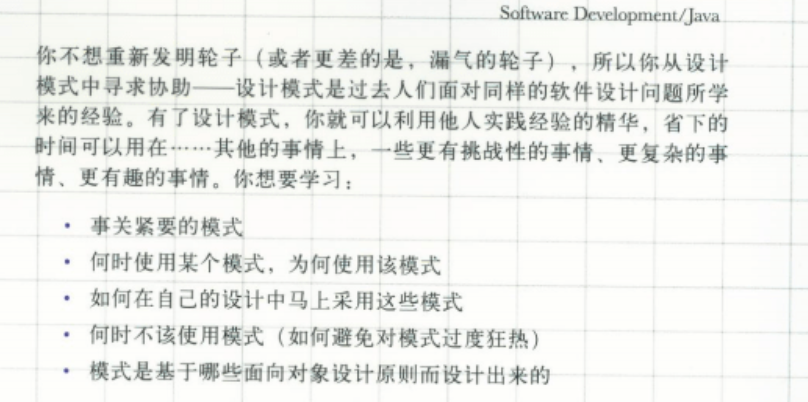
1. **什么是设计模式？**

**设计模式**是软件设计中常见问题的典型解决方案。 每个模式就像一张蓝图， 你可以通过对其进行定制来解决代码中的特定设计问题。

**1.1 我们为什么要学设计模式？**



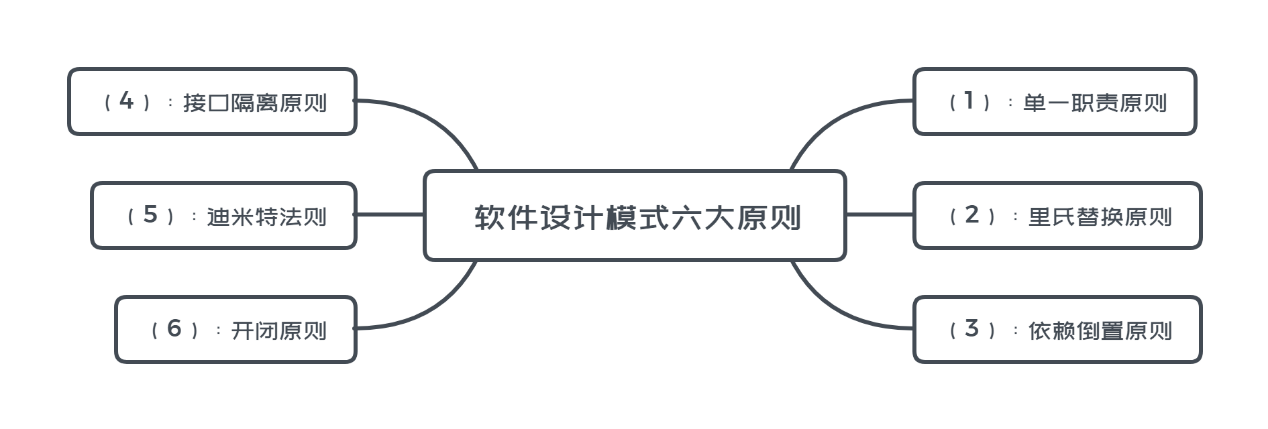
(在[《head first 设计模式》](http://download.csdn.net/detail/justgono/7321467)一书封面找到答案)

简而言之，为了利用他人实践经验，而不是自己造轮子。

**1.2 六大设计原则**

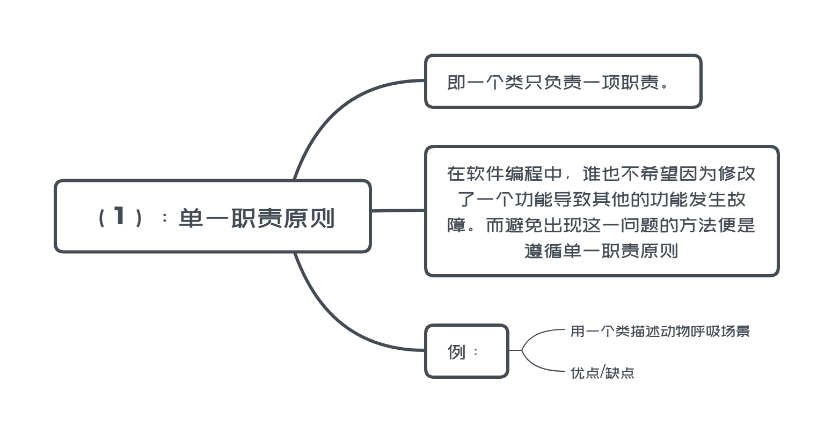
在讲设计模式前，不得不提到这六大原则(一般分为6类)，之后的设计模式也是基于设计原则的展开应用。

