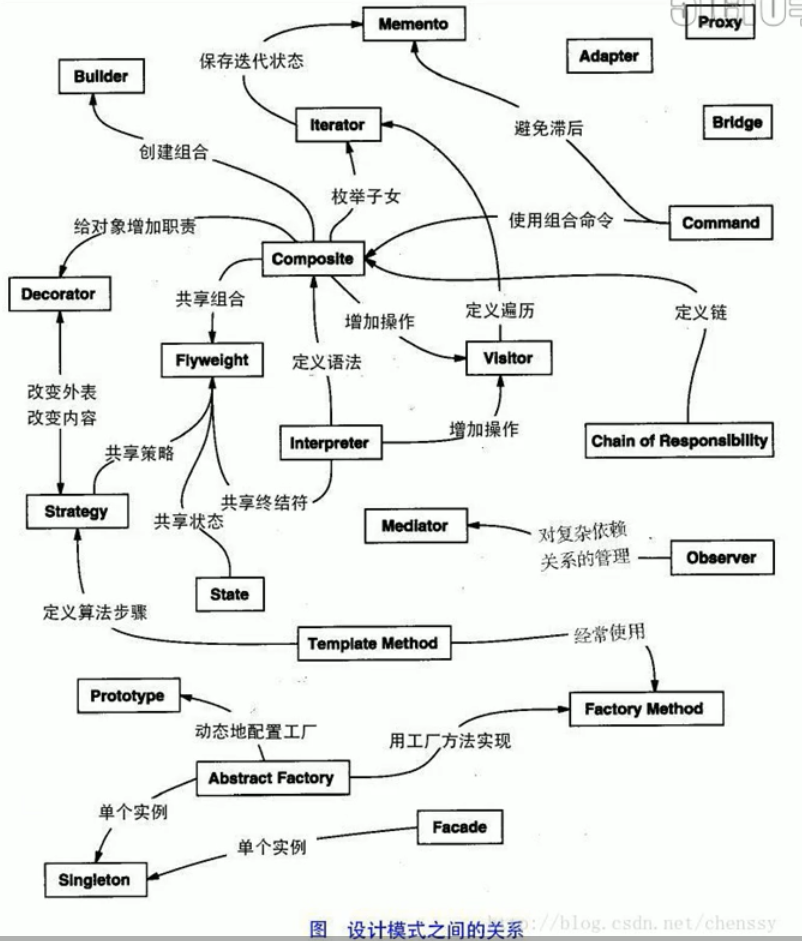
设计模式

# 注：设计模式并不是越高级越好。要选择最适合的，能使用轻量级完成的不要使用重量级。

# 1 模式简介与设计模式原则

## 模式介绍





## 设计模式原则

1. 单一原则：类和接口职责需要单一，降低耦合。
2. 开闭原则：接口和抽象类的使用，能达到对外扩展开发，对内修改关闭。
3. 里氏替换原则：开闭是抽象，里氏替换是实现。
4. 依赖倒置原则：依赖于抽象，不依赖与具体。（面向接口编程）。
5. 接口隔离原则：使用多个隔离接口比单个接口好。降耦合，降依赖，面向接口。
6. 合成复用原则：尽量使用组合聚合，少用继承。
7. 迪米特法则（最少知道原则）：一个实体里尽量少于其他实体发生交互作用。降耦合，减少深度与宽度。

# 2 创建型设计模式

**作用**：用于类创建。

**思想**：具体类封装（抽象出父类）。隐藏具体类的实现（多态）。

**创建模型分两种**：

1. 对象创建模型：对象的创建一部分推迟到另一个对象创建（如IOC注入）。
2. 类创建模型：对象的创建推迟到子类（继承与多态）。

**创建型设计模式有以下几种**：

1. **简单工厂模式**:所有产品子类继承产品抽象父类。工厂类提供工厂生产方法，通过传入参数生成对应的类实例（分支判断+多态）。使用场景较少。

## 2.1 简单工厂模式（静态工厂方法模式）

简单工厂模式又称为静态工厂方法模式，属于对象创建型模型。所有产品子类继承产品抽象父类。工厂类提供静态工厂方法，调用工厂方法时通过传入参数生成对应的类实例（分支判断+多态）。当有新类时继承父类，并在工厂方法添加分支（类多的话复杂性太高，适用子类少的时候）。该模式使用场景较少。

**使用场景**：创建多种对象，而且这些对象有极大的相似点。如果对象个数比较少且固定。可以考虑使用简单工厂模式。

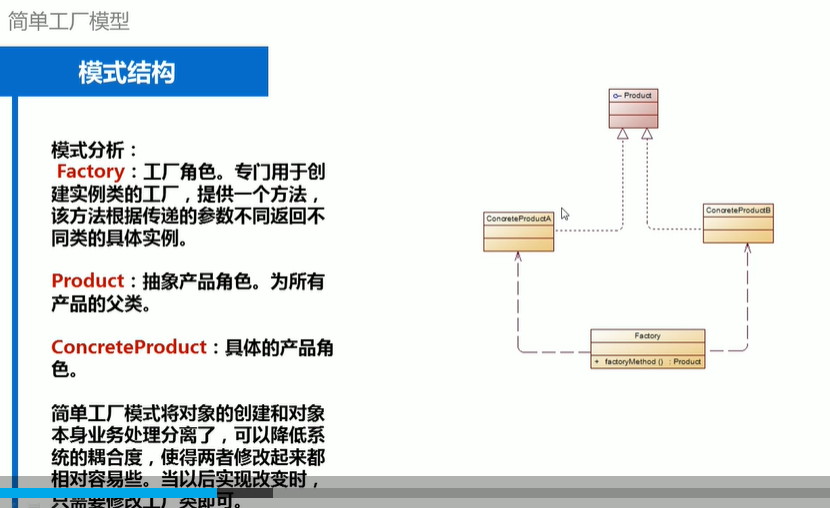
**简单工厂模式优缺点：**

**优点：**

1. 职责分割：有由专门的工厂类创建对象。
2. 客户端无需知道具体类名称（多态）。
3. 引入配置文件,使用参数选择具体类的实现，提高了代码灵活性（参数分支判断）。

