# 设计模式遵循的原则

## 开闭原则（Open Close Principle）

定义：一个软件实体如类、模块和函数应该对扩展开放，对修改关闭。

问题由来：在软件的生命周期内，因为变化、升级和维护等原因需要对软件原有代码进行修改时，可能会给旧代码中引入错误，也可能会使我们不得不对整个功能进行重构，并且需要原有代码经过重新测试。

解决方案：当软件需要变化时，尽量通过扩展软件实体的行为来实现变化，而不是通过修改已有的代码来实现变化。

开闭原则是面向对象设计中最基础的设计原则，它指导我们如何建立稳定灵活的系统。开闭原则可能是设计模式六项原则中定义最模糊的一个了，它只告诉我们对扩展开放，对修改关闭，可是到底如何才能做到对扩展开放，对修改关闭，并没有明确的告诉我们。

## 里氏替换原则（Liskov Substitution Principle）

定义1：如果对每一个类型为 T1的对象 o1，都有类型为 T2 的对象o2，使得以 T1定义的所有程序 P 在所有的对象 o1 都代换成 o2 时，程序 P 的行为没有发生变化，那么类型 T2 是类型 T1 的子类型。

定义2：所有引用基类的地方必须能透明地使用其子类的对象。

问题由来：有一功能P1，由类A完成。现需要将功能P1进行扩展，扩展后的功能为P，其中P由原有功能P1与新功能P2组成。新功能P由类A的子类B来完成，则子类B在完成新功能P2的同时，有可能会导致原有功能P1发生故障。

解决方案：当使用继承时，遵循里氏替换原则。类B继承类A时，除添加新的方法完成新增功能P2外，尽量不要重写父类A的方法，也尽量不要重载父类A的方法。

继承包含这样一层含义：父类中凡是已经实现好的方法（相对于抽象方法而言），实际上是在设定一系列的规范和契约，虽然它不强制要求所有的子类必须遵从这些契约，但是如果子类对这些非抽象方法任意修改，就会对整个继承体系造成破坏。而里氏替换原则就是表达了这一层含义。

继承作为面向对象三大特性之一，在给程序设计带来巨大便利的同时，也带来了弊端。比如使用继承会给程序带来侵入性，程序的可移植性降低，增加了对象间的耦合性，如果一个类被其他的类所继承，则当这个类需要修改时，必须考虑到所有的子类，并且父类修改后，所有涉及到子类的功能都有可能会产生故障。

举例说明继承的风险，我们需要完成一个两数相减的功能，由类A来负责。

**class** A{

**public** **int** func1(**int** a, **int** b){

**return** a-b;

}

}

**public** **class** Client{

**public** **static** **void** main(String[] args){

A a = **new** A();

System.***out***.println("100-50="+a.func1(100, 50));

System.***out***.println("100-80="+a.func1(100, 80));

}

}

运行结果：

100-50=50

100-80=20

后来，我们需要增加一个新的功能：完成两数相加，然后再与100求和，由类B来负责。即类B需要完成两个功能：

两数相减。

两数相加，然后再加100。

由于类A已经实现了第一个功能，所以类B继承类A后，只需要再完成第二个功能就可以了，代码如下：

**class** B **extends** A {

**public** **int** func1(**int** a, **int** b) {

**return** a+b;

}

**public** **int** func2(**int** a, **int** b) {

**return** func1(a , b)+100;

}

}

**public** **class** Client {

**public** **static** **void** main (String[] args){

B b = **new** B();

System.***out***.println("100-50="+b.func1(100, 50));

System.***out***.println("100-80="+b.func1(100, 80));

System.***out***.println("100+20+100="+b.func2(100, 20));

}

}

类B完成后，运行结果：

100-50=150

100-80=180

100+20+100=220

我们发现原本运行正常的相减功能发生了错误。原因就是类B在给方法起名时无意中重写了父类的方法，造成所有运行相减功能的代码全部调用了类B重写后的方法，造成原本运行正常的功能出现了错误。在本例中，引用基类A完成的功能，换成子类B之后，发生了异常。在实际编程中，我们常常会通过重写父类的方法来完成新的功能，这样写起来虽然简单，但是整个继承体系的可复用性会比较差，特别是运用多态比较频繁时，程序运行出错的几率非常大。如果非要重写父类的方法，比较通用的做法是：原来的父类和子类都继承一个更通俗的基类，原有的继承关系去掉，采用依赖、聚合，组合等关系代替

里氏替换原则通俗的来讲就是：子类可以扩展父类的功能，但不能改变父类原有的功能。它包含以下4层含义：

1. 子类可以实现父类的抽象方法，但不能覆盖父类的非抽象方法。
2. 子类中可以增加自己特有的方法。
3. 当子类的方法重载父类的方法时，方法的前置条件（即方法的形参）要比父类方法的输入参数更宽松。
4. 当子类的方法实现父类的抽象方法时，方法的后置条件（即方法的返回值）要比父类更严格。

## 依赖倒置原则（Dependence Inversion Principle）

定义：高层模块不应该依赖低层模块，二者都应该依赖其抽象；抽象不应该依赖细节；细节应该依赖抽象。

问题由来：类A直接依赖类B，假如要将类A改为依赖类C，则必须通过修改类A的代码来达成。这种场景下，类A一般是高层模块，负责复杂的业务逻辑；类B和类C是低层模块，负责基本的原子操作；假如修改类A，会给程序带来不必要的风险。

解决方案：将类A修改为依赖接口I，类B和类C各自实现接口I，类A通过接口I间接与类B或者类C发生联系，则会大大降低修改类A的几率。

依赖倒置原则基于这样一个事实：相对于细节的多变性，抽象的东西要稳定的多。以抽象为基础搭建起来的架构比以细节为基础搭建起来的架构要稳定的多。在java中，抽象指的是接口或者抽象类，细节就是具体的实现类，使用接口或者抽象类的目的是制定好规范和契约，而不去涉及任何具体的操作，把展现细节的任务交给他们的实现类去完成。

面向接口编程比相对于面向实现编程好在什么地方。场景是这样的，母亲给孩子讲故事，只要给她一本书，她就可以照着书给孩子讲故事了。代码如下：

**class** Book{

**public** String getContent(){

**return** "很久很久以前有一个阿拉伯的故事……";

}

}

**class** Mother{

**public** **void** narrate(Book book){

System.***out***.println("妈妈开始讲故事");

System.***out***.println(book.getContent());

}

}

**public** **class** Client{

**public** **static** **void** main(String[] args){

Mother mother = **new** Mother();

mother.narrate(**new** Book());

}

}

运行结果：

妈妈开始讲故事

很久很久以前有一个阿拉伯的故事……

运行良好，假如有一天，需求变成这样：不是给书而是给一份报纸，让这位母亲讲一下报纸上的故事，报纸的代码如下：

**class** Newspaper{

**public** String getContent(){

**return** "林书豪38+7领导尼克斯击败湖人……";

}

}

这位母亲却办不到，因为她居然不会读报纸上的故事，这太荒唐了，只是将书换成报纸，居然必须要修改Mother才能读。假如以后需求换成杂志呢？换成网页呢？还要不断地修改Mother，这显然不是好的设计。原因就是Mother与Book之间的耦合性太高了，必须降低他们之间的耦合度才行。

我们引入一个抽象的接口IReader。读物，只要是带字的都属于读物：

**interface** IReader{

**public** String getContent();

}

Mother类与接口IReader发生依赖关系，而Book和Newspaper都属于读物的范畴，他们各自都去实现IReader接口，这样就符合依赖倒置原则了，代码修改为：

**class** Newspaper **implements** IReader {

**public** String getContent(){

**return** "林书豪17+9助尼克斯击败老鹰……";

}

}

**class** Book **implements** IReader{

**public** String getContent(){

**return** "很久很久以前有一个阿拉伯的故事……";

}

}

**class** Mother{

**public** **void** narrate(IReader reader){

System.***out***.println("妈妈开始讲故事");

System.***out***.println(reader.getContent());

}

}

**public** **class** Client{

**public** **static** **void** main(String[] args){

Mother mother = **new** Mother();

mother.narrate(**new** Book());

mother.narrate(**new** Newspaper());

}

}

运行结果：

妈妈开始讲故事

很久很久以前有一个阿拉伯的故事……

妈妈开始讲故事

林书豪17+9助尼克斯击败老鹰……

这样修改后，无论以后怎样扩展Client类，都不需要再修改Mother类了。这只是一个简单的例子，实际情况中，代表高层模块的Mother类将负责完成主要的业务逻辑，一旦需要对它进行修改，引入错误的风险极大。所以遵循依赖倒置原则可以降低类之间的耦合性，提高系统的稳定性，降低修改程序造成的风险。

采用依赖倒置原则给多人并行开发带来了极大的便利，比如上例中，原本Mother类与Book类直接耦合时，Mother类必须等Book类编码完成后才可以进行编码，因为Mother类依赖于Book类。修改后的程序则可以同时开工，互不影响，因为Mother与Book类一点关系也没有。参与协作开发的人越多、项目越庞大，采用依赖导致原则的意义就越重大。现在很流行的TDD开发模式就是依赖倒置原则最成功的应用。

传递依赖关系有三种方式，以上的例子中使用的方法是接口传递，另外还有两种传递方式：构造方法传递和setter方法传递，相信用过Spring框架的，对依赖的传递方式一定不会陌生。

在实际编程中，我们一般需要做到如下3点：

1. 低层模块尽量都要有抽象类或接口，或者两者都有。
2. 变量的声明类型尽量是抽象类或接口。
3. 使用继承时遵循里氏替换原则。

依赖倒置原则的核心就是要我们面向接口编程，理解了面向接口编程，也就理解了依赖倒置。

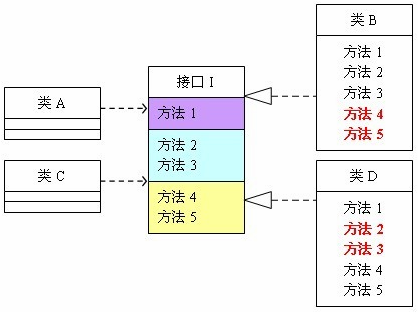
## 接口隔离原则（Interface Segregation Principle）

定义：客户端不应该依赖它不需要的接口；一个类对另一个类的依赖应该建立在最小的接口上。

问题由来：类A通过接口I依赖类B，类C通过接口I依赖类D，如果接口I对于类A和类B来说不是最小接口，则类B和类D必须去实现他们不需要的方法。

解决方案：将臃肿的接口I拆分为独立的几个接口，类A和类C分别与他们需要的接口建立依赖关系。也就是采用接口隔离原则。

举例来说明接口隔离原则：



这个图的意思是：类A依赖接口I中的方法1、方法2、方法3，类B是对类A依赖的实现。类C依赖接口I中的方法1、方法4、方法5，类D是对类C依赖的实现。对于类B和类D来说，虽然他们都存在着用不到的方法（也就是图中红色字体标记的方法），但由于实现了接口I，所以也必须要实现这些用不到的方法。对类图不熟悉的可以参照程序代码来理解，代码如下：

**interface** I {

**public** **void** method1();

**public** **void** method2();

**public** **void** method3();

**public** **void** method4();

**public** **void** method5();

}

**class** A{

**public** **void** depend1(I i){

i.method1();

}

**public** **void** depend2(I i){

i.method2();

}

**public** **void** depend3(I i){

i.method3();

}

}

**class** B **implements** I{

**public** **void** method1() {

System.***out***.println("类B实现接口I的方法1");

}

**public** **void** method2() {

System.***out***.println("类B实现接口I的方法2");

}

**public** **void** method3() {

System.***out***.println("类B实现接口I的方法3");

}

//对于类B来说，method4和method5不是必需的，但是由于接口A中有这两个方法，

//所以在实现过程中即使这两个方法的方法体为空，也要将这两个没有作用的方法进行实现。

**public** **void** method4() {}

**public** **void** method5() {}

}

**class** C{

**public** **void** depend1(I i){

i.method1();

}

**public** **void** depend2(I i){

i.method4();

}

**public** **void** depend3(I i){

i.method5();

}

}

**class** D **implements** I{

**public** **void** method1() {

System.***out***.println("类D实现接口I的方法1");

}

//对于类D来说，method2和method3不是必需的，但是由于接口A中有这两个方法，

//所以在实现过程中即使这两个方法的方法体为空，也要将这两个没有作用的方法进行实现。

**public** **void** method2() {}

**public** **void** method3() {}

**public** **void** method4() {

System.***out***.println("类D实现接口I的方法4");

}

**public** **void** method5() {

System.***out***.println("类D实现接口I的方法5");

}

}

**public** **class** Client{

**public** **static** **void** main(String[] args){

A a = **new** A();

a.depend1(**new** B());

a.depend2(**new** B());

a.depend3(**new** B());

C c = **new** C();

c.depend1(**new** D());

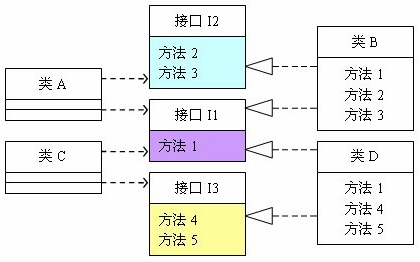
c.depend2(**new** D());

c.depend3(**new** D());

}

}

可以看到，如果接口过于臃肿，只要接口中出现的方法，不管对依赖于它的类有没有用处，实现类中都必须去实现这些方法，这显然不是好的设计。如果将这个设计修改为符合接口隔离原则，就必须对接口I进行拆分。在这里我们将原有的接口I拆分为三个接口，拆分后的设计如图2所示：



**interface** I1 {

**public** **void** method1();

}

**interface** I2 {

**public** **void** method2();

**public** **void** method3();

}

**interface** I3 {

**public** **void** method4();

**public** **void** method5();

}

**class** A{

**public** **void** depend1(I1 i){

i.method1();

}

**public** **void** depend2(I2 i){

i.method2();

}

**public** **void** depend3(I2 i){

i.method3();

}

}

**class** B **implements** I1, I2{

**public** **void** method1() {

System.***out***.println("类B实现接口I1的方法1");

}

**public** **void** method2() {

System.***out***.println("类B实现接口I2的方法2");

}

**public** **void** method3() {

System.***out***.println("类B实现接口I2的方法3");

}

}

**class** C{

**public** **void** depend1(I1 i){

i.method1();

}

**public** **void** depend2(I3 i){

i.method4();

}

**public** **void** depend3(I3 i){

i.method5();

}

}

**class** D **implements** I1, I3{

**public** **void** method1() {

System.***out***.println("类D实现接口I1的方法1");

}

**public** **void** method4() {

System.***out***.println("类D实现接口I3的方法4");

}

**public** **void** method5() {

System.***out***.println("类D实现接口I3的方法5");

}

}

接口隔离原则的含义是：建立单一接口，不要建立庞大臃肿的接口，尽量细化接口，接口中的方法尽量少。也就是说，我们要为各个类建立专用的接口，而不要试图去建立一个很庞大的接口供所有依赖它的类去调用。本文例子中，将一个庞大的接口变更为3个专用的接口所采用的就是接口隔离原则。在程序设计中，依赖几个专用的接口要比依赖一个综合的接口更灵活。接口是设计时对外部设定的“契约”，通过分散定义多个接口，可以预防外来变更的扩散，提高系统的灵活性和可维护性。

说到这里，很多人会觉的接口隔离原则跟之前的单一职责原则很相似，其实不然。其一，单一职责原则原注重的是职责；而接口隔离原则注重对接口依赖的隔离。其二，单一职责原则主要是约束类，其次才是接口和方法，它针对的是程序中的实现和细节；而接口隔离原则主要约束接口接口，主要针对抽象，针对程序整体框架的构建。

采用接口隔离原则对接口进行约束时，要注意以下几点：

1. 接口尽量小，但是要有限度。对接口进行细化可以提高程序设计灵活性是不挣的事实，但是如果过小，则会造成接口数量过多，使设计复杂化。所以一定要适度。
2. 为依赖接口的类定制服务，只暴露给调用的类它需要的方法，它不需要的方法则隐藏起来。只有专注地为一个模块提供定制服务，才能建立最小的依赖关系。
3. 提高内聚，减少对外交互。使接口用最少的方法去完成最多的事情。

运用接口隔离原则，一定要适度，接口设计的过大或过小都不好。设计接口的时候，只有多花些时间去思考和筹划，才能准确地实践这一原则。

## 迪米特法则（最少知道原则）（Demeter Principle）

定义：一个对象应该对其他对象保持最少的了解。

问题由来：类与类之间的关系越密切，耦合度越大，当一个类发生改变时，对另一个类的影响也越大。

解决方案：尽量降低类与类之间的耦合。

自从我们接触编程开始，就知道了软件编程的总的原则：低耦合，高内聚。无论是面向过程编程还是面向对象编程，只有使各个模块之间的耦合尽量的低，才能提高代码的复用率。低耦合的优点不言而喻，但是怎么样编程才能做到低耦合呢？那正是迪米特法则要去完成的。

迪米特法则又叫最少知道原则，最早是在1987年由美国Northeastern University的Ian Holland提出。通俗的来讲，就是一个类对自己依赖的类知道的越少越好。也就是说，对于被依赖的类来说，无论逻辑多么复杂，都尽量地的将逻辑封装在类的内部，对外除了提供的public方法，不对外泄漏任何信息。迪米特法则还有一个更简单的定义：只与直接的朋友通信。首先来解释一下什么是直接的朋友：每个对象都会与其他对象有耦合关系，只要两个对象之间有耦合关系，我们就说这两个对象之间是朋友关系。耦合的方式很多，依赖、关联、组合、聚合等。其中，我们称出现成员变量、方法参数、方法返回值中的类为直接的朋友，而出现在局部变量中的类则不是直接的朋友。也就是说，陌生的类最好不要作为局部变量的形式出现在类的内部。

举一个例子：有一个集团公司，下属单位有分公司和直属部门，现在要求打印出所有下属单位的员工ID。先来看一下违反迪米特法则的设计。

//总公司员工

**class** Employee{

**private** String id;

**public** **void** setId(String id){

**this**.id = id;

}

**public** String getId(){

**return** id;

}

}

//分公司员工

**class** SubEmployee{

**private** String id;

**public** **void** setId(String id){

**this**.id = id;

}

**public** String getId(){

**return** id;

}

}

**class** SubCompanyManager{

**public** List<SubEmployee> getAllEmployee(){

List<SubEmployee> list = **new** ArrayList<SubEmployee>();

