Java设计模式

## 1.****Java中的24种设计模式与7大原则****

**一、创建型模式**

1、抽象工厂模式(Abstract factory pattern): 提供一个接口, 用于创建相关或依赖对象的家族, 而不需要指定具体类.  
2、生成器模式（建造者模式）(Builder pattern): 使用生成器模式封装一个产品的构造过程, 并允许按步骤构造. 将一个复杂对象的构建与它的表示分离, 使得同样的构建过程可以创建不同的表示.  
3、工厂模式(factory method pattern): 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.  
4、原型模式(prototype pattern): 当创建给定类的实例过程很昂贵或很复杂时, 就使用原形模式.  
5、单例模式(Singleton pattern): 确保一个类只有一个实例, 并提供全局访问点.  
6、多例模式(Multition pattern): 在一个解决方案中结合两个或多个模式, 以解决一般或重复发生的问题.  
**二、结构型模式**

1、适配器模式(Adapter pattern): 将一个类的接口, 转换成客户期望的另一个接口. 适配器让原本接口不兼容的类可以合作无间. 对象适配器使用组合, 类适配器使用多重继承.  
2、桥接模式(Bridge pattern): 使用桥接模式通过将实现和抽象放在两个不同的类层次中而使它们可以独立改变.  
3、组合模式(composite pattern): 允许你将对象组合成树形结构来表现"整体/部分"层次结构. 组合能让客户以一致的方式处理个别对象以及对象组合.  
4、装饰者模式(decorator pattern): 动态地将责任附加到对象上, 若要扩展功能, 装饰者提供了比继承更有弹性的替代方案.  
5、外观模式(facade pattern): 提供了一个统一的接口, 用来访问子系统中的一群接口. 外观定义了一个高层接口, 让子系统更容易使用.  
6、亨元模式(Flyweight Pattern): 如想让某个类的一个实例能用来提供许多"虚拟实例", 就使用蝇量模式.  
7、代理模式(Proxy pattern): 为另一个对象提供一个替身或占位符以控制对这个对象的访问.  
**三、行为型模式**

1、责任链模式(Chain of responsibility pattern): 通过责任链模式, 你可以为某个请求创建一个对象链. 每个对象依序检查此请求并对其进行处理或者将它传给链中的下一个对象.  
2、命令模式(Command pattern): 将"请求"封闭成对象, 以便使用不同的请求,队列或者日志来参数化其他对象. 命令模式也支持可撤销的操作.  
3、解释器模式(Interpreter pattern): 使用解释器模式为语言创建解释器.  
4、迭代器模式(iterator pattern): 提供一种方法顺序访问一个聚合对象中的各个元素, 而又不暴露其内部的表示.  
5、中介者模式(Mediator pattern) : 使用中介者模式来集中相关对象之间复杂的沟通和控制方式.  
6、备忘录模式(Memento pattern): 当你需要让对象返回之前的状态时(例如, 你的用户请求"撤销"), 你使用备忘录模式.  
7、观察者模式(observer pattern): 在对象之间定义一对多的依赖, 这样一来, 当一个对象改变状态, 依赖它的对象都会收到通知, 并自动更新.  
8、状态模式(State pattern): 允许对象在内部状态改变时改变它的行为, 对象看起来好象改了它的类.  
9、策略模式(strategy pattern): 定义了算法族, 分别封闭起来, 让它们之间可以互相替换, 此模式让算法的变化独立于使用算法的客户.  
10、模板方法模式(Template pattern): 在一个方法中定义一个算法的骨架, 而将一些步骤延迟到子类中. 模板方法使得子类可以在不改变算法结构的情况下, 重新定义算法中的某些步骤.  
11、访问者模式(visitor pattern): 当你想要为一个对象的组合增加新的能力, 且封装并不重要时, 就使用访问者模式.