**缺点：**

1. 工厂类创建集中了所有产品的创建逻辑，一旦不能工作，影响很大。
2. 增加类个数，一定程度增加了系统复杂性。
3. 扩展困难，添加新产品需要修改工厂逻辑，产品多的时候没法维护。
4. 使用了静态工厂方法，造成工厂角色无法形成基于继承的等级结构。



**简单工厂模式例子：**

**代码：**

**1）抽象父类**，封装子类共有的方法。（缺点：不易于个别子类独有方法扩展，否则修改大，且不能通用）。

*/\*\*  
 \* Pizza类是一个抽象父类。抽象出所有披萨子类共有方法。  
  
 \*/*public abstract class Pizza {  
 */\*\*  
 \* 准备  
 \*/* public abstract void prepare();  
  
 */\*\*  
 \* 制作  
 \*/* public abstract void make();  
  
 */\*\*  
 \* 完成  
 \*/* public abstract void complete();  
  
}

*/\*\*  
 \* 牛奶对象抽象类,供牛奶子类继承  
 \*/*public abstract class AbstractMilk {  
 */\*\*  
 \* 准备  
 \*/* public abstract void prepare();  
  
 */\*\*  
 \* 制作  
 \*/* public abstract void make();  
  
 */\*\*  
 \* 完成  
 \*/* public abstract void complete();  
}

1. **各产品子类**，**继承各自产品族对应父类**，并重写方法。

*/\*\*  
 \* 肉类披萨  
 \*/*public class PizzaMeet extends Pizza{  
 @Override  
 public void prepare() {  
 System.*out*.println("准备Meet");  
 }  
  
 @Override  
 public void make() {  
 System.*out*.println("制作肉类披萨");  
 }  
  
 @Override  
 public void complete() {  
 System.*out*.println("肉类披萨制作完成");  
 }  
}

*/\*\*  
 \* 蔬菜披萨  
 \*/*public class PizzaVegg extends Pizza{  
 @Override  
 public void prepare() {  
 System.*out*.println("准备Vegg");  
 }  
  
 @Override  
 public void make() {  
 System.*out*.println("制作蔬菜披萨");  
 }  
  
 @Override  
 public void complete() {  
 System.*out*.println("蔬菜披萨制作完成");  
 }  
}

*/\*\*  
 \* 苹果牛奶  
 \*/*public class AppleMilk extends AbstractMilk {  
 @Override  
 public void prepare() {  
 System.*out*.println("准备牛奶");  
 }  
  
 @Override  
 public void make() {  
 System.*out*.println("加热牛奶");  
 }  
  
 @Override  
 public void complete() {  
 System.*out*.println("制作牛奶完成");  
 }  
}

1. **核心工厂接口（抽象各产品族创建核心方法）**

*/\*\*  
 \* 工厂接口，抽象出产品族（产品种类，如面包、牛奶、披萨等）各自的核心创建方法。  
 \* --该接口只负责产品族创建的抽象  
 \*/*public interface Factory {  
  
 */\*\*  
 \* 创建披萨对象抽象方法  
 \** ***@param*** *type 披萨类型  
 \** ***@return*** *具体披萨对象  
 \*/* Pizza createPizza(String type);  
  
 */\*\*  
 \* 创建牛奶对象抽象方法  
 \** ***@param*** *type 牛奶类型  
 \** ***@return*** *具体牛奶对象  
 \*/* AbstractMilk createMilk(String type);  
}

1. **产品族共有产品类型具体工厂（如：披萨和牛奶有冷与热两种），实现产品族核心工厂接口。**

*/\*\*  
 \* 热食类型工厂，实现产品族核心接口。完成各产品族热食物对象创建  
 \*/*public class HostFoodFactory implements Factory{  
  
 @Override  
 public Pizza createPizza(String type) {  
 //使用反射取代太多的switch与if判断代码块。后面添加新产品类型，不在需要修改  
 //Class类 代表类的实体，在运行的Java应用程序中表示类和接口  
 //Field类 代表类的成员变量（成员变量也称为类的属性）  
 //Method类 代表类的方法  
 //Constructor类 代表类的构造方法  
 Pizza pizza = null;  
 try {  
 //参考https://www.jianshu.com/p/9be58ee20dee  
 //首字母转大写，用于找类。  
 type = type.substring(0, 1).toUpperCase()+type.substring(1);  
 System.*out*.println("type = [" + type + "]");  
 //获取类Class。还可以使用  
 Class<?> pizzaClass = Class.*forName*("com.zlk.design.creation.abstractFactory." + "Pizza" + type);  
 //获取Class对应的类实例  
 pizza = (Pizza)pizzaClass.newInstance();  
 pizza.prepare();  
 pizza.make(); //热食需要加热制作  
 pizza.complete();  
 } catch (ClassNotFoundException e) {  
 e.printStackTrace();  
 } catch (IllegalAccessException e) {  
 e.printStackTrace();  
 } catch (InstantiationException e) {  
 e.printStackTrace();  
 }  
 /\*  
 Pizza pizza = null;  
 switch (type){  
 case "vegg":  
 pizza = new PizzaVegg();  
 break;  
 case "meet":  
 pizza = new PizzaVegg();  
 break;  
 default:  
 System.out.println("当前没有该类型披萨，你输入类型为："+type);  
 break;  
 }  
  
 if(pizza!=null){  
 pizza.prepare();  
 pizza.make(); //热食需要加热制作  
 pizza.complete();  
 }\*/  
 return pizza;  
 }  
  
 @Override  
 public AbstractMilk createMilk(String type) {  
 AbstractMilk abstractMilk = null;  
 switch (type){  
 case "apple":  
 abstractMilk = new AppleMilk();  
 break;  
 default:  
 System.*out*.println("当前没有该类型披萨，你输入类型为："+type);  
 break;  
 }  
  
 if(abstractMilk !=null){  
 abstractMilk.prepare();  
 abstractMilk.make(); //热食需要加热制作  
 abstractMilk.complete();  
 }  
 return abstractMilk;  
 }  
}

