# **设计模式之禅**

## 设计原则 ：

* + - **同一段在码在两个地方使用的时候就该考虑代码的合理性**
    - **7尽量符合迪米特法则，在实现模板方法的时候抽象的方法可以使用protected来修饰，对于不必暴露的方法可以使用private，尽量少用public**
    - **在使用成员变量的时候最好加上this，在调用父类方法的时候最好加上super，在使用数据类型封装数据的时候，最好先初始化clear**
    - **在学习设计模式的时候不必可以保持设计的模板，可以拆分，组合利用各种设计模式在实际应用中使用最适合当下场景的组合才是关键，在使用建造者模式的时候就组合使用了，模板模式增加了建造者模式的效率**
    - **高层模块应该尽量避免和底层模块的依赖耦合关系**

## 模板模式：

Public abstract class HummerModel{

Public abstract void start(); // 启动

Public abstract void alarm(); // 鸣笛

Public abstract void engineBoom(); //引擎

Public abstract void stop(); //停止

Public void run{ // 模板方法

Start();

Alarm();

engineBoom();

stop();

}

}

在子类继承该父类方法实现，抽象方法即可，这就是简单的模板方法了

模板方法的优点：

* 1. 封装不变部分，扩展可变部分： 将不变的部分可以放在父类的方法里面并且（如上面的run方法），可变和扩展的部分，交给子类来完成（其他的抽象方法）
  2. 提取公共代码部分便于维护 ： 减少代码的失误率，如果相同的代码分散在各个地方，那么要修改方法的时候就需要去找到相同代码的地方，修改不方便，减少了代码的可重用性
  3. 行为父类控制子类实现： 基本的方法子类来实现，行为交给了父类

符合开闭原则

缺点 ： 子类执行的方法的结果会影响父类的结果，在复杂的项目中会给代码的阅读带来难度

应用的场景 ：

1. 多个子类有公有的方法，并且逻辑基本相同的时候；
2. 将重要的复杂的核心算法交由父类完成，其他细节问题交给子类来处理
3. 重构时，将共同的方法抽取到父类当中，通过钩子函数约束其行为

在模板模式中可以通过一个钩子方法来让子类的执行结果控制父类执行结果的走向

在上面的模板类中添加一个方法 : public Boolean isAlarm(){ return true;}; 来控制是否鸣笛

在run方法中 {

If(isAlarm()){ // 子类可以根据自己的行为逻辑完成对应的鸣笛控制，重写该方法

Alarm();

}

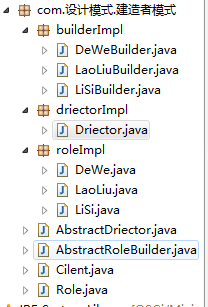
}

### 建造者模式

**建造者模式也叫生成器模式 将一个复杂的对象的创建和表示分离，同样的创建过程可以创建不同的表示**

**建造者模式下面的四个角色：**

* 1. **product 产品类 ： 封装产品的一些抽象方法属性。。。（模板方法）**
  2. **builder抽象的建造者类 ： 规范产品的组建一般由子类实现**
  3. **concreteBuilder 具体的建造者类 ： 实现抽象方法，并返回组建好的对象**
  4. **director 导演类： 负责已有模块的顺序并告诉创建者可以开始创建，在建造者比较庞大的时候可以有多个导演类**

**基本的类图 ：**

**建造者模式的优点 ：**

* 1. **封装性： 使用建造者模式，客户端不需要知道产品内部的组成细节，如同 我们不必知道怎么构建出产品的，产品的具体属性等，产生的对象就是Role类型**
  2. **建造者独立，易于扩展**

**如：LiSiBuilder,LaoLiuBuilder之间是相互独立的易于扩展**

* 1. **便于控制细节风险**

**建造者的具体实现是独立的，对于建造过程的细化对于其他模块不会有任何影响**

**建造者模式使用的场景：**

1. **有相同的方法，不同的实现顺序，得到不同的结果，可以使用建造者模式。**
2. **多个部分或者零件，都可以装配到对象当中，但产生的结果又不相同的时候可以使用建造者模式**
3. **在对象的创建过程中，需要创建其他的对象，而其他的对象又不容易获得的时候可以将其他对象使用建造者模式封装该对象的创建过程（一个对象不易获得在对象设计的时候就不应该出现这样的问题，这个封装的过程是柔化了创建过程）**

**建造者模式注重的是对象的零件类型和装配工艺（顺序），而工厂方法注重的是获取对象的形式关注点是不同的**

### 代理模式

**类图有三个角色**

* 1. **Subject抽象的主题角色： 可以是接口也可以是抽象类，定义了需要实现的方法，无特殊要求**
  2. **具体的Subject角色： 被委托角色，被代理类，业务的具体实现者**
  3. **Proxy代理主题角色： 代理类，负责对真实角色的应用，通过抽象的主题类限制委托给真实角色实现，同时提供了真实方法的预先处理和善后处理**

**简单实现：**

**Public interface Subject{**

**Void request();**

**}**

**Public class RealSubject implements Subject{**

**Public void request(){**

**//具体实现**

**}**

**}**

**Public class Proxy implements Subject{ //代理类**

**Private Subject subject;**

**Public Proxy(Subject subject){**

**This.subject = subject； //构造方法中提供真实对象**

**}**

**Public void request(){**

**This.befor();**

**This.subject.request();**

**This.after();**

**}**

**Public void befor(){**

**//在真实对象处理先做预先处理的事件**

**}**

**Public void after(){**

**//在真实对象处理之后的善后处理**

**}**

**}**

**代理模式的优点：**

* 1. **职责清晰 : 具体的实现对象就是实现具体的逻辑既可以了，不需要关心其他的非本职的工作，通过后期的代理完成一件事，附带的编程简洁清晰**
  2. **高扩展性： 在具体类发生逻辑变化的时候，对代理类没有任何影响，代理类只管调用即可（只要具体类实现了接口）**
  3. **智能化**

**代理模式使用的场景： 代理模式的使用场景非常多，在现实生活中，就像为什么找律师来处理官司一样，很多时候我们只想实现我能简单的需求就能完成一件事的时候，其他的逻辑交给代理类来处理就很方便减少了麻烦，如spring 的AOP就是典型的动态代理**

**动态代理： 将上面的代理类替换成下面的代理类即可**

**Public class SubjectProxy implements InvocationHandler{**

**Object obj = null;**

**Public SubjectProxy(Object obj){  
 this.obj = obj;**

**}**

**Public void invoke(Object proxy,Method method,Object[] args){**

**Method.invoke(obj,args);**

**// args 调用方法的参数 ， obj真实对象**

**}**

**}**

**Public static void main(String[] args){  
 Subject sub = new RealSubject();**

**InvocationHandler handler = new SubjectProxy();**

**Subject proxy = (Subject) Proxy.newProxyInstance(sub.getClass().getClassLoader(),Subject.class().getInterfaces(),handler);**

**Proxy.request();**

**// note : 这里的proxy对象是Proxy的对象 并不是返回真实的对象，这也是代理的重点**

**}**

[Spring-AOP名词解释](https://blog.csdn.net/github_34889651/article/details/51321499)

