理解：

java这么多种设计模式

一类是利用java的特性：

* 单例模式（利用静态内部类，使用时才加载一次的特性）

一类是用高明的技巧处理java中类与类的关系：

java中类与类的关系主要有：继承，实现，依赖，关联（聚合，组合）

继承和实现就是集成类和实现接口。

依赖指的是：**A 类 和 B 类 ，A类中有个方法的参数是B类的对象**

**如：A 中有方法 void methodName(B b);**

关联指的是：**A 类 和 B 类 ，A类中有个类变量是B类的对象**

**而聚合和组合实际上就是关联，区别就B代表的语意 和 A代表 的语意的关系**

**如，A 代表人，B 代表手，这就是组合。**

1.单例模式：

单例模式是指，单例类在所有地方使用的都是同一个对象。即：只能通过静态方法获取对象，不能通过构造方法new对象。

要实现主要有两种方式：

1. 饿汉式（简单粗暴）：

|  |
| --- |
| **public class SingleModel** {  */\* \* 单例模式1 饿汉式 \* 缺陷是：不管是否使用这个类，都会创建出这样一个对象。有一定内存消耗。 \* 好处是，线程安全。 \* \*/* **private static SingleModel** *singleModel*=**new** SingleModel();  **private** SingleModel(){}  **public static SingleModel** getSingleModel(){  **return** *singleModel*;  } } |

2）懒汉式(静态类实现)

|  |
| --- |
| /\*\* 静态内部类 \* 静态成员，只有被创建对象或者被显示调用的时候，才会加载，且只加载一次。  \* Created by horse on 2017/4/30.  \*/  public class SingleModel3 {  public SingleModel3(){  System.out.println("single 初始化");  }  //static成员是位于一个类的外部级别。  private static class loadLazy{  static {  System.out.println("loadlazy jingtai");  }  //final 修饰的变量，不能再被初始化了。  public loadLazy(){ //当 类.静态成员 的时候，并不会执行构造方法，之后实例化才会。  System.out.println("loadlazy");  }  private static final SingleModel3 SINGLE\_MODEL=new SingleModel3();  }  static{  System.out.println("执行静态代码快");  }  public static SingleModel3 getSingleModel(){  //真正调用了getSingleModel方法时，才会加载loadLazy。  return loadLazy.SINGLE\_MODEL;  }  **任何一个类，都只在其被用到的时候才会加载。所以静态内部类，在使用的时候才被加载。即：**  **\* 调用 getSingleModel方法的时候，才会被使用。**  **\* 这样就避免了饿汉式，一开始就创立一个对象。**  **\* 当实例化 或者 类.静态成员 的时候**  **\* 1.获取 类的 .class字节码文件。**  **\* 2. 执行静态代码块 和 初始化静态成员。 静态内部类无任何影响。**  **\* 3. 如果要实例化，那么在堆中的新生代中创建一个对象，并把基本类型初始化。**  **\* 4.调用构造函数。** |

2.代理模式之动态代理：

1 ) 代理模式：

代理模式就是为了隐藏被代理类，并且起到控制代理类行为的作用。

要点：

1. 被代理类和代理类都要实现同一接口。
2. 代理类中要持有被代理类的对象。

|  |
| --- |
| 三要素：抽象接口，被代理类，代理类。 |
| 抽象接口：  public interface KindWomen {  public void throwEye();  public void doSomething()  }起来不 |
| 被代理类：  public class PanJinlian implements KindWomen {  public void throwEye(){  System.out.println("抛媚眼");  }  public void doSomething(){  System.out.println("doSomething");  }  } |
| 代理类：  public class WangPo implements KindWomen {  private KindWomen kindWomen; //持有被代理类的对象。  public WangPo(KindWomen kindWomen){  this.kindWomen=kindWomen;  }  @Override  public void throwEye() {  if(true) //控制突出了代理的功能。  kindWomen.throwEye();  System.out.println("传递给武松"); //加强功能突出了装饰的功能。  }  @Override  public void doSomething() {  kindWomen.doSomething();  }  } |