**七大设计原则：**

1、单一职责原则【SINGLE RESPONSIBILITY PRINCIPLE】：一个类负责一项职责.  
2、里氏替换原则【LISKOV SUBSTITUTION PRINCIPLE】：继承与派生的规则.  
3、依赖倒置原则【DEPENDENCE INVERSION PRINCIPLE】：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。即针对接口编程，不要针对实现编程.  
4、接口隔离原则【INTERFACE SEGREGATION PRINCIPLE】：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.  
5、迪米特法则【LOW OF DEMETER】：低耦合，高内聚.  
6、开闭原则【OPEN CLOSE PRINCIPLE】：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.  
7、组合/聚合复用原则【Composition/Aggregation Reuse Principle(CARP) 】：尽量使用组合和聚合少使用继承的关系来达到复用的原则.

总体来说设计模式分为三大类：

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

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

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

其实还有两类：并发型模式和线程池模式。用一个图片来整体描述一下：



## 1.设计模式的3种类型，七大原则

GoF设计模式是程序员真正掌握面向对象核心思想的必修课,没有阅读这本书就是一种思维的缺失，这本书也是更加透彻了解和掌握《Thinking in Java》必须阅读的。就Java语言体系来说，GoF设计模式是Java基础知识和J2EE框架技术之前的一座隐性的“桥梁”。

GoF设计模式并不是一种具体“技术”，可以看做是一种“思想，灵魂”，展示接口或抽象类在实际案例中的灵活应用，让程序尽可能达到可重用。Java接口和抽象类才是真正体现Java思想的核心所在。

J2EE属于一种框架软件，不同于JavaAPI（属于Toolkist工具箱），不在被动的被使用，而是深入到一个领域中去。J2EE框架软件设计的目的是将一个领域中不变化的东西先定义好，比如整体结构和一些主要职责（数据库操作、事务、安全等），其余就是变化的东西，针对领域中具体应用而产生的不同的变化需求，就是J2EE开发人员要做的工作。

GoF设计模式是Java基础知识和J2EE框架技术之前的一座隐性的“桥梁”？

1）.设计模式更加抽象，J2EE是具体的代码；

2）.设计模式是比J2EE框架软件更小的体系结构，J2EE中许多程序都是使用设计模式来完成的，不具备设计模式知识，很难快速理解J2EE；

3）.J2EE是只适合企业计算的框架软件，但是GoF的设计模式几乎可以用于任何应用。

Java提供了丰富的API，同时又有强大的数据库系统DBS作底层支持，那么编程似乎变成了类似堆积木的简单“拼凑”和“调用”，这是一种误解，对现代编程技术的不了解所致。在真正的面向对象编程OOP中，GoF的《设计模式》为我们提供了一套可复用的面向对象的技术，再配合重构方法Refactoring，所有很少存在简单重复的劳动，再加上Java代码的精炼性和面向对象的纯洁性，编程工作将变成一个让你时刻体验创造快感的激动人心的过程。

GoF的《设计模式》都是以C++为例，我将以Java语言加上UML图形化描述为例来深度讲解设计模式。

创建模式的人是大师，但是拘泥于模式的人永远是工匠。

## 1.1创建型设计模式（Creational Patterns）

创建型设计模式是处理对象创建机制的设计模式

**工厂模式分为三类：**简单工厂模式（Simple Factory）、工厂方法模式（Factory Method）和抽象工厂模式（Abstract Factory），三种模式从前往后逐步抽象，并且更具有一般性。GOF在《设计模式》中将工厂模式分为两类工厂方法模式（Factory Method）和抽象工厂模式（Abstract Factory）。简单工厂模式（Simple Factory）可以看作是工厂方法模式（Factory Method）的一种特例。