<https://www.cnblogs.com/zhanghengscnc/p/8299459.html>



首先分别阐释这六大设计模式：

**1.2.1 单一职责原则**



例：

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

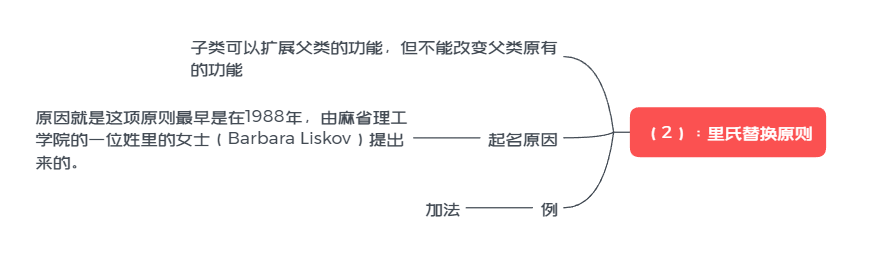
鱼呼吸水

可以看到遵循单一职责原的优点有：

* 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；
* 提高类的可读性，提高系统的可维护性；
* 变更引起的风险降低（在这里就是避免出现类似鱼呼吸空气的情况出现）

但这样修改花销是很大的，这也是遵循单一职责原则缺点。

**1.2.2 里氏替换原则**



这一原则主要是针对继承时使用的，该原则表明子类可以扩展父类的功能，但不能改变父类原有的功能。举例说明如果不遵守这一原则带来的风险是什么

例：

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

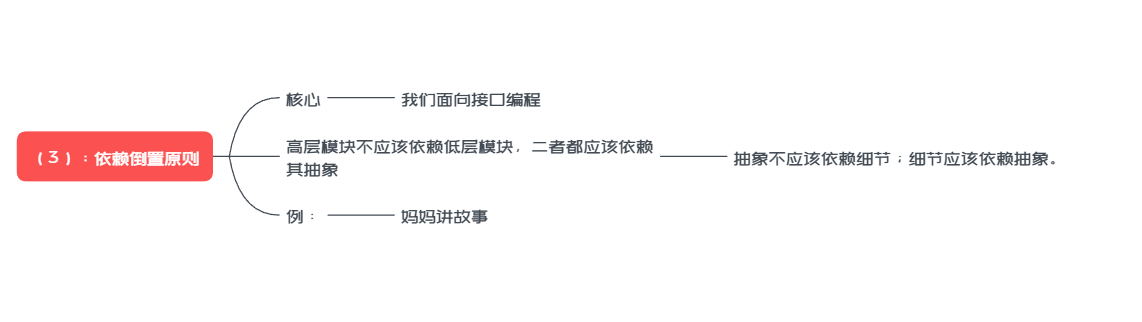
100-80=180

100+20+100=220

会发现这里减法结果出现错误，仔细一看，原来是类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。

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。

**1.2.3 依赖倒置原则**



个人认为这里有个重点概念需要掌握——**接口**，这也是java语言的核心之一

那么接口该怎样理解呢？

接口是一种协议。例如两个开发者，开发时间完全不一致，那么需要两个人的配合开发，则需要一个人讲接口写好，定义好其中所有的变量命名规范、函数定义规范。具体实现类的开发人员则只需要按照接口实现相应功能即可。

也可以简单理解为接口就是一种**特殊的抽象父类**

举个例子说明下：  
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());

}

}

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，narrate函数里传进来的参数是book 而不是Newpaper，如果要读newspaper 则需要修改mother的代码才能读。

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……";

}

}

可以看到这里的implement就是接口的实现，把IReader扩展成了具体的哪一种读物，Newspaper和Book，并且在具体的该类中实现其特有的方法(这里体现为不同的故事)

具体应用：

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());

