结构中的各元素的操作，这里提到了三个事物，一个是对象结构，一个是各元素，一个是操作。那么我们可以这么理解，有这么一个操作，它是作用于一些元素之上的，而这些元素属于某一个对象结构。

好了，最关键的第二句来了，它说使用了访问者模式之后，可以让我们在不改变各元素类的前提下定义作用于这些元素的新操作。这里面的关键点在于前半句，即不改变各元素类的前提下，在这个前提下定义新操作是访问者模式精髓中的精髓。

下面我们来看看访问者模式的类图，去寻找一下访问者模式的角色都有哪些。



我们从类图中可以找到五类角色，两个接口，两种实现类以及一个对象结构。

Visitor接口：它定义了对每一个元素（Element）访问的行为，它的参数就是可以访问的元素，它的方法个数理论上来讲与元素个数（Element的实现类个数）是一样的，从这点不难看出，访问者模式要求元素类的个数不能改变（不能改变的意思是说，如果元素类的个数经常改变，则说明不适合使用访问者模式）。

ConcreteVisitor：具体的访问者，它需要给出对每一个元素类访问时所产生的具体行为。

Element接口：元素接口，它定义了一个接受访问者（accept）的方法，其意义是指，每一个元素都要可以被访问者访问。

ConcreteElement：具体的元素类，它提供接受访问方法的具体实现，而这个具体的实现，通常情况下是使用访问者提供的访问该元素类的方法。

ObjectStructure：这个便是定义当中所提到的对象结构，对象结构是一个抽象表述，具体点可以理解为一个具有容器性质或者复合对象特性的类，它会含有一组元素（Element），并且可以迭代这些元素，供访问者访问。

在上面五个角色当中，最重要的就是最后一个，所谓的访问者模式，就是为了让访问者可以方便的访问对象结构而存在的。关于访问者模式的例子，很多文章和文献使用男人和女人的例子，所以LZ这里就不重复了。

我们来看一个财务方面的简单例子，我们都知道财务都是有账本的，这个账本就可以作为一个对象结构，而它其中的元素有两种，收入和支出，这满足我们访问者模式的要求，即元素的个数是稳定的，因为账本中的元素只能是收入和支出，就像人只有男人和女人一样（尽管现在貌似经常出现第三种，具体第三种是什么请大家自行补脑）。

账本的访问者都会有哪些呢？

LZ不才，对财务只有一个直观的理解，不过这已经足够我们来做例子了，依LZ的了解，查看账本的人可能有这样几种，比如老板，会计事务所的注会，财务主管，等等。而这些人在看账本的时候显然目的和行为是不同的。

我们接下来就试着写一下这个例子，首先我们给出单子的接口，它只有一个方法accept。

```
package com.visitor;

//单个单子的接口（相当于Element）

public interface Bill {
```

```
    void accept(AccountBookViewer viewer);


}
```

其中的方法参数AccountBookViewer是一个账本访问者接口，这个接口稍后给出，我们先给出两个具体的元素，也就是收入单子和消费单子，或者说收入和支出类。

```
package com.visitor;


//消费的单子
public class ConsumeBill implements Bill{


    private double amount;


    private String item;


    public ConsumeBill(double amount, String item) {
        super();
        this.amount = amount;
        this.item = item;
    }
```

```
	public void accept(AccountBookViewer viewer) {
		viewer.view(this);
	}

	public double getAmount() {
		return amount;
	}

	public String getItem() {
		return item;
	}

}


package com.visitor;

//收入单子
public class IncomeBill implements Bill{
```

```
private double amount;

private String item;

public IncomeBill(double amount, String item) {
      super();
      this.amount = amount;
      this.item = item;
}

public void accept(AccountBookViewer viewer) {
      viewer.view(this);
}

public double getAmount() {
      return amount;
}

public String getItem() {
      return item;
```

```
    }

}
```

这两个类没什么特别的，为了使得例子更加形象贴近实际，LZ加入了两个属性，一个是金额，一个是单子的项目，而最关键的还是里面的accept方法，它直接让访问者访问自己，这相当于一次静态分派，当然我们也可以不使用重载而直接给方法不同的名称，这些有关分派的知识LZ稍后介绍。

下面我们给出刚才出现过的账本访问者接口，它有两个方法，如下。

```
package com.visitor;


//账单查看者接口（相当于Visitor）
public interface AccountBookViewer {

    //查看消费的单子
    void view(ConsumeBill bill);

    //查看收入的单子
    void view(IncomeBill bill);

}
```

这两个方法是重载方法，就是在上面的元素类当中用到的，当然你也可以按照访问者模式类图当中的方式去做，将两个方法分别命名为

viewConsumeBill和viewIncomeBill，而一般是建议按照类图上来做的，不过无论怎么写，这并不影响访问者模式的使用。

下面我们给出两个访问者的例子，当然访问者可能会有很多，但是作为例子，我们并不需要写太多，这些访问者都需要实现上面的接口，并且提供两个view方法，也就是他们针对消费的单子和收入的单子都分别要做些什么。

```
package com.visitor;



//老板类，查看账本的类之一

public class Boss implements AccountBookViewer{



	private double totalIncome;



	private double totalConsume;



	//老板只关注一共花了多少钱以及一共收入多少钱，其余并不关心

	public void view(ConsumeBill bill) {

		totalConsume += bill.getAmount();

	}



	public void view(IncomeBill bill) {

		totalIncome += bill.getAmount();
```

```
    }

    public double getTotalIncome() {
        System.out.println("老板查看一共收入多少，数目是：" + totalIncome);
        return totalIncome;
    }

    public double getTotalConsume() {
        System.out.println("老板查看一共花费多少，数目是：" + totalConsume);
        return totalConsume;
    }

}

package com.visitor;

//注册会计师类，查看账本的类之一
public class CPA implements AccountBookViewer{
```

```
        //注会在看账本时，如果是支出，则如果支出是工资，则需要看应该交的税交了没

        public void view(ConsumeBill bill) {

                if (bill.getItem().equals("工资")) {

            System.out.println("注会查看账本时，如果单子的消费目的是发工资，则注会会查看有没有交个人所得税。");

                }

        }

        //如果是收入，则所有的收入都要交税

        public void view(IncomeBill bill) {

                System.out.println("注会查看账本时，只要是收入，注会都要查看公司交税了没。");

        }



}
```

可以看到，这两个类有巨大的差异，老板只关心收入和支出的总额，而注会只关注该交税的是否交税，当然现实当中，二者可能关注的不只是这些，甚至完全不是这些，不过作为例子，这并不是我们关注的重点。

下面该出场的是我们最重要的一个类，账本类，它是当前访问者模式例子中的对象结构，如下。

```
package com.visitor;



import java.util.ArrayList;
```

```
import java.util.List;


//账本类（相当于ObjectStruture）
public class AccountBook {
	//单子列表
	private List<Bill> billList = new ArrayList<Bill>();
	//添加单子
	public void addBill(Bill bill){
		billList.add(bill);
	}
	//供账本的查看者查看账本
	public void show(AccountBookViewer viewer){
		for (Bill bill : billList) {
			bill.accept(viewer);
		}
	}
}
```

可以看到，我们的账本类当中有一个列表，这个列表是元素（Bill）的集合，这便是对象结构的通常表示，它一般会是一堆元素的集合，不过这个集合不一定是列表，也可能是树，链表等等任何数据结构，甚至是若干个数据结构。

有一些文章当中的例子，对象结构还有remove方法，不过这里由于账本比较特殊，是不能删除的，所以为了在简单的基础上尽量与实际情况贴近，LZ就没有加入remove方法。至于show方法，就是账本类的精髓了，它会枚举每一个元素，让访问者访问。

下面我们给出一个简单的客户端，测试一下这个访问者模式。

```
package com.visitor;



public class Client {



      public static void main(String[] args) {

            AccountBook accountBook = new AccountBook();

            //添加两条收入

            accountBook.addBill(new IncomeBill(10000, "卖商
品"));

            accountBook.addBill(new IncomeBill(12000, "卖广告
位"));

            //添加两条支出

            accountBook.addBill(new ConsumeBill(1000, "工
资"));

            accountBook.addBill(new ConsumeBill(2000, "材料
费"));



            AccountBookViewer boss = new Boss();

            AccountBookViewer cpa = new CPA();
```

```
            //两个访问者分别访问账本
            accountBook.show(cpa);
            accountBook.show(boss);

            ((Boss) boss).getTotalConsume();
            ((Boss) boss).getTotalIncome();
        }
}
```

可以看到，两个访问者老板和注会对账本的查看，行为是完全不同的，但是这正是访问者模式的意义所在，它其实是将访问者这部分逻辑独立出去，让其自生自灭。我们可以直观的去理解，上面的代码中，账本以及账本中的元素是非常稳定的，这些几乎不可能改变，而最容易改变的就是访问者这部分。

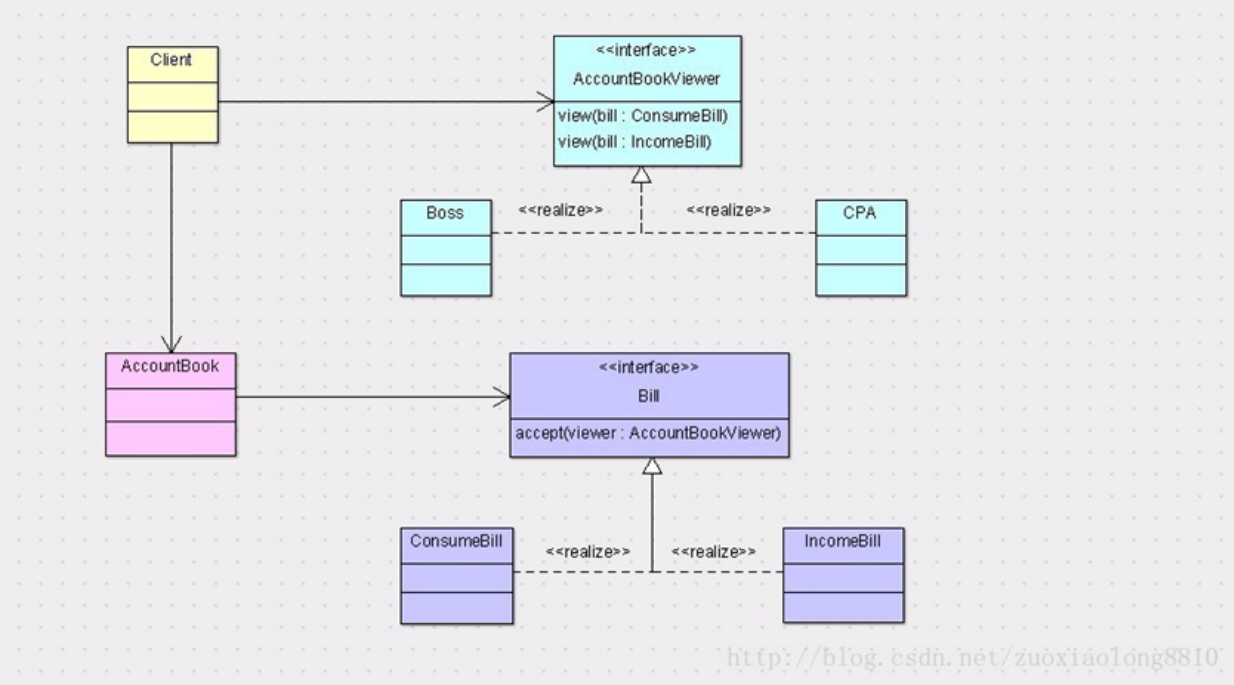
访问者模式最大的优点就是增加访问者非常容易，我们从代码上来看，如果要增加一个访问者，你只需要做一件事即可，那就是写一个类，实现AccountBookViewer接口，然后就可以直接调用AccountBook的show方法去访问账本了。