**角色说明：**

**抽象产品类**：所有产品的抽象父类或接口，一般是1个；

**具体产品类**：抽象产品类的具体实现，也就是具体对象实例化的类，一般是多个；

**抽象工厂类**：所有工厂的抽象父类或接口，一般是1个；

**具体工厂类**：抽象工厂的具体实现，负责某一类别产品的创建，一般是多个。



|  |  |  |  |
| --- | --- | --- | --- |
| 工厂模式 | 抽象产品类 | 抽象工厂类 | 具体工厂类 |
| 工厂方法模式 | **1个**，可以派生出**多个**具体产品类 | **1个**，可以派生出**多个**具体工厂类 | **每个**具体工厂类只能创建**一个**具体产品类实例 |
| 抽象工厂模式 | **多个**，**每个**可以派生出**多个**具体产品类 | **1个**，可以派生出**多个**具体工厂类 | **每个**具体工厂类可以创建**多个**具体产品类实例 |

### 1.1.1 简单工厂模式（Simple Factory）

#### # 定义

简单工厂模式又称静态工厂方法模式，不属于23种GOF设计模式之一。由一个工厂对象决定创建出属于哪一种产品类型的实例。简单工厂模式是工厂方法模式的一个特殊实现。

**组成：**

工厂类：模式的核心，含有一定的业务逻辑；

抽象产品：一般是具体产品要继承的父类或者要实现的接口；

具体产品：具体实现类，工厂类创建的对象是具体产品的一个实例。

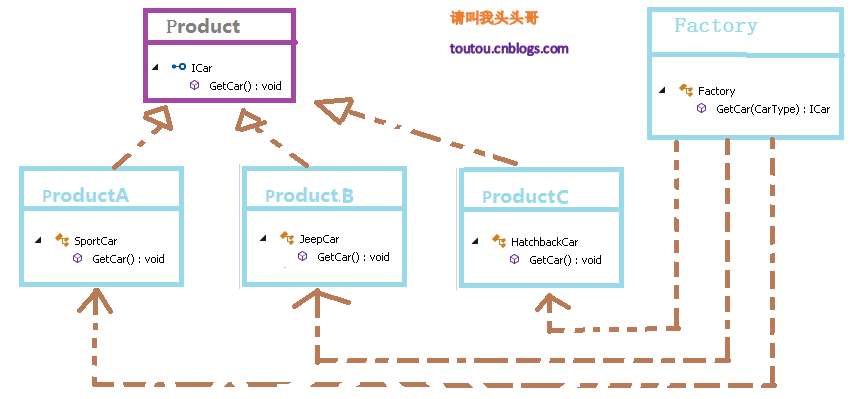
#### # 使用场景

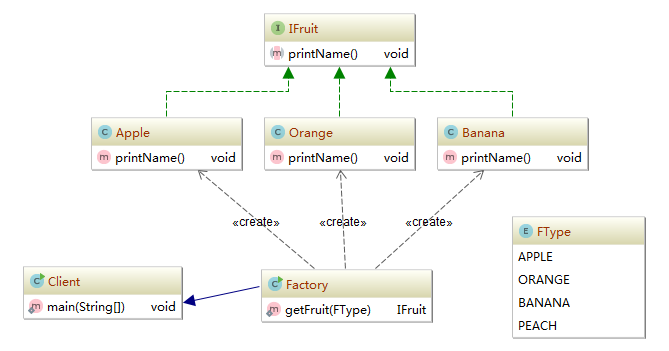
#### # 优缺点

**优点：**简单工厂模式可以根据外界给定信息，决定应该创建哪个具体类的实例，明确区分了各自的职责和权力，有利于整个软件体系结构的优化。

**缺点：**工厂类集中了所有实例的创建逻辑，违反了高内聚责任分配原则。

#### # UML类图





#### # Java代码实例

**抽象产品类：**

*/\*\*  
 \* 1.创建型设计模式之简单工厂模式  
 \* 抽象产品类：IFruit  
 \** ***@author:*** *liangpengju  
 \** ***@date:*** *2017/8/30  
 \** ***@version:*** *1.0  
 \*/***public interface** IFruit {  
 */\*\*  
 \* 水果名称  
 \*/* **public void** printName();  
}