*/\*\*  
 \* 冷食类型工厂，实现产品族核心接口。完成各产品族冷食物对象创建  
 \*/*public class ClodFoodFactory implements Factory{  
  
 @Override  
 public Pizza createPizza(String type) {  
 Pizza pizza = null;  
 switch (type){  
 case "vegg":  
 pizza = new PizzaVegg();  
 break;  
 case "meet":  
 pizza = new PizzaVegg();  
 break;  
 default:  
 System.*out*.println("当前没有该类型披萨，你输入类型为："+type);  
 break;  
 }  
  
 if(pizza!=null){  
 pizza.prepare();  
 //pizza.make(); 冷食不需要加热制作  
 pizza.complete();  
 }  
 return pizza;  
 }  
  
 @Override  
 public AbstractMilk createMilk(String type) {  
 AbstractMilk abstractMilk = null;  
 switch (type){  
 case "apple":  
 abstractMilk = new AppleMilk();  
 break;  
 default:  
 System.*out*.println("当前没有该类型披萨，你输入类型为："+type);  
 break;  
 }  
  
 if(abstractMilk !=null){  
 abstractMilk.prepare();  
 //pizza.make(); 冷食不需要加热制作  
 abstractMilk.complete();  
 }  
 return abstractMilk;  
 }  
}

1. **客户类**，购买产品。

*/\*\*  
 \* 客户类  
 \*/*public class PizzaStore {  
 private SimplePizzaFactory simplePizzaFactory;  
  
 */\*\*IOC  
 \* 构造方法创建，创建PizzaStore传入参数创建simplePizzaFactory  
 \** ***@param*** *simplePizzaFactory 工厂实例  
 \*/* public PizzaStore(SimplePizzaFactory simplePizzaFactory){  
 this.simplePizzaFactory = simplePizzaFactory;  
 }  
 */\*\*  
 \*  
 \** ***@param*** *type 披萨类型  
 \*/* public void orderPizza(String type){  
 Pizza pizza = simplePizzaFactory.createPizza(type);  
 pizza.prepare();  
 pizza.make();  
 pizza.complete();  
 }  
}

**5）测试类**

*/\*\*  
 \* 测试类  
 \*/*public class Test {  
 public static void main(String[] args) {  
 SimplePizzaFactory factory = new SimplePizzaFactory();  
 PizzaStore pizzaStore = new PizzaStore(factory);  
 pizzaStore.orderPizza("meet");  
 System.*out*.println("---------");  
 pizzaStore.orderPizza("vegg");  
 }  
}

**结果输出：**

**准备Meet**

**制作肉类披萨**

**肉类披萨制作完成**

**---------**

**准备Vegg**

**制作蔬菜披萨**

**蔬菜披萨制作完成**

**总结：产品子类继承于抽象产品类。工厂类提供静态工厂方法生成具体的产品（使用分支判断与多态）。添加新产品时需要加入新的产品子类，静态工厂方法需要加新的判断分支。适用于产品子类个数少且比较固定的场景。**

## 2.2 工厂方法模式

工厂方法模式是简单工厂模式的改进。产品子类继承于抽象产品类。抽象出一个工厂接口（包含具体类创建的抽象方法），每个产品会有一个自己实现工厂接口的实现类来实现专门的类创建（类创建模型）。当新增产品时只需要添加对应子类与自己的工厂实现类即可，不用改动其他的工厂实现类。

**引入原因**：为了新增产品时不改动工厂方法（简单工厂类时需要改分支）导致别的产品不能正常使用。（新加入的类使用Spring Boot与Spring Cloud在生产环境中进行热部署、热加载、热启动。由于没有改动其他类（简单工厂类需要改分支），可以无缝部署项目，其他客户可以正常使用系统）。

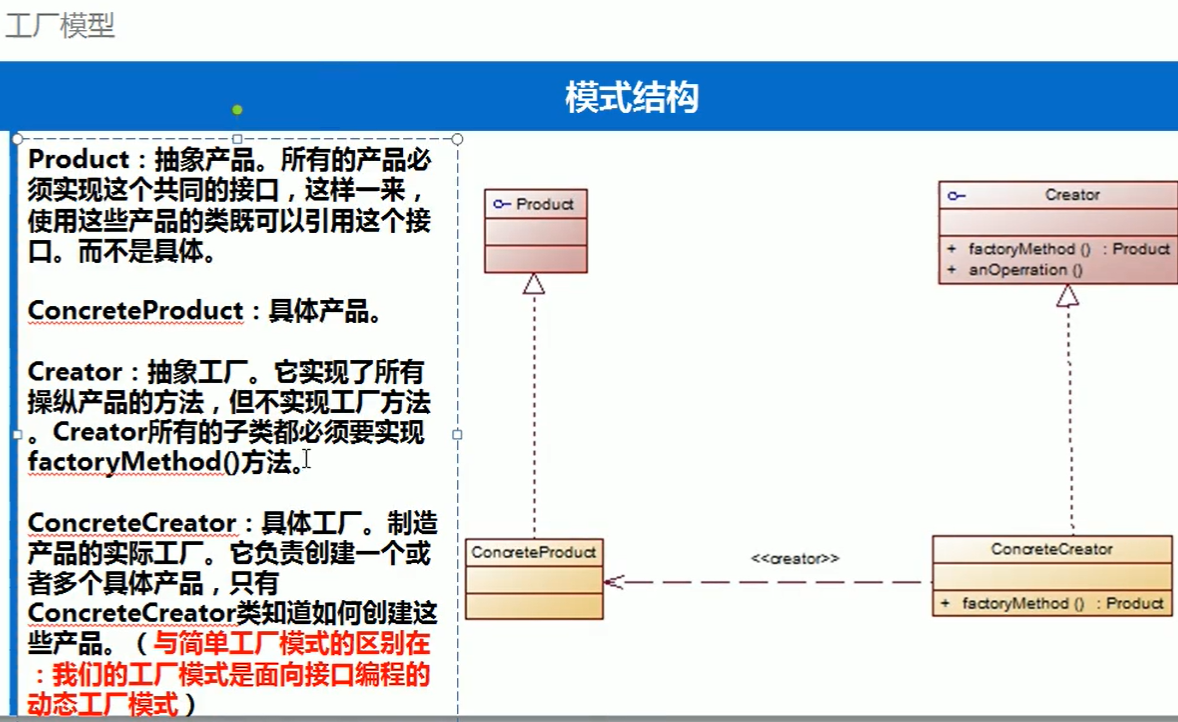
**使用场景**：创建多种对象，而且这些对象有极大的相似点。产品较多，后期加入新产品且不能影响线上系统正常使用。可以考虑使用工厂方法模式。