代理模式的缺陷：

一个被代理类就必须有一个相应的代理类，代码复杂。那么是否存在多个被代理类，一个代理类呢？

2）动态代理：

两元素： 被代理类，抽象接口。

|  |
| --- |
| 抽象接口：  public interface Person {  void say(String string);  } |
| 被代理类：  public void say(String string) {  System.out.println("lalalalla"+string);  } |
| 动态代理的实现：  public class ProxyDemo {  public static void main(String[] args) {  final Student student=new Student();  \* Person person= (Person) Proxy.newProxyInstance(student.getClass().getClassLoader(), student.getClass().getInterfaces(),  new InvocationHandler() {  @Override  public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {  boolean flag=false;  Object object=null;  System.out.println("abc");  if (flag)  object=method.invoke(student,args);  System.out.println("zzzzzzzzzzzzzzzz");  return object;  }  });  \* person.say("zzzz");  }  } |

蓝\*处，就相当于调用了invoke方法

Person person 表示通过Proxy类的方法，创建了一个代理对象，这个代理对象用接口接受（实际上，他们是虚拟的因为并没有代理类）。

**缺陷：动态代理对象只能用接口接收，即代理类必须实现一个接口**

当person.say即：调用了invoke方法，执行红颜色的代码。

注意：动态代理得到的代理对象,调用接口或者类中的任何方法都会执行invoke方法，且通过method.invoke(被代理类,args); 运行调用的方法。

|  |
| --- |
| 首先看到student中有两个方法：  **public class Student implements** Person{   @Override  **public void** say(**String** string) {  **System**.out.println("调用invoke方法 "+string);  }  **public void** eat(){  **System**.out.println("也调用了invoke方法");  } }  再看通过得到的代理对象  person调用两个方法的结果 |
| **person.say(“调用say方法”)；** |
| **abc**  **调用invoke方法 调用say方法**  **zzzzzzzzzzzzzzzz** |
| **person.eat(“调用say方法”)；** |
| **abc**  **也调用了invoke方法**  **zzzzzzzzzzzzzzzz** |

两个结果，除了中间调用的方法不同，前后都是相同的。

基于动态代理，对每一个方法都可以都会执行invoke方法，且能通过

**if** ("eat".equals(method.getName()))  
 object=method.invoke(student,args);

来控制具体方法，因此，里面的代码可以写一些共有代码，例如：

日志记录，性能统计，安全控制，事务处理，异常处理等等

**3）动态代理的扩展应用**

动态代理是 aop切面编程的基础。在spring中，有一个接口BeanPostProcessor接口，用一个公共类，实现这个接口，就会重写两个方法：

@Override  
**public Object** postProcessBeforeInitialization(**Object** bean, **String** beanName) **throws** BeansException {  
 **System**.out.println("第五步"); //spring中bean生命周期经过的第五个阶段  
 **System**.out.println(beanName);  
 **return** bean ;  
}  
  
@Override  
**public Object** postProcessAfterInitialization(**Object** o, **String** s) **throws** BeansException {  
 **System**.out.println(o);  
 **return null**;  
}

而在spring中每一个配置的bean都会在业务代码执行的前后，执行这两个方法。

**这里跟上面的区别，是每个bean对象都会共用一段代码，而上面是一个对象中的每个方法都会共用一段代码。**

**注意：在spring中，一个类可以有多个bean，只要他们在bean标签配置的id不一样就可以了。**

<**bean** id="demo3" class="springDemo.SpringDemo" scope="singleton" ></**bean**>

<**bean** id="demo4" class="springDemo.SpringDemo" scope="prototype" ></**bean**>

**4）Proxy动态代理原理讲解**

java的动态代理的关键是两个类 一个是Proxy 一个是InvocationHandler

Person person= (Person) **Proxy**.newProxyInstance(student.getClass().getClassLoader(),  
 student.getClass().getInterfaces(),  
 **new** InvocationHandler() {  
 @Override  
 **public Object** invoke(**Object** proxy, Method method, **Object**[] args) **throws Throwable** {  
 **boolean** flag=**false**;  
 **Object** object=**null**;  
 **System**.out.println("abc");  
 **if** (flag)  
 object=method.invoke(student,args);  
 **System**.out.println("zzzzzzzzzzzzzzzz");  
 **return** object;  
 }  
 });  
person.say("zzzz");