很简单吧！

如果没使用访问者模式，各位可以自己写一下，你一定会在show方法中写一堆if else，就算不是在show方法，你也一定逃不掉这些if else，它们一定会出现。而且每增加一个访问者，你都需要改你的if

else，代码会显得非常臃肿，而且非常难以扩展和维护。这里由于本章内容较多，限于篇幅，LZ就不给各位写了。

在这里，LZ给出上面的例子的类图供各位参考，类图与标准的类图一模一样。



下面LZ结合上面的例子，给各位分析一下百度百科中对访问者模式的一些描述。

访问者模式的几个特点：

1、访问者模式把数据结构和作用于结构上的操作解耦合，使得操作集合可相对自由地演化。

2、访问者模式适用于数据结构相对稳定算法又易变化的系统。因为访问者模式使得算法操作增加变得容易。若系统数据结构对象易于变化，经常有新的数据对象增加进来，则不适合使用访问者模式。

3、访问者模式的优点是增加操作很容易，因为增加操作意味着增加新的访问者。访问者模式将有关行为集中到一个访问者对象中，其改变不影响系统数据结构。其缺点就是增加新的数据结构很困难。

第一点，数据结构和作用于结构上的操作解耦合，使操作集合可以相对自由的演化，这在上面的例子当中指的是，我们把账本以及账本的元素与查看账本的人解耦，使得这些访问者的行为可以相对独立的变化，这点其实并不难理解。这一点其实说的是访问者模式的优点。

至于剩下的两点，开始提到访问者模式适用于数据结构相对稳定，而算法行为又易变化的系统，这点不难理解，试想一下，如果账本结构不稳定，经常有元素加进来，那么假设有了第三种非支出也非收入的单子，那我们需要做以下两件事。

1）添加一个类ABill，实现Bill接口。

2）在AccountBookViewer接口中添加一个方法view(ABill bill)，并且在所有AccountBookViewer接口的实现类中都增加view(ABill bill)方法的具体实现。

这其中第一件事并不难，而且也符合开闭原则，但是第二件事就值得商榷了。它修改了抽象，导致所有细节都跟着变化，这完全破坏了开闭原则。所以第二点说使用访问者模式的前提是数据结构相对稳定也就不奇怪了。

然而对于算法操作，在访问者模式的使用下，我们可以自由的添加，这个在上面已经提及到，也就是说我们如果要增加查看账本的类，是非常简单的，我们只需要写一个类去实现AccountBookViewer接口，这是开闭原则的完美诠释。

访问者模式中，元素的添加会破坏开闭原则，访问者的添加又符合开闭原则，所以有文献称该模式是倾斜的开闭原则，即一边是符合开闭原则的，一边又是破坏了开闭原则的，有点倾斜的感觉。

从上面的描述，我们可以大致总结出访问者模式的优缺点以及适用性。

优点：

1、使得数据结构和作用于结构上的操作解耦，使得操作集合可以独立变化。

2、添加新的操作或者说访问者会非常容易。

3、将对各个元素的一组操作集中在一个访问者类当中。

4、使得类层次结构不改变的情况下，可以针对各个层次做出不同的操作，而不影响类层次结构的完整性。

5、可以跨越类层次结构，访问不同层次的元素类，做出相应的操作。

缺点：

1、增加新的元素会非常困难。

2、实现起来比较复杂，会增加系统的复杂性。

3、破坏封装，如果将访问行为放在各个元素中，则可以不暴露元素的内部结构和状态，但使用访问者模式的时候，为了让访问者能获取到所关心的信息，元素类不得不暴露出一些内部的状态和结构，就像收入和支出类必须提供访问金额和单子的项目的方法一样。

适用性：

1、数据结构稳定，作用于数据结构的操作经常变化的时候。

2、当一个数据结构中，一些元素类需要负责与其不相关的操作的时候，为了将这些操作分离出去，以减少这些元素类的职责时，可以使用访问者模式。

3、有时在对数据结构上的元素进行操作的时候，需要区分具体的类型，这时使用访问者模式可以针对不同的类型，在访问者类中定义不同的操作，从而去除掉类型判断。

上面优点当中的前三点比较好理解，缺点和适用性当中提到的三点也比较好理解，如果优点当中的后两点不理解也没关系，LZ会在后面给出简单的例子。

在给出后两点优点的例子之前，不得不先解释一下前面提到的分派，分派按照分派的方式可以分为静态分派和动态分派，按照宗量或者说

判断依据的多少，可以分为单分派和多分派，下面LZ简单的介绍下这几种分派方式。

静态分派以及多分派：

静态分派就是按照变量的静态类型进行分派，从而确定方法的执行版本，静态分派在编译时期就可以确定方法的版本。而静态分派最典型的应用就是方法重载，考虑下面一段程序。

```
public class Main {


	public void test(String string){

		System.out.println("string");

	}


	public void test(Integer integer){

System.out.println("integer");

	}


	public static void main(String[] args) {

		String string = "1";

		Integer integer = 1;

		Main main = new Main();

		main.test(integer);
```

```
            main.test(string);

        }

}
```

相信运行结果不需要LZ给出了，会依次打印integer和string，对于test方法，会根据静态类型决定方法版本，而所判断的依据就是，在main类型确定之后，依据test方法的参数类型和参数数量，我们就可以唯一的确定一个重载方法的版本。比如上面的例子，我们确定完main的类型之后，就可以根据test方法是一个参数，并且这个参数是Integer类型还是String类型，就可以确定到底调用哪个重载方法了。

可以看到，在静态分派判断的时候，我们根据多个判断依据（即参数类型和个数）判断出了方法的版本，那么这个就是多分派的概念，因为我们有一个以上的考量标准，也可以称为宗量。所以JAVA是静态多分派的语言。

动态分派以及单分派：

对于动态分派，与静态相反，它不是在编译期确定的方法版本，而是在运行时才能确定。而动态分派最典型的应用就是多态的特性，考虑下面一段程序。

```
package com.visitor1;



interface Person{

        void test();

}

class Man implements Person{

        public void test(){

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

```
        }
}
class Woman implements Person{
        public void test(){
                System.out.println("女人");
        }
}
public class Main {


        public static void main(String[] args) {
                Person man = new Man();
                Person woman = new Woman();
                man.test();
                woman.test();
        }
}
```

这段程序输出结果为依次打印男人和女人，这应该是所有人都预料到的，然而现在的test方法版本，就无法根据man和woman的静态类型去判断了，他们的静态类型都是Person接口，根本无从判断。

显然，产生的输出结果，就是因为test方法的版本是在运行时判断的，这就是动态分派。

动态分派判断的方法是在运行时获取到man和woman的实际引用类型，再确定方法的版本，而由于此时判断的依据只是实际引用类型，只有一个判断依据，所以这就是单分派的概念，这时我们的考量标准只有一个宗量，即变量的实际引用类型。相应的，这说明JAVA是动态单分派的语言。

访问者模式中的伪动态双分派：

上面LZ已经简单的介绍了JAVA语言中的静态多分派和动态单分派，更详细的解释各位可以在其它文献和文章中寻找，这里我们谈谈访问者模式的分派方式。

标题上已经注明了，访问者模式中使用的是伪动态双分派，之所以加了一个“伪”字，是因为一个模式当然不可能更改语言的特性，所以JAVA是动态单分派的语言这点毋庸置疑，而访问者模式只是利用了一些手段达到了看似双分派的效果。

对于动态双分派这个词语，初次接触的读友或许比较迷惑，LZ先给大家一个初步的解释，动态双分派就是在运行时依据两个实际类型去判断一个方法的运行行为，而访问者模式实现的手段是进行了两次动态单分派来达到这个效果。

我们来看上面例子当中账本类中的accept方法的调用。

```
for (Bill bill : billList) {

                    bill.accept(viewer);

            }
```

上面说了访问者模式是使用两次动态单分派达到了依据两个实际类型在运行时判断一个方法版本的效果，那么对于我们现在的例子来说，就是依据biil和viewer两个实际类型决定了view方法的版本，从而决定了accept方法的动作。请注意LZ的用词，是决定accept方法的动作以及决定了view方法的版本。

为什么要强调是accept方法的动作而不是方法的版本，是因为accept方法的版本只需要一次动态分派就可以确定，但是它所产生的动作却需要两次动态分派才能确定。

我们来看下这个accept方法的调用过程，LZ还是分步骤给各位解释。

1、当调用accept方法时，根据bill的实际类型决定是调用ConsumeBill还是IncomeBill的accept方法。

2、这时accept方法的版本已经确定，假如是ConsumeBill，它的accept方法是调用下面这行代码。

```
public void accept(AccountBookViewer viewer) {

      viewer.view(this);