### 原型模式

* + 1. **该模式的简单程度仅次于单例模式和迭代模式，只要在创建对象的时候不是通过new的而是实现该对象的类实现了Cloneable接口，通过使用clone方法复制一个对象来使用**

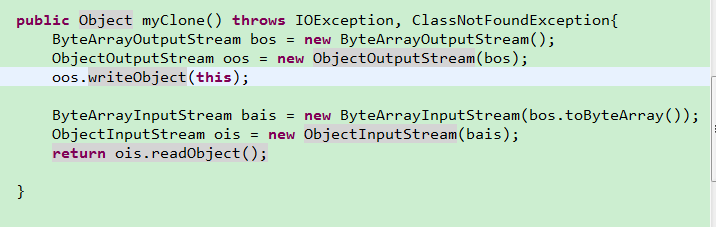
**原型模式的优点：**

* + 1. **性能优良： 使用原型模式创建的对象是基于对内存二进制流的拷贝，要比直接new一个对象性能要好，特别是在一个循环体内大量使用同一个对象的时候，就更能体现其优点了**
    2. **逃避了构造函数的约束： 这是优点也是缺点，因为是在内存的二进制流中进行复制的所以调用该方法创建对象时，是不会通过构造方法的，构造方法中的东西是不会执行的**

**原型模式的使用场景：  
1） 资源优化场景，在初始化一个类的时候，需要消耗大量的数据资源 和硬件资源**

* 1. **性能和安全要求的场景： 通过new产生一个对象的时候需要非常繁琐的数据准备或访问权限则可以使用原型模式**
  2. **一个对象多个修改者的场景： 一个对象需要提供给其他调用者的访问，而且可能调用过程中会修改其值的时候使**

**在原型模式中需要注意的地方是对象的浅拷贝和深拷贝**

* + 1. **浅拷贝 ： java在使用Object的clone()方法的时候使用的是浅拷贝，即在拷贝对象的时候偷懒了，它只是拷贝本对象，对于数组和引用类型的对象是不拷贝的直接使用的原来的那个对象**
    2. **在使用浅拷贝的时候，只有引用类型的成员变量不会拷贝，方法内的变量是会拷贝的，二必须是一个可变的对象而不是原型对象或者不可变对象，才不会被拷贝**
    3. **怎么实现深拷贝： 将需要拷贝的对象在clone方法中添加该对象自己的clone方法，就实现了该对象的深拷贝**
    4. **拷贝副本修正细节信息。。。**
    5. **自己实现深拷贝：**

**需要实现拷贝的对象不要使用final修饰**

### 中介者模式

**中介者模式减少了类间的耦合性，设计结构清晰代码质量也大大的提高了**

**中介者模式的定义：用一个中介对象封装一系列对象之间的交互，中介者模式使得对象之间不需要显示的进行交互，从而使其耦合松散，可以独立的修改他们之间的交互**

**中介者模式的组成：**

1. **Mediator中介者的抽象类，抽象中介类定义统一的接口处理同事角色之间的通信**
2. **Concrete Mediator: 具体中介角色类，中介抽象类的实现，主要负责各个同事对象之间的协作沟通，所以该类中需要依赖每个同事角色**
3. **Colleague同事类：每一个同事角色都知道一个中介者的角色，并且与其他同事进行通信的时候一定会通过中介者协作 ，每一个同事类的行为可以分为两种：**

**3.1） 第一种，处理自身的属性，状态，行为，这种行为叫做自发行为**

**3.2） 第二种，必须依赖中介者才能完成的叫依赖方法**

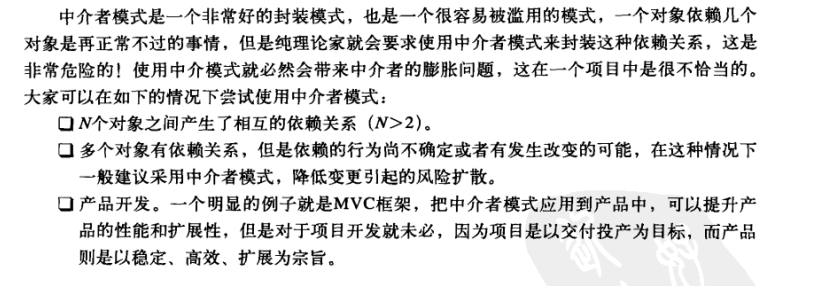
**注意： 同事类必须是构造方法注入中介者类，因为每一个同事类都是依赖于中介者类的，而中介者类可以通过set/get来注入，中介者可以只有部分同事**

**中介者模式的优点： 减少了类间的依赖，把原来一对多的依赖变成了一对一的依赖，同事类只依赖中介者类，减少了依赖，降低了类间的耦合**

**中介者模式的缺点： 中介者类会原来越膨胀，逻辑越来越复杂，随着同事类的增加中介者类越复杂**

**中介者模式的使用场景，在中介者模式在使用上需要有一定的理解，在变成了，没有对象是不存在依赖的，那样这个类就是一个孤岛，而一对多的依赖也是存在的存在即合理所以在使用上并不是说依赖过多了就使用，而是根据类图中类间的交叉依赖，使得结构复杂逻辑不清晰，的时候就可以使用中介者模式**

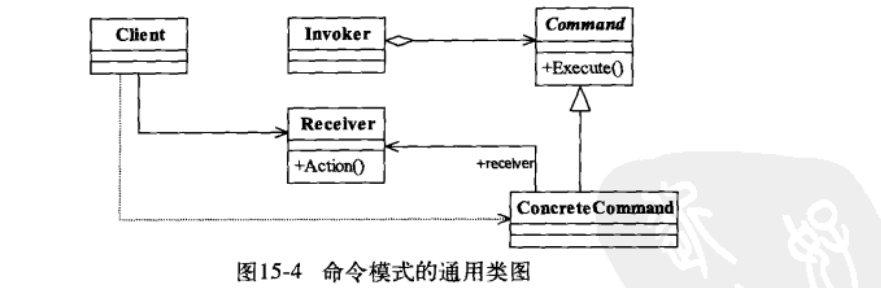
**中介者模式也叫调停者模式**

****

## 命令模式

**命令模式是一种高内聚的模式(将客户端的请求封装成一个对象，通过分析客户端请求的参数来完成对应的处理效果)**

**类图中存在三个角色：**

****

* + 1. **Receive接受者角色：命令执行到这里就应当被执行，如Group的三个实现类**
    2. **Command命令角色： 需要执行的所有命令**
    3. **Invoke调用者角色： 接受到命令并执行命令**