**让我们一步一步的来理清是怎么动态代理的：**

**明确需要代理的类是 student，实现的接口是Person**

**1.通过Proxy.newProxyInstance方法把需要代理的类的 类加载器，接口，和InvocationHandler类传过去。**

**2.这个方法的内部：**

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

**//首先把被代理类的接口赋给intfs**  
  
 **final Class**<?>[] intfs = interfaces.clone();

*// 通过getProxyClass得到student的代理类*

*// 这个代理类 实现了Proxy类 并且 实现了Person 接口。* **Class**<?> cl = getProxyClass0(loader, intfs);  
  
 **final Constructor**<?> cons = cl.getConstructor(constructorParams);

// 这个方法返回的是Proxy0 proxy0=new Proxy0(InvocationHandler h); 返回了Proxy0的一个对象赋给了 Person person  
 **return** cons.newInstance(**new** Object[]{h});  
   
 }  
}

**在经过上述方法之后，我们如愿的得到了student的代理类。**

**这个方法完成了两件事：**

  1.根据参数loader和interfaces调用方法 getProxyClass(loader, interfaces)创建代理类$Proxy0.$Proxy0类 实现了interfaces的接口,并继承了Proxy类.   
        2.实例化$Proxy0并在构造方法中把DynamicSubject传过去,接着$Proxy0调用父类Proxy的构造器,为h赋值,

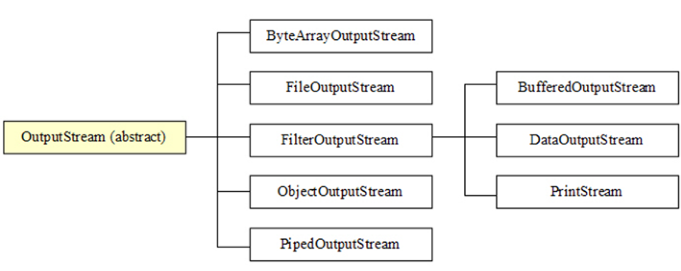
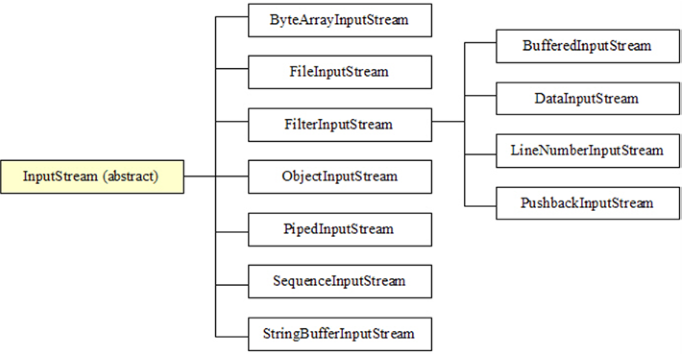
person.say("zzzz"); 是怎么调用invocationHandler中的invoke方法呢？

被代理类中的方法是这样的。

1. **public** **final** **void** say() {
2. **try** {
3. **super**.h.invoke(**this**, m3, **null**);
4. **return**;
5. } **catch** (Error e) {
6. } **catch** (Throwable throwable) {
7. **throw** **new** UndeclaredThrowableException(throwable);
8. }
9. }

**因为在创建代理类的时候把传给newProxyInstance的InvocationHandler对象，传给了代理类。因此调用相应方法的时候，就会调用到InvocationHandler中的invoke 方法。**

3. I0类装饰者模式：

Java的java.io包中囊括了整个流的家族，输出流和输入流的谱系如下所示：  
  


3.1 IO中的装饰器模式

装饰器模式（Decorator Pattern）允许向一个现有的对象添加新的功能，同时又不改变其结构。这种类型的设计模式属于结构型模式，它是作为现有的类的一个包装。

这种设计模式创建了一个装饰类，用来包装原有的类，并在保持类方法签名完整性的前提下，提供了额外的功能。

以InputStream为例，它是一个抽象类:

**public** **abstract** **class** **InputStream** **implements** **Closeable** {

...

...