**优点：**

1. 只需要知道产品的具体工厂，不关心具体类的创建。
2. 新增产品时只需要添加对应子类与自己的工厂实现类即可，不用改动原工厂。满足”开闭原则”。

**缺点：**

每加一个产品，就需要加一个产品子类与一个工厂实现类。类的个数会成倍增加。增加了系统具体类的依赖。



**简单工厂模式例子：**

**代码：**

**1）抽象父类**，封装子类共有的方法。（缺点：不易于个别子类独有方法扩展，否则修改大，且不能通用）。

*/\*\*  
 \* Pizza类是一个抽象父类。抽象出所有披萨子类共有方法。  
 \* 局限性，个别种类特有方法没法添加*

*\*/*public abstract class Pizza {  
 */\*\*  
 \* 准备  
 \*/* public abstract void prepare();  
  
 */\*\*  
 \* 制作  
 \*/* public abstract void make();  
  
 */\*\*  
 \* 完成  
 \*/* public abstract void complete();  
  
}

1. **产品子类**，都继承上面的父类，并重写方法。

*/\*\*  
 \* 肉类披萨  
 \*/*public class PizzaMeet extends Pizza{  
 @Override  
 public void prepare() {  
 System.*out*.println("准备Meet");  
 }  
  
 @Override  
 public void make() {  
 System.*out*.println("制作肉类披萨");  
 }  
  
 @Override  
 public void complete() {  
 System.*out*.println("肉类披萨制作完成");  
 }  
}

*/\*\*  
 \* 蔬菜披萨  
 \*/*public class PizzaVegg extends Pizza{  
 @Override  
 public void prepare() {  
 System.*out*.println("准备Vegg");  
 }  
  
 @Override  
 public void make() {  
 System.*out*.println("制作蔬菜披萨");  
 }  
  
 @Override  
 public void complete() {  
 System.*out*.println("蔬菜披萨制作完成");  
 }  
}

1. **工厂接口**，抽象出工程类创建方法。子类继承时实现工厂方法（类创建模型）。

核心工厂不在负责产品创建，而是将具体产品创建推迟到子类完成。

*/\*\*  
 \* 工厂接口，抽象出工程类创建方法。  
 \* --当有新的产品子类创建时，只需要添加一个具体类去实现工厂接口就行。不需要修改原工厂。但是会多出很多工厂实现了类。*

*\*/*public interface Factory {  
 */\*\*  
 \* 创建披萨对象抽象方法  
 \** ***@return*** *披萨对象  
 \*/* public Pizza createPizza();  
 }

1. **工厂接口实现类**，每个子产品对应一个实现类，负责各自的产品对象创建，对其他产品与原工厂不修改，满足“开闭原则”。

*/\*\*  
 \* 购买肉类披萨类，实现工厂接口  
 \*/*public class PizzaStore1 implements Factory {  
 */\*\*  
 \* 购买披萨  
 \*/* public void orderPizza(){  
 Pizza pizza = createPizza();  
 pizza.prepare();  
 pizza.make();  
 pizza.complete();  
 }  
  
 */\*\*  
 \* 自己的产品创建有自己的工厂实现类完成。当新增产品时，不会影响别的产品与原工厂。  
 \** ***@return*** *Pizza具体产品  
 \*/* @Override  
 public Pizza createPizza() {  
 return new PizzaMeet();  
 }  
}

*/\*\*  
 \* 购蔬菜披萨类，实现工厂接口  
 \*/*public class PizzaStore2 implements Factory {  
 */\*\*  
 \* 购买披萨  
 \*/* public void orderPizza(){  
 Pizza pizza = createPizza();  
 pizza.prepare();  
 pizza.make();  
 pizza.complete();  
 }  
  
 */\*\*  
 \* 自己的产品创建有自己的工厂实现类完成。当新增产品时，不会影响别的产品与原工厂。  
 \** ***@return*** *Pizza具体产品  
 \*/* @Override  
 public Pizza createPizza() {  
 return new PizzaVegg();  
 }  
}

**5）测试类**

*/\*\*  
 \* 测试类  
 \*/*public class Test {  
 public static void main(String[] args) {  
 PizzaStore1 pizzaStore1 = new PizzaStore1();  
 pizzaStore1.orderPizza();  
 System.*out*.println("----------------------------------");  
 PizzaStore2 pizzaStore2 = new PizzaStore2();  
 pizzaStore2.orderPizza();  
 }  
}

**结果输出：**

**准备Meet**

**制作肉类披萨**

**肉类披萨制作完成**

**----------------------------------**

**准备Vegg**

**制作蔬菜披萨**

**蔬菜披萨制作完成**

**总结：产品子类继承于抽象产品类。提供一个核心工厂接口,每个子产品会有一个实现工厂接口的实现类来负责各自产品的创建（类创建模型）。添加新产品时需要加入新的产品子类，再添加一个具体类实现核心工厂接口即可，不需要修改原来存在的工厂。适用于产品子类个数多且后期新增产品较多的场景（热部署）。但是每个产品都需要工厂实现类，导致类个数成倍增加。**