}
```

此时的this是ConsumeBill类型，所以对应于AccountBookViewer接口的view(ConsumeBill bill)方法，此时需要再根据viewer的实际类型确定view方法的版本，如此一来，就完成了动态双分派的过程。

以上的过程就是通过两次动态双分派，第一次对accept方法进行动态分派，第二次对view（类图中的visit方法）方法进行动态分派，从而达到了根据两个实际类型确定一个方法的行为的效果。

而原本我们的做法，通常是传入一个接口，直接使用该接口的方法，此为动态单分派，就像策略模式一样。在这里，show方法传入的viewer接口并不是直接调用自己的view方法，而是通过bill的实际类型先动态分派一次，然后在分派后确定的方法版本里再进行自己的动态分派。

在此之外，还需要再解释一点，在上面第2步，确定view(ConsumeBill bill)方法是静态分派决定的，所以这个并不在此次动态双分派的范畴内，而且静态分派是在编译期就完成的，也就是说，在上述第1步之前就已经完成了对view(ConsumeBill bill)方法版本的选取。况且把静态分派算在内的话，由于静态分派是多分派，这里就不能叫双分派

了，应该叫动态多分派，这显然是不成立的。所以view(ConsumeBill bill)方法的静态分派与访问者模式的动态双分派并没有任何关系。

而且退一步讲，我们完全可以将AccountBookViewer接口中的两个view方法取不同的名字，这样也就完全避免了方法版本确定中静态分派参与的嫌疑，而且这完全不影响访问者模式的效果，可以清楚的看到，标准类图中也是这么建议的。这里LZ写成一样的名字，只是为了方便和更加清晰，而且在只有两个方法的时候这么做也并无不可，但是如果方法多的时候，强烈建议不要取一样的名字，由于静态分派的重载版本往往不是唯一的，所以重载版本过多会造成一定的干扰。

LZ的例子只是为了更加清晰的展示访问者模式，在实际应用中，还是强烈建议各位使用不同的方法名称去命名各个元素的访问方法。

在这里LZ解释静态分派不参与双分派的原因，是因为看到不少文章都多多少少在暗示访问者模式的双分派与view(this)的静态分派相关，这个观点恕LZ不能苟同，动态双分派说到底还是动态分派，是在运行时发生的，它与静态分派有着本质上的区别，不可以说一次动态分派加一次静态分派就是动态双分派，而且访问者模式的双分派本身也是另有所指。

你可能会说，this的类型是动态确定的，这么算下来不也是根据AccountBookViewer的实际类型和Bill的实际类型做了一次动态双分派吗？

答案是当然不能这么算，this的类型可不是动态确定的，你写在哪个类当中，它的静态类型就是哪个类，这是在编译期就确定的，不确定的是它的实际类型，请各位区分开这一点。

如果各位只是为了暂时理解访问者模式的双分派，这样理解倒也不是不可，以后随着理解的加深，会渐渐更加清晰。不过如果你这么理解访问者模式的双分派，一定要搞清楚一点，那就是this的类型是在编译期就可以确定的，而不是在运行时动态确定的，它并不是真正的动态绑定，而且说到底它始终是一个参数类型，参与分派的是它的静态类型，依旧是静态分派的范畴。总之怎么理解是自己的事，LZ只是试图点破一下这里面的分派关系。

上面的分析算是一个铺垫，至于后面对于静态分派的分析，各位可以当做一个讨论，也可以在文章下方发表评论参与进来，LZ不胜欢迎。

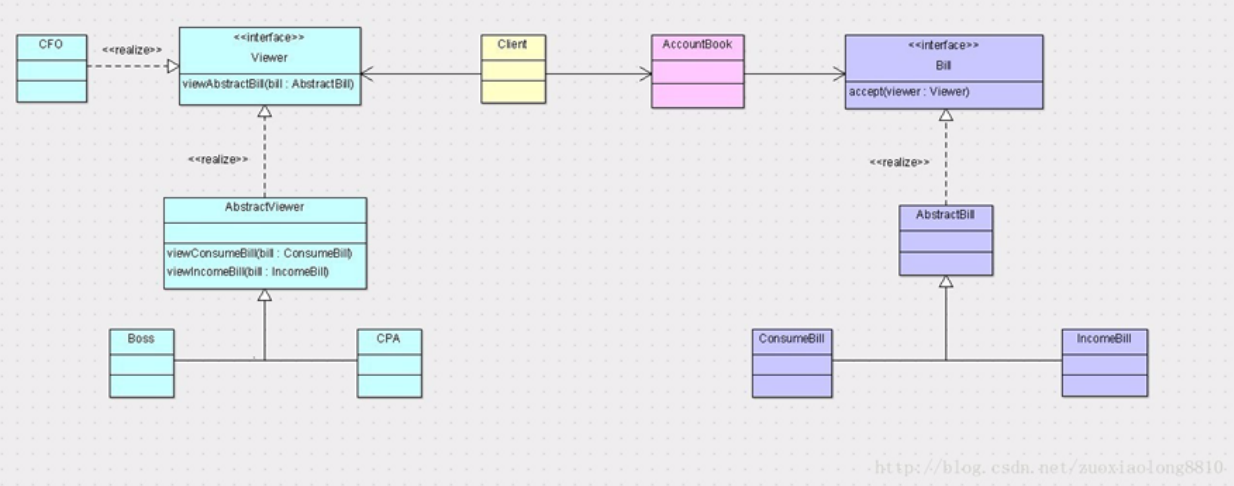
接下来我们还有最后一件事没完成了，那就是前面提到的优点的后两点的体现，因为在前面的例子当中没有用到，所以可能会理解起来会有点困难，这里LZ把上面的例子优化一下，让各位体验一下优点中的后两点是多么强大。

我们先来考虑一个问题，假设我们上面的例子当中再添加一个财务主管，而财务主管不管你是支出还是收入，都要详细的查看你的单子的项目以及金额，简单点说就是财务主管类的两个view方法的代码是一样的。

你可能会说，这好办啊，我们可以复制一下嘛。是的，这当然是一个办法，不过相信对代码有要求的程序猿们一定不喜欢他们的代码里出现一堆的重复代码。而且假设有很多人的访问方式和财务主管一样，对收入和支出的操作一样，那得复制多少代码。

解决方案就是我们可以将元素提炼出层次结构，针对层次结构提供操作的方法，这样就实现了优点当中最后两点提到的针对层次定义操作以及跨越层次定义操作。

这次LZ跟以前的步骤反过来一下，先给出LZ设计的类图，然后我们再来一步步实现针对层次结构做出的访问者模式。



可以看出，我们针对刚才的实现体系，分别抽象出一个层次，现在我们便可以针对层次定义操作了，左边的CFO（财务主管类）便是针对高层次定义的操作，它会统一作用于AbstractBill。

为了支持层次操作，我们需要在处于最低层次的元素类当中添加一些层次结构的条件判断，不过这个是可以接受的，因为这段代码不会被更改，这是符合开闭原则的。

下面LZ直接将代码贴上来，各位可以从中体会一下层次的感觉。

```
package com.visitor;

//单个单子的接口（相当于Element）

public interface Bill {



	void accept(Viewer viewer);



}



package com.visitor;



//抽象单子类，一个高层次的单子抽象

public abstract class AbstractBill implements Bill{



	protected double amount;
```

```
	protected String item;

	public AbstractBill(double amount, String item) {
		super();
		this.amount = amount;
		this.item = item;
	}

	public double getAmount() {
		return amount;
	}

	public String getItem() {
		return item;
	}

}

package com.visitor;
```

```
//收入单子
public class IncomeBill extends AbstractBill{

	public IncomeBill(double amount, String item) {
		super(amount, item);
	}

	public void accept(Viewer viewer) {
		if (viewer instanceof AbstractViewer) {
((AbstractViewer)viewer).viewIncomeBill(this);
			return;
		}
		viewer.viewAbstractBill(this);
	}

}

package com.visitor;

//消费的单子
```

```
public class ConsumeBill extends AbstractBill{

	public ConsumeBill(double amount, String item) {
		super(amount, item);
	}

	public void accept(Viewer viewer) {
		if (viewer instanceof AbstractViewer) {
((AbstractViewer)viewer).viewConsumeBill(this);
			return;
		}
		viewer.viewAbstractBill(this);
	}
}
```

这是元素类的层次结构，可以看到，我们的accept当中出现了if判断，各位看到可能会疑惑了，不是说不让用if else吗，这里怎么又出现了？

请各位仔细体会这里的条件判断的意义，它不是在判断一个接口或者类的具体类型，而是在判断一个层次，基于这一点，这个判断是稳定的，就像上面说的一样，通俗点说，这段代码是不会被更改的。

下面LZ再把分层之后的访问者层次贴上来。

```
package com.visitor;
```

```
//超级访问者接口（它支持定义高层操作）
public interface Viewer{
	void viewAbstractBill(AbstractBill bill);
}



package com.visitor;
//比Viewer接口低一个层次的访问者接口
public abstract class AbstractViewer implements Viewer{
	//查看消费的单子
	abstract void viewConsumeBill(ConsumeBill bill);

	//查看收入的单子
	abstract void viewIncomeBill(IncomeBill bill);

	public final void viewAbstractBill(AbstractBill bill){}
}

package com.visitor;
```

```
//老板类，查看账本的类之一，作用于最低层次结构
public class Boss extends AbstractViewer{
	private double totalIncome;
private double totalConsume;

	//老板只关注一共花了多少钱以及一共收入多少钱，其余并不关心
	public void viewConsumeBill(ConsumeBill bill) {
		totalConsume += bill.getAmount();
	}

	public void viewIncomeBill(IncomeBill bill) {
		totalIncome += bill.getAmount();
	}

	public double getTotalIncome() {
		System.out.println("老板查看一共收入多少，数目是：" + totalIncome);
		return totalIncome;
	}
```

```
	public double getTotalConsume() {
		System.out.println("老板查看一共花费多少，数目是：" + totalConsume);
		return totalConsume;
	}
}


package com.visitor;


//注册会计师类，查看账本的类之一，作用于最低层次结构
public class CPA extends AbstractViewer{
	//注会在看账本时，如果是支出，则如果支出是工资，则需要看应该交的税交了没
	public void viewConsumeBill(ConsumeBill bill) {
		if (bill.getItem().equals("工资")) {
			System.out.println("注会查看账本时，如果单子的消费目的是发工资，则注会会查看有没有交个人所得税。");
		}
	}
	//如果是收入，则所有的收入都要交税
	public void viewIncomeBill(IncomeBill bill) {
```

```
            System.out.println("注会查看账本时，只要是收入，注会都要查看公司交税了没。");

        }

}



package com.visitor;



//财务主管类，查看账本的类之一，作用于高层的层次结构

public class CFO implements Viewer {

        //财务主管对每一个单子都要核对项目和金额

        public void viewAbstractBill(AbstractBill bill) {

            System.out.println("财务主管查看账本时，每一个都核对项目和金额，金额是" + bill.getAmount() + "，项目是" + bill.getItem());