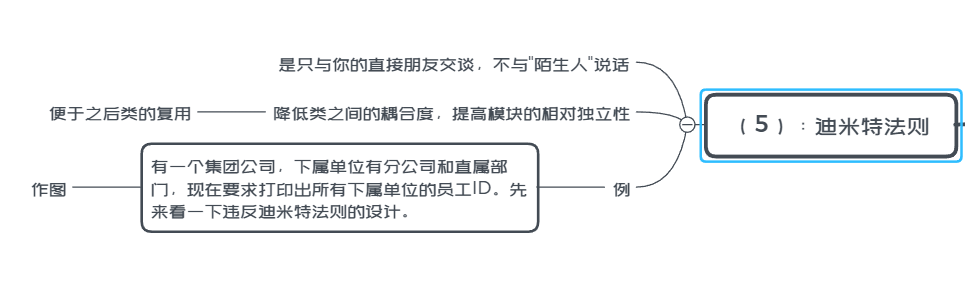
}

}

可以看到母亲的读物变的十分灵活。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。

**1.2.4 接口隔离原则**

即不要在一个接口里面放很多的方法，这样会显得这个类很臃肿。接口应该尽量细化，一个接口对应一个功能模块，同时接口里面的方法应该尽可能的少，使接口更加灵活轻便

**1.2.5 迪米特法则**

例：(直接作图阐释)

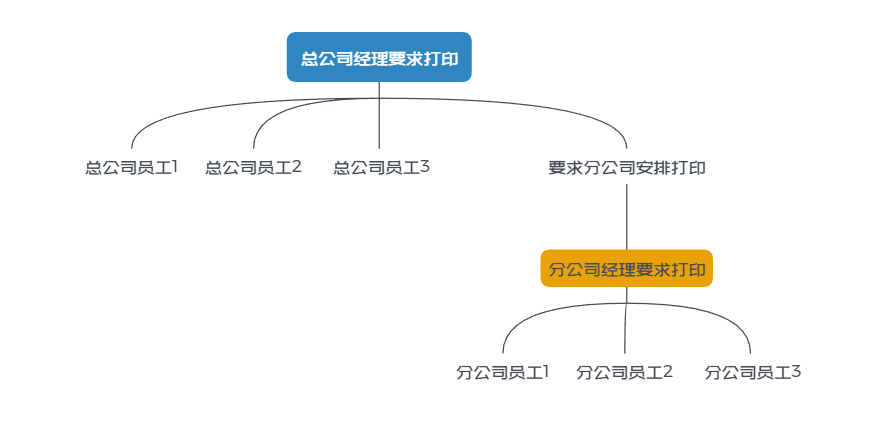
有一个集团公司，下属单位有分公司和直属部门，现在要求打印出所有下属单位的员工ID



先来看一下违反迪米特法则的设计方法：

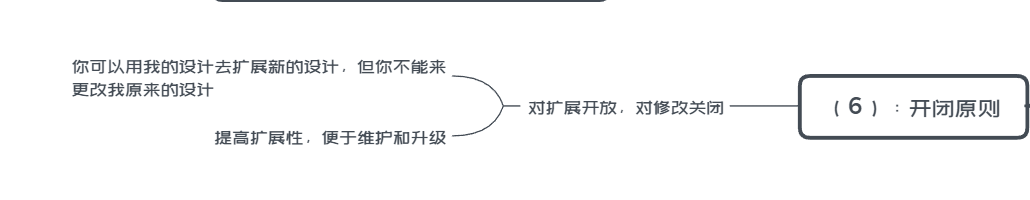


那这个设计的主要问题出在总公司经理类中，根据迪米特法则，只与直接的朋友发生通信，而分公司员工类并不是总公司经理类的直接朋友（以局部变量出现的耦合不属于直接朋友），从逻辑上讲总公司只与他的分公司耦合就行了，与分公司的员工并没有任何联系，这样设计显然是增加了不必要的耦合。按照迪米特法则，应该避免类中出现这样非直接朋友关系的耦合。修改后如下:



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

**1.2.6 开闭原则**



综上再回想一下前面说的5项原则，恰恰是告诉我们用抽象构建框架，用实现扩展细节的注意事项而已：单一职责原则告诉我们实现类要职责单一；里氏替换原则告诉我们不要破坏继承体系；依赖倒置原则告诉我们要面向接口编程；接口隔离原则告诉我们在设计接口的时候要精简单一；迪米特法则告诉我们要降低耦合。而开闭原则是总纲，他告诉我们要对扩展开放，对修改关闭。

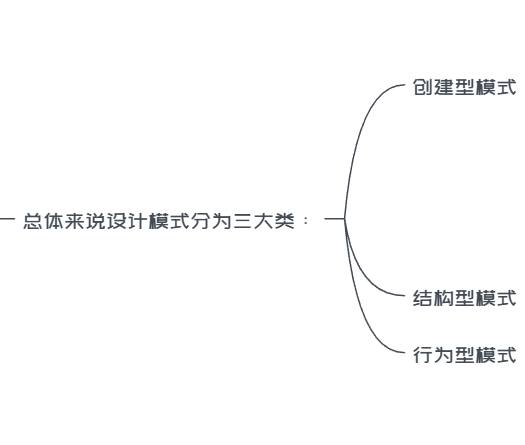
最后说明一下如何去遵守这六个原则。对这六个原则的遵守并不是是和否的问题，而是多和少的问题，也就是说，我们一般不会说有没有遵守，而是说遵守程度的多少。任何事都是过犹不及，设计模式的六个设计原则也是一样，制定这六个原则的目的并不是要我们刻板的遵守他们，而需要根据实际情况灵活运用。对他们的遵守程度只要在一个合理的范围内，就算是良好的设计。我们用一幅图来说明一下。