**模板**

**Public abstract class Receive{ //抽象接受者类**

**Public abstract void doSomething();**

**}**

**Public abstract class Command{ //抽象命令类**

**Protected Receive receive;**

**Public Command(Receive receive){**

**This.receive = receive;**

**}**

**Public abstract void execute();**

**}**

**Public class Receive1 extends Receive{ //对应的请求**

**Public void doSomething(){**

**Syso(“receive1”);**

**}**

**}**

**Public class MyCommand extends Command{ //一个命令**

**Public void execute(){**

**Super.receive.doSomething();**

**}**

**}**

**Public class Invoke{ //调用者**

**Private Command command;**

**Public Invoke(){**

**}**

**Public void action(){**

**This.command.execute();**

**}**

**Public void setCommand(Command command){**

**This.command = command;**

**}**

**Public void getCommand(Command command){**

**Return this.command;**

**}**

**}**

**命令模式的优点：**

* + 1. **类间的解耦，在调用者和接受者之间没有直接的依赖关系，调用者是通过抽象类的Command.execute() 来完成请求的，但是并不知道是那个接受者完成的**
    2. **可扩展性： 对于Command的扩展，而调用者和高层模块Client不产生严重的代码耦合**
    3. **命令模式结合其他模式会更优秀： 命令模式可以结合责任链模式实现命令族解析任务，结合模板方法可以减少Command子类膨胀的问题**

**缺点： 对于命令模式来说最严重的就是任务命令过多的时候，没一个命令的执行都是一个新的命令类，这也是实现扩展性好的前提**

## 责任链模式

**责任链模式的定义： 使得多个对象都有处理请求的机会，让请求的接受者和发送者解耦，将这些对象连成一条链并将请求传递下去，直到有对象处理它为止**

**责任链模式类**

1. **抽象的处理者需要实现请求的处理方法 handlerMessage唯一对外开放的方法 二是定义一个责任链的编排方法 setNext()设置下一个处理者，定义具体的实现者必须实现的方法： 定义能够处理的级别，处理请求的方法**

**如：**

**Public abstract class Hnadler{**

**Protected int level;**

**Protected Handler next;**

**Public Handler(){}**

**Public final void handlerMessage(Request request){**

**If(this.level == request.getLevel){  
 this.response(request);**

**}else{**

**If(next != null){**

**This.next.handlerMessage();**

**}else{**

**Syso(“没人能够处理了”);**

**}**

**}**

**}**

**Public abstract void response(Request request);**

**Public abstract void setNextHandler(Handler handler);**

**Public void setLevel(int level){ //这些个数据类型可以是自己封装的数据类型**

**This.level = level;**

**}**

**}**

**责任链模式的优点： 请求者不需要知道谁是处理者，处理者也不用知道请求者的全貌两者解耦提高系统的灵活性**

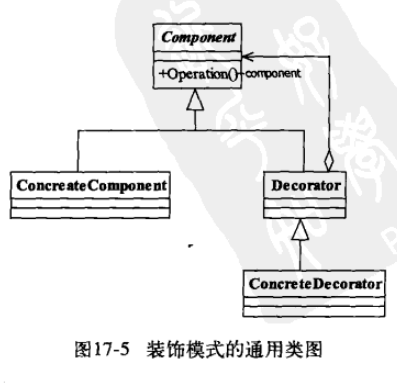
**责任链模式有两个明显的缺点：**

1. **每一个链都从链头遍历到链尾，那么当链的长度足够长的时候就会出现性能问题**
2. **调试不方便，在责任链中链条特别长，环节比较多的时候由于采用了递归调试的时候逻辑比较复杂**

**所以责任链模式需要控制其长度，特别是可以设置一个阈值，超过这个阈值就不再建立链了**

## 装饰者模式

**装饰者模式的定义： 动态的给对象添加一些额外职责，就添加功能来说装饰者模式比生成子类更灵活**

****

**装饰类四个类说明：  
1） Component： 是一个类或者接口是最为原始的对象，也是最核心的类**

1. **ConcreteComponent: 是具体的被修饰的对象**
2. **Decorator : 继承了Component的抽象装饰类，它不一定有抽象方法，但是一定有一个private的Component变量**
3. **ConcreteDecorator: 具体的装饰类，完成自己需要装饰的部分并融合到需要的位置**

**装饰者模式的优点：**

1. **装饰者类和被装饰者类之间不会相互耦合，Component类不需要知道Decorator类，Decorator类从外部对Component进行扩张，Decorator也不用知道具体的Component构件**
2. **装饰模式是继承关系的一个替代方案，我们来看Decorator不管修饰了多少层，返回的都是一个Component对象**
3. **装饰者模式可以动态的扩展一个类的功能**

**装饰者模式的缺点 : 装饰的层数决定了装饰逻辑的复杂度，当装饰层级过多，出现问题的时候需要一层层的分析，所以在使用装饰者模式的时候应该限制装饰者的数量**

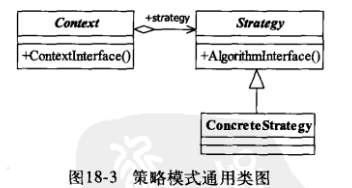
**装饰者模式的使用场景**

1. **需要动态的扩展一个类的功能，或者需要给一个类附加一个功能**
2. **需要动态的给一个类增加功能，并且这些功能可以再动态的撤销**
3. **需要一批兄弟类改装和增加一些功能**

### 策略模式

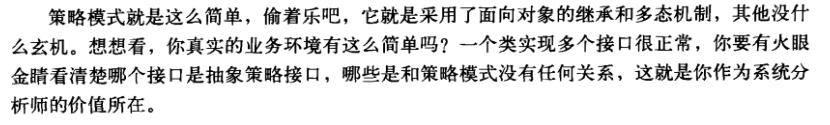
**策略模式： 定义一组算法，将每一个算法都封装起来，并且每一个算法是可以互换的**

**策略模式是用用了面向对象的继承和多态两种机制，非常易于使用**

****

**策略模式的三个角色：**

1. **Context 环境角色： 用于屏蔽高层模块对算法和策略的直接访问，封装可能存在的变化**
2. **Strategy抽象的策略角色：策略算法家族的抽象，一般使用接口，定义每一个算法具有的属性和方法（运算规则）**
3. **ConcreteStrategy 具体的算法类： 实现了抽象策略角色的抽象算法，定义了具体的算法规则**

****

**策略模式应用的优点：**

1. **算法可以自由的切换： 这是策略模式本身的定义，只要实现了抽象策略，就会成为策略家族的一员，通过封装角色的封装，对外可以自由切换成员{**

**在这里看出了和命令模式不同的地方，命令模式是一条命令对应一个执行方案是不可变的，政策模式是可变的，在使用的时候根据不同的场景使用不同的模式即可}**

1. **避免了多重条件判断语句，使用策略模式，可以在其他模块进行使用哪个算法的判断，政策家族对外提供的访问接口就是封装类，简化了操作，避免了多重条件判断**
2. **扩展性良好，想增加一个策略成员只需要实现接口就可以了，符合OCP原则（开闭原则）**

**策略模式的缺点：**

1. **很明显策略越多类就越多，而且每个策略被重用的可能性很低**
2. **在原策略模式上存在必须暴露策略的缺陷，调用类必须知道调用的策略，现在可以通过代理模式，工厂模式，享元模式来解决这个缺点**