        }

}
```

可以看到，LZ这次为了避免迷惑，将view方法的名字全都给改了，在这种情况下，如果你仍然使用相同的名字，那么就算你对静态分派特别熟悉，在层次较多的情况下也会被绕晕的，而且最关键的是，我们改下名字完全不影响访问者模式的效果。

这里要说一下的是，财务主管（CFO）是针对AbstractBill这一层定义的操作，而原来的老板（Boss）和注册会计师（CPA）都是针对ConsumeBill和IncomeBill这一层定义的操作，这时已经产生了跨越层次结构的行为，老板和注册会计师都跨过了抽象单子这一层，直接针对具体的单子定义操作。

我们的账本类是完全不需要变化的，不过为了方便观看，这里再次贴上来。

```
package com.visitor;


import java.util.ArrayList;

import java.util.List;


//账本类（相当于ObjectStruture）

public class AccountBook {

	//单子列表

	private List<Bill> billList = new ArrayList<Bill>();

	//添加单子

	public void addBill(Bill bill){

		billList.add(bill);

	}

	//供账本的查看者查看账本

	public void show(Viewer viewer){

		for (Bill bill : billList) {

			bill.accept(viewer);

		}

	}
```

```
}
```

下面我们就写一个客户端，来看一下针对层次定义的操作是什么效果。

```
package com.visitor;


public class Client {


    public static void main(String[] args) {
        AccountBook accountBook = new AccountBook();
        //添加两条收入
        accountBook.addBill(new IncomeBill(10000, "卖商品"));
        accountBook.addBill(new IncomeBill(12000, "卖广告位"));
        //添加两条支出
        accountBook.addBill(new ConsumeBill(1000, "工资"));
        accountBook.addBill(new ConsumeBill(2000, "材料费"));


        Viewer boss = new Boss();
        Viewer cpa = new CPA();
        Viewer cfo = new CFO();
```

```
            //两个访问者分别访问账本
            accountBook.show(cpa);
            accountBook.show(boss);
            accountBook.show(cfo);

            ((Boss) boss).getTotalConsume();
            ((Boss) boss).getTotalIncome();
        }
}
```

可以看到，我们的注会和老板会针对收入和支出做出不同的操作，而我们财务主管则对所有的单子都是一样的处理方法。

这下好了，如果再出现和财务主管一样对所有单子都是一样操作的人，我们就不需要复制代码了，只需要让他实现Viewer接口就可以了，而如果要像老板和注会一样区分单子的具体类型，则继承AbstractViewer就可以。

目前我们的元素类是两层结构，所以我们现在可以针对这两层定义操作，如果元素类的结构是三层、四层或者N层，我们依然可以使用同样的手法达到现在的效果，比如将收入和支出类再写成抽象类，然后支出又分为出差，工资等等，收入又分为现金收入，债务收入等等，随便怎么分都可以，总之就是出现了三层元素类。那么我们就需要将viewer再抽象一层，从而支持三层定义。

从上面这个例子，我们就可以体会到访问者模式的另外一些优点了，即上面提到的第4和第5点。

这里针对层次结构做出的更改是基于静态分派的分派规则，即没有符合与方法参数列表相同的方法，则会将方法参数自动向上转型去匹配方法版本。

好了，访问者模式已经介绍了不少了，到现在也基本上该结束了。本章的文章有点长，这也是实属无奈，因为访问者模式牵扯的内容比较多，而且这还是LZ尽量缩短文章长度的结果。

各位如果有兴趣，可以去私下试一下支持三层操作定义怎么做，而且对于动态双分派的问题也可以在下方留言讨论。

本次文章就到此吧，十分感谢各位的收看

（二十三）原型模式详解（clone方法源码的简单剖析）

原型模式算是JAVA中最简单的设计模式了，原因是因为它已经被提供了语言级的支持，但是如果提到它的实现原理，又是最复杂的一个设计模式。

下面我们先来看看这个又简单又复杂的设计模式的定义。

定义：用原型实例指定创建对象的种类，并且通过拷贝这些原型创建新的对象。

定义比较简单，总结一下是通过实例指定种类，通过拷贝创建对象。

在JAVA语言中使用原型模式是非常简单的，这是因为Object类当中提供了一个本地方法clone，而JAVA中的任何类只要实现了Cloneable标识接口，就可以使用clone方法来进行对象的拷贝。

我们写一个简单的实例来测试一下，很简单。

```
package com.prototype;


public class Prototype implements Cloneable {


      private int x;

      private int y;
```

```
private int z;

public Prototype() {
      this.x = 2;
      this.y = 3;
      this.z = 4;
}

public void change() {
      this.x = 9;
      this.y = 8;
      this.z = 7;
}

public Prototype clone() {
      Object object = null;
      try {
            object = super.clone();
      } catch (CloneNotSupportedException exception) {
            throw new RuntimeException(exception);
```

```
        }
        return (Prototype) object;
    }

    public String toString() {
        return "[" + x + "," + y + "," + z + "]";
    }

    public static void main(String[] args) {
        Prototype prototype1 = new Prototype();
        prototype1.change();
        System.out.println(prototype1);
        Prototype prototype2 = prototype1.clone();
        System.out.println(prototype2);
    }

}
```

输入结果：

[9,8,7]

[9,8,7]

从输出结果可以看出来，clone方法将prototype1复制了一个，然后赋给了prototype2，这就像复制粘贴一样。值得注意的是，在使用Object.clone()方法去拷贝一个对象时，构造方法是不被执行的，否则prototype2实例中x，y，z的值应该为2，3，4才对，如果你觉得不够直观，可以在构造方法里写一个输出语句试试。

从原型模式的使用方式不难推断出，原型模式常使用于以下场景：

1、对象的创建非常复杂，可以使用原型模式快捷的创建对象。

2、在运行过程中不知道对象的具体类型，可使用原型模式创建一个相同类型的对象，或者在运行过程中动态的获取到一个对象的状态。

对于clone方法，它执行的是浅拷贝，也就是说如果是引用类型的属性，则它不会进行拷贝，而是只拷贝引用。

看下面这个简单的测试，就能看出来了。

```
package com.prototype;


class Field{


	private int a;


	public int getA() {

		return a;
```

```
    }

    public void setA(int a) {
        this.a = a;
    }

}

public class ShallowPrototype implements Cloneable {

    private int x;
    private int y;
    private int z;
    private Field field;

    public ShallowPrototype() {
        this.x = 2;
        this.y = 3;
        this.z = 4;
        this.field = new Field();
```

```
        this.field.setA(5);
    }

    public Field getField() {
        return field;
    }

    public ShallowPrototype clone() {
        Object object = null;
        try {
            object = super.clone();
        } catch (CloneNotSupportedException exception) {
            throw new RuntimeException(exception);
        }
        return (ShallowPrototype) object;
    }

    public String toString() {
        return "[" + x + "," + y + "," + z + "," +
field.getA() + "]";
    }
```

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

		ShallowPrototype prototype1 = new
ShallowPrototype();

		System.out.println(prototype1);

System.out.println(prototype1.getField());

		ShallowPrototype prototype2 = prototype1.clone();

		System.out.println(prototype2);

System.out.println(prototype2.getField());

	}



}
```

输入结果：

```
[2,3,4,5]

com.prototype.Field@de6ced

[2,3,4,5]

com.prototype.Field@de6ced
```

可以看到我们对ShallowPrototype拷贝以后，得到一个实例prototype2，不过当我们输出field属性时，发现它们是引用的同一个对象。这当然不是我们期望得到的结果，这种情况下，我们如果修改prototype1中field的属性a的值，则prototype2中的也会跟着改变。

然而如果要实现深度拷贝，则需要将实现了Cloneable接口并重写了clone方法的类中，所有的引用类型也全部实现Cloneable接口并重写clone方法，而且需要将引用类型的属性全部拷贝一遍。

下面是一个简单的深度拷贝的例子，由上面的例子更改得到。

```
package com.prototype;


class Field implements Cloneable{


	private int a;


	public int getA() {
		return a;
	}


	public void setA(int a) {
		this.a = a;
	}


	protected Field clone() {
		Object object = null;
		try {
```

```
                object = super.clone();
            } catch (CloneNotSupportedException exception) {
                throw new RuntimeException(exception);
            }
            return (Field) object;
        }

}

public class DeepPrototype implements Cloneable {

        private int x;
        private int y;
        private int z;
        private Field field;

        public DeepPrototype() {
            this.x = 2;
            this.y = 3;
            this.z = 4;
```

```
        this.field = new Field();
        this.field.setA(5);
    }

    public Field getField() {
        return field;
    }

    protected DeepPrototype clone() {
        Object object = null;
        try {
            object = super.clone();
            ((DeepPrototype)object).field =
this.field.clone();
        } catch (CloneNotSupportedException exception) {
            throw new RuntimeException(exception);
        }
        return (DeepPrototype) object;
    }

    public String toString() {
```

```
            return "[" + x + "," + y + "," + z + "," +
field.getA() + "]";
      }

      public static void main(String[] args) {
  DeepPrototype prototype1 = new DeepPrototype();
            System.out.println(prototype1);
System.out.println(prototype1.getField());
            DeepPrototype prototype2 = prototype1.clone();
            System.out.println(prototype2);
            System.out.println(prototype2.getField());
      }

}
```

输出结果：

```
[2,3,4,5]
com.prototype.Field@a90653
[2,3,4,5]
com.prototype.Field@de6ced
```

下面我们来看下原型模式的主要优点：

1、由于clone方法是由虚拟机直接复制内存块执行，所以在速度上比使用new的方式创建对象要快。

2、可以基于原型，快速的创建一个对象，而无需知道创建的细节。

3、可以在运行时动态的获取对象的类型以及状态，从而创建一个对象。

然而原型模式的缺点也是相当明显的，主要的缺点就是实现深度拷贝比较困难，需要很多额外的代码量。

不过实际当中我们使用原型模式时，也可以写一个基类实现Cloneable接口重写clone方法，然后让需要具有拷贝功能的子类继承自该类，这是一种节省代码量的常用方式。像上面的例子一样，如果一个类继承自Prototype，则会自动具有拷贝功能。