**例：**

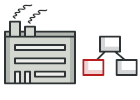
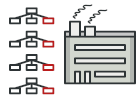
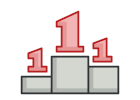
**(a) (b)**

设计a属于良好设计，他们对六项原则的遵守程度都在合理的范围内，且较为均匀。

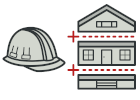
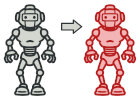
**1.3 设计模式**

****

以创建型模式为例：  
而创建型模式具体分为：

工厂方法 单例 抽象工厂



原型 生成器

面向对象代码经常用到的就是创建对象，创建型设计模式就是把对象的创建和对象的使用分离开，在写类的时候使用创建型模式会使得之后获取对象方便。如果目前不是很理解，也没关系，可以先来看一个例子。

(在这里参考的是Head First 设计模式一书，该书运用了各种生动形象的例子来阐释设计模式，是一本非常适合入门设计模式的书籍，没有之一。

而相比下，元老级的GoF的《设计模式》一书，虽然很薄，但也很抽象，不适合初学者阅读。

(由于时间有限，这里主要以抽象工厂模式和工厂方法为主来讲解

**1.3.1 简单工厂**

假设你有一个披萨店，身为披萨店的主人，你想做一套智能化的披萨制作系统，从而解放生产力。

那么在一开始的订单系统(我们可以认为是原始社会时期)

代码如下：

Pizza orderPizza(){

//为了让系统有弹性，我们很希望这是一个抽象类或接口。但如果这样，这些类或接口就无法直接实例化。

Pizza pizza =new Pizza();

pizza.prepare();

pizza.bake();

pizza.cut();

pizza.box();

return pizza;

}

但是你需要更多披萨类型……  
所以必须增加一些代码，来决定适合的披萨类型，然后再“制造”这个披萨。

Pizza orderPizza(String type){

Pizza pizza ;

if(type.equals("cheese")){

pizza=new CheesePizza();

}else if(type.equals("greek")){

pizza=new GreekPizza();

}else if(type.equals("pepperoni")){

pizza=new PepperoniPizza();

}

pizza.prepare();

pizza.bake();

pizza.cut();

pizza.box();

return pizza;

}

但是压力来自于增加更多的披萨类型  
你发现其他餐厅又多了很多流行风味的披萨：ClamPizza（蛤蜊披萨）、VeggiePizza（素食披萨）。很明显你要赶上他们，需要把这些加入你的菜单中，而最近GreekPizza（希腊披萨）卖的不好，所以你决定将它从菜单中去掉：

Pizza orderPizza(String type){

Pizza pizza ;

if(type.equals("cheese")){

pizza=new CheesePizza();

}//else if(type.equals("greek")){

// pizza=new GreekPizza();

// }

else if(type.equals("pepperoni")){

pizza=new PepperoniPizza ();

}else if(type.equals("clam")){

pizza=new ClamPizza();

}else if(type.equals("veggie")){

pizza=new VeggiePizza();

}

pizza.prepare();

pizza.bake();

pizza.cut();

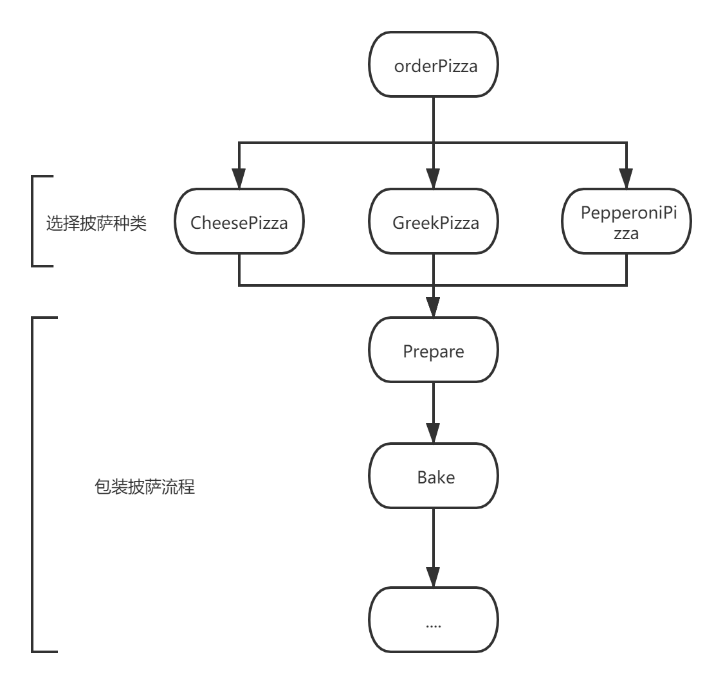
pizza.box();