**政策模式使用的场景：**

1. **在多个类只有算法或者行为上稍有不同的场景**
2. **算法需要自由切换的场景**
3. **需要屏蔽算法规则的场景**

命令模式和策略模式的类图完全一样，代码实现也比较类似，但是两者还是有区别的。

　　● 关注点不同

　　策略模式关注的是算法替换的问题，一个新的算法投产，旧算法退休，或者提供多种算法由调用者自己选择使用，算法的自由更替是它实现的要点。换句话说，策略模式关注的是算法的完整性、封装性，只有具备了这两个条件才能保证其可以自由切换。

　　命令模式则关注的是解耦问题，如何让请求者和执行者解耦是它需要首先解决的，解耦的要求就是把请求的内容封装为一个一个的命令，由接收者执行。由于封装成了命令，就同时可以对命令进行多种处理，例如撤销、记录等。

　　● 角色功能不同  
　　在我们的例子中，策略模式中的抽象算法和具体算法与命令模式的接收者非常相似，但是它们的职责不同。策略模式中的具体算法是负责一个完整算法逻辑，它是不可再拆分的原子业务单元，一旦变更就是对算法整体的变更。

　　而命令模式则不同，它关注命令的实现，也就是功能的实现。例如我们在分支中也提到接收者的变更问题，它只影响到命令族的变更，对请求者没有任何影响，从这方面来说，接收者对命令负责，而与请求者无关。命令模式中的接收者只要符合六大设计原则，完全不用关心它是否完成了一个具体逻辑，它的影响范围也仅仅是抽象命令和具体命令，对它的修改不会扩散到模式外的模块。

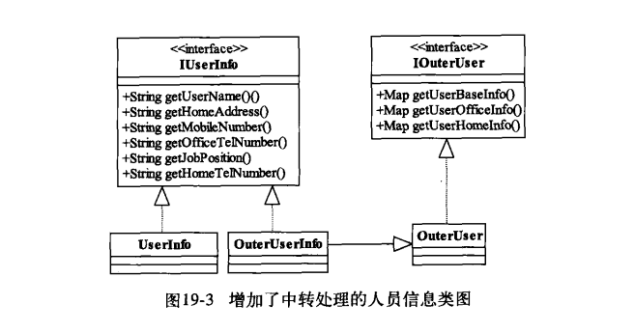
　　当然，如果在命令模式中需要指定接收者，则需要考虑接收者的变化和封装，例如一个老顾客每次吃饭都点同一个厨师的饭菜，那就必须考虑接收者的抽象化问题。

● 使用场景不同  
　　策略模式适用于算法要求变换的场景，而命令模式适用于解耦两个有紧耦合关系的对象场合或者多命令多撤销的场景。

**如果策略模式的策略超过了四个，就要考略加入其它模式来避免策略的暴露和膨胀问题**

## 适配器模式

**适配器模式的定义： 将一个类的接口变换成客户端需要的接口，从而使原本不匹配的两个类能够在一起工作**

****

**通过给一个类添加一层外衣，可以使他变成另外一个类的类型，就可以实现通用**

**类：**

1. **IUserInfo 目标角色：需要包装成的类型**
2. **IOuterUser 被包装的角色，需要适配的角色**
3. **OuterUserInfo : 适配器角色，需要将被包装的原角色变成一个IUserInfo的类型，就需要实现IUserInfo继承 OuterUser**

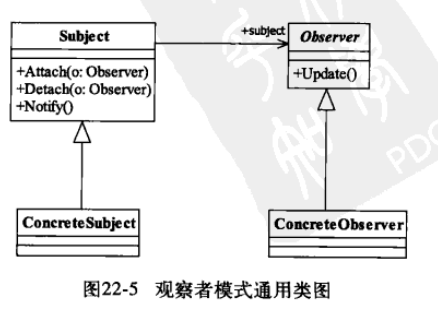
**适配器模式的优点：**

1. **适配器模式可以让两个没有任何关系的类在一起运行**
2. **增加了透明性，实现了目标角色，具体的实现都交给了具体的源角色，对高层模块来说方法是透明的**
3. **提高了类的复用性，需要适配的角色在其他位置的使用不会有任何变化，在目标角色中还可以充当新的演员**
4. **灵活性好，在不需要适配器的时候删掉就可以，不会对代码有任何影响**

**适配器模式适合在开发过后修改问题的时机使用，而不是在开发的时候使用，只有在系统扩展了不符合原来的一些规则的时候使用适配器模式减少风险**

## 观察者模式

**观察者模式的定义: 观察模式也叫订阅模式,在一个对象中添加对多个对象的依赖,如果该对象发生变化会通知其他依赖对象更新和改变**

****

**以下是观察者模式的角色介绍**

* + - 1. **被观察者 Subject: 定义了被观察者需要实现的职责,动态的添加和删除观察者,能够通知所有的观察者,他是一个抽象类,或者实现类,它的职责: 管理观察者,通知观察者**
      2. **Observer观察者：定义了动态接受到消息后对消息的处理**
      3. **ConcreteObserver 具体的观察者 ： 实现更新操作，每一个观察者都可以有自己独特的观察方式**
      4. **ConcreteSubject 具体的被观察者： 需要实现自身需要的动作，并根据不同的动作引发观察者的更新消息**

**观察者模式的优点：**

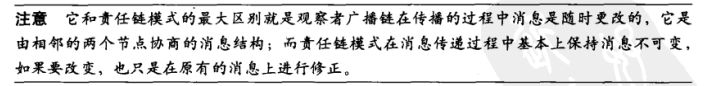
1. **实现了观察者和被观察者的抽象解耦，观察者和被观察者可以非常容易扩张**
2. **建立一套触发机制，可以实现链式的观察模式**

**缺点： 观察者模式在开发的时候需要考虑效率和运行速度，在一个被观察者多个观察者的时候调式会比较复杂，其次java的顺序执行中如果一个观察者的处理时间过长的阻塞显现会影响效率，在这时可以使用异步，多级触发的这种情况应深入考虑可行性**

**观察者模式的使用场景：**

1. **关联行为的场景而不是组合关系，是可以拆分的**
2. **事件多级触发场景**
3. **跨系统的消息交换场景，如消息队列的处理机制**

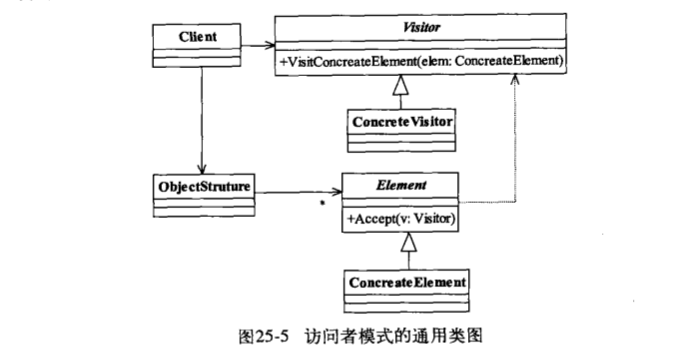
**需要注意的是在观察者模式中，最好只有一个观察者是被观察者，即使一层嵌套的意思，层级嵌套过多，逻辑复杂，维护性差**