下面我们来看看虚拟机中本地方法Object.clone()的源代码，如下。

```
JVM_ENTRY(jobject, JVM_Clone(JNIEnv* env, jobject handle))

   JVMWrapper("JVM_Clone");

   Handle obj(THREAD, JNIHandles::resolve_non_null(handle));

   const KlassHandle klass (THREAD, obj->klass());

   JvmtiVMObjectAllocEventCollector oam;


#ifdef ASSERT

   // Just checking that the cloneable flag is set correct

   if (obj->is_javaArray()) {
```

```
      guarantee(klass->is_cloneable(), "all arrays are
cloneable");

   } else {

      guarantee(obj->is_instance(), "should be instanceOop");

      bool cloneable = klass-
>is_subtype_of(SystemDictionary::Cloneable_klass());

      guarantee(cloneable == klass->is_cloneable(),
"incorrect cloneable flag");

   }

#endif



   // Check if class of obj supports the Cloneable interface.

   // All arrays are considered to be cloneable (See JLS
20.1.5)

   if (!klass->is_cloneable()) {//这里检查了是否实现了
Cloneable接口，如果没实现，会抛出异常
CloneNotSupportException。

      ResourceMark rm(THREAD);

THROW_MSG_0(vmSymbols::java_lang_CloneNotSupportedException()
, klass->external_name());

   }



   // Make shallow object copy
```

```
const int size = obj->size();//取对象大小

oop new_obj = NULL;

if (obj->is_javaArray()) {//如果是数组

    const int length = ((arrayOop)obj())->length();//取长度

    new_obj = CollectedHeap::array_allocate(klass, size,
length, CHECK_NULL);//分配内存，写入元数据信息

} else {

    new_obj = CollectedHeap::obj_allocate(klass, size,
CHECK_NULL);//分配内存，写入元数据信息

}

// 4839641 (4840070): We must do an oop-atomic copy,
because if another thread

// is modifying a reference field in the clonee, a non-
oop-atomic copy might

// be suspended in the middle of copying the pointer and
end up with parts

// of two different pointers in the field.   Subsequent
dereferences will crash.

// 4846409: an oop-copy of objects with long or double
fields or arrays of same

// won't copy the longs/doubles atomically in 32-bit vm's,
so we copy jlongs instead

// of oops.We know objects are aligned on a minimum of an
jlong boundary.
```

```
   // The same is true of StubRoutines::object_copy and the
various oop_copy

   // variants, and of the code generated by the
inline_native_clone intrinsic.

   assert(MinObjAlignmentInBytes >= BytesPerLong, "objects
misaligned");

   Copy::conjoint_jlongs_atomic((jlong*)obj(),
(jlong*)new_obj,

                                (size_t)align_object_size(size)
/ HeapWordsPerLong);//这一步就是真正的COPY内存块了

   // Clear the header

   new_obj->init_mark();//初始化对象头，里面包含了Hashcode，
GC信息，锁信息等，因为拷贝出的对象是一个全新的对象，所以这些
信息需要初始化一下。



   // Store check (mark entire object and let gc sort it out)

   BarrierSet* bs = Universe::heap()->barrier_set();

   assert(bs->has_write_region_opt(), "Barrier set does not
have write_region");

bs->write_region(MemRegion((HeapWord*)new_obj,
size));//write_region最终的实现在一个虚方法里，相当于JAVA的抽
象方法，LZ没找到实现。暂不发表意见。



   // Caution: this involves a java upcall, so the clone
should be
```

```
   // "gc-robust" by this stage.

   if (klass->has_finalizer()) {//如果有finalize方法，则需要注册一下。

      assert(obj->is_instance(), "should be instanceOop");

      new_obj = instanceKlass::register_finalizer(instanceOop(new_obj), CHECK_NULL);

   }



   return JNIHandles::make_local(env, oop(new_obj));//将内存对象转换成JAVA本地对象返回

JVM_END
```

虚拟机的源码比较复杂，而且完全没有相关文献和资料，所以LZ也只能简单的添加一些注释，特别是write_region这个方法，LZ没找到实现在哪里，LZ猜测这个方法的功能是设置对象的边界的，好让GC能够正确的回收内存，但由于没找到实现，所以不敢断言。

在上面的过程中调用了Copy对象的conjoint_jlongs_atomic方法，那个就是真正的复制实例数据的方法，LZ找到了这个方法的实现，给各位看一下。

```
void _Copy_conjoint_jlongs_atomic(jlong* from, jlong* to, size_t count) {

      if (from > to) {

         jlong *end = from + count;

         while (from < end)

            os::atomic_copy64(from++, to++);
```

```
    }
    else if (from < to) {
        jlong *end = from;
        from += count - 1;
        to+= count - 1;
        while (from >= end)
            os::atomic_copy64(from--, to--);
    }
}
```

这是一个操作内存块的方法，其中atomic_copy64这个方法是用汇编语言写的，它确保了在64位的机子下也可以正确的进行内存块的拷贝操作。它的作用很简单，就是把from指针指向的内存的值赋给to指针指向的内存，也就是一个简单的拷贝操作。知道了atomic_copy64方法的作用，上面这个方法的逻辑就非常简单了。

由此可以看出，我们可以将clone方法想象成内存块的复制操作，它的速度比一般的创建对象操作要快。

原型模式的分析就到此结束了，对于虚拟机源码的研究，LZ一直在断断续续的继续着，等设计模式系列写完以后，LZ会写一些虚拟机以及虚拟机源码的相关内容，希望各位能继续支持吧。

感谢各位的收看。

（二十四）解释器模式详解

解释器模式是平时工作当中相对冷门的一个设计模式，也非常的难于理解，百度百科上的解释也非常之少，只是简单的介绍了一下，并且说了一句，可以参考正则表达式为一个实际的应用例子。

不过资料的匮乏并不能阻止我们对真理的探索，下面LZ先将百度百科上的定义以及解决的问题拔到这里，方便各位观看。

定义：给定一个语言，定义它的文法的一种表示，并定义一个解释器，这个解释器使用该表示来解释语言中的句子。

使用场景：解释器模式需要解决的是，如果一种特定类型的问题发生的频率足够高，那么可能就值得将该问题的各个实例表述为一个简单语言中的句子。这样就可以构建一个解释器，该解释器通过解释这些句子来解决该问题。

LZ先给各位解释一下定义当中所提到的文法。文法也称为语法，指的是语言的结构方式。包括词的构成和变化，词组和句子的组织。对于文法来说，我们可以简单的理解为一种语言的规则，那么从解释器模式的定义可以看出，首先我们要先设计一种语言，然后给出语言的文法的表示，而在此基础上，我们采用解释器模式去解释语言中的句子。

要想彻底的理解解释器模式，LZ必须要先普及一下文法的定义，请各位暂且忍受住枯燥的理论知识，后面LZ会将这些理论用各位熟悉的代码诠释一遍。

首先我们来讨论一下上下文无关文法的组成，有四种组成部分。

1，非终结符号集（LZ标注：像JAVA语言中的表达式，程序语句，标识符等）

2，终结符号集（LZ标注：类似JAVA语言中的+，-，*，\，=等）

3，产生式集合，也可以称为规则集合（LZ标注：假设我们记JAVA中的标识符为id，那么下面这句话可以被成视为一条规则 id->a|b...|z|0..|9|_，其中|是或者的意思）

4，一个起始符号，这个符号是非终结符号集的一个元素（LZ标注：JAVA语言使用CompilationUnit（编译单元）作为起始符号。）

上面所说的定义有些抽象，所以LZ在后面加了一些标注，那么上下文无关文法的作用是什么呢？

它可以生成一组由文法导出的语句，这些语句可以根据文法的产生式进行分析，下面LZ给一个《编译原理》一书中的简单例子，为了方便理解，LZ将符号稍微更改了一下。

假设有一上下文无关文法如下：

arithmetic -> arithmetic + number | arithmetic - number | number

number -> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

我们根据这个文法可以得到所有个位数的加减表达式，比如对于 9 + 2 - 1 ，我们可以通过以下步骤推导出来。

arithmetic - >arithmetic - number -> arithmetic + number - number -> number + number - number -> 9 + number - number -> 9 + 2 - number -> 9 + 2 - 1

对于文法来说，一个语句如果能够按照产生式推导出该语句，就称该语句是符合文法的，所以9 + 2 - 1是符合上述文法的一个语句。

在这个文法当中，其中非终结者符号是 arithmetic 和 number， 而终结者符号是 0 - 9 、-、+ 。

我们从文法中可以得知由该文法组成的语句有以下规则。

1、operator的右边必须是一个number。

2、operator的左边必须是一个arithmetic。

3、arithmetic的最右边一定是一个number。

4、由2和3，operator的左边必须是number。

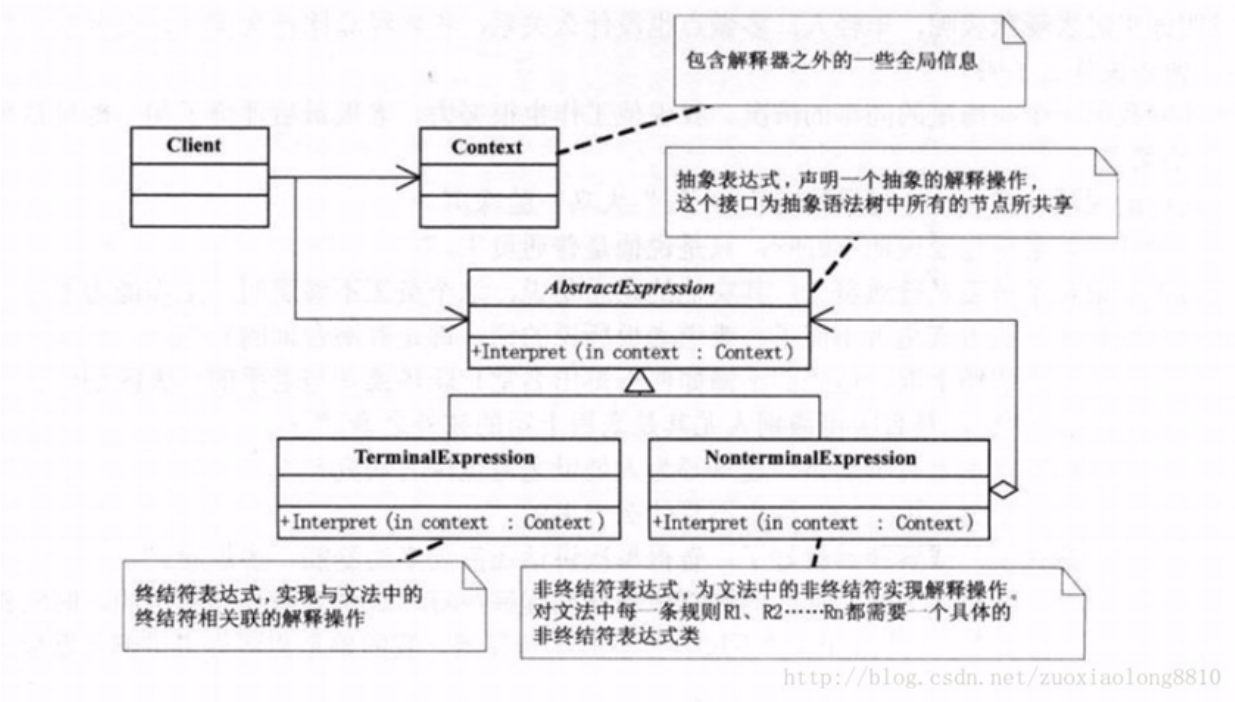
5、由4，number的右边必须是空或者operator。

6、number只能是 0 和 1 - 9 的正整数。

7、operator只能是 - 和 + 。

针对这个文法，我们可以写一个解释器，去计算表达式的结果，而这个解释器就可以使用解释器模式编写。而在编写的过程中，我们需要验证以上的规则，如果违反了规则，则表达式是非法的。为了便于使用程序语言表示，我们只验证以上的后四条规则，这也是由原本的产生式推算出来的规则。