**\*\*Jdk工厂方法模式位置**



**Spring底层如何使用工厂方法模式？IOC**

* 工厂方法模式在Spring底层被广泛的使用，陈某今天举个最常用的例子就是AbstractFactoryBean。
* 这个抽象工厂很熟悉了，这里不再讨论具体的作用。其实现了FactoryBean接口，这个接口中getObject()方法返回真正的Bean实例。

## 2.3 抽象工厂模式

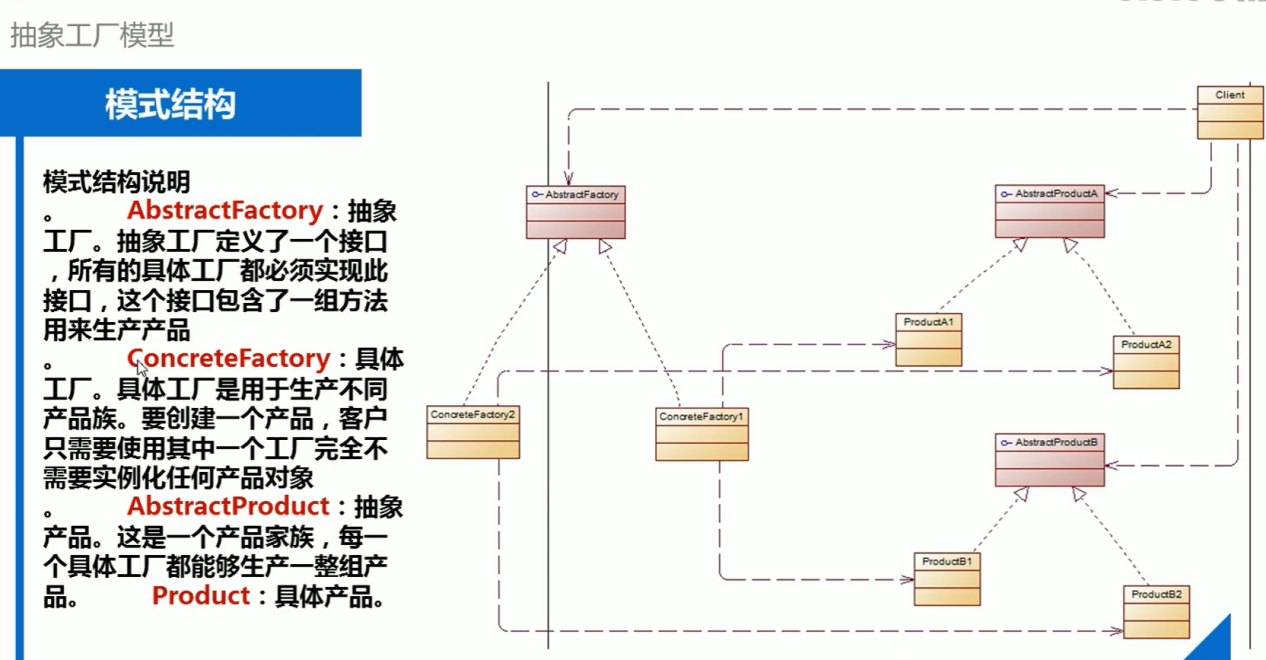
抽象工厂模式与工厂方法模式的区别在于，工厂方法模式适用于产品种类单一的情况。而抽象工厂适用于产品种类（产品族）多，且每个种类还有对应的产品等级。

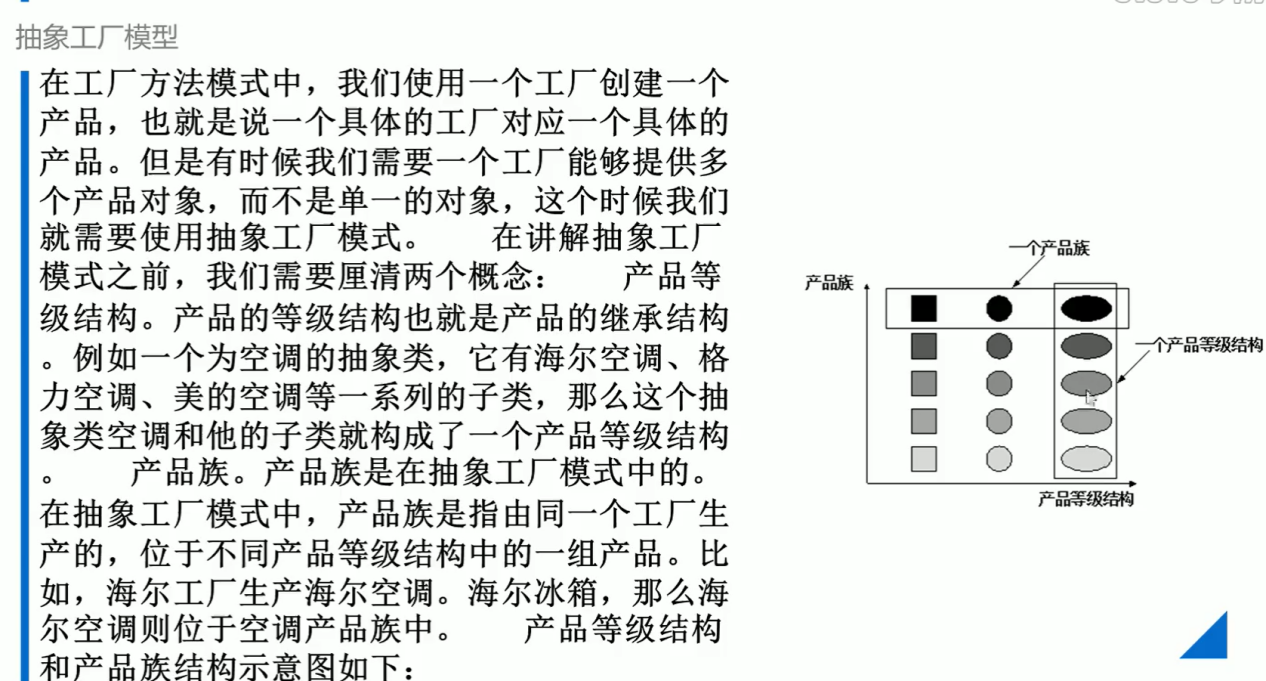
**抽象工厂**：对象创建模型 ，产品族中的每个产品有自己的抽象类。子产品继承各自产品族的抽象类。产品族抽象出一个只包含产品族的核心工厂接口。产品等级的具体实现类去实现产品族核心工厂接口来实现具体类创建与方法调用。

