# 为什么使用设计模式

## 设计经验的总结。使用设计模式是为了可重用代码、让代码更容易被他人理解、保证代码可靠性。目中合理的运用设计模式可以完美的解决很多问题，每种模式在现在中都有相应的原理来与之对应，每一个模式描述了一个在我们周围不断重复发生的问题，以及该问题的核心解决方案，这也是它能被广泛应用的原因

# 二、设计模式分类

## 设计模式分为三大类：

### 创建型模式，共五种：工厂方法模式、抽象工厂模式、单例模式、建造者模式、原型模式。

### 结构型模式，共七种：适配器模式、装饰器模式、代理模式、外观模式、桥接模式、组合模式、享元模式。

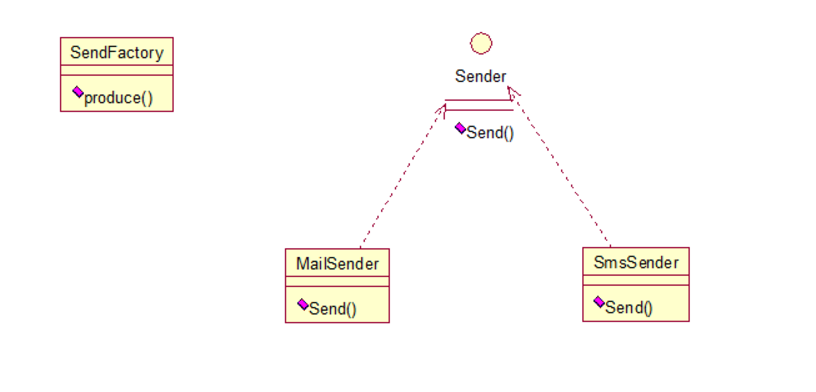
### 行为型模式，共十一种：策略模式、模板方法模式、观察者模式、迭代子模式、责任链模式、命令模式、备忘录模式、状态模式、访问者模式、中介者模式、解释器模式。

# 三、原则

## 对扩展开放，对修改封闭。在程序需要进行拓展的时候，不能去修改原有的代码，而是要扩展原有代码，实现一个热插拔的效果。所以一句话概括就是：为了使程序的扩展性好，易于维护和升级。

# 1、工厂方法模式（Factory Method）

## 1*、普通工厂模式*



### 源码实例

#### 1、接口

|  |
| --- |
| **public** **interface** Sender {  **public** **void** Send();  } |

#### 2、实现类one

|  |
| --- |
| **public** **class** MailSender **implements** Sender {  @Override  **public** **void** Send() {  System.*out*.println("this is mailsender!");  }  } |

#### 3、实现类two

|  |
| --- |
| **public** **class** SmsSender **implements** Sender {  @Override  **public** **void** Send() {  System.*out*.println("this is sms sender!");  }  } |

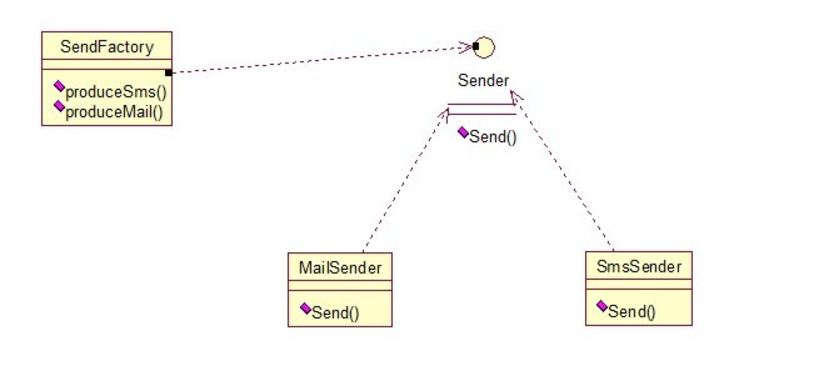
#### 4、工厂类（总，将来这里会进行产出上面的实现类）

|  |
| --- |
| **public** **class** SendFactory {  **public** Sender produce(String type) {  **if** ("mail".equals(type)) {  **return** **new** MailSender();  } **else** **if** ("sms".equals(type)) {  **return** **new** SmsSender();  } **else** {  System.*out*.println("请输入正确的类型!");  **return** **null**;  }  }  } |