return pizza;

}

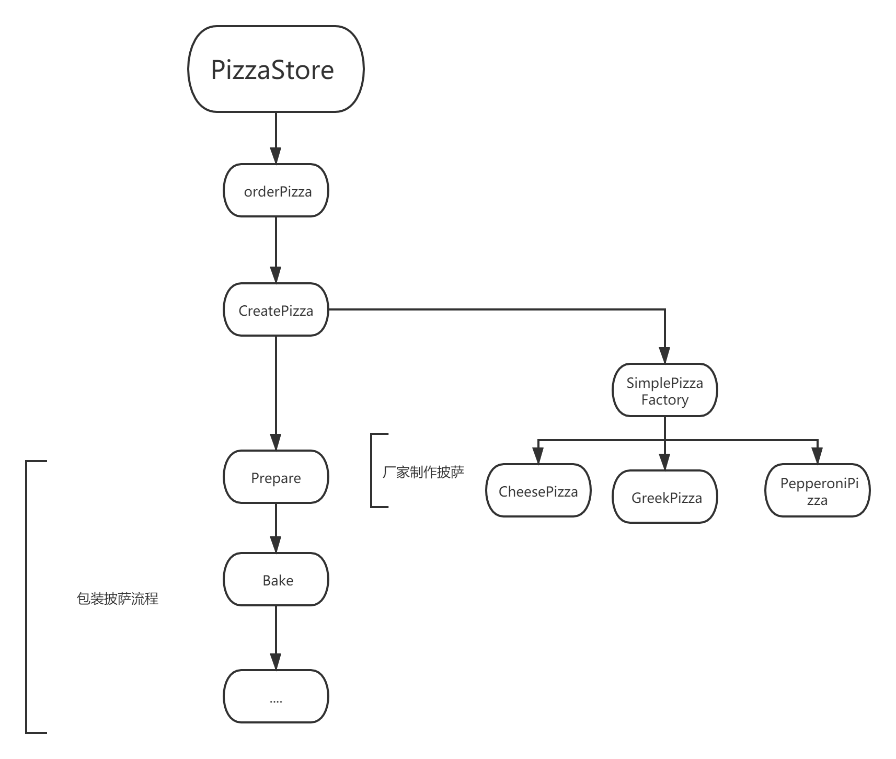
很明显，如果实例化“某些”具体类，将使orderPizza()出问题，而且也无法让orderPizza()对修改关闭；显然这里违背了之前提到的开闭原则。

如何修改呢？



观察代码发现，只有在创建Pizza这里会随着不同披萨类型而发生改变，而pizza打包、切割等环节都是固定的，于是将这一部分抽离出来，即我们把创建对象的代码从orderPizza()方法中抽离。然后把这部分的代码搬到另一个对象中，这个心对象只管如何创建披萨，如果任何对象想要创建披萨，找它就对了。  
我们称这个新对象为“工厂”  
工厂(factory)处理创建对象细节，一旦有了SimplePizzaFactory，orderPizza()就变成此对象的客户。也就是之前提到的，创建型设计模式就是把对象的创建和对象的使用分离开

如图



上面是简单工厂带来的热身，接下来登场的是两个重量级的模式。

**1.3.2 工厂方法模式**  
你的披萨店经营有成，击败了竞争者，现在大家都希望比萨店能在自己家附加有加盟店。每家加盟店都可能想要提供不同风味的披萨(纽约，芝加哥，加州)。如果是分别建不同工厂，不能实现总部的质量控制，容易砸招牌。  
有个做法可让披萨制作轰动局限于PizzaStore类，而同时又能让这些加盟店依然可以自由地制作该区域的风味。所要做的事情，就是把createPizza()方法放回到PizzaStore中，不过要把它设置成“抽象方法”，然后为每个区域风味创建一个PizzaStore的子类。