}

并定义有抽象方法

**public** **abstract** **int** **read**() **throws** IOException;

该抽象方法由具体的子类去实现，通过InputStream的族谱图可以看到，直接继承了InputStream，并且提供某一特定功能的子类有：

* ByteArrayInputStream
* FileInputStream
* ObjectInputStream
* PipedInputStream
* SequenceInputStream
* StringBufferInputStream

这些子类都具有特定的功能，比如说，FileInputStream代表一个文件输入流并提供读取文件内容的功能，ObjectInputStream提供了对象反序列化的功能。

InputStream这个抽象类有一个子类与上述其它子类非常不同，这个子类就是**FilterInputStream**，可参见上图中的InputStream族谱图。

翻开FilterInputStream的代码，我们可以看到，它内部又维护了一个InputStream的成员对象，并且它的所有方法，都是调用这个成员对象的同名方法。换句话说，FilterInputStream它什么事都不做。就是把调用委托给内部的InputStream成员对象。如下所示：

|  |
| --- |
| **public class FilterInputStream extends** InputStream {  **protected volatile** InputStream in;   **protected** FilterInputStream(InputStream in) {  **this**.in = in;  }   **public int** read() **throws IOException** {  **return** in.read();  }   **public int** read(**byte** b[]) **throws IOException** {  **return** read(b, 0, b.length);  }   **public int** read(**byte** b[], **int** off, **int** len) **throws IOException** {  **return** in.read(b, off, len);  }   **public long** skip(**long** n) **throws IOException** {  **return** in.skip(n);  }   **public int** available() **throws IOException** {  **return** in.available();  }   **public void** close() **throws IOException** {  in.close();  }   **public synchronized void** mark(**int** readlimit) {  in.mark(readlimit);  }   **public synchronized void** reset() **throws IOException** {  in.reset();  }   **public boolean** markSupported() {  **return** in.markSupported();  } } |

FilterInputStream的又有其子类，分别是：

* BufferedInputStream
* DataInputStream
* LineNumberInputStream
* PushbackInputStream

虽然从上面代码看FilterInputStream并没有做什么有卵用的事，但是它的子类可不同了，以BufferedInputStream为例，这个类提供了提前读取数据的功能，也就是缓冲的功能。可以看看它的read方法：

|  |
| --- |
| **public synchronized int** read() **throws IOException** {  **if** (pos >= count) {  fill();  **if** (pos >= count)  **return** -1;  }  **return** getBufIfOpen()[pos++] & 0xff; } |

可以看到，当pos>=count时，意即需要提前缓冲一些数据的时候到了，那么就会调用fill()将缓冲区加满，以便后续读取。由于本文只讨论io流的装饰器模式，所以关于具体实现细节将不会展开讨论，比如本文不会讨论fill()方法是如何实现的，在这里可以先将它当做一个黑盒子。

从这里可以开始感受到，BufferedInputStream就是一个装饰者，它能为一个原本没有缓冲功能的InputStream添加上缓冲的功能。

比如我们常用的FileInputStream，它并没有缓冲功能，我们每次调用read，都会向操作系统发起调用索要数据。假如我们通过BufferedInputStream来**装饰**它，那么每次调用read，会预先向操作系统多拿一些数据，这样就不知不觉中提高了程序的性能。如以下代码所示：

BufferedInputStream bis = **new** BufferedInputStream(**new** FileInputStream(**new** File("/home/user/abc.txt")));

同理，对于其它的FilterInputStream的子类，其作用也是一样的，那就是装饰一个InputStream，为它添加它原本不具有的功能。OutputStream以及家属对于装饰器模式的体现，也以此类推。