核心工厂接口的工厂方法以产品族来分（如：面包、披萨、牛奶），工厂实现类按产品等级（如：该类是产冷食物还是热食物），各产品等级的工厂实现类的工厂方法再负责具体类创建（如：是肉类披萨还是蔬菜披萨）。

**产品族**：如面包、牛奶、披萨等。

**产品等级**：如牛奶有冷牛奶与热牛奶。





**代码：**

**1）抽象父类，按产品族（产品种类，如面包、牛奶等）抽象出各自父类**，封装各自产品族子类共有的方法。

*/\*\*  
 \* Pizza类是一个抽象父类。抽象出所有披萨子类共有方法。  
 \* 局限性，个别种类特有方法没法添加（创建工厂时使用多态）。  
 \*/*public abstract class Pizza {  
 */\*\*  
 \* 准备  
 \*/* public abstract void prepare();  
  
 */\*\*  
 \* 制作  
 \*/* public abstract void make();  
  
 */\*\*  
 \* 完成  
 \*/* public abstract void complete();  
  
}

**2）产品子类**，都继承上面的父类，并重写方法。

*/\*\*  
 \* 肉类披萨  
 \*/*public class PizzaMeet extends Pizza{  
 @Override  
 public void prepare() {  
 System.*out*.println("准备Meet");  
 }  
  
 @Override  
 public void make() {  
 System.*out*.println("制作肉类披萨");  
 }  
  
 @Override  
 public void complete() {  
 System.*out*.println("肉类披萨制作完成");  
 }  
}

*/\*\*  
 \* 蔬菜披萨  
 \*/*public class PizzaVegg extends Pizza{  
 @Override  
 public void prepare() {  
 System.*out*.println("准备Vegg");  
 }  
  
 @Override  
 public void make() {  
 System.*out*.println("制作蔬菜披萨");  
 }  
  
 @Override  
 public void complete() {  
 System.*out*.println("蔬菜披萨制作完成");  
 }  
}

**3)工厂类**，通过对静态工厂方法传入参数生产对应的类实例(多态)。

*/\*\*  
 \* 工厂类，用于生产各种披萨  
 \* 通过传入参数生成对应的披萨对象。（多态）  
 \* --当有新的子类创建时，只需要子类继承父类，在工厂方法加一个判断分支即可。但是类多会增*

*\*加复杂性  
 \*/*public class SimplePizzaFactory {  
  
 */\*\*  
 \* 创建披萨对象  
 \** ***@param*** *type 披萨种类  
 \** ***@return*** *披萨对象  
 \*/* public Pizza createPizza(String type){  
 if("meet".equals(type)){  
 return new PizzaMeet();  
 }  
 if("vegg".equals(type)){  
 return new PizzaVegg();  
 }  
 return null;  
 }  
}

**4）客户类**，购买产品。

*/\*\*  
 \* 购买类  
 \*/*public class Store {  
  
 */\*\*  
 \* 消费  
 \** ***@param*** *foodType 产品族（牛奶milk、披萨pizza）  
 \** ***@param*** *proType 产品类型（那种牛奶或者那种披萨）  
 \** ***@param*** *type 种类（冷的clod或者热的host）  
 \*/* public static void consume(String foodType, String proType , String type){  
 if ("pizza".equals(foodType)){  
 if ("clod".equals(type)){  
 if ("vegg".equals(proType)){  
 ClodFoodFactory clodFoodFactory = new ClodFoodFactory();  
 clodFoodFactory.createPizza("vegg");  
 }  
 }  
 if ("host".equals(type)){  
 if ("vegg".equals(proType)){  
 HostFoodFactory hostFoodFactory = new HostFoodFactory();  
 hostFoodFactory.createPizza("vegg");  
 }  
 }  
 }  
 }  
  
 public static void main(String[] args) {  
 *consume*("pizza","vegg","host");  
 }  
}

}

**结果输出**：

type = [Vegg]

准备Vegg

制作蔬菜披萨

蔬菜披萨制作完成

**jdk位置**



Spring 就是一个抽象工厂，可以创建普通的bean ,也可以创建bean工厂。

## 2.4 单例模式

所谓单例，就是整个程序有且仅有一个实例。该类负责创建自己的对象，同时确保只有一个对象被创建。在Java，一般常用在工具类的实现或创建对象需要消耗资源。常用场景线程池、缓存与对话框等。  
**特点**

* 类构造器私有
* 持有自己类型的属性
* 对外提供获取实例的静态方法

**优缺点：**

**优点:**

1. 节省了系统资源；

(2)封装了类的唯一实例。

**缺点:**

(1)没有抽象类，不方便扩展；

(2)职责过重（方法提供对象时可能需要很多逻辑与组装），一定程度上违背“单一职责”原则。

**懒汉模式**  
线程不安全，延迟初始化，严格意义上不是不是单例模式

public class Singleton {

private static Singleton instance;

private Singleton (){}

public static Singleton getInstance() {

//该处多个线程进行初始化时可能导致创建多个对象。线程不安全

if (instance == null) {

instance = new Singleton();

}

return instance;

} }

**饿汉模式**  
线程安全，比较常用，但容易产生垃圾，因为一开始就初始化

public class Singleton {

private static Singleton instance = new Singleton();

private Singleton (){}

public static Singleton getInstance() {

return instance;

} }

**双重锁模式**  
线程安全，延迟初始化。这种方式采用双锁机制，安全且在多线程情况下能保持高性能。

public class Singleton {

private volatile static Singleton singleton;

private Singleton (){}

public static Singleton getSingleton() {

if (singleton == null) {

synchronized (Singleton.class) {

if (singleton == null) {

singleton = new Singleton();