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

SubEmployee emp = **new** SubEmployee();

//为分公司人员按顺序分配一个ID

emp.setId("分公司"+i);

list.add(emp);

}

**return** list;

}

}

**class** CompanyManager{

**public** List<Employee> getAllEmployee(){

List<Employee> list = **new** ArrayList<Employee>();

**for**(**int** i=0; i<30; i++){

Employee emp = **new** Employee();

//为总公司人员按顺序分配一个ID

emp.setId("总公司"+i);

list.add(emp);

}

**return** list;

}

**public** **void** printAllEmployee(SubCompanyManager sub){

List<SubEmployee> list1 = sub.getAllEmployee();

**for**(SubEmployee e:list1){

System.***out***.println(e.getId());

}

List<Employee> list2 = **this**.getAllEmployee();

**for**(Employee e:list2){

System.***out***.println(e.getId());

}

}

}

**public** **class** Client{

**public** **static** **void** main(String[] args){

CompanyManager e = **new** CompanyManager();

e.printAllEmployee(**new** SubCompanyManager());

}

}

现在这个设计的主要问题出在CompanyManager中，根据迪米特法则，只与直接的朋友发生通信，而SubEmployee类并不是CompanyManager类的直接朋友（以局部变量出现的耦合不属于直接朋友），从逻辑上讲总公司只与他的分公司耦合就行了，与分公司的员工并没有任何联系，这样设计显然是增加了不必要的耦合。按照迪米特法则，应该避免类中出现这样非直接朋友关系的耦合。修改后的代码如下:

**class** SubCompanyManager{

**public** List<SubEmployee> getAllEmployee(){

List<SubEmployee> list = **new** ArrayList<SubEmployee>();

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

SubEmployee emp = **new** SubEmployee();

//为分公司人员按顺序分配一个ID

emp.setId("分公司"+i);

list.add(emp);

}

**return** list;

}

**public** **void** printEmployee(){

List<SubEmployee> list = **this**.getAllEmployee();

**for**(SubEmployee e:list){

System.***out***.println(e.getId());

}

}

}

**class** CompanyManager{

**public** List<Employee> getAllEmployee(){

List<Employee> list = **new** ArrayList<Employee>();

**for**(**int** i=0; i<30; i++){

Employee emp = **new** Employee();

//为总公司人员按顺序分配一个ID

emp.setId("总公司"+i);

list.add(emp);

}

**return** list;

}

**public** **void** printAllEmployee(SubCompanyManager sub){

sub.printEmployee();

List<Employee> list2 = **this**.getAllEmployee();

**for**(Employee e:list2){

System.***out***.println(e.getId());

}

}

}

修改后，为分公司增加了打印人员ID的方法，总公司直接调用来打印，从而避免了与分公司的员工发生耦合。

迪米特法则的初衷是降低类之间的耦合，由于每个类都减少了不必要的依赖，因此的确可以降低耦合关系。但是凡事都有度，虽然可以避免与非直接的类通信，但是要通信，必然会通过一个“中介”来发生联系，例如本例中，总公司就是通过分公司这个“中介”来与分公司的员工发生联系的。过分的使用迪米特原则，会产生大量这样的中介和传递类，导致系统复杂度变大。所以在采用迪米特法则时要反复权衡，既做到结构清晰，又要高内聚低耦合。

## 合成/聚合复用原则（Composite Reuse Principle）

定义：原则是尽量使用合成/聚合的方式，而不是使用继承。继承实际上破坏了类的封装性，超类的方法可能会被子类修改。

问题由来：继承复用破坏类的封装特性，它把父类的实现细节直接暴露给了子类，这违背了信息隐藏的原则；如果父类发生了改变，那么子类也要发生相应的改变，这就直接导致了类与类之间的高耦合，不利于类的扩展、复用、维护等，也带来了系统僵硬和脆弱的设计。而用合成和聚合的时候新对象和已有对象的交互往往是通过接口或者抽象类进行的，就可以很好的避免上面的不足，而且这也可以让每一个新的类专注于实现自己的任务，符合单一职责原则。

1. **合成和聚合的区别；依赖和关联**

合成(Composition)和聚合(Aggregation)都是关联(Association)的特殊种类。用C语言来讲，合成是值的聚合(Aggregation by Value)，聚合是则是引用的聚合(Aggregation by Reference)。

**聚合用来表示“拥有”关系或者整体与部分的关系。**代表部分的对象有可能会被多个代表整体的对象所共享，而且不一定会随着某个代表整体的对象被销毁或破坏而被销毁或破坏，部分的生命周期可以超越整体。例如，班级和学生，当班级删除后，学生还能存在，学生可以被培训机构引用。

聚合关系UML类图



**class** Student {

}

**class** Classes {

**private** Student student;

**public** Classes(Student student){

**this**.student = student;

}

}

**合成用来表示一种强得多的“拥有”关系。**在一个合成关系里，部分和整体的生命周期是一样的。一个合成的新对象完全拥有对其组成部分的支配权，包括它们的创建和湮灭等。使用程序语言的术语来说，合成而成的新对象对组成部分的内存分配、内存释放有绝对的责任。

一个合成关系中的成分对象是不能与另一个合成关系共享的。一个成分对象在同一个时间内只能属于一个合成关系。如果一个合成关系湮灭了，那么所有的成分对象要么自己湮灭所有的成分对象（这种情况较为普遍）要么就得将这一责任交给别人（较为罕见）。

例如，一个人由头、四肢和各种器官组成，人与这些具有相同的生命周期，人死了，这些器官也就挂了。房子和房间的关系，当房子没了，房间也不可能独立存在。

合成关系UML类图



**class** Room {

**public** Room createRoom() {

System.out.println(“创建房间”);

returnnew Room();

}

}

**class** House {

**private** Room room;

**public** House() {

room=**new** Room();

}

**public** **void** createHouse() {

room.createRoom();

}

}

1. **依赖和关联**

依赖(Dependency)

依赖是类与类之间的连接，表示一个类依赖于另外一个类的定义。依赖关系仅仅描述了类与类之间的一种使用与被使用的关系，在Java中体现为局部变量、方法的参数或者是对静态方法的调用。

依赖关系UML类图



**static** **class** Boat {

**public** **static** **void** row() {

System.out.println("开动");

}

}

**class** Person {

**public** **void** crossRiver(Boatboat) {

boat.row();

}

**public** **void** fishing() {

Boat boat =**new** Boat() ;

boat.row();

}

**public** **void** patrol() {

Boat.row();

}

}

1. **关联(Association)**

关联是类与类之间的连结。关联关系使一个类知道另外一个类的属性和方法。关联可以是双向的，也可以是单向的。体现在Java中，关联关系是通过成员变量来实现的。

一般关联关系UML类图



**class** Computer{

**public** **void** develop(){

System.out.println("Develop ");

}

}

**class** Person {

**private** Computer computer ;

**public** Person(Computer computer){

**this**.computer = computer ;

}

**public** **void** work() {

computer.develop() ;

System.out.println("work");

}

}

1. **为什么使用合成/聚合复用，而不使用继承复用**

在面向对象的设计里，有两种基本的方法可以在不同的环境中复用已有的设计和实现，即通过组合或通过继承。

**组合**

由于组合可以将已有的对象纳入到新对象中，使之成为新对象的一部分，因此新对象可以调用已有对象的功能，这样做有下面的好处：

1. 新对象存取成分对象的唯一方法是通过成分对象的接口。
2. 这种复用是黑箱复用，因为成分对象的内部细节是新对象所看不见的。
3. 这种复用支持包装。
4. 这种复用所需要的依赖较少。
5. 每一个新的类可以将焦点集中到一个任务上。
6. 这种复用可以在运行时间动态进行，新对象可以动态的引用与成分对象类型相同的对象。

组合复用的缺点就是用组合复用建造的系统会有较多的对象需要管理。

**继承**

组合几乎可以用到任何环境中去，但是继承只能用到一些环境中。

继承复用通过扩展一个已有对象的实现来得到新的功能，基类明显的捕获共同的属性和方法，而子类通过增加新的属性和方法来扩展超类的实现。

继承的优点：

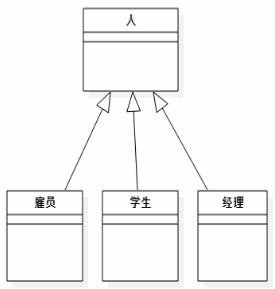
1. 新的实现比较容易，因为基类的大部分功能都可以通过继承自动的进入子类。
2. 修改或扩展继承而来的实现较为容易。

继承的缺点：

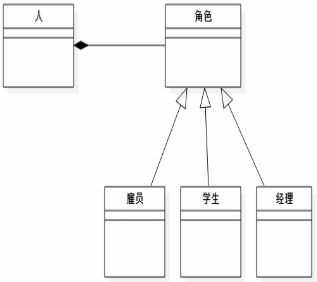
1. 继承复用破坏了包装，因为继承超类的的实现细节暴露给子类。由于超类的内部细节常常对子类是透明的，因此这种复用是透明的复用，又称“白箱”复用。
2. 如果超类的实现发生改变，那么子类的实现也不得不发生改变。因此，当一个基类发生改变时，这种改变就会像水中投入石子引起的水波一样，将变化一圈又一圈的传导到一级又一级的子类，使设计师不得不相应地改变这些子类，以适应超类的变化。
3. 从超类继承而来的实现是静态的，不可能在运行时间内发生改变，因此没有足够的灵活性。

**使用组合还是继承**

按照组合复用原则我们应该首选组合，然后才是继承，使用继承时应该严格的遵守里氏替换原则，必须满足“Is-A”的关系是才能使用继承，而组合却是一种“Has-A”的关系。导致错误的使用继承而不是使用组合的一个重要原因可能就是错误的把“Has-A”当成了“Is-A”。



人被继承到雇员，学生，经理子类。而实际上，雇员、学生和经理分别描述一种角色，而人可以同时有几种不同的角色。比如，一个人既然是经理了就一定是雇员，使用继承来实现角色，则只能使用每一个人具有一种角色，这显然是不合理的。错误的原因就是把角色的等级结构和人的等级结构混淆起来，把Has-A的关系误认为是Is-A的关系，通过下面的改正就可以正确的做到这一点。



从上图可以看出，每一个“人”都可以有一个以上的“角色”，所有一个“人”可以同时是“雇员”，又是“经理”，甚至同时又是“学生”。而且由于“人”与“角色”的耦合是通过合成的，因此，角色可以有动态的变化。一个“人”可以开始是“雇员”，然后晋升为“经理”，然后又由于他参加了MBA课程，又称为了“学生“。

当一个类是另一个类的角色时，不应当使用继承描述这种关系。

**与里氏代换原则联合使用**

里氏代换原则是继承复用的基石。如果在任何可以使用B类型的地方都可以使用S类型，那么S类型才可以称为B类型的子类型(SubType)，而B类型才能称为S类型的基类型(BaseType)。

换言之，只有当每一个S在任何情况下都是一种B的时候，才可以将S设计成B的子类。如果两个类的关系是“Has-A”关系而不是“Is -A”，这两个类一定违反里氏代换原则。

只有两个类满足里氏代换原则，才有可能是“Is -A”关系。

## 单一职责原则（Single responsibility principle）

定义：不要存在多于一个导致类变更的原因。通俗的说，即一个类只负责一项职责。

问题由来：类T负责两个不同的职责：职责P1，职责P2。当由于职责P1需求发生改变而需要修改类T时，有可能会导致原本运行正常的职责P2功能发生故障。

解决方案：遵循单一职责原则。分别建立两个类T1、T2，使T1完成职责P1功能，T2完成职责P2功能。这样，当修改类T1时，不会使职责P2发生故障风险；同理，当修改T2时，也不会使职责P1发生故障风险。

说到单一职责原则，很多人都会不屑一顾。因为它太简单了。稍有经验的程序员即使从来没有读过设计模式、从来没有听说过单一职责原则，在设计软件时也会自觉的遵守这一重要原则，因为这是常识。在软件编程中，谁也不希望因为修改了一个功能导致其他的功能发生故障。而避免出现这一问题的方法便是遵循单一职责原则。虽然单一职责原则如此简单，并且被认为是常识，但是即便是经验丰富的程序员写出的程序，也会有违背这一原则的代码存在。为什么会出现这种现象呢？因为有职责扩散。所谓职责扩散，就是因为某种原因，职责P被分化为粒度更细的职责P1和P2。

比如：类T只负责一个职责P，这样设计是符合单一职责原则的。后来由于某种原因，也许是需求变更了，也许是程序的设计者境界提高了，需要将职责P细分为粒度更细的职责P1，P2，这时如果要使程序遵循单一职责原则，需要将类T也分解为两个类T1和T2，分别负责P1、P2两个职责。但是在程序已经写好的情况下，这样做简直太费时间了。所以，简单的修改类T，用它来负责两个职责是一个比较不错的选择，虽然这样做有悖于单一职责原则。（这样做的风险在于职责扩散的不确定性，因为我们不会想到这个职责P，在未来可能会扩散为P1，P2，P3，P4……Pn。所以记住，在职责扩散到我们无法控制的程度之前，立刻对代码进行重构。）

举例说明，用一个类描述动物呼吸这个场景：

**class** Animal{

**public** **void** breathe(String animal){

System.***out***.println(animal+"呼吸空气");

}

}

**public** **class** Client{

**public** **static** **void** main(String[] args){

Animal animal = **new** Animal();

animal.breathe("牛");

animal.breathe("羊");

animal.breathe("猪");

}

}

运行结果：

牛呼吸空气

羊呼吸空气

猪呼吸空气

程序上线后，发现问题了，并不是所有的动物都呼吸空气的，比如鱼就是呼吸水的。修改时如果遵循单一职责原则，需要将Animal类细分为陆生动物类Terrestrial，水生动物Aquatic，代码如下：

**class** Terrestrial{

**public** **void** breathe(String animal){

System.***out***.println(animal+"呼吸空气");

}

}

**class** Aquatic{

**public** **void** breathe(String animal){

System.***out***.println(animal+"呼吸水");

}

}

**public** **class** Client{

**public** **static** **void** main(String[] args){

Terrestrial terrestrial = **new** Terrestrial();

terrestrial.breathe("牛");

terrestrial.breathe("羊");

terrestrial.breathe("猪");

Aquatic aquatic = **new** Aquatic();

aquatic.breathe("鱼");

}

}

运行结果：

牛呼吸空气

羊呼吸空气

猪呼吸空气

鱼呼吸水

我们会发现如果这样修改花销是很大的，除了将原来的类分解之外，还需要修改客户端。而直接修改类Animal来达成目的虽然违背了单一职责原则，但花销却小的多，代码如下：

**class** Animal{

**public** **void** breathe(String animal){

**if**("鱼".equals(animal)){

System.***out***.println(animal+"呼吸水");

}**else**{

System.***out***.println(animal+"呼吸空气");

}

}

}

**public** **class** Client{

**public** **static** **void** main(String[] args){

Animal animal = **new** Animal();

animal.breathe("牛");

animal.breathe("羊");

animal.breathe("猪");

animal.breathe("鱼");

}

}

可以看到，这种修改方式要简单的多。但是却存在着隐患：有一天需要将鱼分为呼吸淡水的鱼和呼吸海水的鱼，则又需要修改Animal类的breathe方法，而对原有代码的修改会对调用“猪”“牛”“羊”等相关功能带来风险，也许某一天你会发现程序运行的结果变为“牛呼吸水”了。这种修改方式直接在代码级别上违背了单一职责原则，虽然修改起来最简单，但隐患却是最大的。还有一种修改方式：

**class** Animal{

**public** **void** breathe(String animal){

System.***out***.println(animal+"呼吸空气");

}

**public** **void** breathe2(String animal){

System.***out***.println(animal+"呼吸水");

}

}

**public** **class** Client{

**public** **static** **void** main(String[] args){

Animal animal = **new** Animal();

animal.breathe("牛");

animal.breathe("羊");

animal.breathe("猪");

animal.breathe2("鱼");

}

}

可以看到，这种修改方式没有改动原来的方法，而是在类中新加了一个方法，这样虽然也违背了单一职责原则，但在方法级别上却是符合单一职责原则的，因为它并没有动原来方法的代码。这三种方式各有优缺点，那么在实际编程中，采用哪一中呢？其实这真的比较难说，需要根据实际情况来确定。我的原则是：只有逻辑足够简单，才可以在代码级别上违反单一职责原则；只有类中方法数量足够少，才可以在方法级别上违反单一职责原则；

例如本文所举的这个例子，它太简单了，它只有一个方法，所以，无论是在代码级别上违反单一职责原则，还是在方法级别上违反，都不会造成太大的影响。实际应用中的类都要复杂的多，一旦发生职责扩散而需要修改类时，除非这个类本身非常简单，否则还是遵循单一职责原则的好。

遵循单一职责原的优点有：

1. 可以降低类的复杂度，一个类只负责一项职责，其逻辑肯定要比负责多项职责简单的多；
2. 提高类的可读性，提高系统的可维护性；
3. 变更引起的风险降低，变更是必然的，如果单一职责原则遵守的好，当修改一个功能时，可以显著降低对其他功能的影响。

需要说明的一点是单一职责原则不只是面向对象编程思想所特有的，只要是模块化的程序设计，都适用单一职责原则。

# Java23种设计模式

## 创建型

### 工厂模式

**场景问题**

举个生活中常见的例子——组装电脑，我们在组装电脑的时候，通常需要选择一系列的配件，比如CPU、硬盘、内存、主板、电源、机箱等。为讨论使用简单点，只考虑选择CPU和主板的问题。