JDK中的io流的设计是设计模式中装饰器模式的一个经典示范，如果细心发现，JDK中还有许多其它设计模式的体现，比如说监听者模式等等。

3.2 装饰器模式和继承的优劣：

继承和装饰的区别：

**1、以前是通过继承将每一个子类都具备缓冲功能。**

**那么继承体系会复杂，并不利于扩展。**

**2、现在优化思想，单独描述一下缓冲内容。**

**将需要被缓冲的对象。传递进来。也就是，谁需要被缓冲，谁就作为参数传递给缓冲区。**

**这样继承体系就变得很简单。优化了体系结构。**

**3、装饰模式比继承要灵活，避免了继承体系臃肿。**

**而且降低了类于类之间的关系。**

这篇博客：清晰的讲述了装饰模式的好处：

<http://blog.csdn.net/qq_27093465/article/details/53323187>

装饰器模式的好处上面已经说了，但是，继承的好处是什么呢？

父类可以接受它的所有子类！！多态的实现：

|  |
| --- |
| 上面的类的实现，扩展性很差。只要添加了新数据类型，那么就得新添构造函数。还得继续优化。 所以，又有如下的优化。 找到所有构造函数的参数的共同类型。通过多态的形式。可以提高扩展性。   1. /\*\* 2. \* 通过多态的形式。可以提高扩展性。 3. \* 现在传入的参数就可以是MyTextReader、MyMediaReader、... 4. \* 后期再出现新的子类，也可以接受 5. \*/ 6. **class** MyBufferReader **extends** MyReader { 7. **private** MyReader r; 9. MyBufferReader(MyReader r) { 10. } 11. } |

3.3 （重点）装饰者模式的概念和示例

什么是装饰模式：

在不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。是通过创建一个包装对象来实现的。

装饰模式的特点：

（1） 装饰对象和真实对象有相同的接口。这样客户端对象就能以和真实对象相同的方式和装饰对象交互。  
（2） 装饰对象包含一个真实对象的引用（reference）  
（3） 装饰对象接受所有来自客户端的请求。它把这些请求转发给真实的对象。  
（4） 装饰对象可以在转发这些请求以前或以后增加一些附加功能。这样就确保了在运行时，不用修改给定对象的结构就可以在外部增加附加的功能。在面向对象的设计中，通常是通过继承来实现对给定类的功能扩展。

模拟IO类的装饰者示例：

InputStream:

**public abstract class** Input {  
 /\*\*  
 \* 抽象类的方法可以实现，也可以不实现。  
 \* 但这个类要实例化，必须通过继承它的子类来实现。  
 \*/  
 **public void** read(){  
 **System**.out.println("read 流");  
 };  
 **public void** write(){  
 **System**.out.println("write 流");  
 };  
  
}

继承了它的子类FilterInputStream：

**public class FilterInput extends** Input {  
 **private** Input in;  
 **public** FilterInput(){  
 }  
 **public** FilterInput(Input in){  
 **this**.in=in;  
 }  
 **public void** read() {  
 in.read();  
 }  
 **public void** write() {  
 in.read();  
 }  
}

装饰类：BufferInputStream:

**public class BufferInput extends FilterInput**{  
 **protected** Input in;  
 **public** BufferInput(Input in) {  
 **this**.in=in;  
 }  
 **public void** read() {  
 buffer();  
 in.read();  
 }  
 **public void** write() {  
 buffer(); //给传过来的input家族的类增加功能。  
 in.write();  
 }  
 **private void** buffer(){  
 **System**.out.println("添加缓冲功能");  
 }  
}

使用装饰类和不使用装饰类的对比：

**public class DecoratorTest** {  
 @org.junit.Test  
 **public void** test(){  
 // Input in=new Input(); 不能直接实例化  
 Input in=**new** FilterInput(**new** FileInput()); //这样也是不行的。因为FilterInput中持有一个Input类型  
 in.read();   
 **BufferInput** bufferInput=**new** BufferInput(in);  
 bufferInput.read();  
 }  
}