}

}

}

return singleton;

} }

双重检查模式，进行了两次的判断，第一次是为了避免不要的实例，第二次是为了进行同步，避免多线程问题。由于singleton=new Singleton()对象的创建在JVM中可能会进行重排序，在多线程访问下存在风险，使用volatile修饰signleton实例变量有效，解决该问题。

**静态内部类单例模式（推荐）**

public class Singleton {

private Singleton(){

}

public static Singleton getInstance(){

return Inner.instance;

}

private static class Inner {

private static final Singleton instance = new Singleton();

} }

只有第一次调用getInstance方法时，虚拟机才加载 Inner 并初始化instance ，只有一个线程可以获得对象的初始化锁，其他线程无法进行初始化，保证对象的唯一性。目前此方式是所有单例模式中最推荐的模式，但具体还是根据项目选择。  
**枚举单例模式**

public enum Singleton {

INSTANCE;}

默认枚举实例的创建是线程安全的，并且在任何情况下都是单例。实际上

* 枚举类隐藏了私有的构造器。
* 枚举类的域 是相应类型的一个实例对象  
  那么枚举类型日常用例是这样子的：

public enum Singleton {

INSTANCE

//doSomething 该实例支持的行为

//可以省略此方法，通过Singleton.INSTANCE进行操作

public static Singleton get Instance() {

return Singleton.INSTANCE;

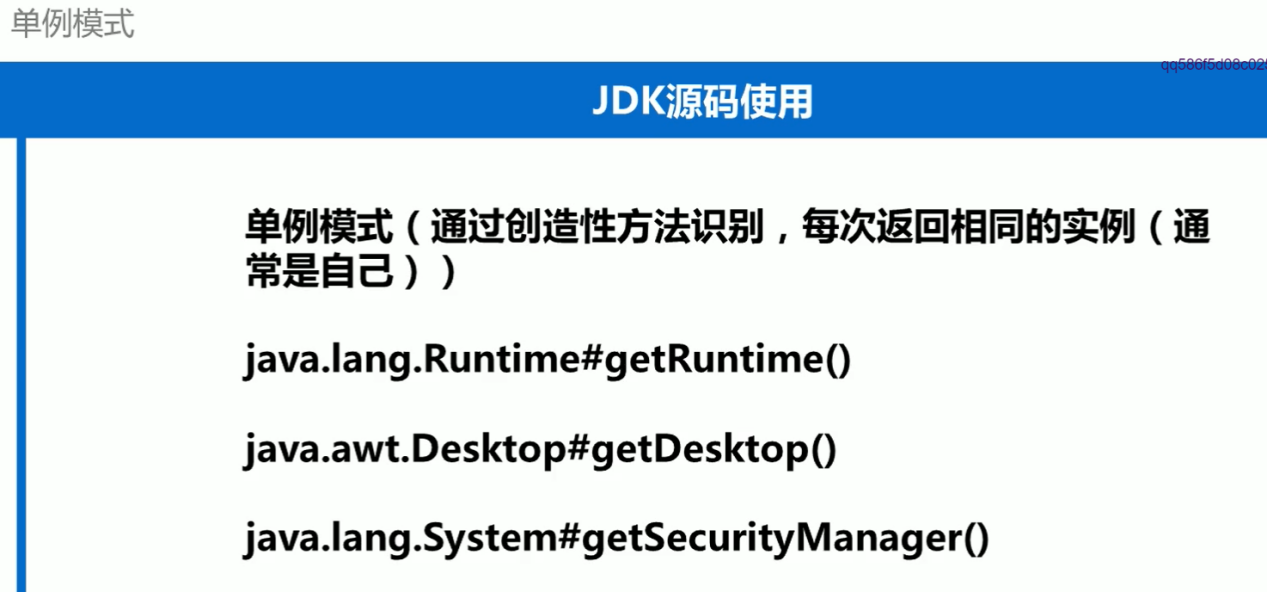
}}

枚举单例模式在《Effective Java》中推荐的单例模式之一。但枚举实例在日常开发是很少使用的，就是很简单以导致可读性较差。  
在以上所有的单例模式中，推荐静态内部类单例模式。主要是非常直观，即保证线程安全又保证唯一性。  
众所周知，单例模式是创建型模式，都会新建一个实例。那么一个重要的问题就是反序列化。当实例被写入到文件到反序列化成实例时，我们需要重写readResolve方法，以让实例唯一。

private Object readResolve() throws ObjectStreamException{

return singleton;}

**Jdk源码位置：**



**spring** 中加载单例的过程都是在BeanFactory接口中定义的getBean（…）

## 2.5 建造者模式

创造者模式的就是由**创造者接口**抽象产品各部件的创建方法。**具体的创造者**实现创造者接口后实现产品各部件细节创建（字段组装与实例返回）。**指挥者**会通过拿到的具体的创造者去调用对应的部件创建方法并返回具体产品对象实例。目的是使用相同的创建过程可以创建不同的类。