事实上，在选择CPU的时候，面临一系列的问题，比如品牌、型号、针脚数目、主频等问题，只有把这些问题都确定下来，才能确定具体的CPU。

同样，在选择主板的时候，也有一系列问题，比如品牌、芯片组、集成芯片、总线频率等问题，也只有这些都确定了，才能确定具体的主板。

选择不同的CPU和主板，是每个客户在组装电脑的时候，向装机公司提出的要求，也就是我们每个人自己拟定的装机方案。

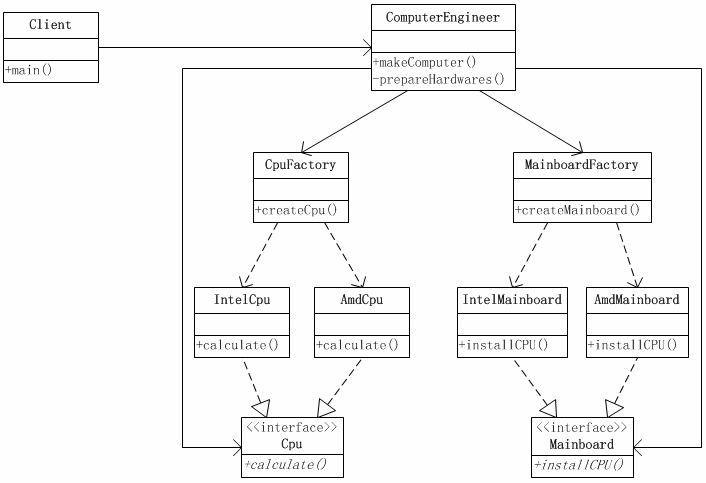
在最终确定这个装机方案之前，还需要整体考虑各个配件之间的兼容性。比如：CPU和主板，如果使用Intel的CPU和AMD的主板是根本无法组装的。因为Intel的CPU针脚数与AMD主板提供的CPU插口不兼容，就是说如果使用Intel的CPU根本就插不到AMD的主板中，所以装机方案是整体性的，里面选择的各个配件之间是有关联的。

对于装机工程师而言，他只知道组装一台电脑，需要相应的配件，但是具体使用什么样的配件，还得由客户说了算。也就是说装机工程师只是负责组装，而客户负责选择装配所需要的具体的配件。因此，当装机工程师为不同的客户组装电脑时，只需要根据客户的装机方案，去获取相应的配件，然后组装即可。

**工厂模式的解决方案**

考虑客户的功能，需要选择自己需要的CPU和主板，然后告诉装机工程师自己的选择，接下来就等着装机工程师组装电脑了。

对装机工程师而言，只是知道CPU和主板的接口，而不知道具体实现，很明显可以用上简单工厂模式或工厂方法模式。为了简单，这里选用简单工厂。客户告诉装机工程师自己的选择，然后装机工程师会通过相应的工厂去获取相应的实例对象。



CPU接口与具体实现

**public** **interface** Cpu {

**public** **void** calculate();

}

**public** **class** IntelCpu **implements** Cpu {

//CPU的针脚数

**private** **int** pins = 0;

**public** IntelCpu(**int** pins){

**this**.pins = pins;

}

@Override

**public** **void** calculate() {

System.***out***.println("Intel CPU的针脚数：" + pins);

}

}

**public** **class** AmdCpu **implements** Cpu {

//CPU的针脚数

**private** **int** pins = 0;

**public** AmdCpu(**int** pins){

**this**.pins = pins;

}

@Override

**public** **void** calculate() {

System.***out***.println("AMD CPU的针脚数：" + pins);

}

}

主板接口与具体实现

**public** **interface** Mainboard {

**public** **void** installCPU();

}

**public** **class** IntelMainboard **implements** Mainboard {

//CPU插槽的孔数

**private** **int** cpuHoles = 0;

/\*\*

\* 构造方法，传入CPU插槽的孔数

\* **@param** cpuHoles

\*/

**public** IntelMainboard(**int** cpuHoles){

**this**.cpuHoles = cpuHoles;

}

@Override

**public** **void** installCPU() {

System.***out***.println("Intel主板的CPU插槽孔数是：" + cpuHoles);

}

}

**public** **class** AmdMainboard **implements** Mainboard {

//CPU插槽的孔数

**private** **int** cpuHoles = 0;

/\*\*

\* 构造方法，传入CPU插槽的孔数

\* **@param** cpuHoles

\*/

**public** AmdMainboard(**int** cpuHoles){

**this**.cpuHoles = cpuHoles;

}

@Override

**public** **void** installCPU() {

System.***out***.println("AMD主板的CPU插槽孔数是：" + cpuHoles);

}

}

CPU与主板工厂类

**public** **class** CpuFactory {

**public** **static** Cpu createCpu(**int** type){

Cpu cpu = **null**;

**if**(type == 1){

cpu = **new** IntelCpu(755);

}**else** **if**(type == 2){

cpu = **new** AmdCpu(938);

}

**return** cpu;

}

}

**public** **class** MainboardFactory {

**public** **static** Mainboard createMainboard(**int** type){

Mainboard mainboard = **null**;

**if**(type == 1){

mainboard = **new** IntelMainboard(755);

}**else** **if**(type == 2){

mainboard = **new** AmdMainboard(938);

}

**return** mainboard;

}

}

装机工程师类与客户类运行结果如下：

**public** **class** ComputerEngineer {

//定义组装机需要的CPU

**private** Cpu cpu = **null**;

//定义组装机需要的主板

**private** Mainboard mainboard = **null**;

**public** **void** makeComputer(**int** cpuType , **int** mainboard){

/\*\*

\* 组装机器的基本步骤

\*/

//1:首先准备好装机所需要的配件

prepareHardwares(cpuType, mainboard);

//2:组装机器

//3:测试机器

//4：交付客户

}

**private** **void** prepareHardwares(**int** cpuType , **int** mainboard){

//这里要去准备CPU和主板的具体实现，为了示例简单，这里只准备这两个

//可是，装机工程师并不知道如何去创建，怎么办呢？

//直接找相应的工厂获取

**this**.cpu = CpuFactory.createCpu(cpuType);

**this**.mainboard = MainboardFactory.createMainboard(mainboard);

//测试配件是否好用

**this**.cpu.calculate();

**this**.mainboard.installCPU();

}

}

**public** **class** Client {

**public** **static** **void** main(String[]args){

ComputerEngineer cf = **new** ComputerEngineer();

cf.makeComputer(1,1);

}

}

运行结果如下：

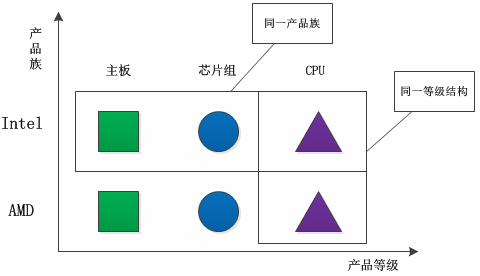


### 抽象工厂模式

每一个模式都是针对一定问题的解决方案。抽象工厂模式与工厂模式的最大区别就在于，工厂模式针对的是一个产品等级结构；而抽象工厂模式则需要面对多个产品等级结构。

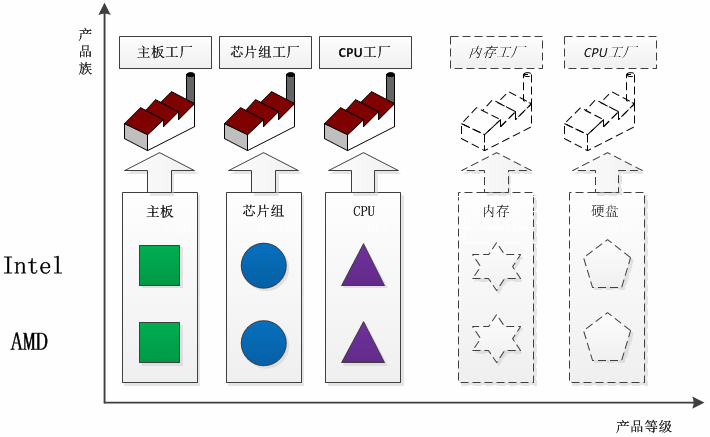
在学习抽象工厂具体实例之前，应该明白两个重要的概念：产品族和产品等级。

所谓产品族，是指位于不同产品等级结构中，功能相关联的产品组成的家族。比如AMD的主板、芯片组、CPU组成一个家族，Intel的主板、芯片组、CPU组成一个家族。而这两个家族都来自于三个产品等级：主板、芯片组、CPU。一个等级结构是由相同的结构的产品组成，示意图如下：

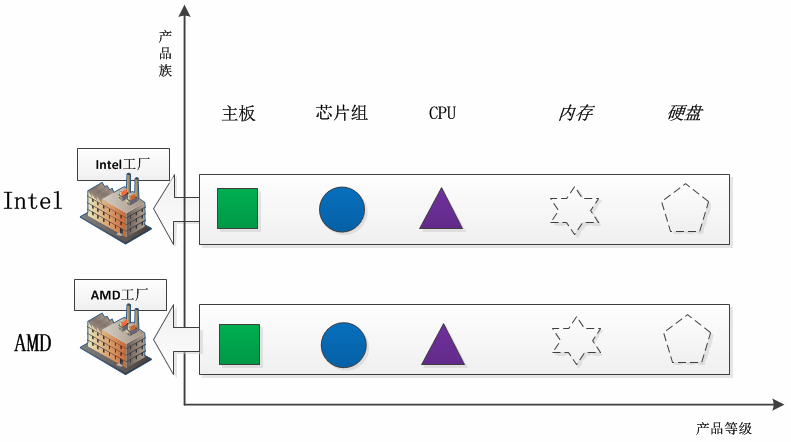


显然，每一个产品族中含有产品的数目，与产品等级结构的数目是相等的。产品的等级结构与产品族将产品按照不同方向划分，形成一个二维的坐标系。横轴表示产品的等级结构，纵轴表示产品族，上图共有两个产品族，分布于三个不同的产品等级结构中。只要指明一个产品所处的产品族以及它所属的等级结构，就可以唯一的确定这个产品。

上面所给出的三个不同的等级结构具有平行的结构。因此，如果采用工厂方法模式，就势必要使用三个独立的工厂等级结构来对付这三个产品等级结构。由于这三个产品等级结构的相似性，会导致三个平行的工厂等级结构。随着产品等级结构的数目的增加，工厂方法模式所给出的工厂等级结构的数目也会随之增加。如下图：



那么，是否可以使用同一个工厂等级结构来对付这些相同或者极为相似的产品等级结构呢？当然可以的，而且这就是抽象工厂模式的好处。同一个工厂等级结构负责三个不同产品等级结构中的产品对象的创建。



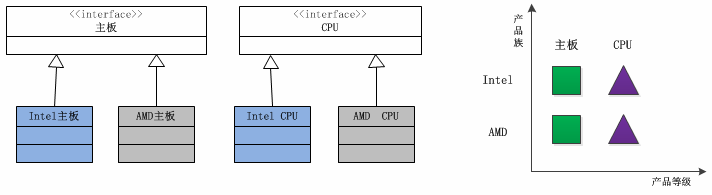
可以看出，一个工厂等级结构可以创建出分属于不同产品等级结构的一个产品族中的所有对象。显然，这时候抽象工厂模式比简单工厂模式、工厂方法模式更有效率。对应于每一个产品族都有一个具体工厂。而每一个具体工厂负责创建属于同一个产品族，但是分属于不同等级结构的产品。

**抽象工厂模式结构**

抽象工厂模式是对象的创建模式，它是工厂模式的进一步推广。

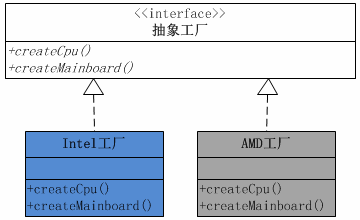
假设一个子系统需要一些产品对象，而这些产品又属于一个以上的产品等级结构。那么为了将消费这些产品对象的责任和创建这些产品对象的责任分割开来，可以引进抽象工厂模式。这样的话，消费产品的一方不需要直接参与产品的创建工作，而只需要向一个公用的工厂接口请求所需要的产品。

通过使用抽象工厂模式，可以处理具有相同（或者相似）等级结构中的多个产品族中的产品对象的创建问题。如下图所示：

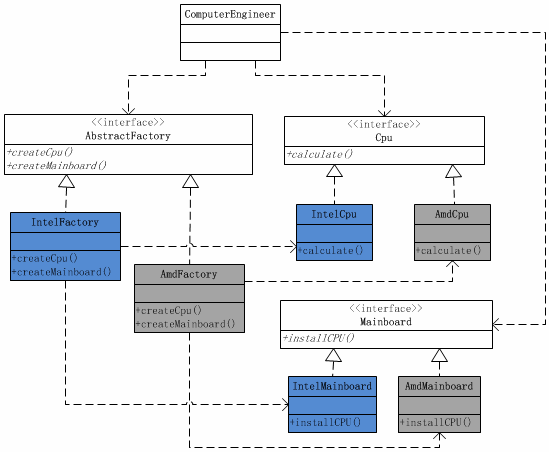


由于这两个产品族的等级结构相同，因此使用同一个工厂族也可以处理这两个产品族的创建问题，这就是抽象工厂模式。

根据产品角色的结构图，就不难给出工厂角色的结构设计图。



可以看出，每一个工厂角色都有两个工厂方法，分别负责创建分属不同产品等级结构的产品对象。



CPU接口与具体实现

**public** **interface** Cpu {

**public** **void** calculate();

}

**public** **class** IntelCpu **implements** Cpu {

//CPU的针脚数

**private** **int** pins = 0;

**public** IntelCpu(**int** pins){

**this**.pins = pins;

}

@Override

**public** **void** calculate() {

System.***out***.println("Intel CPU的针脚数：" + pins);

}

}

**public** **class** AmdCpu **implements** Cpu {

//CPU的针脚数

**private** **int** pins = 0;

**public** AmdCpu(**int** pins){

**this**.pins = pins;

}

@Override

**public** **void** calculate() {

System.***out***.println("AMD CPU的针脚数：" + pins);

}

}

主板接口与具体实现

**public** **interface** Mainboard {

**public** **void** installCPU();

}

**public** **class** IntelMainboard **implements** Mainboard {

//CPU插槽的孔数

**private** **int** cpuHoles = 0;

/\*\*

\* 构造方法，传入CPU插槽的孔数

\* **@param** cpuHoles

\*/

**public** IntelMainboard(**int** cpuHoles){

**this**.cpuHoles = cpuHoles;

}

@Override

**public** **void** installCPU() {

System.***out***.println("Intel主板的CPU插槽孔数是：" + cpuHoles);

}

}

**public** **class** AmdMainboard **implements** Mainboard {

//CPU插槽的孔数

**private** **int** cpuHoles = 0;

/\*\*

\* 构造方法，传入CPU插槽的孔数

\* **@param** cpuHoles

\*/

**public** AmdMainboard(**int** cpuHoles){

**this**.cpuHoles = cpuHoles;

}

@Override

**public** **void** installCPU() {

System.***out***.println("AMD主板的CPU插槽孔数是：" + cpuHoles);

}

}

CPU与主板工厂类

**public** **class** CpuFactory {

**public** **static** Cpu createCpu(**int** type){

Cpu cpu = **null**;

**if**(type == 1){

cpu = **new** IntelCpu(755);

}**else** **if**(type == 2){

cpu = **new** AmdCpu(938);

}

**return** cpu;

}

}

**public** **class** MainboardFactory {

**public** **static** Mainboard createMainboard(**int** type){

Mainboard mainboard = **null**;

**if**(type == 1){

mainboard = **new** IntelMainboard(755);

}**else** **if**(type == 2){

mainboard = **new** AmdMainboard(938);

}

**return** mainboard;

}

}

抽象工厂类和实现类

**public** **interface** AbstractFactory {

//创建CPU对象

**public** Cpu createCpu();

//创建主板对象

**public** Mainboard createMainboard();

}

**public** **class** IntelFactory **implements** AbstractFactory {

@Override

**public** Cpu createCpu() {

**return** **new** IntelCpu(755);

}

@Override

**public** Mainboard createMainboard() {

**return** **new** IntelMainboard(755);

}

}

**public** **class** AmdFactory **implements** AbstractFactory {

@Override

**public** Cpu createCpu() {

**return** **new** IntelCpu(938);

}

@Override

**public** Mainboard createMainboard() {

**return** **new** IntelMainboard(938);

}

}

装机工程师类与客户类运行结果如下：

**public** **class** ComputerEngineer {

//定义组装机需要的CPU

**private** Cpu cpu = **null**;

//定义组装机需要的主板

**private** Mainboard mainboard = **null**;

**public** **void** makeComputer(AbstractFactory af){

/\*\*

\* 组装机器的基本步骤

\*/

//1:首先准备好装机所需要的配件

prepareHardwares(af);

//2:组装机器

//3:测试机器

//4：交付客户

}

**private** **void** prepareHardwares(AbstractFactory af){

//这里要去准备CPU和主板的具体实现，为了示例简单，这里只准备这两个

//可是，装机工程师并不知道如何去创建，怎么办呢？

//直接找相应的工厂获取

**this**.cpu = af.createCpu();

**this**.mainboard = af.createMainboard();

//测试配件是否好用

**this**.cpu.calculate();

**this**.mainboard.installCPU();

}

}

**public** **class** Client {

**public** **static** **void** main(String[]args){

//创建装机工程师对象

ComputerEngineer cf = **new** ComputerEngineer();

//客户选择并创建需要使用的产品对象

AbstractFactory af = **new** IntelFactory();

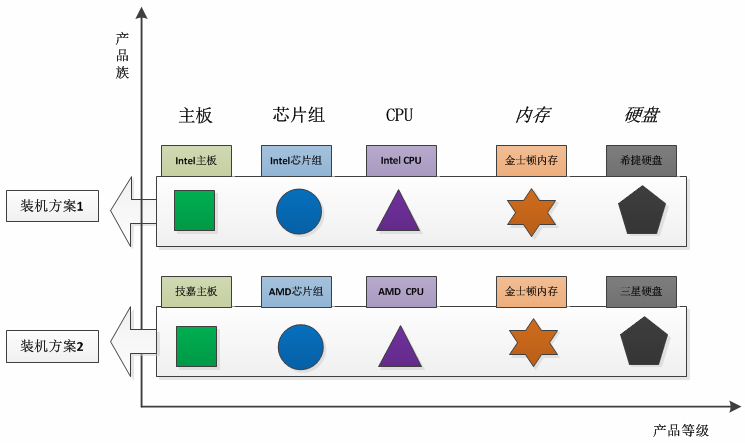
//告诉装机工程师自己选择的产品，让装机工程师组装电脑

cf.makeComputer(af);