输出结果：

文件流

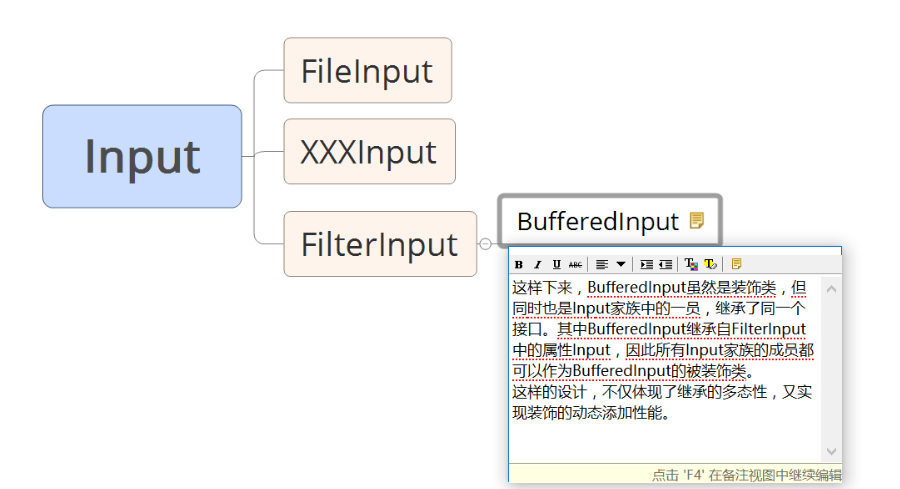
read 流

添加缓冲功能

文件流

read 流

分析：

  
装饰类和被装饰类拥有同一个接口。

4.适配器模式

4.1 适配器模式示例：

适配器模式，简单的说就是，在不改变旧的类的情况下，在新的接口中使用它。

这样的一个例子：

公司本来使用的是

**public interface** SourceOrderApi {  
 **public void** updateDate(**String** orderId,**String** date,**String** client);  
}

它的实现类

**public class SourceOrderApiImp implements** SourceOrderApi {  
 @Override  
 **public void** updateDate(**String** orderId, **String** date, **String** client) {  
 **System**.out.println(client+"已将订单"+orderId+"的有效期延长至"+date);  
 }  
}

调用的时候

**public void** testAdapter1(){  
 SourceOrderApi sourceOrderApi = **new** SourceOrderApiImp();  
 sourceOrderApi.updateDate("123456", "2014-10-15", "user");  
}

但是，公司项目客户提出了一个新的接口需求：

**public interface** AppOrderApi {  
 //只需要传入订单Id即可  
 **public void** updateDate(**String** orderId,**String** client);  
}

但是，我们原来的接口是updateDate(String orderId,String date,String client);

如果要把这个接口改成新的接口，那么很多地方需要改动，这样非常的麻烦。我们就可以通过适配器模式来实现新接口的类：

**public class AppOrderApiImp implements** AppOrderApi {  
 **private** SourceOrderApi sourceOrderApi;  
 **public** AppOrderApiImp(){  
 sourceOrderApi=**new** SourceOrderApiImp();  
 }  
 @Override  
 **public void** updateDate(**String** orderId, **String** client) {  
 //这样就完成了旧接口到新接口的适配  
 sourceOrderApi.updateDate(orderId,"",client);  
 }  
}

这样，我们就可以采用新接口调用旧接口。

4.2 适配器模式的概念

Target：适配目标，也就是上面的新接口。

adaptee:需要适配的源，也就是上面的旧接口。

adapter:适配器。

这是网上一个博客中的一段话：

|  |
| --- |
| 只需要将原接口转化为客户希望的另一个接口，就是适配器模式！  转化无非就是1.继承原类或者实现原接口  2.持有原接口的对象  再实现目标接口。。。。  那么第一种就是类的适配，第二种就是对象的适配！仅此而已。。。。 |

4.3 缺省的适配器模式

缺省适配(Default Adapter)模式为一个接口提供缺省实现，这样子类型可以从这个缺省实现进行扩展，而不必从原有接口进行扩展。作为适配器模式的一个特例，缺省是适配模式在JAVA语言中有着特殊的应用。

鲁智深的故事

　　和尚要做什么呢？吃斋、念经、打坐、撞钟、习武等。如果设计一个和尚接口，给出所有的和尚都需要实现的方法，那么这个接口应当如下：

复制代码

public interface 和尚 {

public void 吃斋（）；

public void 念经（）；

public void 打坐（）；

public void 撞钟（）；

public void 习武（）；

public String getName();

}

复制代码

　　显然，所有的和尚类都应当实现接口所定义的全部方法，不然就根本通不过JAVA语言编辑器。像下面的鲁智深类就不行。

复制代码

public class 鲁智深 implements 和尚{

public void 习武(){

拳打镇关西；

大闹五台山；

大闹桃花村；

火烧瓦官寺；

倒拔垂杨柳；

}

public String getName(){

return "鲁智深";

}

}

复制代码

　　由于鲁智深只实现了getName()和习武()方法，而没有实现任何其他的方法。因此，它根本就通不过Java语言编译器。鲁智深类只有实现和尚接口的所有的方法才可以通过Java语言编译器，但是这样一来鲁智深就不再是鲁智深了。以史为鉴，可以知天下。研究一下几百年前鲁智深是怎么剃度成和尚的，会对Java编程有很大的启发。不错，当初鲁达剃度，众僧说：“此人形容丑恶、相貌凶顽，不可剃度他",但是长老却说：”此人上应天星、心地刚直。虽然时下凶顽，命中驳杂，久后却得清净。证果非凡，汝等皆不及他。”