**给披萨店使用框架**

bastract class PizzaStore(){

public Pizza orderPizza(String type){

Pizza pizza;

pizza=createPizza(type);

pizza.prepare();

pizza.bake();

pizza.cut();

pizza.box();

return pizza;

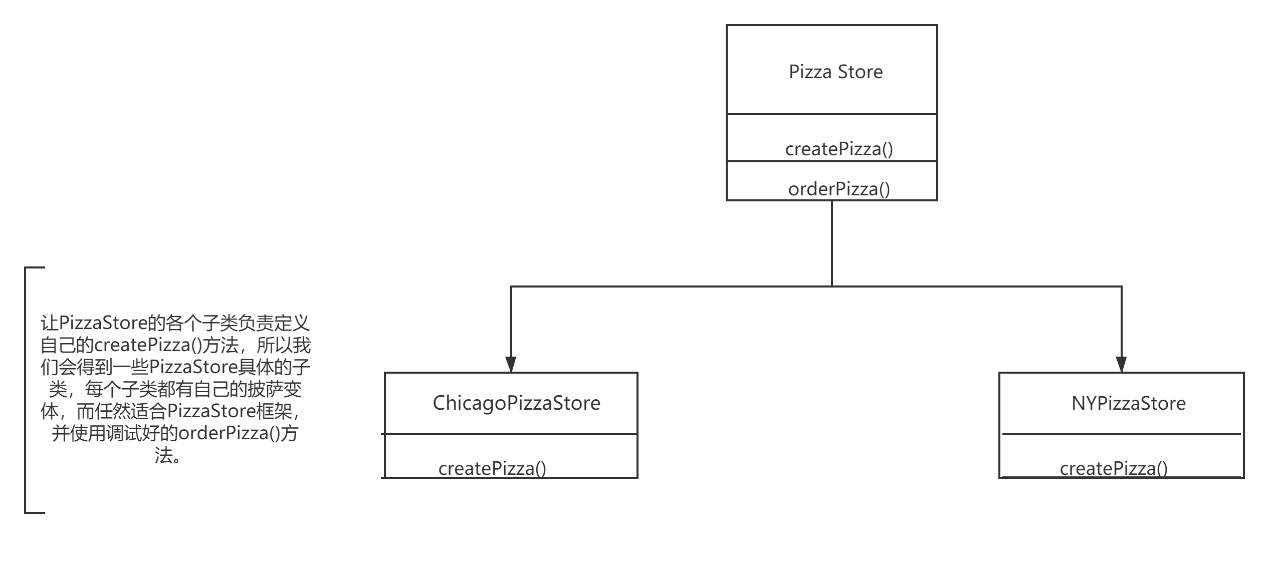
}

abstract Pizza createPizza(String type);