****

**异步处理需要考虑线程安全问题和队列的问题**

## 访问者模式

**访问者模式的定义: 封装了某种数据结构中的各元素的操作,它可以在不改变数据结构的前提下给元素提供新的操作**

****

1. **Visitor抽象的访问者; 声明可以被访问的对象, 通过VisitConreateElement(Object) 传入的参数决定了那种数据结构可以被访问**
2. **ConcreteVisitor: 具体的访问者实现类, 对传入的数据结构做新的操作处理**
3. **Element : 数据结构的抽象方法, 提供了一个允许那种访问者访问的抽象方法**
4. **ConcreateElement: 具体的实现方法,实现抽象方法和自身的属性和方法,通过accept方法完成回调Visitor.VisitConreateElement(this); 来调用访问者的方法处理自身的数据结构**
5. **ObjectStruture: 是一个element的容器，一般情况下不会实现，只用在使用的时候通过一个Set,Map,List就可以完成相关操作**

**模板**

**Public interface Visitor{ //抽象的访问者**

**Void visit(Element element);**

**Void vidit(ConreateElement element);**

**}**

**Public class ConreateVisitor implments Visitor{ //具体的访问者**

**Public void visit(Element element){**

**Element.doSomething();**

**}**

**}**

**Public abstract class Element{ //抽象的数据类型**

**Private int age;**

**Private String name;**

**Public abstract void doSomething(Visitor v);**

**Get/set…**

**}**

**Public class CommonElement extends Element{ //具体的数据类型**

**Private String commonProperty;**

**Public void doSomething(Visitor v){**

**v.visit(this);**

**}**

**}**

**Client{**

**Main{**

**Element e = new CommonElement();**

**Visitor v = new ConreateVisitor();**

**e.doSomething(v);**

**}**

**}**

**访问者模式的优点：**

1. **符合单一职责原则： ConreateElement下面的类只负责加载数据，visitor负责展现，两个不同的职责明确的分开了**
2. **优秀的扩展性： 由于职责是分开的继续增加数据的操作是非常快捷的**
3. **灵活性非常高： 在数据类型不同计算方式不同的情况下，只用给访问者要访问的类型数据做对应处理就能实现**

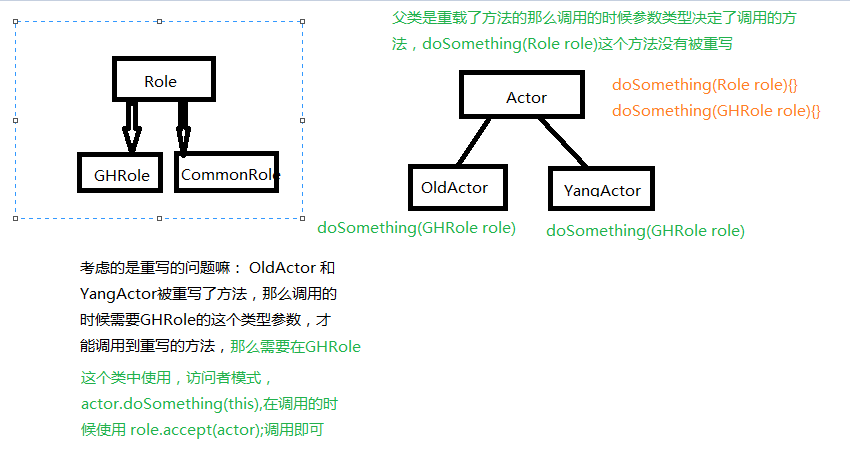
**访问者模式的缺点：**

1. **具体的Element需要对访问者公布细节**
2. **具体element变更困难**
3. **违背了依赖倒置原则，抛弃了对接口的实现，直接使用实现类来创建对象**

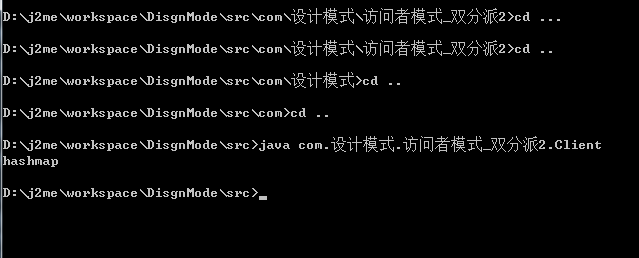
**访问者模式的实现场景：**

1. **当一个对象的结构含有不同的接口，而你想对这些对象的具体类实现依赖操作**
2. **需要对一个 对象结构中的类进行不同的操作，并且不相关，你想避免这些操作污染了这些类**

**访问模式的双分派模式**

****

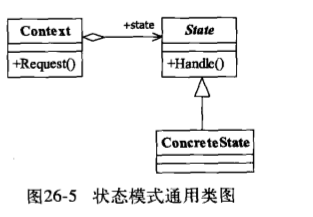
**关于java命令行运行的问题**

****

## 状态模式

**状态模式的定义： 当一个类的状态发生改变时，其行为也会有所改变，这个对象看起来像变成另外一个类的样子**

**状态模式的核心是封装**

**状态模式的三个角色：**

1. **state抽象的状态类，抽象出每一个状态共有的动作，并封装了环境角色以实现状态的切换和交换切换后的行为**
2. **ConcreteState: 具体的环境类： 实现了一个具体状态所对应的动作，并通过环境角色切换状态（本状态如何过渡到其他状态）**
3. **Context 环境类： 根据不同的环境切换状态，封装了所有的状态对象，并在切换状态的时候将Context对象传递过去，让下一个状态的动作属性有对应的环境对象**

**模板：**

**Public abstract class Status{ //抽象状态类**

**Protected Context context; //需要一个环境类对象，才能够实时改变状态，也可以使用其他办法，但是一定得使环境类的状态变化才行**

**Public void setContext(Context context){**

**This.context = context;**

**}**

**Public Context getContext(){**

**Return this.context;**

**}**

**Public abstract void handle1();**

**Public abstract void handle2();**

**}**

**【 \*\*\* //环境角色有两个不成文的约束： \*\*\* 】**

**1 ） 将所有的状态申明为该类的常量**

**2） 环境对象具有抽象状态类的抽象方法，具体的实现使用了委托处理**

**Public class Context{**

**Public static final Status status1 = new Status1();**

**Public static final Status status2 = new Status2();**

**Private Status status;**

**Public Context(){**

**}**

**Public setStatus(Status status){**

**This.status = status;**

**This.status.setContext(this); //给状态类该环境对象，使切换后的状态能够继续改变环境状态，并不至于空指针**

**}**

**Public void handle1(){  
 this.status.handle1();**

**}**

**Public void handle2(){**

**This.status.hanle2();**

**}**

**}**

**Public class Status1 extends Status{**

**Public void handle1(){**

**Syso(“handle1()”);**

**}**

**Public void handle2(){**

**Syso(“no something”);**

**This.context.setStatus(Context.status2); //切换状态**

**This.context.handle2();**

**}**

**}**

**// 具体类有两个职责，一个是处理完该状态必须完成的任务，二是决定是否过渡到其他状态**

**Public class Status2 extends Status{**

**Public void handle1(){  
 syso(“no something”);**

**This.context.setStatus(Context.status1);**

**This.conetxt.handle1();**

**}**

**Public void handle2(){**

**Syso(“do something”);**

**}**

**}**

**Main{**

**Context context = new Context();**

**Context.setStatus(Context.status1);**

**Context.handle1();**

**Context.handle2(); //切换状态 status2**

**Context.handle1(); //切换状态 status1**

**Context.handle2(); //**

//隐藏了状态的切换，只看到了行为的不同

**}**

**状态结构的优点：**

1. **结构清晰： 避免使用过多的 switch ….. case 和if..else if语句避免程序的复杂性，提高了程序的可维护性**
2. **遵循设计原则： 很好的体现了开闭原则和单一职责原则，每一个状态都是一个子类，需要增加状态的时候就增加子类就行了，需要增加操作方法，在单个状态中增加就可以了**
3. **封装的非常好： 状态的切换是在内部完成的，外部对状态的切换的实现是不知情的**