}

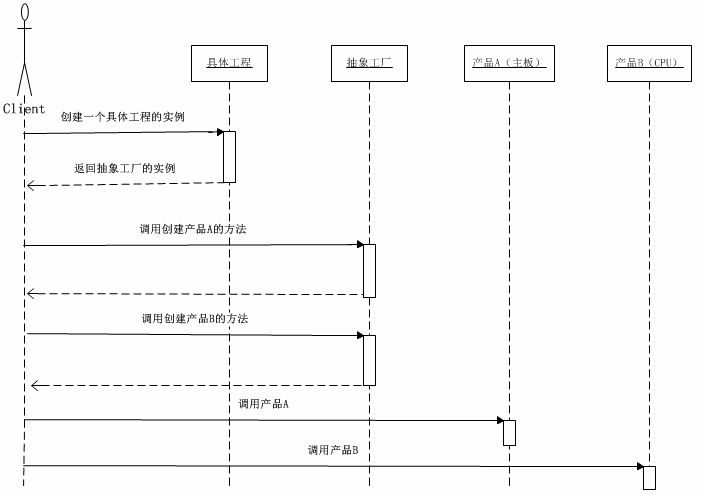
}

抽象工厂的功能是为一系列相关对象或相互依赖的对象创建一个接口。一定要注意，这个接口内的方法不是任意堆砌的，而是一系列相关或相互依赖的方法。比如上面例子中的主板和CPU，都是为了组装一台电脑的相关对象。不同的装机方案，代表一种具体的电脑系列。



由于抽象工厂定义的一系列对象通常是相关或相互依赖的，这些产品对象就构成了一个产品族，也就是抽象工厂定义了一个产品族。

这就带来非常大的灵活性，切换产品族的时候，只要提供不同的抽象工厂实现就可以了，也就是说现在是以一个产品族作为一个整体被切换。



**在什么情况下应当使用抽象工厂模式**

1. 一个系统不应当依赖于产品类实例如何被创建、组合和表达的细节，这对于所有形态的工厂模式都是重要的。
2. 这个系统的产品有多于一个的产品族，而系统只消费其中某一族的产品。
3. 同属于同一个产品族的产品是在一起使用的，这一约束必须在系统的设计中体现出来。（比如：Intel主板必须使用Intel CPU、Intel芯片组）
4. 系统提供一个产品类的库，所有的产品以同样的接口出现，从而使客户端不依赖于实现。

**抽象工厂模式的优点**

1. 分离接口和实现

客户端使用抽象工厂来创建需要的对象，而客户端根本就不知道具体的实现是谁，客户端只是面向产品的接口编程而已。也就是说，客户端从具体的产品实现中解耦。

1. 使切换产品族变得容易

　　因为一个具体的工厂实现代表的是一个产品族，比如上面例子的从Intel系列到AMD系列只需要切换一下具体工厂。

**抽象工厂模式的缺点**

1. 不太容易扩展新的产品

　　如果需要给整个产品族添加一个新的产品，那么就需要修改抽象工厂，这样就会导致修改所有的工厂实现类。

### 单例模式

单例模式确保某一个类只有一个实例，而且自行实例化并向整个系统提供这个实例。这个类称为单例类。

**单例模式的特点：**

1. 单例类只能有一个实例。
2. 单例类必须自己创建自己的唯一实例。
3. 单例类必须给所有其他对象提供这一实例。

**饿汉式单例类**

**public** **class** EagerSingleton {

**private** **static** EagerSingleton *instance* = **new** EagerSingleton();

//私有默认构造方法

**private** EagerSingleton(){}

//静态工厂方法

**public** **static** EagerSingleton getInstance(){

**return** *instance*;

}

}

上面的例子中，在这个类被加载时，静态变量instance会被初始化，此时类的私有构造子会被调用。这时候，单例类的唯一实例就被创建出来了。

饿汉式其实是一种比较形象的称谓。既然饿，那么在创建对象实例的时候就比较着急，饿了嘛，于是在装载类的时候就创建对象实例。

**private** **static** EagerSingleton *instance* = **new** EagerSingleton();

饿汉式是典型的空间换时间，当类装载的时候就会创建类的实例，不管你用不用，先创建出来，然后每次调用的时候，就不需要再判断，节省了运行时间。

**懒汉式单例类**

**public** **class** LazySingleton {

**private** **static** LazySingleton *instance* = **null**;

//私有默认构造方法

**private** LazySingleton(){}

//静态工厂方法

**public** **static** **synchronized** LazySingleton getInstance(){

**if**(*instance* == **null**){

*instance* = **new** LazySingleton();

}

**return** *instance*;

}

}

上面的懒汉式单例类实现里对静态工厂方法使用了同步化，以处理多线程环境。

懒汉式其实是一种比较形象的称谓。既然懒，那么在创建对象实例的时候就不着急。会一直等到马上要使用对象实例的时候才会创建，懒人嘛，总是推脱不开的时候才会真正去执行工作，因此在装载对象的时候不创建对象实例。

**private** **static** LazySingleton *instance* = **null**;

懒汉式是典型的时间换空间,就是每次获取实例都会进行判断，看是否需要创建实例，浪费判断的时间。当然，如果一直没有人使用的话，那就不会创建实例，则节约内存空间

由于懒汉式的实现是线程安全的，这样会降低整个访问的速度，而且每次都要判断。那么有没有更好的方式实现呢？

**双重检查加锁**

可以使用“双重检查加锁”的方式来实现，就可以既实现线程安全，又能够使性能不受很大的影响。那么什么是“双重检查加锁”机制呢？

所谓“双重检查加锁”机制，指的是：并不是每次进入getInstance方法都需要同步，而是先不同步，进入方法后，先检查实例是否存在，如果不存在才进行下面的同步块，这是第一重检查，进入同步块过后，再次检查实例是否存在，如果不存在，就在同步的情况下创建一个实例，这是第二重检查。这样一来，就只需要同步一次了，从而减少了多次在同步情况下进行判断所浪费的时间。

“双重检查加锁”机制的实现会使用关键字volatile，它的意思是：被volatile修饰的变量的值，将不会被本地线程缓存，所有对该变量的读写都是直接操作共享内存，从而确保多个线程能正确的处理该变量。

注意：在java1.4及以前版本中，很多JVM对于volatile关键字的实现的问题，会导致“双重检查加锁”的失败，因此“双重检查加锁”机制只能用在java5及以上的版本。

**public** **class** Singleton {

**private** **volatile** **static** Singleton *instance* = **null**;

**private** Singleton(){}

**public** **static** Singleton getInstance(){

//先检查实例是否存在，如果不存在才进入下面的同步块

**if**(*instance* == **null**){

//同步块，线程安全的创建实例

**synchronized** (Singleton.**class**) {

//再次检查实例是否存在，如果不存在才真正的创建实例

**if**(*instance* == **null**){

*instance* = **new** Singleton();

}

}

}

**return** *instance*;

}

}

这种实现方式既可以实现线程安全地创建实例，而又不会对性能造成太大的影响。它只是第一次创建实例的时候同步，以后就不需要同步了，从而加快了运行速度。

提示：由于volatile关键字可能会屏蔽掉虚拟机中一些必要的代码优化，所以运行效率并不是很高。因此一般建议，没有特别的需要，不要使用。也就是说，虽然可以使用“双重检查加锁”机制来实现线程安全的单例，但并不建议大量采用，可以根据情况来选用。

根据上面的分析，常见的两种单例实现方式都存在小小的缺陷，那么有没有一种方案，既能实现延迟加载，又能实现线程安全呢？

**Lazy initialization holder class模式**

这个模式综合使用了Java的类级内部类和多线程缺省同步锁的知识，很巧妙地同时实现了延迟加载和线程安全。

什么是类级内部类:

简单点说，类级内部类指的是，有static修饰的成员式内部类。如果没有static修饰的成员式内部类被称为对象级内部类。

类级内部类相当于其外部类的static成分，它的对象与外部类对象间不存在依赖关系，因此可直接创建。而对象级内部类的实例，是绑定在外部对象实例中的。

类级内部类中，可以定义静态的方法。在静态方法中只能够引用外部类中的静态成员方法或者成员变量。

类级内部类相当于其外部类的成员，只有在第一次被使用的时候才被会装载。

多线程缺省同步锁的知识:

大家都知道，在多线程开发中，为了解决并发问题，主要是通过使用synchronized来加互斥锁进行同步控制。但是在某些情况中，JVM已经隐含地为您执行了同步，这些情况下就不用自己再来进行同步控制了。这些情况包括：

1. 由静态初始化器（在静态字段上或static{}块中的初始化器）初始化数据时
2. 访问final字段时
3. 在创建线程之前创建对象时
4. 线程可以看见它将要处理的对象时

解决方案的思路:

要想很简单地实现线程安全，可以采用静态初始化器的方式，它可以由JVM来保证线程的安全性。比如前面的饿汉式实现方式。但是这样一来，不是会浪费一定的空间吗？因为这种实现方式，会在类装载的时候就初始化对象，不管你需不需要。

如果现在有一种方法能够让类装载的时候不去初始化对象，那不就解决问题了？一种可行的方式就是采用类级内部类，在这个类级内部类里面去创建对象实例。这样一来，只要不使用到这个类级内部类，那就不会创建对象实例，从而同时实现延迟加载和线程安全。

示例代码如下：

**public** **class** Singleton {

**private** Singleton(){}

/\*\*

\* 类级的内部类，也就是静态的成员式内部类，该内部类的实例与外部类的实例

\* 没有绑定关系，而且只有被调用到时才会装载，从而实现了延迟加载。

\*/

**private** **static** **class** SingletonHolder{

//静态初始化器，由JVM来保证线程安全

**private** **static** Singleton *instance* = **new** Singleton();

}

**public** **static** Singleton getInstance(){

**return** SingletonHolder.*instance*;

}

}

当getInstance方法第一次被调用的时候，它第一次读取SingletonHolder.instance，导致SingletonHolder类得到初始化；而这个类在装载并被初始化的时候，会初始化它的静态域，从而创建Singleton的实例，由于是静态的域，因此只会在虚拟机装载类的时候初始化一次，并由虚拟机来保证它的线程安全性。

这个模式的优势在于，getInstance方法并没有被同步，并且只是执行一个域的访问，因此延迟初始化并没有增加任何访问成本。

**单例和枚举**

按照《高效Java 第二版》中的说法：单元素的枚举类型已经成为实现Singleton的最佳方法。用枚举来实现单例非常简单，只需要编写一个包含单个元素的枚举类型即可。

**public** **enum** Singleton {

//定义一个枚举的元素，它就代表了Singleton的一个实例。

***uniqueInstance***;

//单例可以有自己的操作

**public** **void** singletonOperation(){

//功能处理

}

}

使用枚举来实现单实例控制会更加简洁，而且无偿地提供了序列化机制，并由JVM从根本上提供保障，绝对防止多次实例化，是更简洁、高效、安全的实现单例的方式。

### 建造者模式

建造模式可以将一个产品的内部表象（internal representation）与产品的生产过程分割开来，从而可以使一个建造过程生成具有不同的内部表象的产品对象。

**产品的内部表象:**

一个产品常有不同的组成成分作为产品的零件，这些零件有可能是对象，也有可能不是对象，它们通常又叫做产品的内部表象（internal representation）。不同的产品可以有不同的内部表象，也就是不同的零件。使用建造模式可以使客户端不需要知道所生成的产品有哪些零件，每个产品的对应零件彼此有何不同，是怎么建造出来的，以及怎么组成产品。

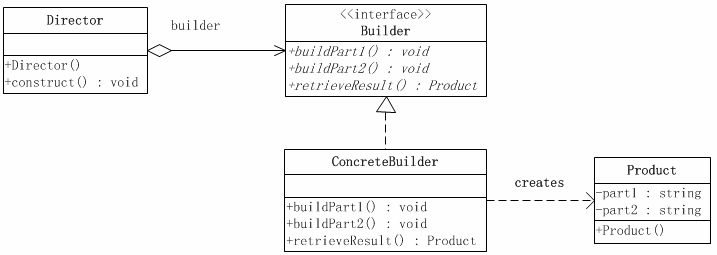
**对象性质的建造:**

有些情况下，一个对象会有一些重要的性质，在它们没有恰当的值之前，对象不能作为一个完整的产品使用。比如，一个电子邮件有发件人地址、收件人地址、主题、内容、附录等部分，而在最起码的收件人地址得到赋值之前，这个电子邮件不能发送。

有些情况下，一个对象的一些性质必须按照某个顺序赋值才有意义。在某个性质没有赋值之前，另一个性质则无法赋值。这些情况使得性质本身的建造涉及到复杂的商业逻辑。这时候，此对象相当于一个有待建造的产品，而对象的这些性质相当于产品的零件，建造产品的过程是建造零件的过程。由于建造零件的过程很复杂，因此，这些零件的建造过程往往被“外部化”到另一个称做建造者的对象里，建造者对象返还给客户端的是一个全部零件都建造完毕的产品对象。

建造模式利用一个导演者对象和具体建造者对象一个个地建造出所有的零件，从而建造出完整的产品对象。建造者模式将产品的结构和产品的零件的建造过程对客户端隐藏起来，把对建造过程进行指挥的责任和具体建造者零件的责任分割开来，达到责任划分和封装的目的。

**建造模式的结构:**



在这个示意性的系统里，最终产品Product只有两个零件，即part1和part2。相应的建造方法也有两个：buildPart1()和buildPart2()、同时可以看出本模式涉及到四个角色，它们分别是：

**抽象建造者（Builder）角色：**给 出一个抽象接口，以规范产品对象的各个组成成分的建造。一般而言，此接口独立于应用程序的商业逻辑。模式中直接创建产品对象的是具体建造者 (ConcreteBuilder)角色。具体建造者类必须实现这个接口所要求的两种方法：一种是建造方法(buildPart1和 buildPart2)，另一种是返还结构方法(retrieveResult)。一般来说，产品所包含的零件数目与建造方法的数目相符。换言之，有多少 零件，就有多少相应的建造方法。

**具体建造者（ConcreteBuilder）角色：**担任这个角色的是与应用程序紧密相关的一些类，它们在应用程序调用下创建产品的实例。这个角色要完成的任务包括：1.实现抽象建造者Builder所声明的接口，给出一步一步地完成创建产品实例的操作。2.在建造过程完成后，提供产品的实例。

**导演者（Director）角色：**担任这个角色的类调用具体建造者角色以创建产品对象。应当指出的是，导演者角色并没有产品类的具体知识，真正拥有产品类的具体知识的是具体建造者角色。

**产品（Product）角色：**产品便是建造中的复杂对象。一般来说，一个系统中会有多于一个的产品类，而且这些产品类并不一定有共同的接口，而完全可以是不相关联的。

导演者角色是与客户端打交道的角色。导演者将客户端创建产品的请求划分为对各个零件的建造请求，再将这些请求委派给具体建造者角色。具体建造者角色是做具体建造工作的，但是却不为客户端所知。

一般来说，每有一个产品类，就有一个相应的具体建造者类。这些产品应当有一样数目的零件，而每有一个零件就相应地在所有的建造者角色里有一个建造方法。

产品类Product:

**public** **class** Product {

//定义一些关于产品的操作

**private** String part1;

**private** String part2;

**public** String getPart1() {

**return** part1;

}

**public** **void** setPart1(String part1) {

**this**.part1 = part1;

}

**public** String getPart2() {

**return** part2;

}

**public** **void** setPart2(String part2) {

**this**.part2 = part2;

}

}

抽象建造者类Builder:

**public** **interface** Builder {

**public** **void** buildPart1();

**public** **void** buildPart2();

**public** Product retrieveResult();

}

具体建造者类ConcreteBuilder:

**public** **class** ConcreteBuilder **implements** Builder {

**private** Product product = **new** Product();

//产品零件建造方法1

@Override

**public** **void** buildPart1() {

//构建产品的第一个零件

　　　　product.setPart1("编号：9527");

}

//产品零件建造方法2

@Override

**public** **void** buildPart2() {

//构建产品的第二个零件

　　　　product.setPart2("名称：XXX");

}

//产品返还方法

@Override

**public** Product retrieveResult() {

**return** product;

}

}

导演者类Director:

**public** **class** Director {

//持有当前需要使用的建造器对象

**private** Builder builder;

//构造方法，传入建造器对象

**public** Director(Builder builder){

**this**.builder = builder;

}

//产品构造方法，负责调用各个零件建造方法

**public** **void** construct(){

builder.buildPart1();

builder.buildPart2();

}

}

客户端类Client:

**public** **class** Client {

**public** **static** **void** main(String[]args){

Builder builder = **new** ConcreteBuilder();

Director director = **new** Director(builder);

director.construct();

Product product = builder.retrieveResult();

System.***out***.println(product.getPart1());

System.***out***.println(product.getPart2());

}

}

客户端负责创建导演者和具体建造者对象。然后，客户端把具体建造者对象交给导演者，导演者操作具体建造者，开始创建产品。当产品完成后，建造者把产品返还给客户端。

把创建具体建造者对象的任务交给客户端而不是导演者对象，是为了将导演者对象与具体建造者对象的耦合变成动态的，从而使导演者对象可以操纵数个具体建造者对象中的任何一个。

**建造模式分成两个很重要的部分：**

1. 一个部分是Builder接口，这里是定义了如何构建各个部件，也就是知道每个部件功能如何实现，以及如何装配这些部件到产品中去；
2. 另外一个部分是Director，Director是知道如何组合来构建产品，也就是说Director负责整体的构建算法，而且通常是分步骤地来执行。

不管如何变化，建造模式都存在这么两个部分，一个部分是部件构造和产品装配，另一个部分是整体构建的算法。认识这点是很重要的，因为在建造模式中，强调的是固定整体构建的算法，而灵活扩展和切换部件的具体构造和产品装配的方式。

再直白点说，建造模式的重心在于分离构建算法和具体的构造实现，从而使得构建算法可以重用。具体的构造实现可以很方便地扩展和切换，从而可以灵活地组合来构造出不同的产品对象。