我们先来看下解释器模式的类图，引自《大话设计模式》。



可以看到类图中有四个角色，抽象表达式（AbstractExpression）、终结符表达式（TerminalExpression）、

非终结符表达式（NonterminalExpression）以及上下文（Context）。

四个角色所负责的任务在类图中已有解释，LZ这里不再重复，这里要说的是，这里具体的表达式类个数是不定的。

换句话说，终结符表达式（TerminalExpression）和非终结符表达式（NonterminalExpression）的个数都是根据文法需要而定的，并非是一成不变。

下面我们就使用上述的解释器模式的结构去写一个解释器，用于解释上面的加减表达式，首先我们先写一个上下文，它记录了一些全局信息，提供给表达式类使用，如下。

```
package com.interpreter;



import java.util.ArrayList;

import java.util.List;



//上下文

public class Context {



	private int result;//结果



	private int index;//当前位置



	private int mark;//标志位
```

```
private char[] inputChars;//输入的字符数组

private List<Integer> operateNumbers = new ArrayList<Integer>(2);//操作数

private char operator;//运算符

public Context(char[] inputChars) {
    super();
    this.inputChars = inputChars;
}

public int getResult() {
    return result;
}

public void setResult(int result) {
    this.result = result;
}
```

```
public boolean hasNext(){
      return index != inputChars.length;
}

public char next() {
      return inputChars[index++];
}

public char current(){
      return inputChars[index];
}

public List<Integer> getOperateNumbers() {
      return operateNumbers;
}

public void setLeftOperateNumber(int operateNumber) {
      this.operateNumbers.add(0, operateNumber);
}
```

```
    public void setRightOperateNumber(int operateNumber) {
        this.operateNumbers.add(1, operateNumber);
    }

    public char getOperator() {
        return operator;
    }

    public void setOperator(char operator) {
        this.operator = operator;
    }

    public void mark(){
        mark = index;
    }

    public void reset(){
        index = mark;
    }
}
```

上下文的各个属性，都是表达式在计算过程中需要使用的，也就是类图中所说的全局信息，其中的操作数和运算符是模拟的计算机中寄存器加减指令的执行方式。下面我们给出抽象的表达式，它只是定义一个解释操作。

```
package com.interpreter;


//抽象表达式，定义一个解释操作

public interface Expression {


	void interpreter(Context context);


}
```

下面便是最重要的四个具体表达式了，这其中对应于上面文法提到的终结符和非终结符，如下。

```
package com.interpreter;


//算数表达式（非终结符表达式，对应arithmetic）

public class ArithmeticExpression implements Expression {


	public void interpreter(Context context) {

		context.setResult(getResult(context));//计算结果

		context.getOperateNumbers().clear();//清空操作数
```

```
context.setLeftOperateNumber(context.getResult());//将结果压入左操作数
        }

        private int getResult(Context context){
                int result = 0;
                switch (context.getOperator()) {
                case '+':
                        result = context.getOperateNumbers().get(0) + context.getOperateNumbers().get(1);
                        break;
                case '-':
                        result = context.getOperateNumbers().get(0) - context.getOperateNumbers().get(1);
                        break;
                default:
                        break;
                }
                return result;
        }

}
```

```
package com.interpreter;


//非终结符表达式，对应number
public class NumberExpression implements Expression{


	public void interpreter(Context context) {
		//设置操作数
		Integer operateNumber =
Integer.valueOf(String.valueOf(context.current()));
		if (context.getOperateNumbers().size() == 0) {
context.setLeftOperateNumber(operateNumber);
			context.setResult(operateNumber);
		}else {
context.setRightOperateNumber(operateNumber);
			Expression expression = new
ArithmeticExpression();//转换成算数表达式
			expression.interpreter(context);
		}
	}
```

```
}

package com.interpreter;

//终结符表达式，对应-、+
public class OperatorExpression implements Expression{

	public void interpreter(Context context) {
		context.setOperator(context.current());//设置运算符
	}

}

package com.interpreter;

//终结符表达式，对应0、1、2、3、4、5、6、7、8、9
public class DigitExpression implements Expression{

	public void interpreter(Context context) {
```

```
            Expression expression = new NumberExpression();//
如果是数字，则直接转为number表达式

            expression.interpreter(context);

      }



}
```

这四个类就是简单的解释操作，值得一提的就是其中的两次转换，这个在稍后LZ会解释一下。

下面本来该是客户端程序了，不过由于我们的例子较为复杂，客户端的代码会比较臃肿，所以LZ抽出了一个语法分析类，分担了一些客户端的任务，在标准解释器模式的类图中是没有这个类的。

各位可以把它的代码想象成在客户端里面就好，这并不影响各位理解解释器模式本身，语法分析器的代码如下。

```
package com.interpreter;



//语法解析器（如果按照解释器模式的设计，这些代码应该是在客户
端，为了更加清晰，我们添加一个语法解析器）

public class GrammarParser {



      //语法解析

      public void parse(Context context) throws Exception{

            while (context.hasNext()) {

                  Expression expression = null;
```

```
                    switch (context.current()) {
                        case '+':
                        case '-':
                            checkGrammar(context);
                            expression = new
OperatorExpression();
                            break;
                        case '0':
                        case '1':
                        case '2':
                        case '3':
                        case '4':
                        case '5':
                        case '6':
                        case '7':
                        case '8':
                        case '9':
                            context.mark();
                            checkGrammar(context,
context.current());
                            context.reset();
```

```
                                expression = new
DigitExpression();

                                break;

                        default:

                                throw new RuntimeException("语
法错误！");//无效符号

                }

                expression.interpreter(context);

                context.next();

            }

        }

        //检查语法

        private void checkGrammar(Context context,char current)
{

            context.next();

            if (context.hasNext() && context.current() != '+'
&& context.current() != '-') {

                throw new RuntimeException("语法错误！");//
第5条

            }

            try {

Integer.valueOf(String.valueOf(current));

            } catch (Exception e) {
```

```
                throw new RuntimeException("语法错误！");//
第6条
            }
        }
        //检查语法
        private void checkGrammar(Context context){
            if (context.getOperateNumbers().size() == 0) {//
第4条
                throw new RuntimeException("语法错误！");
            }
            if (context.current() != '+' && context.current()
!= '-') {//第7条
                throw new RuntimeException("语法错误！");
            }
        }

}
```

可以看到，我们的语法分析器不仅做了简单的分析语句，从而得出相应表达式的工作，还做了一个工作，就是语法的正确性检查。

下面我们写个客户端去计算几个表达式试一下。

```
package com.interpreter;
```

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



public class Client {


	public static void main(String[] args) {
		List<String> inputList = new ArrayList<String>();
		//三个正确的，三个错误的
inputList.add("1+2+3+4+5+6+7+8+9");
inputList.add("1-2+3-4+5-6+7-8+9");
		inputList.add("9");
		inputList.add("-1+2+3+5");
		inputList.add("1*2");
		inputList.add("11+2+3+9");


		GrammarParser grammarParser = new
GrammarParser();//语法分析器


		for (String input : inputList) {
```

```
                Context context = new
Context(input.toCharArray());

                try {

                        grammarParser.parse(context);//语法分
析器会调用解释器解释表达式

                        System.out.println(input + "=" +
context.getResult());

                } catch (Exception e) {

                        System.out.println("语法错误，请输入
正确的表达式！");