**具体产品类：**

*/\*\*  
 \* 1.创建型设计模式之简单工厂模式  
 \* 角色：具体产品类，接口IFruit具体实现类  
 \** ***@author:*** *liangpengju  
 \** ***@date:*** *2017/8/30  
 \** ***@version:*** *1.0  
 \*/***public class** Banana **implements** IFruit{  
 @Override  
 **public void** printName() {  
 System.***out***.println(**"This is Banana..."**);  
 }  
}

*/\*\*  
 \* 1.创建型设计模式之简单工厂模式  
 \* 角色：具体产品类，接口IFruit具体实现类  
 \** ***@author:*** *liangpengju  
 \** ***@date:*** *2017/8/30  
 \** ***@version:*** *1.0  
 \*/***public class** Apple **implements** IFruit{  
 @Override  
 **public void** printName() {  
 System.***out***.println(**"This is Apple..."**);  
 }  
}

*/\*\*  
 \* 1.创建型设计模式之简单工厂模式  
 \* 角色：具体产品类，接口IFruit具体实现类  
 \** ***@author:*** *liangpengju  
 \** ***@date:*** *2017/8/30  
 \** ***@version:*** *1.0  
 \*/***public class** Orange **implements** IFruit {  
 @Override  
 **public void** printName() {  
 System.***out***.println(**"This is Orange..."**);  
 }  
}

**核心工厂类：实现1**

*/\*\*  
 \* 1.创建型设计模式之简单工厂模式  
 \* 角色：简单工厂类  
 \** ***@author:*** *liangpengju  
 \** ***@date:*** *2017/8/30  
 \** ***@version:*** *1.0  
 \*/***public class** Factory **extends** Client {  
  
 **public static** IFruit getFruit(FType fType) **throws** Exception {  
 **switch** (fType){  
 **case *APPLE***:  
 **return new** Apple();  
 **case *BANANA***:  
 **return new** Banana();  
 **case *ORANGE***:  
 **return new** Orange();  
 **default**:  
 **throw new** Exception(**"不存在该实例"**);  
 }  
 }  
}

**客户端：测试Factory核心工厂**

*/\*\*  
 \* 1.创建型设计模式之简单工厂模式  
 \* 角色：客户端  
 \** ***@author:*** *liangpengju  
 \** ***@date:*** *2017/8/30  
 \** ***@version:*** *1.0  
 \*/***public class** Client {  
  
 **public static void** main(String[] args) {  
 IFruit iFruit;  
 **try**{  
 *//Apple* iFruit = Factory.*getFruit*(FType.***APPLE***);  
 iFruit.printName();  
 *//Banana* iFruit = Factory.*getFruit*(FType.***BANANA***);  
 iFruit.printName();  
 *//Orange* iFruit = Factory.*getFruit*(FType.***ORANGE***);  
 iFruit.printName();  
 *//PEACH 没有该实例* iFruit = Factory.*getFruit*(FType.***PEACH***);  
 iFruit.printName();  
 }**catch** (Exception e){  
 System.***out***.println(e.getMessage());  
 }  
 }  
}

*/\*\*  
 \* 枚举类型  
 \*/***public enum** FType {  
 ***APPLE***,  
 ***ORANGE***,  
 ***BANANA***,  
 ***PEACH***;  
}

**核心工厂类：实现2**

*/\*\*  
 \* 1.创建型设计模式之简单工厂模式  
 \* 角色：简单工厂类  
 \* 作用：使用反射的方式创建具体产品  
 \** ***@author:*** *liangpengju  
 \** ***@date:*** *2017/8/30  
 \** ***@version:*** *1.0  
 \*/***public class** Factory2 {  
  
 */\*\*  
 \* 静态工厂方法，创建各种各样的水果  
 \** ***@param clazz*** *\** ***@return*** *\*/* **public static** IFruit createFruit(Class clazz){  
 IFruit iFruit = **null**;  
 **try** {  
 *//根据类名创建一个实例* iFruit= (IFruit) Class.*forName*(clazz.getName()).newInstance();  
 } **catch** (InstantiationException e) {  
 System.***out***.println(**"实例对象初始化异常"**);  
 } **catch** (IllegalAccessException e) {  
 System.***out***.println(**"安全权限访问异常"**);  
 } **catch** (ClassNotFoundException e) {  
 System.***out***.println(**"没有找到具体的产品类"**);  
 }  
 **return** iFruit;  
 }  
}