**在什么情况下使用建造模式**

1. 需要生成的产品对象有复杂的内部结构，每一个内部成分本身可以是对象，也可以仅仅是一个对象（即产品对象）的一个组成部分。
2. 需要生成的产品对象的属性相互依赖。建造模式可以强制实行一种分步骤进行的建造过程，因此，如果产品对象的一个属性必须在另一个属性被赋值之后才可以被赋值，使用建造模式是一个很好的设计思想。
3. 在对象创建过程中会使用到系统中的其他一些对象，这些对象在产品对象的创建过程中不易得到。

### 原型模式

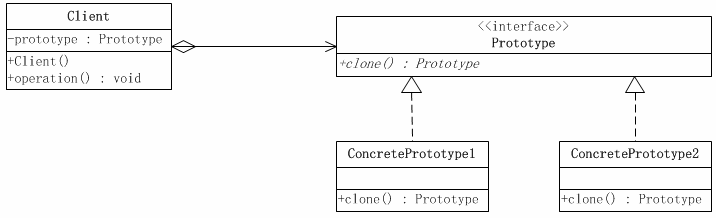
通过给出一个原型对象来指明所有创建的对象的类型，然后用复制这个原型对象的办法创建出更多同类型的对象。这就是选型模式的用意。

**原型模式的结构**

原型模式要求对象实现一个可以“克隆”自身的接口，这样就可以通过复制一个实例对象本身来创建一个新的实例。这样一来，通过原型实例创建新的对象，就不再需要关心这个实例本身的类型，只要实现了克隆自身的方法，就可以通过这个方法来获取新的对象，而无须再去通过new来创建。

原型模式有两种表现形式：（1）简单形式、（2）登记形式，这两种表现形式仅仅是原型模式的不同实现。

**简单形式的原型模式:**



这种形式涉及到三个角色：

1. 客户(Client)角色：客户类提出创建对象的请求。
2. 抽象原型(Prototype)角色：这是一个抽象角色，通常由一个Java接口或Java抽象类实现。此角色给出所有的具体原型类所需的接口。
3. 具体原型（Concrete Prototype）角色：被复制的对象。此角色需要实现抽象的原型角色所要求的接口。

抽象原型角色:

**public** **interface** Prototype{

//克隆自身的方法

**public** Object clone();

}

具体原型角色:

**public** **class** ConcretePrototype1 **implements** Prototype {

**public** Prototype clone(){

//最简单的克隆，新建一个自身对象，由于没有属性就不再复制值了

Prototype prototype = **new** ConcretePrototype1();

**return** prototype;

}

}

**public** **class** ConcretePrototype2 **implements** Prototype {

**public** Prototype clone(){

//最简单的克隆，新建一个自身对象，由于没有属性就不再复制值了

Prototype prototype = **new** ConcretePrototype2();

**return** prototype;

}

}

客户端角色:

**public** **class** Client {

//持有需要使用的原型接口对象

**private** Prototype prototype;

//构造方法，传入需要使用的原型接口对象

**public** Client(Prototype prototype){

**this**.prototype = prototype;

}

**public** **void** operation(Prototype example){

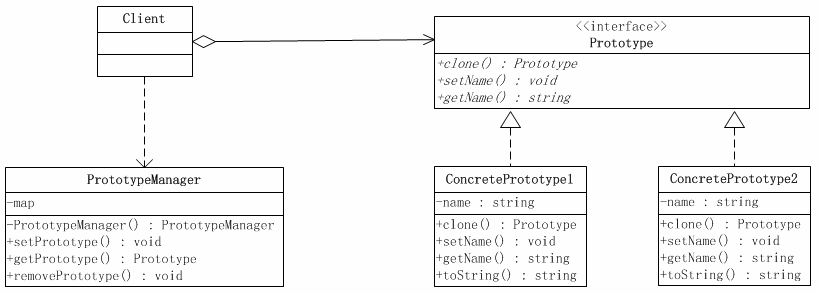
//需要创建原型接口的对象

Prototype copyPrototype = prototype.clone();

}

}

**登记形式的原型模式**



作为原型模式的第二种形式，它多了一个原型管理器(PrototypeManager)角色，该角色的作用是：创建具体原型类的对象，并记录每一个被创建的对象。

抽象原型角色:

**public** **interface** Prototype{

**public** Prototype clone();

**public** String getName();

**public** **void** setName(String name);

}

具体原型角色:

**public** **class** ConcretePrototype1 **implements** Prototype {

**private** String name;

**public** Prototype clone(){

ConcretePrototype1 prototype = **new** ConcretePrototype1();

prototype.setName(**this**.name);

**return** prototype;

}

**public** String toString(){

**return** "Now in Prototype1 , name = " + **this**.name;

}

@Override

**public** String getName() {

**return** name;

}

@Override

**public** **void** setName(String name) {

**this**.name = name;

}

}

**public** **class** ConcretePrototype2 **implements** Prototype {

**private** String name;

**public** Prototype clone(){

ConcretePrototype2 prototype = **new** ConcretePrototype2();

prototype.setName(**this**.name);

**return** prototype;

}

**public** String toString(){

**return** "Now in Prototype2 , name = " + **this**.name;

}

@Override

**public** String getName() {

**return** name;

}

@Override

**public** **void** setName(String name) {

**this**.name = name;

}

}

原型管理器角色保持一个聚集，作为对所有原型对象的登记，这个角色提供必要的方法，供外界增加新的原型对象和取得已经登记过的原型对象。

**public** **class** PrototypeManager {

//用来记录原型的编号和原型实例的对应关系

**private** **static** Map<String,Prototype> *map* = **new** HashMap<String,Prototype>();

//私有化构造方法，避免外部创建实例

**private** PrototypeManager(){}

/\*\*

\* 向原型管理器里面添加或是修改某个原型注册

\* **@param** prototypeId 原型编号

\* **@param** prototype 原型实例

\*/

**public** **synchronized** **static** **void** setPrototype(String prototypeId , Prototype prototype){

*map*.put(prototypeId, prototype);

}

/\*\*

\* 从原型管理器里面删除某个原型注册

\* **@param** prototypeId 原型编号

\*/

**public** **synchronized** **static** **void** removePrototype(String prototypeId){

*map*.remove(prototypeId);

}

/\*\*

\* 获取某个原型编号对应的原型实例

\* **@param** prototypeId 原型编号

\* **@return** 原型编号对应的原型实例

\* **@throws** Exception 如果原型编号对应的实例不存在，则抛出异常

\*/

**public** **synchronized** **static** Prototype getPrototype(String prototypeId) **throws** Exception{

Prototype prototype = *map*.get(prototypeId);

**if**(prototype == **null**){

**throw** **new** Exception("您希望获取的原型还没有注册或已被销毁");

}

**return** prototype;

}

}

客户端角色:

**public** **class** Client {

**public** **static** **void** main(String[]args){

**try**{

Prototype p1 = **new** ConcretePrototype1();

PrototypeManager.setPrototype("p1", p1);

//获取原型来创建对象

Prototype p3 = PrototypeManager.getPrototype("p1").clone();

p3.setName("张三");

System.***out***.println("第一个实例：" + p3);

//有人动态的切换了实现

Prototype p2 = **new** ConcretePrototype2();

PrototypeManager.setPrototype("p1", p2);

//重新获取原型来创建对象

Prototype p4 = PrototypeManager.getPrototype("p1").clone();

p4.setName("李四");

System.***out***.println("第二个实例：" + p4);

//有人注销了这个原型

PrototypeManager.removePrototype("p1");

//再次获取原型来创建对象

Prototype p5 = PrototypeManager.getPrototype("p1").clone();

p5.setName("王五");

System.***out***.println("第三个实例：" + p5);

}**catch**(Exception e){

e.printStackTrace();

}

}

}

**两种形式的比较**

1. 简单形式和登记形式的原型模式各有其长处和短处。
2. 如果需要创建的原型对象数目较少而且比较固定的话，可以采取第一种形式。在这种情况下，原型对象的引用可以由客户端自己保存。
3. 如果要创建的原型对象数目不固定的话，可以采取第二种形式。在这种情况下，客户端不保存对原型对象的引用，这个任务被交给管理员对象。在复制一个原型对象之前，客户端可以查看管理员对象是否已经有一个满足要求的原型对象。如果有，可以直接从管理员类取得这个对象引用；如果没有，客户端就需要自行复制此原型对象。

**原型模式的优点**

原型模式允许在运行时动态改变具体的实现类型。原型模式可以在运行期间，由客户来注册符合原型接口的实现类型，也可以动态地改变具体的实现类型，看起来接口没有任何变化，但其实运行的已经是另外一个类实例了。因为克隆一个原型就类似于实例化一个类。

**原型模式的缺点**

原型模式最主要的缺点是每一个类都必须配备一个克隆方法。配备克隆方法需要对类的功能进行通盘考虑，这对于全新的类来说不是很难，而对于已经有的类不一定很容易，特别是当一个类引用不支持序列化的间接对象，或者引用含有循环结构的时候。

**Java中的克隆方法**

Java的所有类都是从java.lang.Object类继承而来的，而Object类提供protected Object clone()方法对对象进行复制，子类当然也可以把这个方法置换掉，提供满足自己需要的复制方法。对象的复制有一个基本问题，就是对象通常都有对其他的对象的引用。当使用Object类的clone()方法来复制一个对象时，此对象对其他对象的引用也同时会被复制一份。

Java语言提供的Cloneable接口只起一个作用，就是在运行时期通知Java虚拟机可以安全地在这个类上使用clone()方法。通过调用这个clone()方法可以得到一个对象的复制。由于Object类本身并不实现Cloneable接口，因此如果所考虑的类没有实现Cloneable接口时，调用clone()方法会抛出CloneNotSupportedException异常。

**克隆满足的条件**

clone()方法将对象复制了一份并返还给调用者。所谓“复制”的含义与clone()方法是怎么实现的。一般而言，clone()方法满足以下的描述：

1. 对任何的对象x，都有：x.clone()!=x。换言之，克隆对象与原对象不是同一个对象。
2. 对任何的对象x，都有：x.clone().getClass() == x.getClass()，换言之，克隆对象与原对象的类型一样。
3. 如果对象x的equals()方法定义其恰当的话，那么x.clone().equals(x)应当成立的。

在JAVA语言的API中，凡是提供了clone()方法的类，都满足上面的这些条件。JAVA语言的设计师在设计自己的clone()方法时，也应当遵守着三个条件。一般来说，上面的三个条件中的前两个是必需的，而第三个是可选的。

**浅克隆和深克隆**

无论你是自己实现克隆方法，还是采用Java提供的克隆方法，都存在一个浅度克隆和深度克隆的问题。

**浅度克隆:**只负责克隆按值传递的数据（比如基本数据类型、String类型），而不复制它所引用的对象，换言之，所有的对其他对象的引用都仍然指向原来的对象。

**深度克隆:**除了浅度克隆要克隆的值外，还负责克隆引用类型的数据。那些引用其他对象的变量将指向被复制过的新对象，而不再是原有的那些被引用的对象。换言之，深度克隆把要复制的对象所引用的对象都复制了一遍，而这种对被引用到的对象的复制叫做间接复制。

深度克隆要深入到多少层，是一个不易确定的问题。在决定以深度克隆的方式复制一个对象的时候，必须决定对间接复制的对象时采取浅度克隆还是继续采用深度克隆。因此，在采取深度克隆时，需要决定多深才算深。此外，在深度克隆的过程中，很可能会出现循环引用的问题，必须小心处理。

**利用序列化实现深度克隆**

把对象写到流里的过程是序列化(Serialization)过程；而把对象从流中读出来的过程则叫反序列化(Deserialization)过程。应当指出的是，写到流里的是对象的一个拷贝，而原对象仍然存在于JVM里面。

在Java语言里深度克隆一个对象，常常可以先使对象实现Serializable接口，然后把对象（实际上只是对象的拷贝）写到一个流里（序列化），再从流里读回来（反序列化），便可以重建对象。

**public** Object deepClone() **throws** IOException, ClassNotFoundException{

//将对象写到流里

ByteArrayOutputStream bos = **new** ByteArrayOutputStream();

ObjectOutputStream oos = **new** ObjectOutputStream(bos);

oos.writeObject(**this**);

//从流里读回来

ByteArrayInputStream bis = **new** ByteArrayInputStream(bos.toByteArray());

ObjectInputStream ois = **new** ObjectInputStream(bis);

**return** ois.readObject();

}

这样做的前提就是对象以及对象内部所有引用到的对象都是可序列化的，否则，就需要仔细考察那些不可序列化的对象可否设成transient，从而将之排除在复制过程之外。

浅度克隆显然比深度克隆更容易实现，因为Java语言的所有类都会继承一个clone()方法，而这个clone()方法所做的正式浅度克隆。

有一些对象，比如线程(Thread)对象或Socket对象，是不能简单复制或共享的。不管是使用浅度克隆还是深度克隆，只要涉及这样的间接对象，就必须把间接对象设成transient而不予复制；或者由程序自行创建出相当的同种对象，权且当做复制件使用。

## 构建型

### 适配器模式

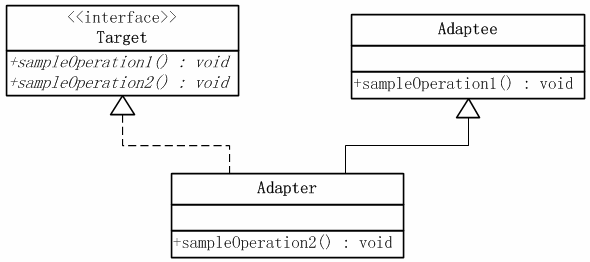
适配器模式把一个类的接口变换成客户端所期待的另一种接口，从而使原本因接口不匹配而无法在一起工作的两个类能够在一起工作。

**适配器模式的结构**

适配器模式有类的适配器模式和对象的适配器模式两种不同的形式。

**类适配器模式**

类的适配器模式把适配的类的API转换成为目标类的API。



在上图中可以看出，Adaptee类并没有sampleOperation2()方法，而客户端则期待这个方法。为使客户端能够使用Adaptee类，提供一个中间环节，即类Adapter，把Adaptee的API与Target类的API衔接起来。Adapter与Adaptee是继承关系，这决定了这个适配器模式是类的：

模式所涉及的角色有：

**目标(Target)角色：**这就是所期待得到的接口。注意：由于这里讨论的是类适配器模式，因此目标不可以是类。

**源(Adapee)角色：**现在需要适配的接口。

**适配器(Adaper)角色：**适配器类是本模式的核心。适配器把源接口转换成目标接口。显然，这一角色不可以是接口，而必须是具体类。

**public** **interface** Target {

//这是源类Adaptee也有的方法

**public** **void** sampleOperation1();

//这是源类Adaptee没有的方法

**public** **void** sampleOperation2();

}

上面给出的是目标角色的源代码，这个角色是以一个JAVA接口的形式实现的。可以看出，这个接口声明了两个方法：sampleOperation1()和sampleOperation2()。而源角色Adaptee是一个具体类，它有一个sampleOperation1()方法，但是没有sampleOperation2()方法。

**public** **class** Adaptee {

**public** **void** sampleOperation1(){}

}

适配器角色Adapter扩展了Adaptee,同时又实现了目标(Target)接口。由于Adaptee没有提供sampleOperation2()方法，而目标接口又要求这个方法，因此适配器角色Adapter实现了这个方法。

**public** **class** Adapter **extends** Adaptee **implements** Target {

/\*\*

\* 由于源类Adaptee没有方法sampleOperation2()

\* 因此适配器补充上这个方法

\*/

@Override

**public** **void** sampleOperation2() {

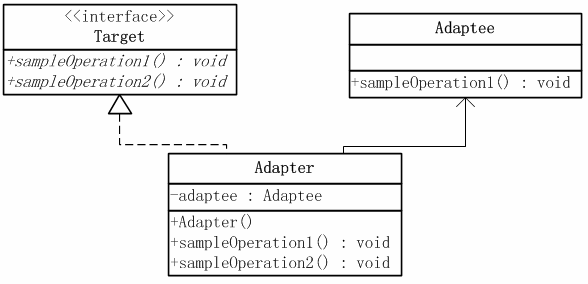
//写相关的代码

}

}

**对象适配器模式**

与类的适配器模式一样，对象的适配器模式把被适配的类的API转换成为目标类的API，与类的适配器模式不同的是，对象的适配器模式不是使用继承关系连接到Adaptee类，而是使用委派关系连接到Adaptee类。



从上图可以看出，Adaptee类并没有sampleOperation2()方法，而客户端则期待这个方法。为使客户端能够使用Adaptee类，需要提供一个包装(Wrapper)类Adapter。这个包装类包装了一个Adaptee的实例，从而此包装类能够把Adaptee的API与Target类的API衔接起来。Adapter与Adaptee是委派关系，这决定了适配器模式是对象的。

**public** **interface** Target {

//这是源类Adaptee也有的方法

**public** **void** sampleOperation1();

//这是源类Adaptee没有的方法

**public** **void** sampleOperation2();

}

**public** **class** Adaptee {

**public** **void** sampleOperation1(){}

}

**public** **class** Adapter {

**private** Adaptee adaptee;

**public** Adapter(Adaptee adaptee){

**this**.adaptee = adaptee;

}

/\*\*

\* 源类Adaptee有方法sampleOperation1

\* 因此适配器类直接委派即可

\*/

**public** **void** sampleOperation1(){

**this**.adaptee.sampleOperation1();

}

/\*\*

\* 源类Adaptee没有方法sampleOperation2

\* 因此由适配器类需要补充此方法

\*/

**public** **void** sampleOperation2(){

//写相关的代码

}

}

**类适配器和对象适配器的权衡**

1. 类适配器使用对象继承的方式，是静态的定义方式；而对象适配器使用对象组合的方式，是动态组合的方式。
2. 对于类适配器，由于适配器直接继承了Adaptee，使得适配器不能和Adaptee的子类一起工作，因为继承是静态的关系，当适配器继承了Adaptee后，就不可能再去处理 Adaptee的子类了。

对于对象适配器，一个适配器可以把多种不同的源适配到同一个目标。换言之，同一个适配器可以把源类和它的子类都适配到目标接口。因为对象适配器采用的是对象组合的关系，只要对象类型正确，是不是子类都无所谓。