                }

            }

        }



}
```

输出结果：

1+2+3+4+5+6+7+8+9=45

1-2+3-4+5-6+7-8+9=5

9=9

语法错误，请输入正确的表达式！

语法错误，请输入正确的表达式！

语法错误，请输入正确的表达式！

可以看到，前三个表达式是符合我们的文法规则的，而后三个都不符合规则，所以提示了错误，这样的结果，与我们文法所表述的规则是相符的。

LZ需要提示的是，这里面本来是客户端使用解释器来解释语句的，不过由于我们抽离出了语法分析器，所以由语法分析器调用解释器来解释语句，这消除了客户端对解释器的关联，与标准类图不符，不过这其实只是我们所做的简单的改善而已，并不影响解释器模式的结构。

另外，上面的例子当中，还有两点是LZ要提一下的。LZ为了方便理解，已经尽量的将例子简化，不过其中有两个地方的转换是值得注意的。

1、一个是操作数满足条件时，会产生一个ArithmeticExpression表达式。

2、另外一个是从DigitExpression直接转换成NumberExpression的地方，这其实和第1点一样，都是对文法规则的使用，不过这个更加清晰。我们可以清楚的看到，0-9的数字或者说DigitExpression只对应唯一一种方式的非终结者符号，就是number，所以我们直接转换成NumberExpression。

不过我们的转换是由终结者符号反向转换成非终结者符号的顺序，也就是相当于从抽象语法树的低端向上转换的顺序。其实相当于LZ省去了抽象语法树的潜在构建过程，直接开始解释表达式。

我们看上面的类图中，非终结者表达式有一条到抽象表达式的聚合线，那其实是将非终结者表达式按照产生式分解的过程，这会是一个递归的过程，而我们省去了这一步，直接采用反向计算的方式。

然后再说说我们的语法分析器，它的工作就是将终结者符号对应上对应的表达式，可以看到它里面的swich结构就是用来选取表达式的。实际当中，我们当然不会写这么糟糕的swich结构，我们可以使用很多方式优化它。当然，语法分析器的另外一个工作就是检查语法的正确性，这点可以从两个check方法明显的看到。

不过很遗憾，在日常工作当中，我们使用到解释器模式的概率几乎为0，因为写一个解释器就基本相当于创造了一种语言，这对于大多数人来说，是几乎不可能接到的工作。不过我们了解一下解释器模式，还是对我们有好处的。

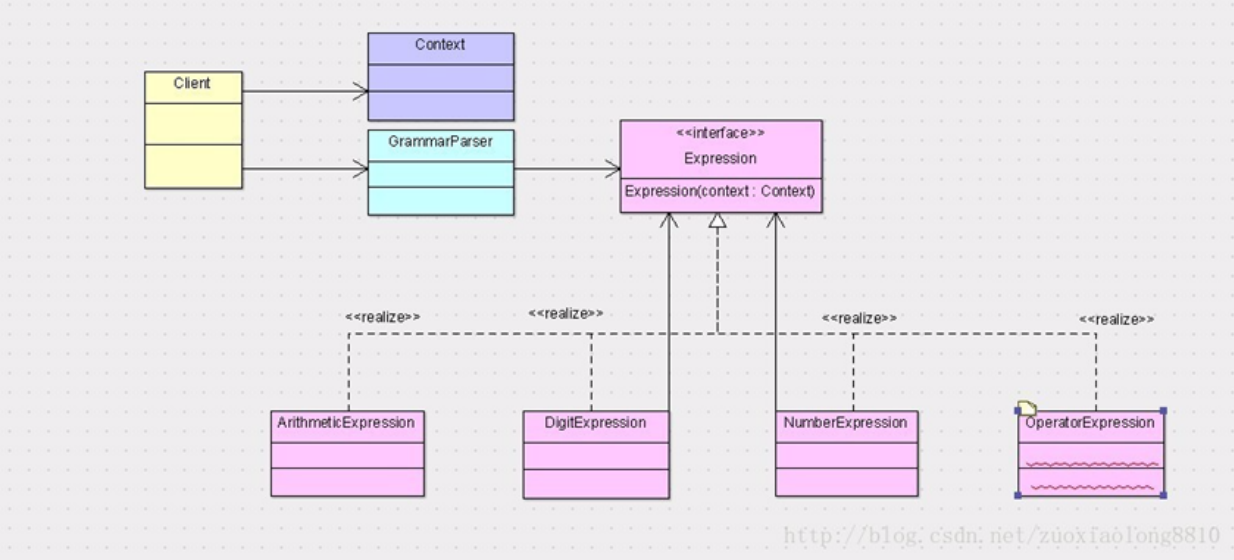
前面已经提到过解释器模式适用的场景，我们这里结合上面的例子总结一下解释器模式的优点：

1、由于我们使用具体的终止符和非终止符去解释文法，所以会比较易于编写。

2、可以比较方便的修改和扩展文法规则。

相对于优点来说，它的缺点也非常明显，那就是由于我们几乎针对每一个规则都定义了一个类，所以如果一个文法的规则比较多，那对于文法的维护工作也会变得非常困难。

下面LZ将我们例子的类图贴上来，各位参考一下。



针对这个类图，LZ简单的说两点。

1、Client本来是关联的Expression接口，不过由于中间加了个语法分析器（GrammerParser），所以变成了Client关联语法分析器，语法分析器再关联Expression接口。

2、由于我们采用反向计算的方式，所以非终结者表达式没有到Expression接口的聚合线，而是由两条关联线代替了两条聚合线。

解释器模式的分享就到此结束了，各位只要大致了解一下即可，如果实在理解不了，也不必死抠这个模式。

学习设计模式，有时候就像追女生一样，如果你怎么追都追不上，那说明缘分未到，该放弃的时候就得放弃，说不定哪天缘分到了，不需要你追，你心中的她自然就上钩了。所以如果有哪一个设计模式一时半会理解不了，请不要着急去理解它，不理解的原因是因为你的积累还不够，如果霸王硬上弓的话，往往会被设计模式暴揍一顿，最后还得自己付医药费。

LZ只想说，何必呢？

到这篇文章为止，LZ已经将所有24种设计模式全部讲解了一遍，其中有好有坏，有对有错。不过不管怎样，LZ本人的收获还是很大的，也很感谢这当中支持LZ的猿友。设计模式系列或许还会有最后一篇，内容自然是对24种设计模式的总结，LZ最近也在为此而准备着，敬请各位猿友擦亮双眼期待吧。

一个系列结束了，不代表LZ的学习之路结束了，在上一章已经提到过，LZ最近在研究虚拟机源码，在接下来的时间里，LZ或许会写一些与虚拟机相关的内容，如果有哪位猿友对虚拟机有兴趣的话，可以继续关注下LZ。

感谢各位的收看，我们下次再见。

24种设计模式的总结以及学习设计模式的几点建议

迄今为止，LZ已经将24种设计模式介绍完了，其中包括GOF23种设计模式以及简单工厂模式，这些设计模式之间并不是完全独立的，而是互相之间，会有一些相同的影子，下面我们来一起总结下这24种设计模式。

模式分类 & 传送门 & 对比维度说明

- 设计原则：　设计模式（总纲）
- 创建型：　单例模式　简单工厂模式　工厂方法模式　抽象工厂模式　建造者模式　原型模式
- 结构型：　代理模式　适配器模式　装饰器模式　桥接模式　组合模式　享元模式　外观模式
- 行为型：　观察者模式　模板方法模式　命令模式　状态模式　职责链模式　解释器模式　中介者模式　访问者模式　策略模式　备忘录模式　迭代器模式

以上便是设计模式的分类以及各个模式的传送门，可以看到其中行为型模式的个数为最多，结构型次之，创建型设计模式最少。

在写这篇文章的时候，LZ考虑的最多的一个问题就是，从哪几个维度去对比设计模式能让大家更加清楚的看出各个设计模式的区别与联系，思来想去，LZ决定从以下几个维度去对比设计模式。

• 设计原则：描述每个设计模式都遵循了哪些设计原则，破坏了哪些设计原则。

• 常用场景：描述各个设计模式大部分情况下，都会在哪些场景下出现。

• 使用概率：主要指在普遍的工作当中，该设计模式出现的频率，若是类库或是开源框架提供的功能中包含该模式，则也会计算其频率。

• 复杂度：特指一个设计模式在实现的时候的复杂度，主要的衡量标准是类的数量、类之间的耦合关系。

• 变化点：设计模式很大的一个意义在于容纳变化，掌握一个设计模式的变化点是非常重要的一件事。

• 选择关键点：当选择使用一个设计模式的时候，指出最关键的选择点在哪里。

• 逆鳞：龙有逆鳞，不可触摸，同样的，设计模式也有逆鳞，有些地方是不能碰的。

• 相关设计模式：与其它设计模式的关系。

创建型设计模式

单例模式

设计原则：无

• 常用场景：应用中有对象需要是全局的且唯一

• 使用概率：99.99999%

• 复杂度：低

• 变化点：无

• 选择关键点：一个对象在应用中出现多个实例是否会引起逻辑上或者是程序上的错误

• 逆鳞：在以为是单例的情况下，却产生了多个实例

• 相关设计模式

• 原型模式：单例模式是只有一个实例，原型模式每拷贝一次都会创造一个新的实例。

简单工厂模式

设计原则：遵循单一职责、违背开闭原则

• 常用场景：需要在一堆产品中选择其中一个产品

• 使用概率：99.99999%

• 复杂度：低

• 变化点：产品的种类

• 选择关键点：一种产品是否可根据某个参数决定它的种类

• 逆鳞：工厂类不能正常工作

• 相关设计模式

• 工厂方法模式：工厂方法模式是简单工厂模式的进一步抽象化，在这两者之间做选择，主要看将工厂进一步抽象化是否有必要，通常情况下，如果工厂的作用仅仅是用来制造产品，则没必要使用工厂方法模式。

工厂方法模式

设计原则：遵循单一职责、依赖倒置、开闭原则

• 常用场景：一种场景是希望工厂与产品的种类对客户端保持透明，给客户端提供一致的操作，另外一种是不同的工厂和产品可以提供客户端不同的服务或功能

• 使用概率：60%

• 复杂度：中低

• 变化点：工厂与产品的种类

• 选择关键点：工厂类和产品类是否是同生同灭的关系

• 逆鳞：无

• 相关设计模式

• 抽象工厂模式：工厂方法模式与抽象工厂模式最大的区别在于，在工厂方法模式中，工厂创造的是一个产品，而在抽象工厂模式中，工厂创造的是一个产品族。

抽象工厂模式

设计原则：遵循单一职责、依赖倒置、开闭原则

• 常用场景：需要一个接口可以提供一个产品族，且不必知道产品的具体种类

• 使用概率：30%

• 复杂度：中

• 变化点：工厂与产品的种类

• 选择关键点：产品族是否需要一起提供，且是否有一致的接口

• 逆鳞：无

• 相关设计模式

• 建造者模式：两者都是建造一批对象或者说产品，不同的是两者的目的和实现手段，在建造者模式中，是为了复用对象的构建过程而定义了一个指挥者，而在抽象工厂模式中，是为了提供一个这批对象的创建接口而定义了抽象工厂接口。

建造者模式

设计原则：遵循单一职责、开闭原则

• 常用场景：需要构建一批构建过程相同但表示不同的产品，而构建过程非常复杂

• 使用概率：10%

• 复杂度：中

• 变化点：产品的表示

• 选择关键点：各个产品的构建过程是否相同

• 逆鳞：指挥者不能正常工作

原型模式

• 设计原则：无

• 常用场景：需要在运行时动态的创建指定实例种类的对象，或是需要复用其状态

- 使用概率：10%

- 复杂度：中低

- 变化点：无

- 选择关键点：创建出来的对象是否可以立即投入使用

- 逆鳞：在以为是深度拷贝的情况下，却未实现深度拷贝

结构型设计模式

代理模式

设计原则：体现功能复用

- 常用场景：需要修改或屏蔽某一个或若干个类的部分功能，复用另外一部分功能，可使用静态代理，若是需要拦截一批类中的某些方法，在方法的前后插入一些一致的操作，假设这些类有一致的接口，可使用JDK的动态代理，否则可使用cglib

- 使用概率：99.99999%

- 复杂度：中高

- 变化点：静态代理没有变化点，动态代理的变化点为具有相同切入点的类

- 选择关键点：静态代理选择的关键点是是否要复用被代理的部分功能，动态代理选择的关键点在于能否在将被代理的这一批类当中，找出相同的切入点

- 逆鳞：切入点的不稳定

- 相关设计模式

• 适配器模式：对于适配器模式当中的定制适配器，它与静态代理有着相似的部分，二者都有复用功能的作用，不同的是，静态代理会修改一部分原有的功能，而适配器往往是全部复用，而且在复用的同时，适配器还会将复用的类适配一个接口