　　原来如此！看来只要这里也应上一个天星的话，问题就解决了！使用面向对象的语言来说，“应”者，实现也；“天星”者，抽象类也。

复制代码

public abstract class 天星 implements 和尚 {

public void 吃斋(){}

public void 念经(){}

public void 打坐(){}

public void 撞钟(){}

public void 习武(){}

public String getName(){

return null;

}

}

复制代码

　　鲁智深类继承抽象类“天星”

复制代码

public class 鲁智深 extends 天星{

public void 习武(){

拳打镇关西；

大闹五台山；

大闹桃花村；

火烧瓦官寺；

倒拔垂杨柳；

}

public String getName(){

return "鲁智深";

}

}

复制代码

　　这个抽象的天星类便是一个适配器类，鲁智深实际上借助于适配器模式达到了剃度的目的。此适配器类实现了和尚接口所要求的所有方法。但是与通常的适配器模式不同的是，此适配器类给出的所有的方法的实现都是“平庸”的。这种“平庸化”的适配器模式称作缺省适配模式。

　　在很多情况下，必须让一个具体类实现某一个接口，但是这个类又用不到接口所规定的所有的方法。通常的处理方法是，这个具体类要实现所有的方法，那些有用的方法要有实现，那些没有用的方法也要有空的、平庸的实现。

　　这些空的方法是一种浪费，有时也是一种混乱。除非看过这些空方法的代码，程序员可能会以为这些方法不是空的。即便他知道其中有一些方法是空的，也不一定知道哪些方法是空的，哪些方法不是空的，除非看过这些方法的源代码或是文档。

　　缺省适配模式可以很好的处理这一情况。可以设计一个抽象的适配器类实现接口，此抽象类要给接口所要求的每一种方法都提供一个空的方法。就像帮助了鲁智深的“上应天星”一样，此抽象类可以使它的具体子类免于被迫实现空的方法。

缺省适配模式的结构

　　缺省适配模式是一种“平庸”化的适配器模式。

public interface AbstractService {

public void serviceOperation1();

public int serviceOperation2();

public String serviceOperation3();

}

复制代码

public class ServiceAdapter implements AbstractService{

@Override

public void serviceOperation1() {

}

@Override

public int serviceOperation2() {

return 0;

}

@Override

public String serviceOperation3() {

return null;

}

}

复制代码

　　可以看到，接口AbstractService要求定义三个方法，分别是serviceOperation1()、serviceOperation2()、serviceOperation3()；而抽象适配器类ServiceAdapter则为这三种方法都提供了平庸的实现。因此，任何继承自抽象类ServiceAdapter的具体类都可以选择它所需要的方法实现，而不必理会其他的不需要的方法。

**适配器模式的用意**是要改变源的接口，以便于目标接口相容。**缺省适配的用意稍有不同**，它是为了方便建立一个不平庸的适配器类而提供的一种平庸实现。

　　在任何时候，如果不准备实现一个接口的所有方法时，就可以使用“缺省适配模式”制造一个抽象类，给出所有方法的平庸的具体实现。这样，从这个抽象类再继承下去的子类就不必实现所有的方法了。

5.代理模式、装饰模式、适配器模式的三种方式的异同。

其实适配器模式，装饰者模式，和代理模式其实感觉原理都是差不多的，都是持有一个对象，然后对其对象进行操作。

代理模式： 代理类，被代理类，接口。

代理类中持有被代理类，且两者都继承同一接口。

装饰模式：装饰类，被装饰类，接口。

装饰类中持有被装饰类，且两者都继承同一接口。

适配器模式：被适配类，目标接口，适配类。

把被适配类抽象成源接口，适配类中持有被适配类或者接口，实现目 标接口。

可以看出来，代理模式和装饰模式两者的区别非常的小，都是持有旧类对象，然后通过新类中对其操作，突出控制就是代理模式，突出功能增强就是装饰模式。

而适配器模式，两者接口变了，从旧接口变成新接口。但都会持有旧的对象。

实现同一接口的好处：这样客户端对象就能以和真实对象相同的方式和装饰对象交互。