1. 对于类适配器，适配器可以重定义Adaptee的部分行为，相当于子类覆盖父类的部分实现方法。

对于对象适配器，要重定义Adaptee的行为比较困难，这种情况下，需要定义Adaptee的子类来实现重定义，然后让适配器组合子类。虽然重定义Adaptee的行为比较困难，但是想要增加一些新的行为则方便的很，而且新增加的行为可同时适用于所有的源。

1. 对于类适配器，仅仅引入了一个对象，并不需要额外的引用来间接得到Adaptee。

对于对象适配器，需要额外的引用来间接得到Adaptee。

建议尽量使用对象适配器的实现方式，多用合成/聚合、少用继承。当然，具体问题具体分析，根据需要来选用实现方式，最适合的才是最好的。

**适配器模式的优点**

1. 更好的复用性

系统需要使用现有的类，而此类的接口不符合系统的需要。那么通过适配器模式就可以让这些功能得到更好的复用。

1. 更好的扩展性

　　在实现适配器功能的时候，可以调用自己开发的功能，从而自然地扩展系统的功能。

**适配器模式的缺点**

1. 过多的使用适配器，会让系统非常零乱，不易整体进行把握。比如，明明看到调用的是A接口，其实内部被适配成了B接口的实现，一个系统如果太多出现这种情况，无异于一场灾难。因此如果不是很有必要，可以不使用适配器，而是直接对系统进行重构。

**缺省适配模式**

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。

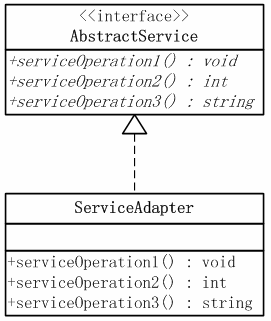
在很多情况下，必须让一个具体类实现某一个接口，但是这个类又用不到接口所规定的所有的方法。通常的处理方法是，这个具体类要实现所有的方法，那些有用的方法要有实现，那些没有用的方法也要有空的、平庸的实现。

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。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();

}

**public** **class** ServiceAdapter **implements** AbstractService{

@Override

**public** **void** serviceOperation1() {

}

@Override

**public** **int** serviceOperation2() {

**return** 0;

}

@Override

**public** String serviceOperation3() {

**return** **null**;

}

}

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。

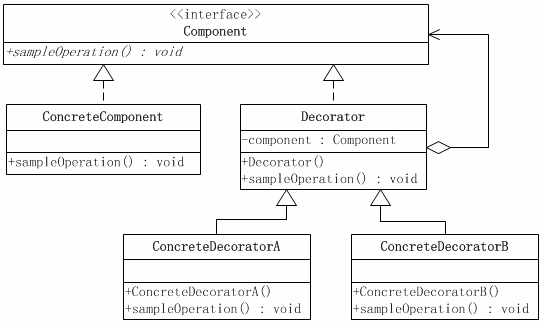
在任何时候，如果不准备实现一个接口的所有方法时，就可以使用“缺省适配模式”制造一个抽象类，给出所有方法的平庸的具体实现。这样，从这个抽象类再继承下去的子类就不必实现所有的方法了。

### 装饰者模式

装饰模式又名包装(Wrapper)模式。装饰模式以对客户端透明的方式扩展对象的功能，是继承关系的一个替代方案。

**装饰模式的结构**

装饰模式以对客户透明的方式动态地给一个对象附加上更多的责任。换言之，客户端并不会觉得对象在装饰前和装饰后有什么不同。装饰模式可以在不使用创造更多子类的情况下，将对象的功能加以扩展。



在装饰模式中的角色有：

**抽象构件(Component)角色：**给出一个抽象接口，以规范准备接收附加责任的对象。

**具体构件(ConcreteComponent)角色：**定义一个将要接收附加责任的类。

**装饰(Decorator)角色：**持有一个构件(Component)对象的实例，并定义一个与抽象构件接口一致的接口。

**具体装饰(ConcreteDecorator)角色：**负责给构件对象“贴上”附加的责任。

抽象构件角色:

**public** **interface** Component {

**public** **void** sampleOperation();

}

具体构件角色:

**public** **class** ConcreteComponent **implements** Component {

@Override

**public** **void** sampleOperation() {

// 写相关的业务代码

}

}

装饰角色:

**public** **class** Decorator **implements** Component{

**private** Component component;

**public** Decorator(Component component){

**this**.component = component;

}

@Override

**public** **void** sampleOperation() {

// 委派给构件

component.sampleOperation();

}

}

具体装饰角色:

**public** **class** ConcreteDecoratorA **extends** Decorator {

**public** ConcreteDecoratorA(Component component) {

**super**(component);

}

@Override

**public** **void** sampleOperation() {

**super**.sampleOperation();

// 写相关的业务代码

}

}

**public** **class** ConcreteDecoratorB **extends** Decorator {

**public** ConcreteDecoratorB(Component component) {

**super**(component);

}

@Override

**public** **void** sampleOperation() {

**super**.sampleOperation();

// 写相关的业务代码

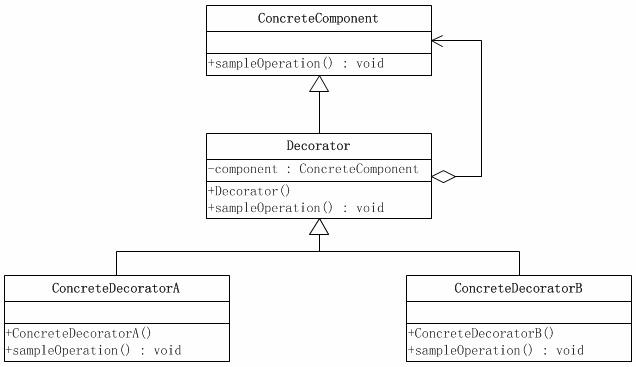
}

}

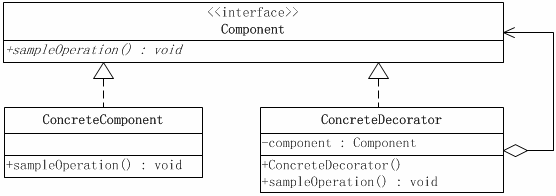
**装饰模式的简化**

大多数情况下，装饰模式的实现都要比上面给出的示意性例子要简单

如果只有一个ConcreteComponent类，那么可以考虑去掉抽象的Component类（接口），把Decorator作为一个ConcreteComponent子类。如下图所示：



如果只有一个ConcreteDecorator类，那么就没有必要建立一个单独的Decorator类，而可以把Decorator和ConcreteDecorator的责任合并成一个类。甚至在只有两个ConcreteDecorator类的情况下，都可以这样做。如下图所示：



**半透明的装饰模式**

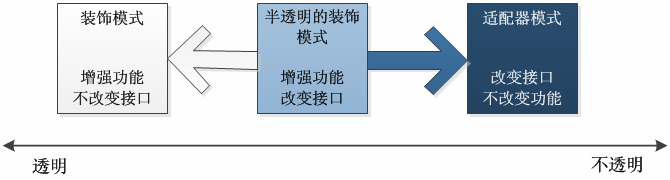
然而，纯粹的装饰模式很难找到。装饰模式的用意是在不改变接口的前提下，增强所考虑的类的性能。在增强性能的时候，往往需要建立新的公开的方法。这就导致了大多数的装饰模式的实现都是“半透明”的，而不是完全透明的。换言之，允许装饰模式改变接口，增加新的方法。这意味着客户端可以声明ConcreteDecorator类型的变量，从而可以调用ConcreteDecorator类中才有的方法：。

半透明的装饰模式是介于装饰模式和适配器模式之间的。适配器模式的用意是改变所考虑的类的接口，也可以通过改写一个或几个方法，或增加新的方法来增强或改变所考虑的类的功能。大多数的装饰模式实际上是半透明的装饰模式，这样的装饰模式也称做半装饰、半适配器模式。

装饰模式和适配器模式都是“包装模式(Wrapper Pattern)”，它们都是通过封装其他对象达到设计的目的的，但是它们的形态有很大区别。

理想的装饰模式在对被装饰对象进行功能增强的同时，要求具体构件角色、装饰角色的接口与抽象构件角色的接口完全一致。而适配器模式则不然，一般而言，适配器模式并不要求对源对象的功能进行增强，但是会改变源对象的接口，以便和目标接口相符合。

装饰模式有透明和半透明两种，这两种的区别就在于装饰角色的接口与抽象构件角色的接口是否完全一致。透明的装饰模式也就是理想的装饰模式，要求具体构件角色、装饰角色的接口与抽象构件角色的接口完全一致。相反，如果装饰角色的接口与抽象构件角色接口不一致，也就是说装饰角色的接口比抽象构件角色的接口宽的话，装饰角色实际上已经成了一个适配器角色，这种装饰模式也是可以接受的，称为“半透明”的装饰模式，如下图所示。



在适配器模式里面，适配器类的接口通常会与目标类的接口重叠，但往往并不完全相同。换言之，适配器类的接口会比被装饰的目标类接口宽。

显然，半透明的装饰模式实际上就是处于适配器模式与装饰模式之间的灰色地带。如果将装饰模式与适配器模式合并成为一个“包装模式”的话，那么半透明的装饰模式倒可以成为这种合并后的“包装模式”的代表。

**装饰模式的优点**

1. 装饰模式与继承关系的目的都是要扩展对象的功能，但是装饰模式可以提供比继承更多的灵活性。装饰模式允许系统动态决定“贴上”一个需要的“装饰”，或者除掉一个不需要的“装饰”。继承关系则不同，继承关系是静态的，它在系统运行前就决定了。
2. 通过使用不同的具体装饰类以及这些装饰类的排列组合，设计师可以创造出很多不同行为的组合。

**装饰模式的缺点**

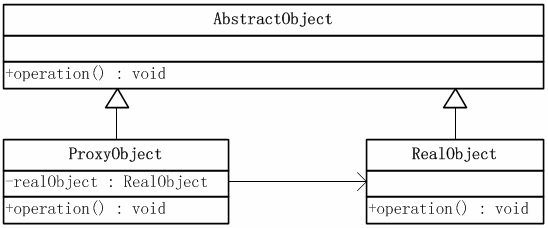
由于使用装饰模式，可以比使用继承关系需要较少数目的类。使用较少的类，当然使设计比较易于进行。但是，在另一方面，使用装饰模式会产生比使用继承关系更多的对象。更多的对象会使得查错变得困难，特别是这些对象看上去都很相像。

### 代理模式

代理模式给某一个对象提供一个代理对象，并由代理对象控制对原对象的引用。

**代理模式的结构**

所谓代理，就是一个人或者机构代表另一个人或者机构采取行动。在一些情况下，一个客户不想或者不能够直接引用一个对象，而代理对象可以在客户端和目标对象之间起到中介的作用。



在代理模式中的角色：

**抽象对象角色：**声明了目标对象和代理对象的共同接口，这样一来在任何可以使用目标对象的地方都可以使用代理对象。

**目标对象角色：**定义了代理对象所代表的目标对象。

**代理对象角色：**代理对象内部含有目标对象的引用，从而可以在任何时候操作目标对象；代理对象提供一个与目标对象相同的接口，以便可以在任何时候替代目标对象。代理对象通常在客户端调用传递给目标对象之前或之后，执行某个操作，而不是单纯地将调用传递给目标对象。

抽象对象角色：

**public** **abstract** **class** AbstractObject {

//操作

**public** **abstract** **void** operation();

}

目标对象角色：

**public** **class** RealObject **extends** AbstractObject {

@Override

**public** **void** operation() {

//一些操作

System.***out***.println("一些操作");

}

}

代理对象角色：

**public** **class** ProxyObject **extends** AbstractObject{

RealObject realObject = **new** RealObject();

@Override

**public** **void** operation() {

//调用目标对象之前可以做相关操作

System.***out***.println("before");

realObject.operation();

//调用目标对象之后可以做相关操作

System.***out***.println("after");

}

}

客户端：

**public** **class** Client {

**public** **static** **void** main(String[] args) {

// **TODO** Auto-generated method stub

AbstractObject obj = **new** ProxyObject();

obj.operation();

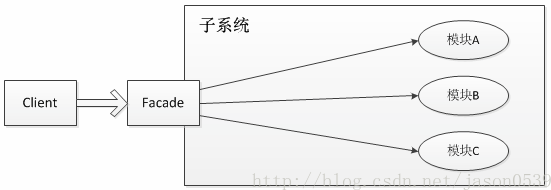
}

}

从上面的例子可以看出代理对象将客户端的调用委派给目标对象，在调用目标对象的方法之前跟之后都可以执行特定的操作。

### 外观模式

外观模式（Facade）,他隐藏了系统的复杂性，并向客户端提供了一个可以访问系统的接口。这种类型的设计模式属于结构性模式。为子系统中的一组接口提供了一个统一的访问接口，这个接口使得子系统更容易被访问或者使用。



简单来说，该模式就是把一些复杂的流程封装成一个接口供给外部用户更简单的使用。这个模式中，设计到3个角色。

**门面角色：**外观模式的核心。它被客户角色调用，它熟悉子系统的功能。内部根据客户角色的需求预定了几种功能的组合。

**子系统角色:**实现了子系统的功能。它对客户角色和Facade时未知的。它内部可以有系统内的相互交互，也可以由供外界调用的接口。

**客户角色:**通过调用Facede来完成要实现的功能。

子系统角色中的类：

**public** **class** ModuleA {

//示意方法

**public** **void** testA(){

System.***out***.println("调用ModuleA中的testA方法");

}

}

**public** **class** ModuleB {

//示意方法

**public** **void** testB(){

System.***out***.println("调用ModuleB中的testB方法");

}

}

**public** **class** ModuleC {

//示意方法

**public** **void** testC(){

System.***out***.println("调用ModuleC中的testC方法");

}

}

门面角色类：

**public** **class** Facade {

//示意方法，满足客户端需要的功能

**public** **void** test(){

ModuleA a = **new** ModuleA();

a.testA();

ModuleB b = **new** ModuleB();

b.testB();

ModuleC c = **new** ModuleC();

c.testC();

}

}

客户端角色类：

**public** **class** Client {

**public** **static** **void** main(String[] args) {

Facade facade = **new** Facade();

facade.test();

}

}

Facade类其实相当于A、B、C模块的外观界面，有了这个Facade类，那么客户端就不需要亲自调用子系统中的A、B、C模块了，也不需要知道系统内部的实现细节，甚至都不需要知道A、B、C模块的存在，客户端只需要跟Facade类交互就好了，从而更好地实现了客户端和子系统中A、B、C模块的解耦，让客户端更容易地使用系统。

**门面模式的优点**

1. 松散耦合

门面模式松散了客户端与子系统的耦合关系，让子系统内部的模块能更容易扩展和维护。

1. 简单易用

门面模式让子系统更加易用，客户端不再需要了解子系统内部的实现，也不需要跟众多子系统内部的模块进行交互，只需要跟门面类交互就可以了。

1. 更好的划分访问层次

通过合理使用Facade，可以帮助我们更好地划分访问的层次。有些方法是对系统外的，有些方法是系统内部使用的。把需要暴露给外部的功能集中到门面中，这样既方便客户端使用，也很好地隐藏了内部的细节。

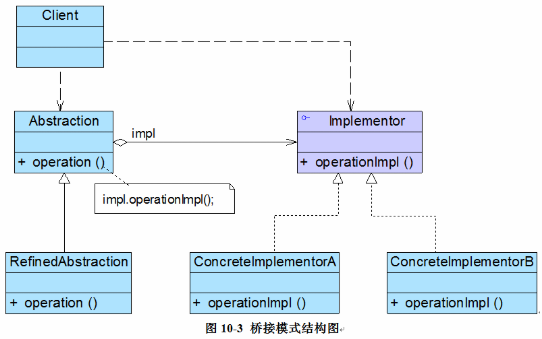
### 桥接模式

将抽象部分与它的实现部分分离，使它们都可以独立地变化。它是一种对象结构型模式，又称为柄体(Handle and Body)模式或接口(Interface)模式。

**桥接模式的结构**

桥接模式用一种巧妙的方式处理多层继承存在的问题，用抽象关联取代了传统的多层继承，将类之间的静态继承关系转换为动态的对象组合关系，使得系统更加灵活，并易于扩展，同时有效控制了系统中类的个数。

桥接模式的结构与其名称一样，存在一条连接两个继承等级结构的桥，桥接模式结构如图10-3所示：



在桥接模式结构图中包含如下几个角色：

**Abstraction（抽象类）：**用于定义抽象类的接口，它一般是抽象类而不是接口，其中定义了一个Implementor（实现类接口）类型的对象并可以维护该对象，它与Implementor之间具有关联关系，它既可以包含抽象业务方法，也可以包含具体业务方法。

**RefinedAbstraction（扩充抽象类）：**扩充由Abstraction定义的接口，通常情况下它不再是抽象类而是具体类，它实现了在Abstraction中声明的抽象业务方法，在RefinedAbstraction中可以调用在Implementor中定义的业务方法。

**Implementor（实现类接口）：**定义实现类的接口，这个接口不一定要与Abstraction的接口完全一致，事实上这两个接口可以完全不同，一般而言，Implementor接口仅提供基本操作，而Abstraction定义的接口可能会做更多更复杂的操作。Implementor接口对这些基本操作进行了声明，而具体实现交给其子类。通过关联关系，在Abstraction中不仅拥有自己的方法，还可以调用到Implementor中定义的方法，使用关联关系来替代继承关系。

**ConcreteImplementor（具体实现类）：**具体实现Implementor接口，在不同的ConcreteImplementor中提供基本操作的不同实现，在程序运行时，ConcreteImplementor对象将替换其父类对象，提供给抽象类具体的业务操作方法。

对于“实现部分”维度，典型的实现类接口及具体实现类代码如下所示：

**public interface** Implementor {

**public** **void** operationImpl();

}

**public** **class** ImplementorImpl **extends** Implementor {

@Override

**public** **void** operationImpl() {

//具体实现代码

}

}

对于另一“抽象部分”维度而言，其典型的抽象类代码如下所示：

**public** **abstract** **class** Abstraction {

**protected** Implementor impl; //定义实现类接口对象

**public** **void** setImpl(Implementor impl) {

**this**.impl=impl;

}

**public** **abstract** **void** operation(); //声明抽象业务方法

}

在抽象类Abstraction中定义了一个实现类接口类型的成员对象impl，再通过注入的方式给该对象赋值，一般将该对象的可见性定义为protected，以便在其子类中访问Implementor的方法，其子类一般称为扩充抽象类或细化抽象类(RefinedAbstraction)，典型的RefinedAbstraction类代码如下所示：