#### 5、测试类 （传入相应参数获取相关实现类）

|  |
| --- |
| **public** **class** FactoryTest {  **public** **static** **void** main(String[] args) {  SendFactory factory = **new** SendFactory();  Sender sender = factory.produce("sms");  sender.Send();  }  } |

## 2、多个工厂方法模式，是对普通工厂方法模式的改进，在普通工厂方法模式中，如果传递的字符串出错，则不能正确创建对象，而多个工厂方法模式是提供多个工厂方法，分别创建对象。关系图：



### 源码实例，只要修改上面的工厂类即可

#### 1、工厂类

|  |
| --- |
| **public** **class** SendFactory {  **public** Sender produceMail(){  **return** **new** MailSender();  }  **public** Sender produceSms(){  **return** **new** SmsSender();  }  } |

#### 2、测试类（直接调用相应对象的方法即可）

|  |
| --- |
| **public** **class** FactoryTest {  **public** **static** **void** main(String[] args) {  SendFactory factory = **new** SendFactory();  Sender sender = factory.produceMail();  sender.Send();  }  } |

## 3、静态工厂方法模式，将上面的多个工厂方法模式里的方法置为静态的，不需要创建实例，直接调用即可。

### 源码实例（只要将工程类中的方法设置为静态即可）

#### 1、工厂类

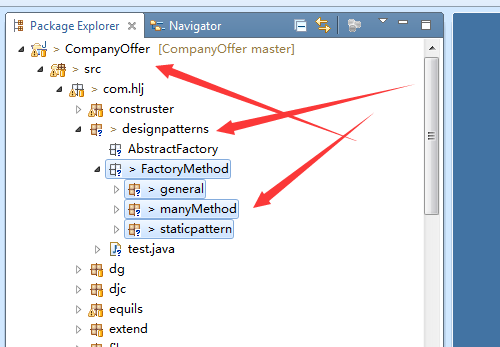
|  |
| --- |
| **public** **class** SendFactory {    **public** **static** Sender produceMail(){  **return** **new** MailSender();  }    **public** **static** Sender produceSms(){  **return** **new** SmsSender();  }  } |

#### 2、测试类（直接引用即可）

|  |
| --- |
| **public** **class** FactoryTest {  **public** **static** **void** main(String[] args) {  Sender sender = SendFactory.*produceMail*();  sender.Send();  }  } |

## 4、总结

总体来说，工厂模式适合：凡是出现了大量的产品需要创建，并且具有共同的接口时，可以通过工厂方法模式进行创建。在以上的三种模式中，第一种如果传入的字符串有误，不能正确创建对象，第三种相对于第二种，不需要实例化工厂类，所以，大多数情况下，我们会选用第三种——静态工厂方法模式。



# 2、抽象工厂模式（Abstract Factory）

工厂方法模式有一个问题就是，类的创建依赖工厂类，也就是说，如果想要拓展程序（工厂中的方法扩展），必须对工厂类进行修改，这违背了闭包原则，所以，从设计角度考虑，有一定的问题，如何解决？就用到抽象工厂模式，创建多个工厂类，这样一旦需要增加新的功能，直接增加新的工厂类就可以了，不需要修改之前的代码。因为抽象工厂不太好理解，我们先看看图，然后就和代码，就比较容易理解。

### 源码实例

#### 1、接口

|  |
| --- |
| **public** **interface** Sender {  **public** **void** Send();  } |

#### 2、实现类one

|  |
| --- |
| **public** **class** MailSender **implements** Sender {  @Override  **public** **void** Send() {  System.*out*.println("this is mailsender!");  }  } |

