工厂模式：主要用来实例化有共同接口的类，工厂模式可以动态决定应该实例化那一个类。  
**工厂模式的形态**  
工厂模式主要用一下几种形态：  
1：简单工厂（Simple Factory）。  
2：工厂方法（Factory Method）。  
3：抽象工厂（Abstract Factory）。  
  
**简单工厂（Simple Factory）**  
又叫静态工厂，是工厂模式三中状态中结构最为简单的。主要有一个静态方法，用来接受参数，并根据参数来决定返回实现同一接口的不同类的实例。我们来看一个具体的例子：  
假设一家工厂，几生产洗衣机，有生产冰箱，还有空调等等..  
我们先为所有产品定义一个共同的产品接口

1. **public** **interface** Product{}

接着我们让这个工厂的所有产品都必须实现此接口

1. **public** **class** Washer **implements** Product{
2. **public** Washer(){
3. System.out.println("洗衣机被制造了");
4. }
5. }
7. **public** **class** Icebox **implements** Product{
8. **public** Icebox(){
9. System.out.println("冰箱被制造了");
10. }
11. }
13. **public** **class** AirCondition **implements** Product{
14. **public** Icebox(){
15. System.out.println("空调被制造了");
16. }
17. }

接下来我们来写一个工厂类，有它来负责生产以上的产品

1. **public** **class** SimpleFactory {
2. //返回产品对象Product
3. **public**  **static**  Product factory(String productName) **throws** Exception{
4. **if**(productName.equals("Washer")){
5. **return** **new** Washer();
6. }**else** **if**(productName.equals("Icebox")){
7. **return** **new** Icebox();
8. }**else** **if**(productName.equals("AirCondition")){
9. **return** **new** AirCondition();
10. }**else**{
11. **throw** **new** Exception("没有该产品");
12. }
13. }
14. }

好了，有了这个工厂类，我们就可以开始下定单了，SimpleFactory将根据不同的定单类决定生产什么产品。

1. **public** **static** **void** main(String[] args) {
2. **try** {
3. SimpleFactory.factory("Washer");
4. SimpleFactory.factory("Icebox");
5. SimpleFactory.factory("AirCondition");
6. } **catch** (Exception e) {
7. e.printStackTrace();
8. }
9. }

由上面的代码可以看出，简单工厂的核心就是一个SimpleFactory类，他拥有必要的逻辑判断能力和所有产品的创建权利，我们只需要向把定单给他，就能得到我们想要的产品。这使用起来似乎非常方便。  
但，实际上，这个SimpleFactory有很多的局限。首先，我们每次想要增加一种新产品的时候，都必须修改SimpleFactory的原代码。其次，当我们拥有很多很多产品的时候，而且产品之间又存在复杂的层次关系的时候，这个类必须拥有复杂的逻辑判断能力，其代码量也将不断地激增，这对以后的维护简直就是恐怖两个字...  
还有就是，整个系统都严重依赖SimpleFactory类，只要SimpleFactory类一出问题，系统就进入不能工作的状态，这也是最为致命的一点....  
以上的不足将在工厂模式的另外两种状态中得到解决。  
  
**工厂方法（Factory Method）**  
上面的代码告诉我们，简单工厂并不简单，它是整个模式的核心，一旦他出了问题，整个模式都将受影响而不能工作，为了降低风险和为日后的维护、扩展做准备，我们需要对它进行重构，引入工厂方法。

第一个类：先定义产品类型，第二个类：生产产品

我们再来定义一个产品接口

1. **public** **interface** Product{}//定义产品的类型

工厂方法为工厂类定义了接口，用多态来削弱了工厂类的职能，以下是工厂接口的定义：

1. **public** **interface** Factory{ //生产产品
2. **public** Product create();
3. }

实现了产品类型接口的产品类

1. **public** **class** Washer **implements** Product{
2. **public** Washer(){
3. System.out.println("洗衣机被制造了");
4. }
5. }
7. **public** **class** Icebox **implements** Product{
8. **public** Icebox(){
9. System.out.println("冰箱被制造了");
10. }
11. }
13. **public** **class** AirCondition **implements** Product{
14. **public** Icebox(){
15. System.out.println("空调被制造了");
16. }
17. }

工厂方法的核心部分，也就是具体创建产品对象的具体工厂类

1. //创建洗衣机的工厂
2. **public** **class** CreateWasher **implements** Factory{
3. **public** Product create(){
4. **return** **new** Washer();
5. }
6. }
8. //创建冰箱的工厂
9. **public** **class** CreateIcebox **implements** Factory{
10. **public** Product create(){
11. **return** **new** Icebox();
12. }
13. }
15. //创建空调的工厂
16. **public** **class** CreateAirCondition **implements** Factory{
17. **public** Product create(){
18. **return** **new** AirCondition();
19. }
20. }