**客户端：测试Factory2核心工厂**

*/\*\*  
 \* 1.创建型设计模式之简单工厂模式  
 \* 角色：客户端  
 \* 作用：测试Factory2创建具体产品实例  
 \** ***@author:*** *liangpengju  
 \** ***@date:*** *2017/8/30  
 \** ***@version:*** *1.0  
 \*/***public class** Client2 {  
  
 **public static void** main(String[] args) {  
 IFruit iFruit;  
 **try**{  
 *//Apple* iFruit = Factory2.*createFruit*(Apple.**class**);  
 iFruit.printName();  
 *//Banana* iFruit = Factory2.*createFruit*(Banana.**class**);  
 iFruit.printName();  
 *//Orange* iFruit = Factory2.*createFruit*(Orange.**class**);  
 iFruit.printName();  
 }**catch** (Exception e){  
 System.***out***.println(e.getMessage());  
 }  
 }  
}

#### #JDK 应用实例

### 1.1.2 工厂方法模式（Factory Method）

#### # 定义

工厂方法模式又称为多态性工厂方法模式，核心的工厂类不再负责所有产品的创建，而是将具体的创建工作交给工厂子类去做。核心工厂变成一个抽象工厂的角色，仅负责给出具体工厂子类必须实现的接口，而不参与哪一个具体产品类应该被实例化这种细节。

解决了简单工厂模式的一些问题，完全实现了“开-闭”原则，实现了可扩展；更加复杂的层次结构，可用于产品结果复杂的情景。

#### # 使用场景

Person person = new Person(); 很多情况下对象创建时需要做初始化工作，比如：赋值、查询数据库等，如果初始化工作是很长一段代码，就会把很多工作都放在一个方法中执行，相当于将很多鸡蛋放在一个篮子里，有悖于Java面向对象的原则。面向对象的封装（Encapsulation）和委托（Delegation）是尽可能将过长的代码委托“分割”成每段，再将每段“封装”起来，减少段与段直接的耦合性，这样就可以分数风险，以后如需要修改，只需要修改每段即可。

* 当一个类不能预料它必须创建哪一种类型的对象时；
* 你想要局部化理解哪些类被创建；
* 当您有继承于相同子类的类时，或者它们仅仅只是实现同一接口的不相关的类，又或者这些类实例的方法是相同的并且可以交替使用；
* 当你想要将实际类型的客户端与正在被实例化的客户端进行隔离时。

#### # 优缺点

**优点：**

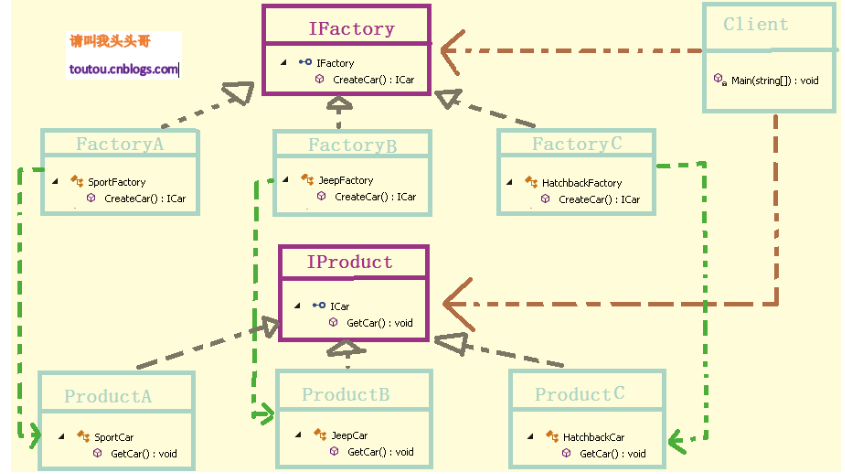
* 客户端不需要知道它要创建对象实例的每一个子类，它仅仅需要指定一个抽象类或接口和工厂类对象；
* 工厂类封装对象的创建方法，如果创建过程非常复杂时会很有用；
* 屏蔽产品类：产品类如何实现，客户端无需关心，只关心产品的接口，只要接口保持不变，系统上层的模块就不会发生变化；
* 典型的解耦框架：高层模块只需要知道抽象产品类，无需关心具体产品类，符合迪米特法则，符合依赖倒置原则，符合里氏替换原则；
* 多态性：客户端代码可以做到与特点应用无关，适用于任何实体类。