#### 3、实现类two

|  |
| --- |
| **public** **class** SmsSender **implements** Sender {  @Override  **public** **void** Send() {  System.*out*.println("this is sms sender!");  }  } |

#### 4、提供实现类接口，将来继承它的都是工厂

|  |
| --- |
| **public** **interface** Provider {  **public** Sender produce();  } |

#### 5、实现类one工厂

|  |
| --- |
| **public** **class** SendMailFactory **implements** Provider {  @Override  **public** Sender produce(){  **return** **new** MailSender();  }  } |

#### 6、实现类2接口

|  |
| --- |
| **public** **class** SendSmsFactory **implements** Provider{  @Override  **public** Sender produce() {  **return** **new** SmsSender();  }  } |

#### 7、测试类，实现调用

|  |
| --- |
| **public** **class** Test {    **public** **static** **void** main(String[] args) {  //使用哪个，new哪个对象即可（和我在中科软中遇到的情况情况是一样的）  Provider provider = **new** SendMailFactory();  Sender sender = provider.produce();  sender.Send();  }  } |

### 总结

如果你现在想增加一个功能：发及时信息，则只需做一个实现类，实现Sender接口，同时做一个工厂类，实现Provider接口，就OK了，无需去改动现成的代码。这样做，拓展性较好！

# 3、单例模式（Singleton）

单例对象（Singleton）是一种常用的设计模式。在Java应用中，单例对象能保证在一个JVM中，该对象只有一个实例存在。这样的模式有几个好处：

1、某些类创建比较频繁，对于一些大型的对象，这是一笔很大的系统开销。

2、省去了new操作符，降低了系统内存的使用频率，减轻GC压力。

3、有些类如交易所的核心交易引擎，控制着交易流程，如果该类可以创建多个的话，系统完全乱了。（比如一个军队出现了多个司令员同时指挥，肯定会乱成一团），所以只有使用单例模式，才能保证核心交易服务器独立控制整个流程。

## 源码实例

## 1、简单的单例类

|  |
| --- |
| **public** **class** Singleton {    /\* 持有私有静态实例，防止被引用，此处赋值为null，目的是实现延迟加载 \*/  **private** **static** Singleton *instance* = **null**;    /\* 私有构造方法，防止被实例化 \*/  **private** Singleton() {  }    /\* 静态工程方法，创建实例 \*/  **public** **static** Singleton getInstance() {  **if** (*instance* == **null**) {  *instance* = **new** Singleton();  }  **return** *instance*;  }    /\* 如果该对象被用于序列化，可以保证对象在序列化前后保持一致 \*/  **public** Object readResolve() {  **return** *instance*;  }  } |

## 解释：这个类可以满足基本要求，但是，像这样毫无线程安全保护的类，如果我们把它放入多线程的环境下，肯定就会出现问题了，如何解决？我们首先会想到对getInstance方法加synchronized关键字，如下：

## 2、解决线程安全问题

|  |
| --- |
| /\*\*  \* 解决线程安全 synchronized 上锁  \*/  **public** **static** **synchronized** Singleton getInstance() {  **if** (*instance* == **null**) {  *instance* = **new** Singleton();  }  **return** *instance*; } |

## 解释：但是，synchronized关键字锁住的是这个对象，这样的用法，在性能上会有所下降，因为每次调用getInstance()，都要对对象上锁，事实上，只有在第一次创建对象的时候需要加锁，之后就不需要了，所以，这个地方需要改进。我们改成下面这个：

## 3、解决上述问题

|  |
| --- |
| **public** **class** SingletonFinal {    **private** **static** SingletonFinal *instance* = **null**;    **private** SingletonFinal() {  }    **private** **static** **synchronized** **void** syncInit() {  **if** (*instance* == **null**) {  *instance* = **new** SingletonFinal();  }  }    **public** **static** SingletonFinal getInstance() {  **if** (*instance* == **null**) {  *syncInit*();  }  **return** *instance*;  }  } |