**public** **class** RefinedAbstraction **extends** Abstraction {

**public** **void** operation() {

//业务代码

impl.operationImpl(); //调用实现类的方法

//业务代码

}

}

对于客户端而言，可以针对两个维度的抽象层编程，在程序运行时再动态确定两个维度的子类，动态组合对象，将两个独立变化的维度完全解耦，以便能够灵活地扩充任一维度而对另一维度不造成任何影响。

桥接模式是设计Java虚拟机和实现JDBC等驱动程序的核心模式之一，应用较为广泛。在软件开发中如果一个类或一个系统有多个变化维度时，都可以尝试使用桥接模式对其进行设计。桥接模式为多维度变化的系统提供了一套完整的解决方案，并且降低了系统的复杂度。

**桥接模式优点**

1. 分离抽象接口及其实现部分。桥接模式使用“对象间的关联关系”解耦了抽象和实现之间固有的绑定关系，使得抽象和实现可以沿着各自的维度来变化。所谓抽象和实现沿着各自维度的变化，也就是说抽象和实现不再在同一个继承层次结构中，而是“子类化”它们，使它们各自都具有自己的子类，以便任何组合子类，从而获得多维度组合对象。
2. 在很多情况下，桥接模式可以取代多层继承方案，多层继承方案违背了“单一职责原则”，复用性较差，且类的个数非常多，桥接模式是比多层继承方案更好的解决方法，它极大减少了子类的个数。
3. 桥接模式提高了系统的可扩展性，在两个变化维度中任意扩展一个维度，都不需要修改原有系统，符合“开闭原则”。

**桥接模式缺点**

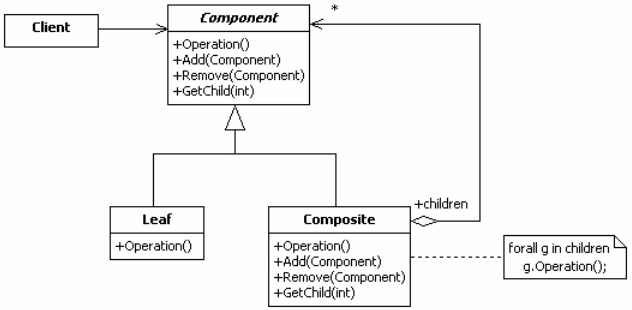
1. 桥接模式的使用会增加系统的理解与设计难度，由于关联关系建立在抽象层，要求开发者一开始就针对抽象层进行设计与编程。
2. 桥接模式要求正确识别出系统中两个独立变化的维度，因此其使用范围具有一定的局限性，如何正确识别两个独立维度也需要一定的经验积累。

### 组合模式

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。

组合模式，就是在一个对象中包含其他对象，这些被包含的对象可能是终点对象（不再包含别的对象），也有可能是非终点对象（其内部还包含其他对象，或叫组对象），我们将对象称为节点，即一个根节点包含许多子节点，这些子节点有的不再包含子节点，而有的仍然包含子节点，以此类推。很明显，这是树形结构，终结点叫叶子节点，非终节点（组节点）叫树枝节点，第一个节点叫根节点。同时也类似于文件目录的结构形式：文件可称之为终节点，目录可称之为非终节点（组节点）。

**组合模式结构**



**Component :** 组合中的对象声明接口，在适当的情况下，实现所有类共有接口的默认行为。声明一个接口用于访问和管理 Component 的子部件。

**public** **abstract** **class** Component {

**protected** String name;

**public** Component(String name) {

**this**.name = name;

}

**public** **abstract** **void** Add(Component c);

**public** **abstract** **void** Remove(Component c);

**public** **abstract** **void** Display(**int** depth);

}

**Leaf :** 表示叶节点对象。叶子节点没有子节点。

**public** **class** Leaf **extends** Component {

**public** Leaf(String name) {

**super**(name);

}

@Override

**public** **void** Add(Component c) {

System.***out***.println("Can not add to a leaf");

}

@Override

**public** **void** Remove(Component c) {

System.***out***.println("Can not remove from a leaf");

}

@Override

**public** **void** Display(**int** depth) {

String temp = "";

**for** (**int** i = 0; i < depth; i++)

temp += '-';

System.***out***.println(temp + name);

}

}

**Composite :** 定义枝节点行为，用来存储子部件，在 Component 接口中实现与子部件相关的操作。例如 Add 和 Remove。

**public** **class** Composite **extends** Component {

**private** List<Component> children = **new** ArrayList<Component>();

**public** Composite(String name) {

**super**(name);

}

@Override

**public** **void** Add(Component c) {

children.add(c);

}

@Override

**public** **void** Remove(Component c) {

children.remove(c);

}

@Override

**public** **void** Display(**int** depth) {

String temp = "";

**for** (**int** i = 0; i < depth; i++)

temp += '-';

System.***out***.println(temp + name);

**for** (Component c : children) {

c.Display(depth + 2);

}

}

}

**Client :** 通过 Component 接口操作结构中的对象。

**public** **class** CompositePattern {

**public** **static** **void** main(String[] args) {

Composite root = **new** Composite("root");

root.Add(**new** Leaf("Leaf A"));

root.Add(**new** Leaf("Leaf B"));

Composite compX = **new** Composite("Composite X");

compX.Add(**new** Leaf("Leaf XA"));

compX.Add(**new** Leaf("Leaf XB"));

root.Add(compX);

Composite compXY = **new** Composite("Composite XY");

compXY.Add(**new** Leaf("Leaf XYA"));

compXY.Add(**new** Leaf("Leaf XYB"));

compX.Add(compXY);

root.Display(1);

}

}

**组合模式的优点**

1. 可以清楚地定义分层次的复杂类型，表示对象的全部层次或者部分层次  ，它让客户端忽略了层次的差异，方便对整个层次经行控制。
2. 客户端可以一致的使用一个组合模式或对单个对象，不必关心处理的是单个对象还是整个组合结构，简化了客户端的代码。
3. 在组合模式种增加新的容器构件和叶子构件都很方便，无需对现有类库进行任何修改，符合开闭原则。
4. 为树形结构的面向对象实现提供了一种灵活的解决方案，通过叶子对象和容器对象的递归组合可以形成复杂的树形机构，但对树形结构的控制却很简单。

**组合模式的缺点**

1. 在增加新的构件时就比较难。而且难以限定，有时候希望在一个容器中能有某些特定的对象，例如在某个文件夹只能有image或者gif等。这个就比较难以实现。

**组合模式适用场景**

1. 在具有整体和部分的层次结构中希望通过一种忽略整体与个体之间差异的，客户端一致对待的情况。
2. 在一个使用面向对象语言开发的系统中需要处理一个树形结构的。
3. 在一个系统中能分离出叶子和容器的，而且他们的类型还固定不变，需要增加一些新的类型

### 享元模式

享元模式是对象的结构模式。享元模式以共享的方式高效地支持大量的细粒度对象。

**享元模式的结构**

享元模式采用一个共享来避免大量拥有相同内容对象的开销。这种开销最常见、最直观的就是内存的损耗。享元对象能做到共享的关键是区分内蕴状态(Internal State)和外蕴状态(External State)。

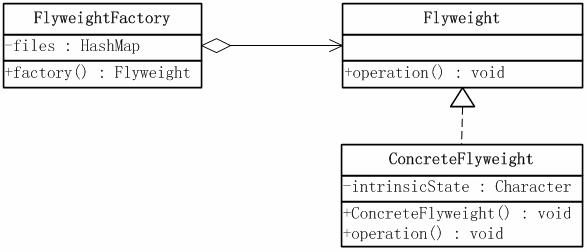
一个内蕴状态是存储在享元对象内部的，并且是不会随环境的改变而有所不同。因此，一个享元可以具有内蕴状态并可以共享。

一个外蕴状态是随环境的改变而改变的、不可以共享的。享元对象的外蕴状态必须由客户端保存，并在享元对象被创建之后，在需要使用的时候再传入到享元对象内部。外蕴状态不可以影响享元对象的内蕴状态，它们是相互独立的。

享元模式可以分成单纯享元模式和复合享元模式两种形式。

**单纯享元模式**

在单纯的享元模式中，所有的享元对象都是可以共享的。



单纯享元模式所涉及到的角色如下：

**抽象享元(Flyweight)角色 ：**给出一个抽象接口，以规定出所有具体享元角色需要实现的方法。

**具体享元(ConcreteFlyweight)角色：**实现抽象享元角色所规定出的接口。如果有内蕴状态的话，必须负责为内蕴状态提供存储空间。

**享元工厂(FlyweightFactory)角色 ：**本角色负责创建和管理享元角色。本角色必须保证享元对象可以被系统适当地共享。当一个客户端对象调用一个享元对象的时候，享元工厂角色会检查系统中是否已经有一个符合要求的享元对象。如果已经有了，享元工厂角色就应当提供这个已有的享元对象；如果系统中没有一个适当的享元对象的话，享元工厂角色就应当创建一个合适的享元对象。

抽象享元角色类：

**public** **interface** Flyweight {

//一个示意性方法，参数state是外蕴状态

**public** **void** operation(String state);

}

具体享元角色类ConcreteFlyweight有一个内蕴状态，在本例中一个Character类型的intrinsicState属性代表，它的值应当在享元对象被创建时赋予。所有的内蕴状态在对象创建之后，就不会再改变了。

如果一个享元对象有外蕴状态的话，所有的外部状态都必须存储在客户端，在使用享元对象时，再由客户端传入享元对象。这里只有一个外蕴状态，operation()方法的参数state就是由外部传入的外蕴状态。

**public** **class** ConcreteFlyweight **implements** Flyweight {

**private** Character intrinsicState = **null**;

/\*\*

\* 构造函数，内蕴状态作为参数传入

\* **@param** state

\*/

**public** ConcreteFlyweight(Character state){

**this**.intrinsicState = state;

}

/\*\*

\* 外蕴状态作为参数传入方法中，改变方法的行为，

\* 但是并不改变对象的内蕴状态。

\*/

@Override

**public** **void** operation(String state) {

System.***out***.println("Intrinsic State = " + **this**.intrinsicState);

System.***out***.println("Extrinsic State = " + state);

}

}

享元工厂角色类，必须指出的是，客户端不可以直接将具体享元类实例化，而必须通过一个工厂对象，利用一个factory()方法得到享元对象。一般而言，享元工厂对象在整个系统中只有一个，因此也可以使用单例模式。

当客户端需要单纯享元对象的时候，需要调用享元工厂的factory()方法，并传入所需的单纯享元对象的内蕴状态，由工厂方法产生所需要的享元对象。

**public** **class** FlyweightFactory {

**private** Map<Character,Flyweight> files = **new** HashMap<Character,Flyweight>();

**public** Flyweight factory(Character state){

//先从缓存中查找对象

Flyweight fly = files.get(state);

**if**(fly == **null**){

//如果对象不存在则创建一个新的Flyweight对象

fly = **new** ConcreteFlyweight(state);

//把这个新的Flyweight对象添加到缓存中

files.put(state, fly);

}

**return** fly;

}

}

客户端类

**public** **class** Client {

**public** **static** **void** main(String[] args) {

FlyweightFactory factory = **new** FlyweightFactory();

Flyweight fly = factory.factory(**new** Character('a'));

fly.operation("First Call");

fly = factory.factory(**new** Character('b'));

fly.operation("Second Call");

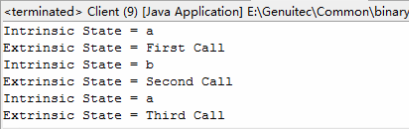
fly = factory.factory(**new** Character('a'));

fly.operation("Third Call");

}

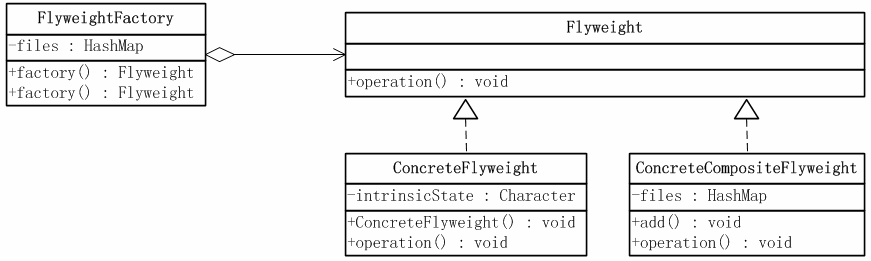
}

虽然客户端申请了三个享元对象，但是实际创建的享元对象只有两个，这就是共享的含义。运行结果如下：



**复合享元模式**

在单纯享元模式中，所有的享元对象都是单纯享元对象，也就是说都是可以直接共享的。还有一种较为复杂的情况，将一些单纯享元使用合成模式加以复合，形成复合享元对象。这样的复合享元对象本身不能共享，但是它们可以分解成单纯享元对象，而后者则可以共享。



复合享元角色所涉及到的角色如下：

**抽象享元(Flyweight)角色 ：**给出一个抽象接口，以规定出所有具体享元角色需要实现的方法。

**具体享元(ConcreteFlyweight)角色：**实现抽象享元角色所规定出的接口。如果有内蕴状态的话，必须负责为内蕴状态提供存储空间。

**复合享元(ConcreteCompositeFlyweight)角色 ：**复合享元角色所代表的对象是不可以共享的，但是一个复合享元对象可以分解成为多个本身是单纯享元对象的组合。复合享元角色又称作不可共享的享元对象。

**享元工厂(FlyweightFactory)角色 ：**本角 色负责创建和管理享元角色。本角色必须保证享元对象可以被系统适当地共享。当一个客户端对象调用一个享元对象的时候，享元工厂角色会检查系统中是否已经有 一个符合要求的享元对象。如果已经有了，享元工厂角色就应当提供这个已有的享元对象；如果系统中没有一个适当的享元对象的话，享元工厂角色就应当创建一个 合适的享元对象。

抽象享元角色类:

**public** **interface** Flyweight {

//一个示意性方法，参数state是外蕴状态

**public** **void** operation(String state);

}

具体享元角色类:

**public** **class** ConcreteFlyweight **implements** Flyweight {

**private** Character intrinsicState = **null**;

/\*\*

\* 构造函数，内蕴状态作为参数传入

\* **@param** state

\*/

**public** ConcreteFlyweight(Character state){

**this**.intrinsicState = state;

}

/\*\*

\* 外蕴状态作为参数传入方法中，改变方法的行为，

\* 但是并不改变对象的内蕴状态。

\*/

@Override

**public** **void** operation(String state) {

System.***out***.println("Intrinsic State = " + **this**.intrinsicState);

System.***out***.println("Extrinsic State = " + state);

}

}

复合享元对象是由单纯享元对象通过复合而成的，因此它提供了add()这样的聚集管理方法。由于一个复合享元对象具有不同的聚集元素，这些聚集元素在复合享元对象被创建之后加入，这本身就意味着复合享元对象的状态是会改变的，因此复合享元对象是不能共享的。

复合享元角色实现了抽象享元角色所规定的接口，也就是operation()方法，这个方法有一个参数，代表复合享元对象的外蕴状态。一个复合享元对象的所有单纯享元对象元素的外蕴状态都是与复合享元对象的外蕴状态相等的；而一个复合享元对象所含有的单纯享元对象的内蕴状态一般是不相等的，不然就没有使用价值了。

**public** **class** ConcreteCompositeFlyweight **implements** Flyweight {

**private** Map<Character,Flyweight> files = **new** HashMap<Character,Flyweight>();

/\*\*

\* 增加一个新的单纯享元对象到聚集中

\*/

**public** **void** add(Character key , Flyweight fly){

files.put(key,fly);

}

/\*\*

\* 外蕴状态作为参数传入到方法中

\*/

@Override

**public** **void** operation(String state) {

Flyweight fly = **null**;

**for**(Object o : files.keySet()){

fly = files.get(o);

fly.operation(state);

}

}

}

享元工厂角色提供两种不同的方法，一种用于提供单纯享元对象，另一种用于提供复合享元对象。

**public** **class** FlyweightFactory {

**private** Map<Character,Flyweight> files = **new** HashMap<Character,Flyweight>();

/\*\*

\* 复合享元工厂方法

\*/

**public** Flyweight factory(List<Character> compositeState){

ConcreteCompositeFlyweight compositeFly = **new** ConcreteCompositeFlyweight();

**for**(Character state : compositeState){

compositeFly.add(state,**this**.factory(state));

}

**return** compositeFly;

}

/\*\*

\* 单纯享元工厂方法

\*/

**public** Flyweight factory(Character state){

//先从缓存中查找对象

Flyweight fly = files.get(state);

**if**(fly == **null**){

//如果对象不存在则创建一个新的Flyweight对象

fly = **new** ConcreteFlyweight(state);

//把这个新的Flyweight对象添加到缓存中

files.put(state, fly);

}

**return** fly;

}

}

客户端角色

**public** **class** Client {

**public** **static** **void** main(String[] args) {

List<Character> compositeState = **new** ArrayList<Character>();

compositeState.add('a');

compositeState.add('b');

compositeState.add('c');

compositeState.add('a');

compositeState.add('b');

FlyweightFactory flyFactory = **new** FlyweightFactory();

Flyweight compositeFly1 = flyFactory.factory(compositeState);

Flyweight compositeFly2 = flyFactory.factory(compositeState);

compositeFly1.operation("Composite Call");

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

System.***out***.println("复合享元模式是否可以共享对象：" + (compositeFly1 == compositeFly2));

Character state = 'a';

Flyweight fly1 = flyFactory.factory(state);

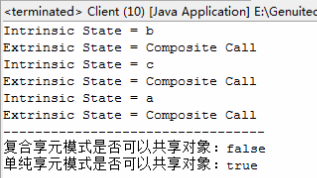
Flyweight fly2 = flyFactory.factory(state);

System.***out***.println("单纯享元模式是否可以共享对象：" + (fly1 == fly2));