**状态模式的缺点：**

**子类过多，类过于膨胀，在状态过多的时候，状态分类就会很多，不好管理，**

**状态模式应用的场景：**

1. **对象随着状态改变，行为也发生改变的时候： 如权限管理，权限不同执行的动作也会不一样**
2. **条件分支语句的替换者**

**问题 ： 建造者模式 + 状态模式**

## 享元模式

**享元技术是池技术的重要实现，享元技术使得共享对象可以有效的支持大量的细粒度的对象**

**要求细粒度的对象，可将对象的信息分为内部状态和外部状态**

1. **内部状态： 内部状态时对象可以共享出来的信息，存储在享元对象的内部并不会随着环境改变而改变，如 name age ，不必直接存储在具体的某个对象当中，是可以共享的**
2. **外部状态：外部状态时是对象**得以依赖的一个标记**，是随着环境改变而改变的，是不可以共享的状态，它是一批对象的统一标识，是惟一的一个索引**

#### 桥梁模式（桥接模式）

**桥梁模式的定义：将抽象和实现解耦**

**模板：**

**Public abstract class Product{**

**Private Property property;**

**Public abstract Object doSomething();**

**Public abstract Object doAnything();**

**}**

**Public abstract class Crop{**

**Private Product product;**

**Public Crop(Product product){**

**This.product = product;**

**}**

**Public void make(){**

**This.product.doSomething();**

**This.product.doAnything();**

**}**

**}**

**Public class HouseProduct extends Product{**

**Public int doSomething(){**

**Syos();**

**}**

**Public int doAnything(){**

**Syos();**

**}**

**}**

**Public class WanDa extends Crop{**

**Public WanDa(Product product){ super(product)}**

**}**

**Main{**

**Product p = new HouseProduct();**

**Crop c = new WanDa(p);**

**c.make;**

**}**

**桥梁模式使用了类间的聚合，继承，重写等常用功能，但是它却提供了稳定清晰的架构**

**桥梁模式的优点：  
1） 抽象和实现的分离： 这是桥梁模式最大的优点，解决了继承的缺点，在该模式下实现可以不受抽象制约，不再绑定到一个固定的抽象层次上**

**2 ) 优秀的扩展能力**

**3） 实现细节对客户透明，客户不关心实现的细节，它已经由抽象层通过聚合关系完成了封装**

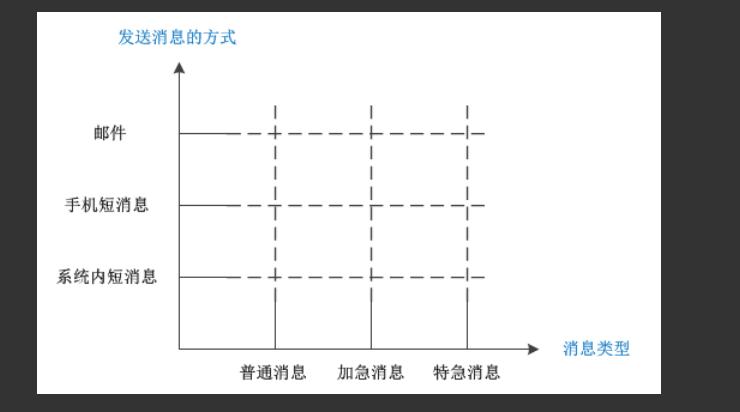
**桥梁模式的使用场景：  
1） 不希望或者不适合用继承的场景，在通过继承过渡无法细化设计粒度的场景，可以使用桥梁模式**

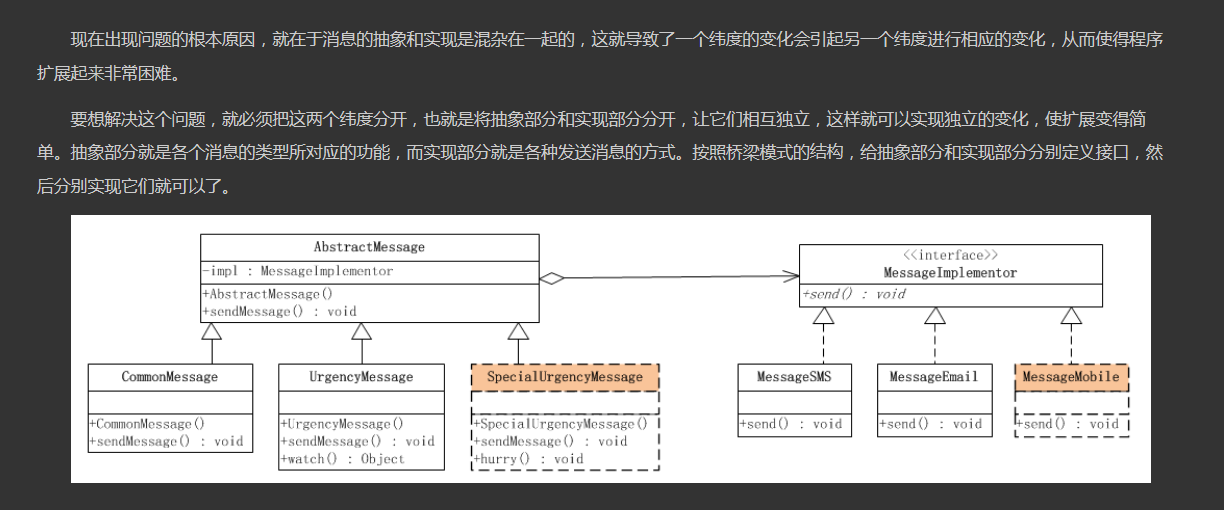
**2） 接口或者抽象不稳定的场景：**

**3） 重要性要求比较高的场景： 设计的粒度越细被重用的可能性就会越大，而采用继承则受父类的限制，不可能出现太细的颗粒度**

**Note : 桥梁模式是非常简单的，在使用的时候主要考虑如何拆分抽象和实现，并不是一旦涉及到了继承就考虑使用该模式，桥梁模式的意图是封装变化，尽量把可能变化的因素封装到最细，最小的逻辑单元，避免扩散风险**

根据业务的功能要求，业务的变化具有两个维度，一个维度是抽象的消息，包括普通消息、加急消息和特急消息，这几个抽象的消息本身就具有一定的关系，加急消息和特急消息会扩展普通消息；另一个维度是在具体的消息发送方式上，包括系统内短消息、邮件和手机短消息，这几个方式是平等的，可被切换的方式。