## 解释：因为我们只需要在创建类的时候进行同步，所以只要将创建和getInstance()分开，单独为创建加synchronized关键字，也是可以的：

# 4、适配器模式

## 解释：

适配器模式（Adapter Pattern）是作为两个不兼容的接口之间的桥梁。

这种类型的设计模式属于结构型模式，它结合了两个独立接口的功能。

举例

举个真实的例子，读卡器是作为内存卡和笔记本之间的适配器。您将内存卡插入读卡器，

再将读卡器插入笔记本，这样就可以通过笔记本来读取内存卡。

何时使用

1、系统需要使用现有的类，而此类的接口不符合系统的需要。

2、想要建立一个可以重复使用的类，用于与一些彼此之间没有太大关联的一些类，包括一些可能在将来引进的类一起工作，这些源类不一定有一致的接口。

3、通过接口转换，将一个类插入另一个类系中。（比如老虎和飞禽，现在多了一个飞虎，在不增加实体的需求下，增加一个适配器，在里面包容一个虎对象，实现飞的接口。）

## 1、测试场景

我们有一个 *MediaPlayer* 接口和一个实现了 *MediaPlayer* 接口的实体类 *AudioPlayer*。默认情况下，*AudioPlayer* 可以播放 mp3 格式的音频文件。

我们还有另一个接口 *AdvancedMediaPlayer* 和实现了 *AdvancedMediaPlayer* 接口的实体类。该类可以播放 vlc 和 mp4 格式的文件。

我们想要让 *AudioPlayer* 播放其他格式的音频文件。为了实现这个功能，我们需要创建一个实现了 *MediaPlayer* 接口的适配器类 *MediaAdapter*，并使用 *AdvancedMediaPlayer* 对象来播放所需的格式。

*AudioPlayer* 使用适配器类 *MediaAdapter* 传递所需的音频类型，不需要知道能播放所需格式音频的实际类。*AdapterPatternDemo*，我们的演示类使用 *AudioPlayer* 类来播放各种格式。

## 1、媒体播放器接口，MediaPlayer

|  |
| --- |
| **public** **interface** MediaPlayer {  **public** **void** play(String audioType, String fileName);  } |

## 2、其他播放器接口，包含可以看Vlc和Mp4的格式的播放

|  |
| --- |
| **public** **interface** AdvancedMediaPlayer {  **public** **void** playVlc(String fileName);  **public** **void** playMp4(String fileName);  } |

## 3、VlcPlayer实现接口类

|  |
| --- |
| **public** **class** VlcPlayer **implements** AdvancedMediaPlayer{  @Override  **public** **void** playVlc(String fileName) {  System.*out*.println("Playing vlc file. Name: "+ fileName);  }  @Override  **public** **void** playMp4(String fileName) {  //什么也不做  }  } |

## 4、Mp4Player实现接口类

|  |
| --- |
| **public** **class** Mp4Player **implements** AdvancedMediaPlayer{  @Override  **public** **void** playVlc(String fileName) {  //什么也不做  }  @Override  **public** **void** playMp4(String fileName) {  System.*out*.println("Playing mp4 file. Name: "+ fileName);  }  } |

## 5、适配器，实现了MediaPlayer

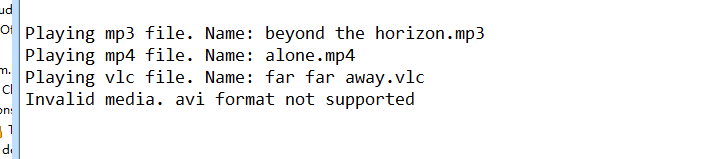
|  |
| --- |
| **public** **class** MediaAdapter **implements** MediaPlayer {  AdvancedMediaPlayer advancedMusicPlayer;  **public** MediaAdapter(String audioType){  **if**(audioType.equalsIgnoreCase("vlc") ){  advancedMusicPlayer = **new** VlcPlayer();  } **else** **if** (audioType.equalsIgnoreCase("mp4")){  advancedMusicPlayer = **new** Mp4Player();  }  }  @Override  **public** **void** play(String audioType, String fileName) {  **if**(audioType.equalsIgnoreCase("vlc")){  advancedMusicPlayer.playVlc(fileName);  }**else** **if**(audioType.equalsIgnoreCase("mp4")){  advancedMusicPlayer.playMp4(fileName);  }  }  } |