}

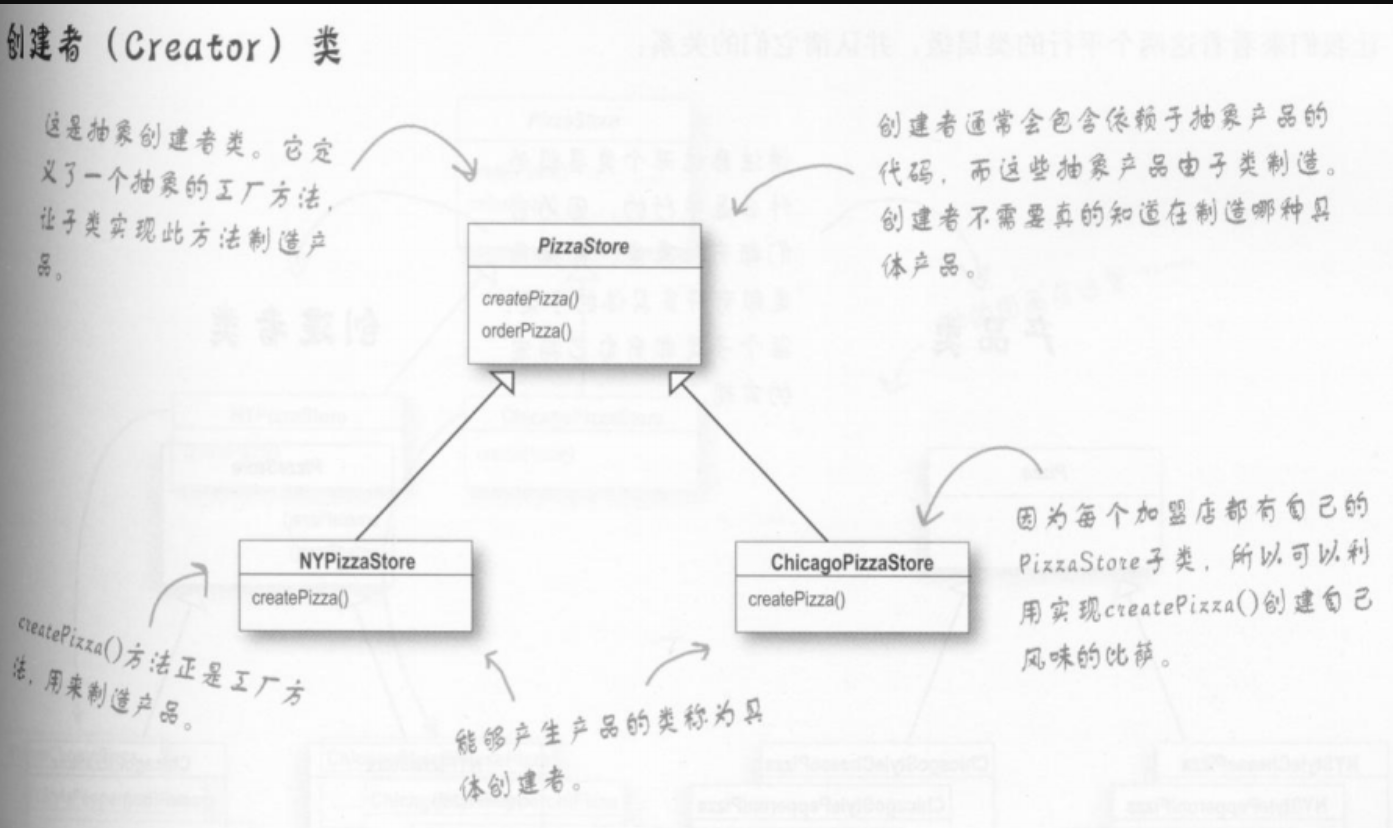
现在已近有一个PizzaStore作为超类，让每个域类型(NYPizzaFactory，ChicagoPizzaFactory，CaliforniaPizzaFactory)都继承这个PizzaStore，每个子类各自决定如何制造披萨，

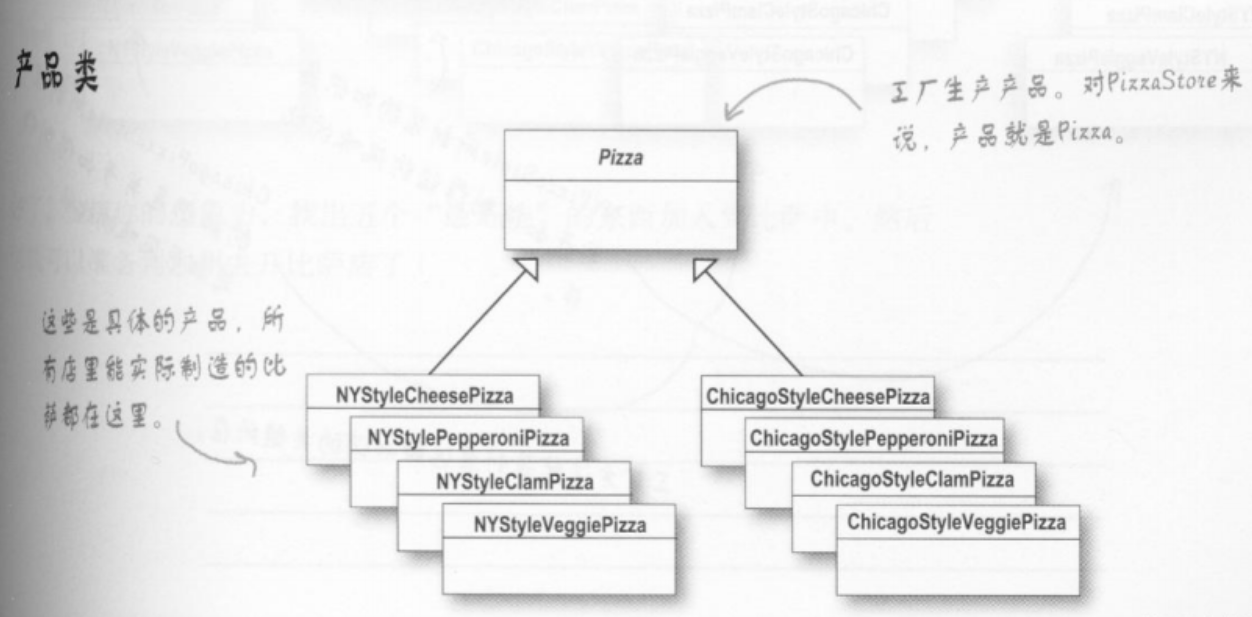
如图：



工厂方法模式通过让子类决定改创建的对象是什么，来达到将对象创建的过程封装的目的

这里分为两类，从两个维度说明：





定义工厂方法模式  
工厂方法模式定义了一个创建对象的接口，但由子类决定要实例化的类是哪一个。工厂方法让类把实例化推迟到子类。  
工厂方法与简单工厂的区别：工厂方法的子类看起来很像简单工厂。简单工厂把全部的事情在一个地方处理完成，而工厂方法却是创建一个框架，让子类决定如何实现