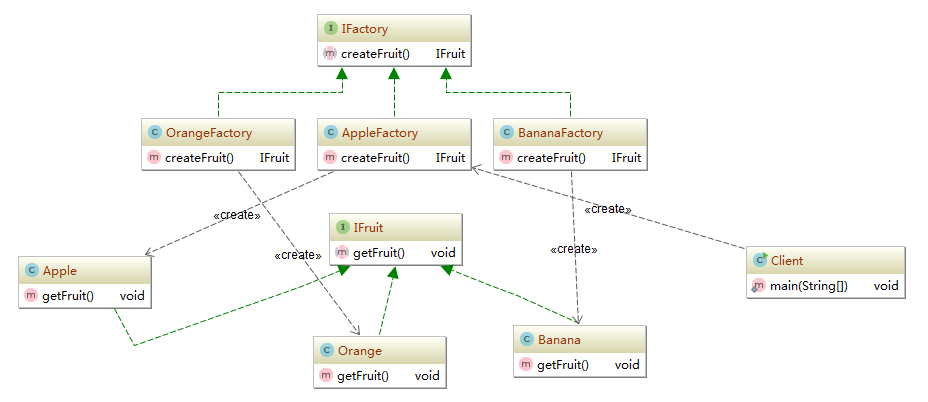
**缺点：**

不重新编译就没有办法改变一个实现类。

需要抽象产品类和具体产品类作为factory method的载体，需要增加一个类层次。

#### # UML类图





#### # Java代码实例

**抽象工厂类**

*/\*\*  
 \* 2.创建型设计模式之工厂方法模式  
 \* 角色：抽象工厂接口  
 \** ***@author:*** *liangpengju  
 \** ***@date:*** *2017/9/1  
 \** ***@version:*** *1.0  
 \*/***public interface** IFactory {  
  
 *//创建水果实例对象* IFruit createFruit();  
}

**具体工厂类**

*/\*\*  
 \* 2.创建型设计模式之工厂方法模式 Factory Method Pattern  
 \* 角色：抽象工厂类  
 \* 作用：用于生产Apple  
 \** ***@author:*** *liangpengju  
 \** ***@date:*** *2017/9/1  
 \** ***@version:*** *1.0  
 \*/***public class** AppleFactory **implements** IFactory {  
 @Override  
 **public** IFruit createFruit() {  
 **return new** Apple();  
 }  
}

*/\*\*  
 \* 2.创建型设计模式之工厂方法模式 Factory Method Pattern  
 \* 角色：抽象工厂类  
 \* 作用：用于生产Banana  
 \** ***@author:*** *liangpengju  
 \** ***@date:*** *2017/9/1  
 \** ***@version:*** *1.0  
 \*/***public class** BananaFactory **implements** IFactory {  
 @Override  
 **public** IFruit createFruit() {  
 **return new** Banana();  
 }  
}

*/\*\*  
 \* 2.创建型设计模式之工厂方法模式 Factory Method Pattern  
 \* 角色：抽象工厂类  
 \* 作用：用于生产Orange  
 \** ***@author:*** *liangpengju  
 \** ***@date:*** *2017/9/1  
 \** ***@version:*** *1.0  
 \*/***public class** OrangeFactory **implements** IFactory {  
 @Override  
 **public** IFruit createFruit() {  
 **return new** Orange();  
 }  
}