## 6、音频播放器，要实现所有的功能，里面包含了适配器对象

|  |
| --- |
| **public** **class** AudioPlayer **implements** MediaPlayer {  MediaAdapter mediaAdapter;  @Override  **public** **void** play(String audioType, String fileName) {  //播放 mp3 音乐文件的内置支持  **if**(audioType.equalsIgnoreCase("mp3")){  System.*out*.println("Playing mp3 file. Name: "+ fileName);  }  //mediaAdapter 提供了播放其他文件格式的支持  **else** **if**(audioType.equalsIgnoreCase("vlc")  || audioType.equalsIgnoreCase("mp4")){  mediaAdapter = **new** MediaAdapter(audioType);  mediaAdapter.play(audioType, fileName);  }  **else**{  System.*out*.println("Invalid media. "+  audioType + " format not supported");  }  }  } |

## 7、开始测试

|  |
| --- |
| **public** **class** TestMain {  **public** **static** **void** main(String[] args) {  AudioPlayer audioPlayer = **new** AudioPlayer();  audioPlayer.play("mp3", "beyond the horizon.mp3");  audioPlayer.play("mp4", "alone.mp4");  audioPlayer.play("vlc", "far far away.vlc");  audioPlayer.play("avi", "mind me.avi");  }  } |



# 5、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。**

**应用实例：** 1、孙悟空有 72 变，当他变成"庙宇"后，他的根本还是一只猴子，但是他又有了庙宇的功能。 2、不论一幅画有没有画框都可以挂在墙上，但是通常都是有画框的，并且实际上是画框被挂在墙上。在挂在墙上之前，画可以被蒙上玻璃，装到框子里；这时画、玻璃和画框形成了一个物体。

### 步骤 1、创建一个接口。

*Shape.java*

public interface Shape {

void draw();

}

### 步骤 2、创建实现接口的实体类。

*Rectangle.java*

public class Rectangle implements Shape {

@Override

public void draw() {

System.out.println("Shape: Rectangle");

}

}

*Circle.java*

public class Circle implements Shape {

@Override

public void draw() {

System.out.println("Shape: Circle");

}

}

### 步骤 3、创建实现了 *Shape* 接口的抽象装饰类。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。

*ShapeDecorator.java*

public abstract class ShapeDecorator implements Shape {

protected Shape decoratedShape;

public ShapeDecorator(Shape decoratedShape){

this.decoratedShape = decoratedShape;

}

public void draw(){

decoratedShape.draw();

}

}

### 步骤 4、创建扩展了 *ShapeDecorator* 类的实体装饰类。

*RedShapeDecorator.java*

public class RedShapeDecorator extends ShapeDecorator {

public RedShapeDecorator(Shape decoratedShape) {

super(decoratedShape);

}

@Override

public void draw() {

decoratedShape.draw();

setRedBorder(decoratedShape);

}

private void setRedBorder(Shape decoratedShape){

System.out.println("Border Color: Red");

}

}

### 步骤 5、使用 *RedShapeDecorator* 来装饰 *Shape* 对象。

*DecoratorPatternDemo.java*

public class DecoratorPatternDemo {

public static void main(String[] args) {

Shape circle = new Circle();

Shape redCircle = new RedShapeDecorator(new Circle());

Shape redRectangle = new RedShapeDecorator(new Rectangle());

System.out.println("Circle with normal border");

circle.draw();

System.out.println("\nCircle of red border");

redCircle.draw();

System.out.println("\nRectangle of red border");

redRectangle.draw();

}

}

### 步骤 6、验证输出。

Circle with normal border

Shape: Circle

Circle of red border

Shape: Circle

Border Color: Red

Rectangle of red border

Shape: Rectangle

Border Color: Red