**1.3.3 抽象工厂模式：**

回到披萨制造工厂, 随着你的加盟店越开越多，一个工厂产能有限，要开发生产太多品类，分散了精力。只能多开几个工厂，把加盟店划分区域，一个区域一个工厂，负责此区域内加盟店的披萨需求。也就是，将工厂也分区域分类别开。本着抽象、封装的原则，我们可以把抽象成接口，因为工厂都是负责做披萨而已。只不过不同工厂做的步骤、原料不一样（实现不同，但上层并不关心）

**建造原料工厂**  
我们要建造一个工厂来生产原料，这个工厂负责创建原料家族中的每一种原料。

interface PizzaIngredientFactory{

public Dough createDough();

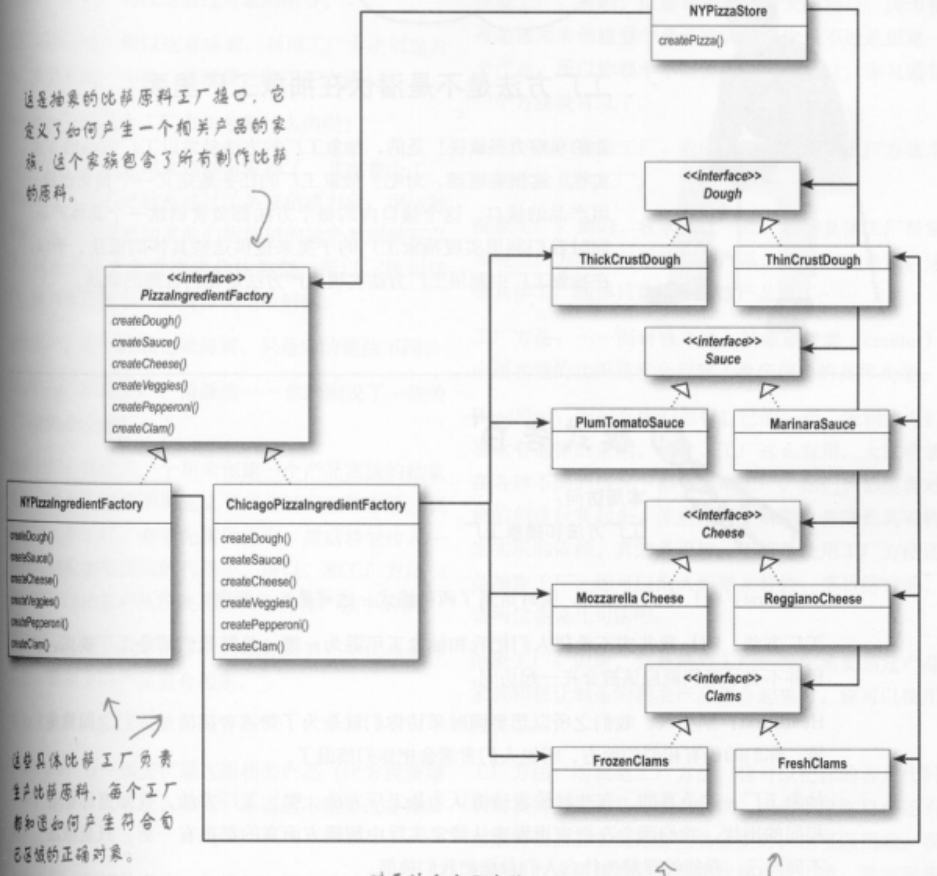
public Sauce createSauce();

public Cheese createCheese();

public Veggies[] createVeggies();

public Pepperoni createPepperoni();

public Clams createClams();

}  


那么这两种模式区别在哪呢？

相同：都是用来创建对象。

不同：工厂方法使用的是继承，抽象工厂使用的是组合。

优点：工厂方法只负责从具体类型中解耦，抽象工厂适合将一群相关的产品集合起来

可以看到随着披萨店规模越来越大，所需要的披萨智能化生产系统也越来越复杂，但同时效率也会越来越高。