****

****

# 模式对比

### 结构模式的对比

## 装饰者模式 VS代理模式

**1.1 ） 代理模式和装饰者模式的对比： 代理模式： 是把当前对象的行为交给一个代理对象来管理执行，代理类负责接口限定是否可以调用真实角色，以及是否对发送到真实角色的对象进行变形，不会对被代理对象的方法做任何调整 ,代理的极致开发就是AOP,在使用spring开发的时候是必然用到的,,它就是使用代理和反射的技术**

**1.2） 装饰者模式在保证接口不变的情况下加强类的功能,它保证的是被修饰对象的功能比原来的丰富(当然也可以更弱),但是不会做准入判断和准入参数过滤,如是否执行类的功能,是否对参数进行过滤这不是装饰者模式关心的装饰者模式使用相对较少,在java.io.\*这个包下面有很多都是使用了装饰者模式：**

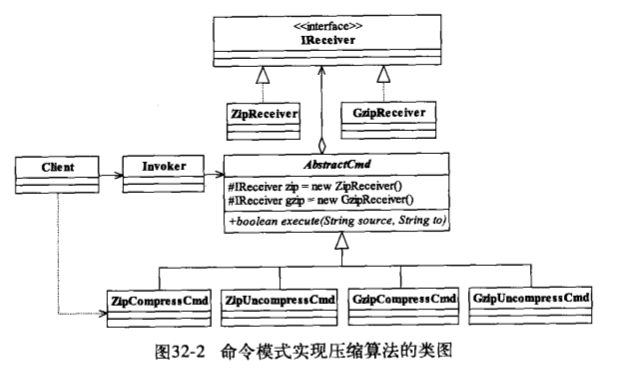
**OutputStream os = new DataOutputStream(new FileOutputStream(“java.text”)); 这就是典型的装饰者模式**

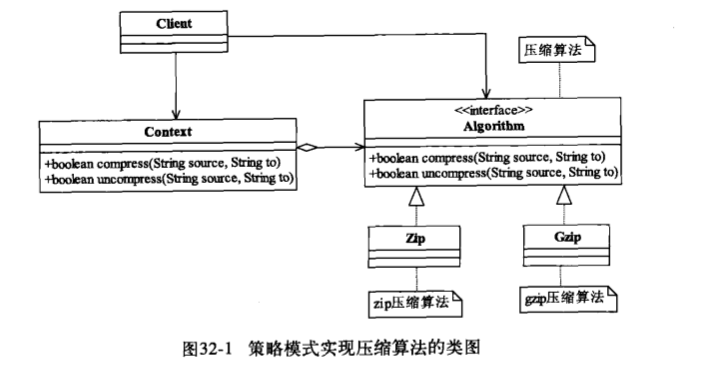
## 装饰者模式VS适配器模式

装饰者模式和适配器的类图没有多少相似点，但是他们都是有包装的作用，都是通过委托方式实现其功能，不同点是装饰者模式包装的是自己的兄弟类，同一个家族的，而适配器模式是修饰非血缘关系的类，将非血缘关系的类伪装成本家族的对象，它的本质还是 非相同接口的对象

1. 意图不同，装饰者模式是加强对象的功能，类中把一只丑小鸭强化成了一个美丽自信的鸭子，它不改变类的行为和属性，只是增加功能，使得美丽更美丽，强壮更强壮，安全更安全，适配器模式关注的是转化，它的主要意图是两个不同对象的转化，可以天鹅当鸭子看，也可以把鸭子当天鹅看
2. 施与对象不同 ： 装饰者模式装饰的是同族，只有在相同的属性和行为下才能对比出是增强了还是减弱了，适配器模式是必须两个不同的对象才能完成，不能同宗，否则就是拥有同样的行为属性不需要适配了’
3. 场景不同，装饰者模式可以在任何阶段使用，适配器模式是一种补救的模式，一般出现在构建完成的项目中，作为紧急处理的一种手段
4. 扩展不同： 装饰者模式易于扩展和修改，适配器模式是在两个对象间架起了桥梁，建立容易去掉就比较困难，需要从系统的整体去考虑

## 策略模式 VS 命令模式

在命令模式没有接受者这个类图的话那么策略模式的类图和命令模式的类图就非常相似但是他们还是有很大的不同：  




1. 关注点不同 ， 策略模式关注的是算法的替换问题，一个新的算法投产旧的算法就会退休，或者提供多种方法调用者自己选择，算法的自由更替是他实现的要点。策略模式关注的是算法的完整性，封装性，只有保证这两个条件才能自由切换

命令模式关注的是解耦，如何让请求者和执行者解耦是它首先要做的事情，解耦的要求是将一个个的请求封装成一个个的命令，由接受者执行，由于封装成了命令，就可以对命令执行多种处理，如 撤销，记录

1. 角色功能不同， 政策模式的抽象算法和具体算法 和 命令模式的抽象接受者和具体接受者很相似 ，但是他们的职责是不一样的，策略模式中的具体算法负责一个完整的算法逻辑，他是不再拆分的原子单元，一旦变更就是算法整体的变更

而命令模式关注的是命令的实现，也就是功能的实现 例如： 我们在分支中对接受者对象进行变更，对请求者没有任何影响，它只是对命令族做了变更，从这方面来说接受者对命令族负责和请求者没有关系，命令模式中的接受者只需要符合六大设计原则，完全不用关心它是否完成一个具体的逻辑，它的影响范围仅仅是抽象命令和具体命令对它的修改不会扩散到模块外的模块

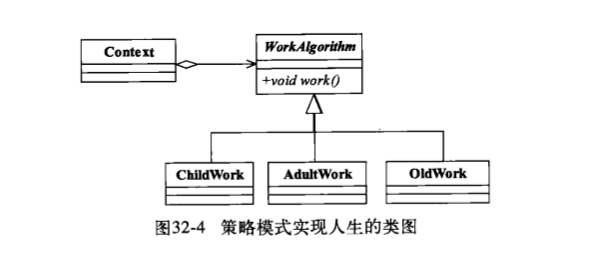
如果在命令模式中需要设计接受者那么需要考虑接受者的变化和封装

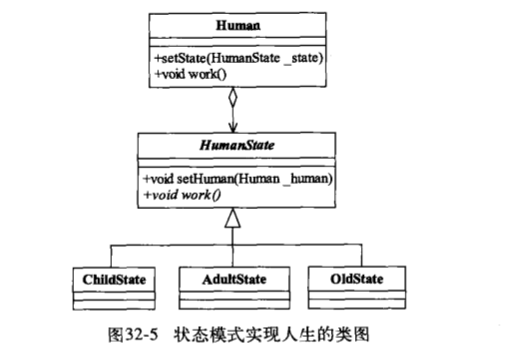
1. 使用场景不同，命令模式使用的场景是解耦两个紧耦合的对象或者有多命令多撤销的场景，策略模式适用于算法要求变换的场景