**抽象产品类**

*/\*\*  
 \* 2.创建型设计模式之工厂方法模式 Factory Method Pattern  
 \* 角色：抽象产品类  
 \** ***@author:*** *liangpengju  
 \** ***@date:*** *2017/9/1  
 \** ***@version:*** *1.0  
 \*/***public interface** IFruit {  
  
 *//获取水果名称* **void** getFruit();  
}

具体产品类

*/\*\*  
 \* 2.创建型设计模式之工厂方法模式 Factory Method Pattern  
 \* 角色：具体产品类  
 \* 作用：苹果类  
 \** ***@author:*** *liangpengju  
 \** ***@date:*** *2017/9/1  
 \** ***@version:*** *1.0  
 \*/***public class** Apple **implements** IFruit {  
 @Override  
 **public void** getFruit() {  
 System.***out***.println(**"Hi,This is Apple!"**);  
 }  
}

*/\*\*  
 \* 2.创建型设计模式之工厂方法模式 Factory Method Pattern  
 \* 角色：具体产品类  
 \* 作用：香蕉类  
 \** ***@author:*** *liangpengju  
 \** ***@date:*** *2017/9/1  
 \** ***@version:*** *1.0  
 \*/***public class** Banana **implements** IFruit {  
  
 @Override  
 **public void** getFruit() {  
 System.***out***.println(**"Hi,This is Banana!"**);  
 }  
}

*/\*\*  
 \* 2.创建型设计模式之工厂方法模式 Factory Method Pattern  
 \* 角色：具体产品类  
 \* 作用：桔子类  
 \** ***@author:*** *liangpengju  
 \** ***@date:*** *2017/9/1  
 \** ***@version:*** *1.0  
 \*/***public class** Orange **implements** IFruit {  
  
 @Override  
 **public void** getFruit() {  
 System.***out***.println(**"Hi,This is Orange!"**);  
 }  
}

客户端

*/\*\*  
 \* 2.创建型设计模式之工厂方法模式 Factory Method Pattern  
 \* 角色：客户端  
 \* 作用：用于测试工厂方法模式  
 \** ***@author:*** *liangpengju  
 \** ***@date:*** *2017/9/1  
 \** ***@version:*** *1.0  
 \*/***public class** Client {  
  
 **public static void** main(String[] args) {  
 *//抽象工厂* IFactory iFactory = **new** AppleFactory();  
 *//抽象产品* IFruit iFruit = iFactory.createFruit();  
 iFruit.getFruit();  
 }  
}

#### # JDK应用实例

### 1.1.3 抽象工厂模式（Abstract Factory）

#### # 定义

工厂方法模式又称为多态性工厂方法模式，核心的工厂类不再负责所有产品的创建，而是将具体的创建工作交给工厂子类去做。核心工厂变成一个抽象工厂的角色，仅负责给出具体工厂子类必须实现的接口，而不参与哪一个具体产品类应该被实例化这种细节。

解决了简单工厂模式的一些问题，完全实现了“开-闭”原则，实现了可扩展；更加复杂的层次结构，可用于产品结果复杂的情景。

#### # 使用场景

Person person = new Person(); 很多情况下对象创建时需要做初始化工作，比如：赋值、查询数据库等，如果初始化工作是很长一段代码，就会把很多工作都放在一个方法中执行，相当于将很多鸡蛋放在一个篮子里，有悖于Java面向对象的原则。面向对象的封装（Encapsulation）和委托（Delegation）是尽可能将过长的代码委托“分割”成每段，再将每段“封装”起来，减少段与段直接的耦合性，这样就可以分数风险，以后如需要修改，只需要修改每段即可。

#### # 优缺点

#### # UML类图

#### # Java代码实例

#### # JDK应用实例

## 2.简单工厂模式