从上面创建产品对象的代码可以看出，工厂方法和简单工厂的主要区别是，简单工厂是把创建产品的职能都放在一个类里面，而工厂方法则把不同的产品放在实现了工厂接口的不同工厂类里面，这样就算其中一个工厂类出了问题，其他工厂类也能正常工作，互相不受影响，以后增加新产品，也只需要新增一个实现工厂接口工厂类，就能达到，不用修改已有的代码。但工厂方法也有他局限的地方，那就是当面对的产品有复杂的等级结构的时候，例如，工厂除了生产家电外产品，还生产手机产品，这样一来家电是手机就是两大产品家族了，这两大家族下面包含了数量众多的产品，每个产品又有多个型号，这样就形成了一个复杂的产品树了。如果用工厂方法来设计这个产品家族系统，就必须为每个型号的产品创建一个对应的工厂类，当有数百种甚至上千种产品的时候，也必须要有对应的上百成千个工厂类，这就出现了传说的类爆炸，对于以后的维护来说，简直就是一场灾难.....  
  
**抽象工厂（Factory Method）**  
抽象工厂：意的意图在于创建一系列互相关联或互相依赖的对象。<<Java设计模式>>  
我自己觉得抽象工厂是在工厂方法的基础上引进了分类管理的概念....  
工厂方法用来创建一个产品，它没有分类的概念，而抽象工厂则用于创建一系列产品，所以产品分类成了抽象工厂的重点，  
我们继续用上面的例子来说明：  
工厂生产的所有产品都用都用大写字母来标明它们的型号，比如冰箱，就有“冰箱-A",“冰箱-B",同样，其他的产品也都是遵守这个编号规则，于是就有了一下产品家族树  
  
冰箱：

1. 冰箱-A
2. 冰箱-B

洗衣机：

1. 洗衣机-A
2. 洗衣机-B

我们可以为冰箱和洗衣机分别定义两个产品接口，以对他们进行分类，

1. //洗衣机接口
2. **public** **interface** Washer{
3. }
5. //冰箱接口
6. **public** **interface** Icebox{
7. }

接着，我们分别创建这两个接口的具体产品

1. //洗衣机-A
2. **public** **class** WasherA **implements** Washer{
3. **public** WasherA(){
4. System.out.println("洗衣机-A被制造了");
5. }
6. }
8. //洗衣机-B
9. **public** **class** WasherB **implements** Washer{
10. **public** WasherB(){
11. System.out.println("洗衣机-B被制造了");
12. }
13. }
15. //冰箱-A
16. **public** **class** IceboxA **implements** Icebox{
17. **public** IceboxA(){
18. System.out.println("冰箱-A被制造了");
19. }
20. }
22. //冰箱-B
23. **public** **class** IceboxB **implements** Icebox{
24. **public** IceboxB(){
25. System.out.println("冰箱-B被制造了");
26. }
27. }

到此，产品部分我们准备好了，接下来我们来处理工厂部分，我们先来定义工厂行为接口

1. **public** **interface** Factory{
2. //生产产品
3. **public** Washer createWasher();
4. **public** Icebox createIcebox();
5. }

接下来我创造具体的工厂类，我们根据上面产品的接口，把型号A的产品分为一类，由一个工厂来管理，把型号为B的产品有另一个工厂管理，根据这个分类，我们可以实现如下的两个具体工厂类

1. //创建生产型号为A的产品工厂
2. **public** **class** FactoryA **implements** Factory{
3. //创建洗衣机-A
4. **public** Washer createWasher(){
5. **return** **new** WasherA();
6. }
8. //创建冰箱-A
9. **public** Icebox createIcebox(){
10. **return** **new** IceboxA();
11. }
12. }
14. //创建生产型号为B的产品工厂
15. **public** **class** FactoryB **implements** Factory{
16. //创建洗衣机-B
17. **public** Washer createWasher(){
18. **return** **new** WasherB();
19. }
21. //创建冰箱-B
22. **public** Icebox createIcebox(){
23. **return** **new** IceboxB();
24. }
25. }

这样，我们的抽象工厂就完成了。有上面可以看出，在运用上我觉得工厂方法和抽象工厂，都有自己的应用场景，并没有什么优劣之分，但在应用抽象工厂之前，要先对创建的对象进行系统的分类，这点很重要，好的产品分类规则能为具体工厂类的选择调用和以后的扩展提供清晰的思路.