## 策略模式 VS 状态模式

策略模式和状态模式的分析角度是大相庭径的，策略模式是通过分析每一个年龄段的工作方式不同而得出三个不同的 算法逻辑，

状态模式是从人的生长规律来分析的，随着状态的改变行为也随之改变对不同的业务需求，有很多种实现方式，问题是关注的重点是什么，是人的生长规律还是工作逻辑找准焦点在哪 在选择对应的设计模式





区分：

1. 环境角色的职责不同 ：两者都有一个环境类，策略模式的环境类是做一个算法的切换，起一个委托的作用

状态模式的环境类 不仅有委托的功能，还有登记状态变化的功能，和具体的状态类共同完成状态的切换和行为的随之改变

1. 解决的问题的重点不同： 策略模式主要关注的是算法的改变对外界的变化影响降低到最低的程度，它保证算法可以自由切换

状态模式 解决的是内部状态发生变化后行为随之的变化，它的出发点是事物的状态，封装状态而暴露行为，一个对象的内部状态改变了，从外部看就好像是行为改变了

1. 应用场景不同： 策略模式是一系列相互平行，可互相替换的算法封装后的结果： 这就要求了算法必须是平行的不然就封装了一堆垃圾，替换后会产生“坏味道”

状态模式 需要一个有状态有行为描述才能使用状态模式

1. 结构复杂度不一样： 策略模式的复杂度比较简单易于扩张 ，状态模式相对复杂点，状态是内部改变的并且随之改变了外部的行为

#### 责任链模式和观察者模式（触发链模式）

1. 链中的消息对象不同： 从首节点到尾节点两个链中传递的消息对象是不同的责任链模式下基本上不会改变消息的结构，虽然每一个节点都会产生消费，但是结构不变，但是触发链模式下是允许结构改变的，链中传递的消息对象是可以改变的只要上下级对传递的对象了解即可，它只要求两个相邻的节点之间的消息对象是固定的
2. 上下级节点关系不同： 在责任链模式中，上下节点没有关系，都是从链首传来的，上级节点是什么没有关系，只会按照自己的逻辑来处理传递对象

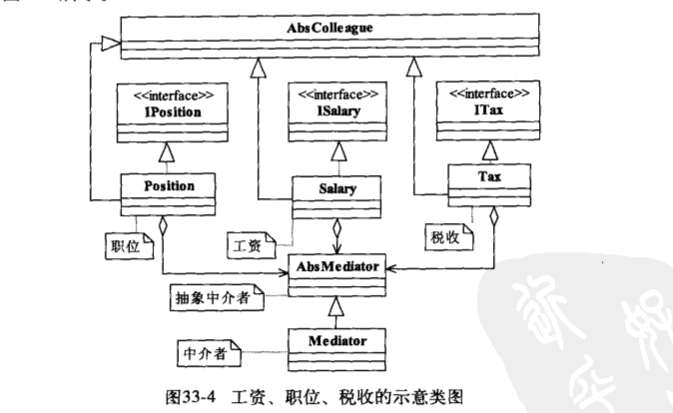
触发链模式下，上下级节点关系亲密，下级节点对上级节点膜拜，上级节点对下级节点绝对的信任，链中任意相邻节点都是一个牢固的独立团体

1. 消息分销的渠道不同： 责任链模式是通过链首传递一个消息对象，消息对象沿着链条向链尾移动，以固定的方向，

触发链模式 是观察者模式实现的当一个消息传递进来的时候，可以通过多种方式传递，可以以广播方式传递，可以以跳跃的方式传递，这取决于消息的逻辑

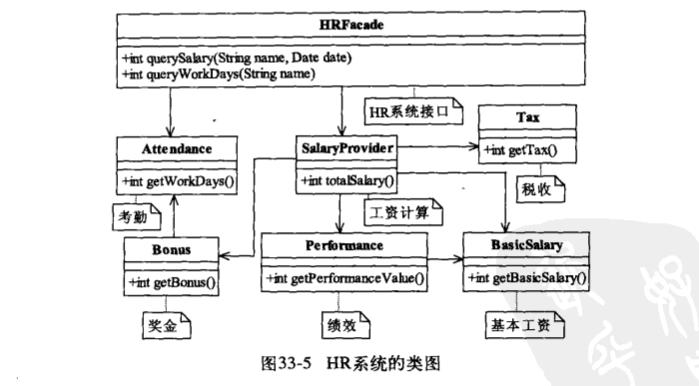
## 中介者模式VS门面模式

1. **中介者模式:**

****

**通过中介者模式是降低其他对象之间的耦合,而不是中介者和对象的耦合,因为每个对象都只知道中介者,不知道其他对象的存在,中介者完成了对象之间的逻辑部分解耦了其他对象间的耦合**

**上图 采用中介者模式后,三个对象之间已经相互独立了,全部委托中介者完成,如果不使用中介者模式那么其他对象之间就会对另外两个对象产生关联,这必定会增加系统的复杂度,中介者封装了同事之间的逻辑关系,方便系统的维护和扩展**

****

**使用门面模式避免了子系统的复杂逻辑的外泄,确保子系统业务逻辑的单纯性,即使业务流程变更,影响的也是子系统内部的功能**

**主要区别:**

**门面模式主要是封装隔绝 中介者模式是调节同事之间的关系**

**门面模式,只是增加了一个门面,它对子系统没有增加任何功能,子系统脱离了门面系统完全可以独立运行**

**中介者模式,增加了业务功能,它把各个同事之间原有的耦合转移到了中介者,同事不可能脱离中介者独立运行,除非增加系统的复杂性降低可扩展性**

**知晓状态不同:**

1. **门面模式 : 子系统并不知道有门面的存在,对中介者来说每一个同事类都知道有一个中介者的存在,因为需要依靠中介者和同事进行交流**
2. **门面模式是一种简单的封装,所有的业务都是交给子系统完成的,中介者模式,需要有一个中心,由中心协调同事类完成,并且中心本身也具有一定的业务逻辑完成部分业务**

## 包装模式群

包装模式包括: 门面模式, 代理模式, 适配器模式, 桥梁模式 , 装饰者模式

1. 代理模式 : 不希望展示一个对象的实现细节的时候,还可以用在限制一个对象使用的场景中
2. 装饰者模式: 是一种特殊的代理模式,不在改变接口的前提下实现功能的增强或者减弱,就扩展性而言他比子类更灵活,在已经发布项目后,可以轻松的通过装饰者模式 完成功能的扩展,增强
3. 适配器模式: 主要是接口的转换,把一个对象的接口转换成系统希望的接口,这里的系统不一定是一个应用,可能是某个环境, 比如通过接口转换可以屏蔽外界接口,一面外界接口深入系统内部,从而提高系统的稳定性和可靠性
4. 桥梁模式: 是在抽象层产生耦合,解决自行扩展的问题,它可以使两个有耦合关系的对象可以独立的扩展而互不影响
5. 门面模式: 是对一个粗粒度的封装,它提供一个方便访问子系统的接口,不具有任何的业务逻辑,仅仅是一个访问复杂系统的快速通道,没有他子系统照样能够运行,有了它方便访问

混编模式暂时放弃