}

}

运行结果如下：



**享元模式的优缺点**

享元模式的优点在于它大幅度地降低内存中对象的数量。但是，它做到这一点所付出的代价也是很高的：

1. 享元模式使得系统更加复杂。为了使对象可以共享，需要将一些状态外部化，这使得程序的逻辑复杂化。
2. 享元模式将享元对象的状态外部化，而读取外部状态使得运行时间稍微变长。

## 行为型

### 责任链模式

责任链模式是一种对象的行为模式。在责任链模式里，很多对象由每一个对象对其下家的引用而连接起来形成一条链。请求在这个链上传递，直到链上的某一个对象决定处理此请求。发出这个请求的客户端并不知道链上的哪一个对象最终处理这个请求，这使得系统可以在不影响客户端的情况下动态地重新组织和分配责任。

**责任链模式的结构**

责任链模式涉及到的角色如下所示：

**抽象处理者(Handler)角色：**定义出一个处理请求的接口。如果需要，接口可以定义 出一个方法以设定和返回对下家的引用。这个角色通常由一个Java抽象类或者Java接口实现。上图中Handler类的聚合关系给出了具体子类对下家的引用，抽象方法handleRequest()规范了子类处理请求的操作。

**具体处理者(ConcreteHandler)角色：**具体处理者接到请求后，可以选择将请求处理掉，或者将请求传给下家。由于具体处理者持有对下家的引用，因此，如果需要，具体处理者可以访问下家

抽象处理者角色:

**public** **abstract** **class** Handler {

/\*\*

\* 持有后继的责任对象

\*/

**protected** Handler successor;

/\*\*

\* 示意处理请求的方法，虽然这个示意方法是没有传入参数的

\* 但实际是可以传入参数的，根据具体需要来选择是否传递参数

\*/

**public** **abstract** **void** handleRequest();

/\*\*

\* 取值方法

\*/

**public** Handler getSuccessor() {

**return** successor;

}

/\*\*

\* 赋值方法，设置后继的责任对象

\*/

**public** **void** setSuccessor(Handler successor) {

**this**.successor = successor;

}

}

具体处理者角色:

**public** **class** ConcreteHandler **extends** Handler {

/\*\*

\* 处理方法，调用此方法处理请求

\*/

@Override

**public** **void** handleRequest() {

/\*\*

\* 判断是否有后继的责任对象

\* 如果有，就转发请求给后继的责任对象

\* 如果没有，则处理请求

\*/

**if**(getSuccessor() != **null**)

{

System.***out***.println("放过请求");

getSuccessor().handleRequest();

}**else**

{

System.***out***.println("处理请求");

}

}

}

客户端类:

**public** **class** Client {

**public** **static** **void** main(String[] args) {

//组装责任链

Handler handler1 = **new** ConcreteHandler();

Handler handler2 = **new** ConcreteHandler();

handler1.setSuccessor(handler2);

//提交请求

handler1.handleRequest();

}

}

可以看出，客户端创建了两个处理者对象，并指定第一个处理者对象的下家是第二个处理者对象，而第二个处理者对象没有下家。然后客户端将请求传递给第一个处理者对象。

由于本示例的传递逻辑非常简单：只要有下家，就传给下家处理；如果没有下家，就自行处理。因此，第一个处理者对象接到请求后，会将请求传递给第二个处理者对象。由于第二个处理者对象没有下家，于是自行处理请求。

**纯的与不纯的责任链模式**

一个纯的责任链模式要求一个具体的处理者对象只能在两个行为中选择一个：一是承担责任，而是把责任推给下家。不允许出现某一个具体处理者对象在承担了一部分责任后又 把责任向下传的情况。

在一个纯的责任链模式里面，一个请求必须被某一个处理者对象所接收；在一个不纯的责任链模式里面，一个请求可以最终不被任何接收端对象所接收。

纯的责任链模式的实际例子很难找到，一般看到的例子均是不纯的责任链模式的实现。有些人认为不纯的责任链根本不是责任链模式，这也许是有道理的。但是在实际的系统里，纯的责任链很难找到。如果坚持责任链不纯便不是责任链模式，那么责任链模式便不会有太大意义了。

### 命令模式

命令模式属于对象的行为模式。命令模式又称为行动(Action)模式或交易(Transaction)模式。

命令模式把一个请求或者操作封装到一个对象中。命令模式允许系统使用不同的请求把客户端参数化，对请求排队或者记录请求日志，可以提供命令的撤销和恢复功能。

**命令模式的结构**

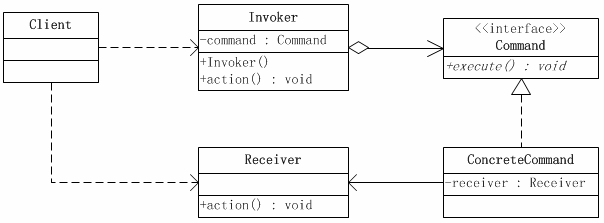
命令模式是对命令的封装。命令模式把发出命令的责任和执行命令的责任分割开，委派给不同的对象。

每一个命令都是一个操作：请求的一方发出请求要求执行一个操作；接收的一方收到请求，并执行操作。命令模式允许请求的一方和接收的一方独立开来，使得请求的一方不必知道接收请求的一方的接口，更不必知道请求是怎么被接收，以及操作是否被执行、何时被执行，以及是怎么被执行的。

命令允许请求的一方和接收请求的一方能够独立演化，从而具有以下的优点：

1. 命令模式使新的命令很容易地被加入到系统里。
2. 允许接收请求的一方决定是否要否决请求。
3. 能较容易地设计一个命令队列。
4. 可以容易地实现对请求的撤销和恢复。
5. 在需要的情况下，可以较容易地将命令记入日志。

下面以一个示意性的系统，说明命令模式的结构。



命令模式涉及到五个角色，它们分别是：

**客户端(Client)角色：**创建一个具体命令(ConcreteCommand)对象并确定其接收者。

**命令(Command)角色：**声明了一个给所有具体命令类的抽象接口。

**具体命令(ConcreteCommand)角色：**定义一个接收者和行为之间的弱耦合；实现execute()方法，负责调用接收者的相应操作。execute()方法通常叫做执行方法。

**请求者(Invoker)角色：**负责调用命令对象执行请求，相关的方法叫做行动方法。

**接收者(Receiver)角色：**负责具体实施和执行一个请求。任何一个类都可以成为接收者，实施和执行请求的方法叫做行动方法。

接收者角色类：

**public** **class** Receiver {

//真正执行命令相应的操作

**public** **void** action(){

System.***out***.println("执行操作");

}

}

抽象命令角色类：

**public** **interface** Command {

//执行方法

**void** execute();

}

具体命令角色类：

**public** **class** ConcreteCommand **implements** Command {

//持有相应的接收者对象

**private** Receiver receiver = **null**;

//构造方法

**public** ConcreteCommand(Receiver receiver){

**this**.receiver = receiver;

}

@Override

**public** **void** execute() {

//通常会转调接收者对象的相应方法，让接收者来真正执行功能

receiver.action();

}

}

请求者角色类：

**public** **class** Invoker {

//持有命令对象

**private** Command command = **null**;

//构造方法

**public** Invoker(Command command){

**this**.command = command;

}

//行动方法

**public** **void** action(){

command.execute();

}

}

客户端角色类：

**public** **class** Client {

**public** **static** **void** main(String[] args) {

//创建接收者

Receiver receiver = **new** Receiver();

//创建命令对象，设定它的接收者

Command command = **new** ConcreteCommand(receiver);

//创建请求者，把命令对象设置进去

Invoker invoker = **new** Invoker(command);

//执行方法

invoker.action();

}

}

**命令模式的优点**

1. 更松散的耦合

命令模式使得发起命令的对象——客户端，和具体实现命令的对象——接收者对象完全解耦，也就是说发起命令的对象完全不知道具体实现对象是谁，也不知道如何实现。

1. 更动态的控制

命令模式把请求封装起来，可以动态地对它进行参数化、队列化和日志化等操作，从而使得系统更灵活。

1. 很自然的复合命令

命令模式中的命令对象能够很容易地组合成复合命令，也就是宏命令，从而使系统操作更简单，功能更强大。

1. 更好的扩展性

由于发起命令的对象和具体的实现完全解耦，因此扩展新的命令就很容易，只需要实现新的命令对象，然后在装配的时候，把具体的实现对象设置到命令对象中，然后就可以使用这个命令对象，已有的实现完全不用变化。

### 解释器模式

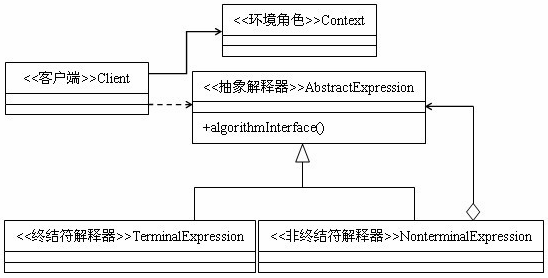
解释器模式是类的行为模式。给定一个语言之后，解释器模式可以定义出其文法的一种表示，并同时提供一个解释器。客户端可以使用这个解释器来解释这个语言中的句子。

给定一个语言(如由abcdef六个字符组成的字符串集合),定义它的文法的一种表示(S::=abA\*ef,A::=cd)并定义一个解释器,解释器使用该表示来解释语言中的句子.

其中的解释器类似一个翻译机

**解释器模式的结构**

下面就以一个示意性的系统为例，讨论解释器模式的结构。系统的结构图如下所示：



模式所涉及的角色如下所示：

**抽象表达式(Expression)角色：**声明一个所有的具体表达式角色都需要实现的抽象接口。这个接口主要是一个interpret()方法，称做解释操作。

**终结符表达式(Terminal Expression)角色：**实现了抽象表达式角色所要求的接口，主要是一个interpret()方法；文法中的每一个终结符都有一个具体终结表达式与之相对应。比如有一个简单的公式R=R1+R2，在里面R1和R2就是终结符，对应的解析R1和R2的解释器就是终结符表达式。

**非终结符表达式(Nonterminal Expression)角色：**文法中的每一条规则都需要一个具体的非终结符表达式，非终结符表达式一般是文法中的运算符或者其他关键字，比如公式R=R1+R2中，“+"就是非终结符，解析“+”的解释器就是一个非终结符表达式。

**环境(Context)角色：**这个角色的任务一般是用来存放文法中各个终结符所对应的具体值，比如R=R1+R2，我们给R1赋值100，给R2赋值200。这些信息需要存放到环境角色中，很多情况下我们使用Map来充当环境角色就足够了。

**实例**

如下我们通过对算术表达式的解释来看一个解释器模式的实现,

如表达式m+n+p中,如果我们使用解释器模式对该表达式进行解释,那么m,n,p代表的三个字母可以看成是终结符号,而+代表的运算符则可以看成是非终结符号

首先建立抽象解释器表示数学运算:

**public** **abstract** **class** ArithmeticExpression {

**public** **abstract** **int** interptet();

}

解释器中定义了interptet()方法,ArithmeticExpression有两个直接子类,NumExpression,和OperatorExpression

建立NumExpression,对数字进行解释:

**public** **class** NumExpression **extends** ArithmeticExpression {

**private** **int** num;

**public** NumExpression(**int** \_num) {

num = \_num;

}

@Override

**public** **int** interptet() {

**return** num;

}

}

建立OperatorExpression,对运算符进行解释:

**public** **abstract** **class** OperatorExpression **extends** ArithmeticExpression {

**protected** ArithmeticExpression mArithmeticExpression1, mArithmeticExpression2;

**public** OperatorExpression(ArithmeticExpression \_arithmeticExpression1,

ArithmeticExpression \_arithmeticExpression2) {

mArithmeticExpression1 = \_arithmeticExpression1;

mArithmeticExpression2 = \_arithmeticExpression2;

}

}

AdditionExpression,OperatorExpression的直接子类,加法运算解释器：

**public** **class** AdditionExpression **extends** OperatorExpression {

**public** AdditionExpression(ArithmeticExpression \_arithmeticExpression1,

ArithmeticExpression \_arithmeticExpression2) {

**super**(\_arithmeticExpression1, \_arithmeticExpression2);

}

@Override

**public** **int** interptet() {

**return** mArithmeticExpression1.interptet() + mArithmeticExpression2.interptet();

}

}

新增业务逻辑处理类：

**public** **class** Calculator {

**protected** Stack<ArithmeticExpression> mArithmeticExpressionStack = **new** Stack<>();

**public** Calculator(String expression) {

ArithmeticExpression arithmeticExpression1, arithmeticExpression2;

String[] elements = expression.split(" ");

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

**switch** (elements[i].charAt(0)) {

**case** '+':

arithmeticExpression1 = mArithmeticExpressionStack.pop();

arithmeticExpression2 = **new** NumExpression(Integer.*valueOf*(elements[++i]));

mArithmeticExpressionStack.push(**new** AdditionExpression(arithmeticExpression1

, arithmeticExpression2));

**break**;

**default**:

mArithmeticExpressionStack.push(**new** NumExpression(Integer.*valueOf*(elements[i])));

**break**;

}

}

}

**public** **int** calculate() {

**return** mArithmeticExpressionStack.pop().interptet();

}

}

客户端调用：

// 解释计算123+124+125+126的运算结果

Calculator calculator = **new** Calculator("123+124+125+126");

Log.d(TAG, "setBtnClick: -->" + calculator.calculate());

**解释器模式的优点**

1. 灵活性强,如上边的例子,当我们想对文法规则进行扩展延伸时,只需要增加相应的非终结符解释器,并在构建语法树的时候使用新增的解释器对象进行具体的解释即可.

**解释器模式的缺点**

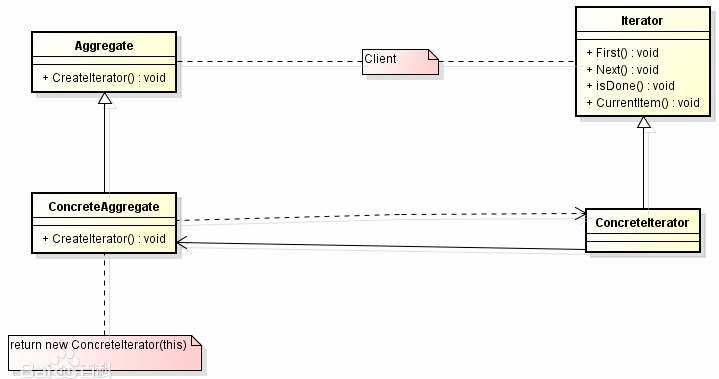
1. 因为每一条文法都可以对应至少一个解释器,会生成大量的类,导致后期维护困难,而且对应复杂的文法,构建语法树会显得异常繁琐.

### 迭代器模式

迭代器模式（Iterator），提供一种方法顺序访问一个聚合对象中的各种元素，而又不暴露该对象的内部表示。

**迭代器模式的角色构成**

1. **迭代器角色（Iterator）:**定义遍历元素所需要的方法，一般来说会有这么三个方法：取得下一个元素的方法next()，判断是否遍历结束的方法hasNext()），移出当前对象的方法remove(),
2. **具体迭代器角色（Concrete Iterator）：**实现迭代器接口中定义的方法，完成集合的迭代。
3. **容器角色(Aggregate):** 一般是一个接口，提供一个iterator()方法，例如java中的Collection接口，List接口，Set接口等
4. **具体容器角色（ConcreteAggregate）：**就是抽象容器的具体实现类，比如List接口的有序列表实现ArrayList，List接口的链表实现LinkList，Set接口的哈希列表的实现HashSet等。



定义迭代器角色(Iterator)：

**public** **interface** Iterator {

**public** **boolean** hasNext();

**public** Object next();

}

定义具体迭代器角色(Concrete Iterator)：

**public** **class** ConcreteIterator **implements** Iterator {

**private** List list = **null**;

**private** **int** index;

**public** ConcreteIterator(List list) {

**super**();

**this**.list = list;

}

@Override

**public** **boolean** hasNext() {

**if** (index >= list.getSize()) {

**return** **false**;

} **else** {

**return** **true**;

}

}

@Override

**public** Object next() {

Object object = list.get(index);

index++;

**return** object;

}

}

定义容器角色(Aggregate)：

//定义集合可以进行的操作

**public** **interface** List {

**public** **void** add(Object obj);

**public** Object get(**int** index);

**public** Iterator iterator();

**public** **int** getSize();

}

定义具体容器角色(ConcreteAggregate)：

**public** **class** ConcreteAggregate **implements** List{

**private** Object[] list;

**private** **int** size=0;

**private** **int** index=0;

**public** ConcreteAggregate(){

index=0;

size=0;

list=**new** Object[100];

}

@Override

**public** **void** add(Object obj) {

list[index++]=obj;

size++;

}

@Override

**public** Iterator iterator() {

**return** **new** ConcreteIterator(**this**);

}

@Override

**public** Object get(**int** index) {

**return** list[index];

}

@Override

**public** **int** getSize() {

**return** size;

}

}

代码测试：

**public** **class** IteratorTest {

**public** **static** **void** main(String[] args) {

List list=**new** ConcreteAggregate();

list.add("a");

list.add("b");

list.add("c");

list.add("d");

Iterator it=list.iterator();

**while**(it.hasNext()){

System.***out***.println(it.next());

}

}

}

**迭代器模式的优点**

1. 简化了遍历方式，对于对象集合的遍历，还是比较麻烦的，对于数组或者有序列表，我们尚可以通过游标来取得，但用户需要在对集合了解很清楚的前提下，自行遍历对象，但是对于hash表来说，用户遍历起来就比较麻烦了。而引入了迭代器方法后，用户用起来就简单的多了。
2. 可以提供多种遍历方式，比如说对有序列表，我们可以根据需要提供正序遍历，倒序遍历两种迭代器，用户用起来只需要得到我们实现好的迭代器，就可以方便的对集合进行遍历了。
3. 封装性良好，用户只需要得到迭代器就可以遍历，而对于遍历算法则不用去关心。

**迭代器模式的缺点**

1. 对于比较简单的遍历（像数组或者有序列表），使用迭代器方式遍历较为繁琐，大家可能都有感觉，像ArrayList，我们宁可愿意使用for循环和get方法来遍历集合。

### 中介模式

### 备忘录模式

### 观察者模式

### 状态模式

### 策略模式

### 模板方法模式

### 访问者模式