适配器模式

• 设计原则：遵循开闭原则、体现功能复用

• 常用场景：需要使用一个类的功能，但是该类的接口不符合使用场合要求的接口，可使用定制适配器，又或者是有一个接口定义的行为过多，则可以定义一个缺省适配器，让子类选择性的覆盖适配器的方法

• 使用概率：40%

• 复杂度：中

• 变化点：无

• 选择关键点：定制适配器的选择关键点在于是否有更加优良的替代方案，缺省适配器的选择关键点在于接口中的方法是否可以不全部提供，且都有缺省方案

• 逆鳞：无

• 相关设计模式

• 装饰器模式：对于适配器模式中的定制适配器与装饰器模式，二者都是使用组合加继承的手段，不同的是，适配器模式的目的在于适配接口，装饰器模式的目的在于动态的添加功能，且可以叠加。

装饰器模式

• 设计原则：遵循迪米特、单一职责、开闭原则，破坏里氏替换，体现功能复用

• 常用场景：一个类需要动态的添加功能，且这些功能可以相互叠加

• 使用概率：99.99999%

• 复杂度：中

• 变化点：动态添加的功能或者说装饰器

• 选择关键点：添加的功能是否需要动态组装

• 逆鳞：无

桥接模式

• 设计原则：遵循单一职责、迪米特、开闭原则，体现功能复用

• 常用场景：一个对象有多个维度的变化，需要将这些维度抽离出来，让其独立变化

• 使用概率：20%

• 复杂度：中高

• 变化点：维度的扩展与增加

• 选择关键点：是否可以将对象拆分成多个不相关的维度

• 逆鳞：无

组合模式

设计原则：遵循依赖倒置、开闭原则，破坏接口隔离

• 常用场景：当有一个结构可以组合成树形结构，且需要向客户端提供一致的操作接口，使得客户端操作忽略简单元素与复杂元素

- 使用概率：30%
- 复杂度：中
- 变化点：节点的数量
- 选择关键点：对外提供一致操作接口的结构是否可转化为树形结构
- 逆鳞：结构不稳定或结构中的节点有递归关系

享元模式

设计原则：无

- 常用场景：一些状态相同的对象被大量的重复使用
- 使用概率：90%
- 复杂度：中
- 变化点：无
- 选择关键点：被共享的对象是否可以将外部状态提取出来
- 逆鳞：没有将外部状态提取完全

外观模式

设计原则：遵循迪米特

- 常用场景：一个子系统需要对外提供服务
- 使用概率：60%
- 复杂度：中
- 变化点：无
- 选择关键点：子系统对外提供服务是否需要依赖很多的类

• 逆鳞：子系统对外提供的服务的变化或子系统本身的不稳定

• 相关设计模式

• 中介者模式：二者都是为了处理复杂的耦合关系，不同的是外观模式处理的是类之间复杂的依赖关系，中介者模式处理的是对象之间复杂的交互关系

行为型设计模式

观察者模式

设计原则：遵循迪米特、开闭原则

• 常用场景：需要将观察者与被观察者解耦或者是观察者的种类不确定

• 使用概率：40%

• 复杂度：中

• 变化点：观察者的种类与个数

• 选择关键点：观察者与被观察者是否是多对一的关系

• 逆鳞：观察者之间有过多的细节依赖

模板方法模式

设计原则：破坏里氏替换，体现功能复用

• 常用场景：一批子类的功能有可提取的公共算法骨架

• 使用概率：80%

• 复杂度：中低

• 变化点：算法骨架内各个步骤的具体实现

• 选择关键点：算法骨架是否牢固

• 逆鳞：无

命令模式

设计原则：遵循迪米特、开闭原则

• 常用场景：行为的请求者与行为的处理者耦合度过高

• 使用概率：20%

• 复杂度：中高

• 变化点：命令的种类

• 选择关键点：请求者是否不需要关心命令的执行只知道接受者

• 逆鳞：命令的种类无限制增长

• 相关设计模式

• 职责链模式：容易将二者关联在一起的原因是，二者都是为了处理请求或者命令而存在的，而且二者都是为了将请求者与响应者解耦，不同的是命令模式中，客户端需要知道一个命令的接受者，在创建命令的时候就把接受者与命令绑定在一起发送给调用者，而职责链模式中，客户端并不关心最终处理请求的对象是谁，客户端只是封装一个请求对象，随后交给职责链的头部而已，也正因为这样，二者的实现方式，有着很大的区别

状态模式

设计原则：遵循单一职责、依赖倒置、开闭原则

• 常用场景：一个对象在多个状态下行为不同，且这些状态可互相转换

• 使用概率：20%

• 复杂度：中

• 变化点：状态的种类

• 选择关键点：这些状态是否经常在运行时需要在不同的动态之间相互转换

• 逆鳞：无

• 相关设计模式

• 策略模式：二者的实现方式非常相似，策略接口与状态接口，具体的策略与具体的状态以及二者都拥有的上下文，如果看它们的类图，会发现几乎一模一样，而二者不同的地方就在于，状态模式经常会在处理请求的过程中更改上下文的状态，而策略模式只是按照不同的算法处理算法逻辑，而且从实际场景来讲，顾名思义，状态模式改变的是状态，策略模式改变的是策略

职责链模式

设计原则：遵循迪米特

• 常用场景：一个请求的处理需要多个对象当中的一个或几个协作处理

• 使用概率：15%

• 复杂度：中

• 变化点：处理链的长度与次序

- 选择关键点：对于每一次请求是否每个处理的对象都需要一次处理机会
- 逆鳞：无

解释器模式

- 设计原则：遵循单一职责
- 常用场景：有一种语言被频繁的使用
- 使用概率：0.00009%
- 复杂度：中高
- 变化点：语言的规则
- 选择关键点：被频繁使用的语言是否可用文法表示
- 逆鳞：语言的规则无限制增长或规则十分不稳定

中介者模式

- 设计原则：遵循迪米特，破坏单一职责
- 常用场景：一个系列的对象交互关系十分复杂
- 使用概率：10%
- 复杂度：中
- 变化点：对象之间的交互

- 选择关键点：复杂的交互关系是否有共性可被中介者承担
- 逆鳞：中介者无法工作

访问者模式

- 设计原则：遵循倾斜的开闭原则
- 常用场景：作用于一个数据结构之上的操作经常变化
- 使用概率：5%
- 复杂度：高
- 变化点：数据结构之上的操作
- 选择关键点：数据结构是否稳定以及操作是否经常变化
- 逆鳞：数据结构的不稳定

策略模式

- 设计原则：遵循单一职责、依赖倒置、迪米特、开闭原则
- 常用场景：算法或者策略需要经常替换
- 使用概率：60%
- 复杂度：中
- 变化点：策略的种类

- 选择关键点：客户端是否依赖于某一个或若干个具体的策略
- 逆鳞：无

备忘录模式

- 设计原则：遵循迪米特、开闭原则
- 常用场景：需要在对象的外部保存该对象的内部状态
- 使用概率：5%
- 复杂度：中
- 变化点：无
- 选择关键点：是否可以在必要的时候捕捉到对象的内部状态
- 逆鳞：大对象的备份

迭代器模式

- 设计原则：遵循迪米特
- 常用场景：需要迭代访问一个聚合对象中的各个元素，且不暴露该聚合对象内部的表示
- 使用概率：99.99999%

- 复杂度：中

- 变化点：聚合对象的种类

- 选择关键点：客户端是否关心遍历的次序

- 逆鳞：无

- 相关设计模式

- 访问者模式：二者都是迭代的访问一个聚合对象中的各个元素，不同的是，访问者模式中，扩展开放的部分在作用于对象的操作上，而迭代器模式中，扩展开放的部分在聚合对象的种类上，而且二者的实现方式也有着很大的区别。

结束语

以上便是24种设计模式的各个特点与部分模式的对比，如果总结的过程当中有疏漏或是错误，请各位不吝赐教，LZ感激不尽。

此外需要说明的是，上面的概率当中有的会出现99.99999%这样的数字，这是因为这个模式已经嵌入到JAVA类库或是我们常用的开源框架当中（标注一下：LZ总结的主要针对WEB开发，android的开发LZ并未接触过，所以不包括在此列），比如迭代器模式，只要你使用过ArrayList或者HashSet，LZ就认为这个模式被使用。这个使用频率从某种意义上讲，可以认为是该模式的重要程度，当然由于这个频率是LZ制定的，所以仅代表个人观点。

这里再针对设计模式的学习，给各位提一点点建议，仅供参考，请各位吸优排差，次序不分先后。

1、之前说过，学习设计模式除了努力之外还要靠缘分，所以如果有设计模式当时怎么看都不明白，可以暂且放下，之后说不定哪天你突然之间就明白了。（此话并非虚言，LZ很多次的顿悟常发生在上厕所、洗澡、回家路上等一些学习之外的时候。）

2、对于已经在工作的人来说，可以常思考一下，有没有哪个设计模式可以改善现有的系统架构，但不要轻易付诸实践。

3、学习设计模式之前，一定要先整明白UML类图，什么关联，依赖，聚合，组合等等都得搞明白儿的，否则学习起来也依然会很吃力。

4、对于初学者，一定要在弄清楚标准的实现代码之后，写一个属于自己的例子，哪怕是比葫芦画瓢，然后仔细体会设计模式使用前后的差异，主要从扩展性和类（类包括客户端，而不仅仅指设计模式中的角色）的职责两个方面。

5、一定要将设计模式的变化点搞清楚，这点非常重要，甚至重要程度高于设计模式的场景、实现方式以及类和对象之间的耦合关

系，很多时候，设计模式的滥用就是因为变化点没搞清楚，以至于该变化的没变化，不该变化的经常变化，增加系统的负担。

6、设计模式不是一次性学习完就可以扔掉不看的东西，而是要经常回过头来看看，说不定每一次你都有不一样的体会，而且一般情况下，这些体会会越来越深刻，越来越透彻。

7、如果可能的话，多研究一些开源框架，去找找它们里面的设计模式。

LZ暂时也就只能想到这些，如果以后有想到再补充吧，各位如果有什么好的建议也可以与LZ分享一下。

到此为止，整个设计模式系列就真真正正的彻底结束了，当初写的时候也没想到自己可以真的坚持下来，虽说整整26篇文章不算多，但是LZ确实花费了大量的时间和精力，值得欣慰的是LZ本人也得到了巨大的收获，不仅仅是对设计模式的理解日益加深，而且还得了不少猿友的支持，让LZ对分享这一道路更加坚定。

以后的编程之路还很长，对于LZ来说，编程并不仅仅是工作，而是一份事业，它给了LZ荣誉、金钱、成就感等等很多东西，希望各位至少在年轻的时候不要被一些悲观化的观点所干扰，特别是对编程有着热爱的猿友们，极致才能成就大道，但凡在一个领域有所成就者，大都是钻研了数十年的成果。

当然，人各有志，LZ无法左右他人，但LZ已经决定了自己的路，学火影里鸣人的一句话，“这就是我的忍道。”

来源： http://www.cnblogs.com/zuoxiaolong/p/pattern26.html