**适用场景**：产品又较多的相同点，其他部分相似。创建具体产品实例时候逻辑繁杂（各字段的来源需要很多逻辑处理）。

**建造者模式与工厂设计模式异同**：

**相同点**：都是用来创建具体类实例。

**不同点**：工厂设计模式只关心具体类的创建。而建造者模式更关心类创建的细节（如字段的来源与组装逻辑等）。

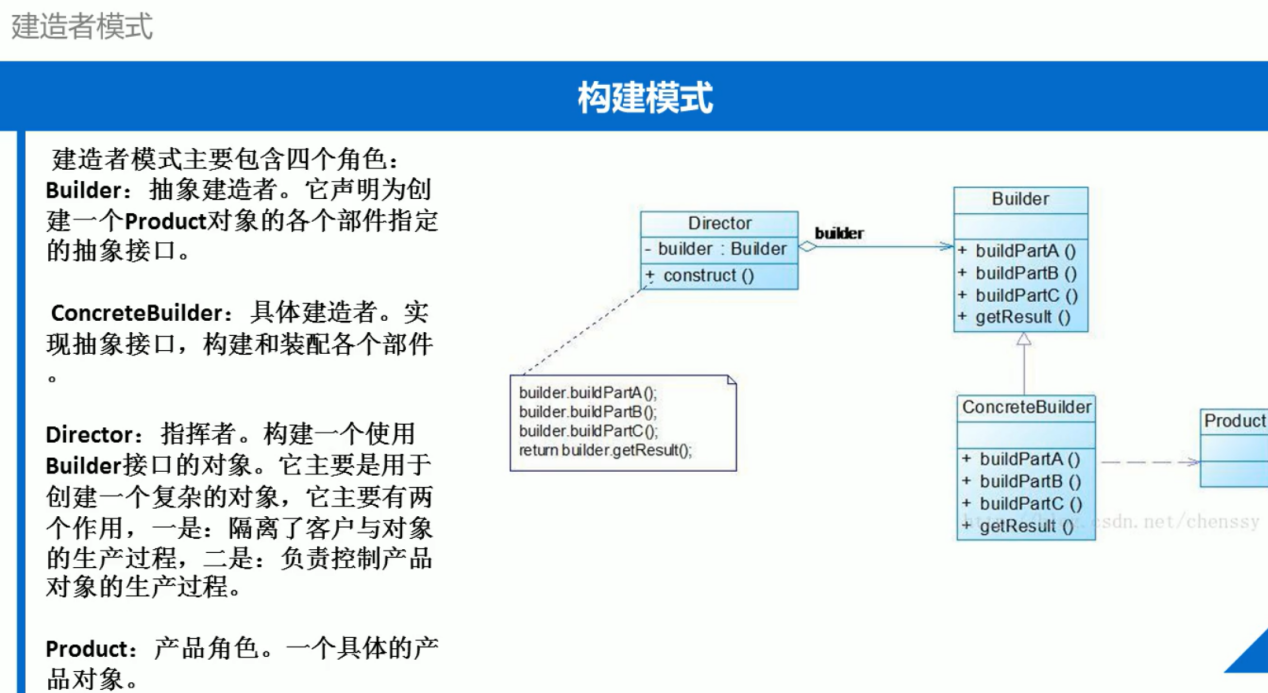
**创造者模式优缺点**：

优点：(1)复杂的产品创建进行了分离。

(2)每一个具体的创造者相对独立，与其他具体创造者无关。（面向接口编程）

缺点:(1)产品间要有较多相同点，其他部分相似。差异大不实用，导致使用范围受限。

(2)产品内部的变化复杂，可能需要定义很多的具体建造者。这种变化会导致系统很庞大。



**代码：**

1. **产品类Product**。一般创建时，字段来源需要复杂的逻辑组装。

*/\*\* product:产品角色，具体的产品对象  
 \* 机器人类  
 \*/*public class Robot {  
 //实际使用时字段可能定义的对象，需要复杂的组装。  
 private String head;  
 private String body;  
 private String legs;  
  
 public String getHead() {  
 return head;  
 }  
  
 public void setHead(String head) {  
 this.head = head;  
 }  
  
 public String getBody() {  
 return body;  
 }  
  
 public void setBody(String body) {  
 this.body = body;  
 }  
  
 public String getLegs() {  
 return legs;  
 }  
  
 public void setLegs(String legs) {  
 this.legs = legs;  
 }  
}

1. **Builder 建造者接口**。提供产品各部件细节创建方法与对象返回方法。

*/\*\*  
 \* Builder 机器人抽象构造者接口，相当于工厂设计模式的抽象工厂。  
 \* 区别在于：工厂设计模式关注具体类创建，不关注类创建细节。建造者模式更关注创建类的细节。  
 \*/*public interface RobotBuilder {  
 //细节，字段复制的来源，组装形式等。如果是对象将更为复杂  
 public void createhead();  
 //细节  
 public void createbody();  
 //细节  
 public void createLegs();  
  
 */\*\*  
 \* 相同点：工厂设计模式与构造者模式相同点都是用来创建对象  
 \* 不同点：工厂设计模式关注具体类创建，不关注类创建细节（工厂模式只有一个创建方法）。建造者模式更关注创建类的细节（比工厂模式多了字段的具体创建方法）。  
 \** ***@return*** *Robot 具体对象  
 \*/* public Robot createRobot();  
  
}

3）**ConcreteBuilder具体构造者类。实现类创建个部件细节方法与产品对象方法。**

*/\*\*  
 \* ConcreteBuilder具体构造者类，实现抽象的构造者接口。进行类的具体组装并返回类实例。  
 \*/*public class SmartRobotBuilderImpl implements RobotBuilder {  
 Robot robot = new Robot();  
 @Override  
 public void createhead() {  
 //可能有许多的逻辑与工艺才能得到该值  
 String head = "Head";  
 String smartChip = "Smart";  
 //将其装载，\*\*别人调用不在需要了解具体细节\*\*。  
 String com = head + smartChip;  
 robot.setHead(com);  
 }  
  
 @Override  
 public void createbody() {  
 robot.setBody("body");  
 }  
  
 @Override  
 public void createLegs() {  
 String leg = "Legs";  
 String cutLeg = "Sinele";  
 String finallegs = leg + cutLeg;  
 robot.setLegs(finallegs);  
 }  
  
 @Override  
 public Robot createRobot() {  
 return robot;  
 }  
}

4）**Director 指挥者**，通过传入的具体创造者类调用具体创建方法后返回产品对象实例。

*/\*\*  
 \* Director 指挥官，指挥具体的类创建  
 \*/*public class Worker {  
 */\*\*  
 \* 传入需要创建的构造者生产具体的产品实例  
 \** ***@param*** *builder 具体构造者  
 \*/* public Robot builderRobot(RobotBuilder builder){  
 //调用具体构造方法中的方法对 对象进行具体创建  
 builder.createhead();  
 builder.createbody();  
 builder.createLegs();  
 return builder.createRobot();  
 }  
}

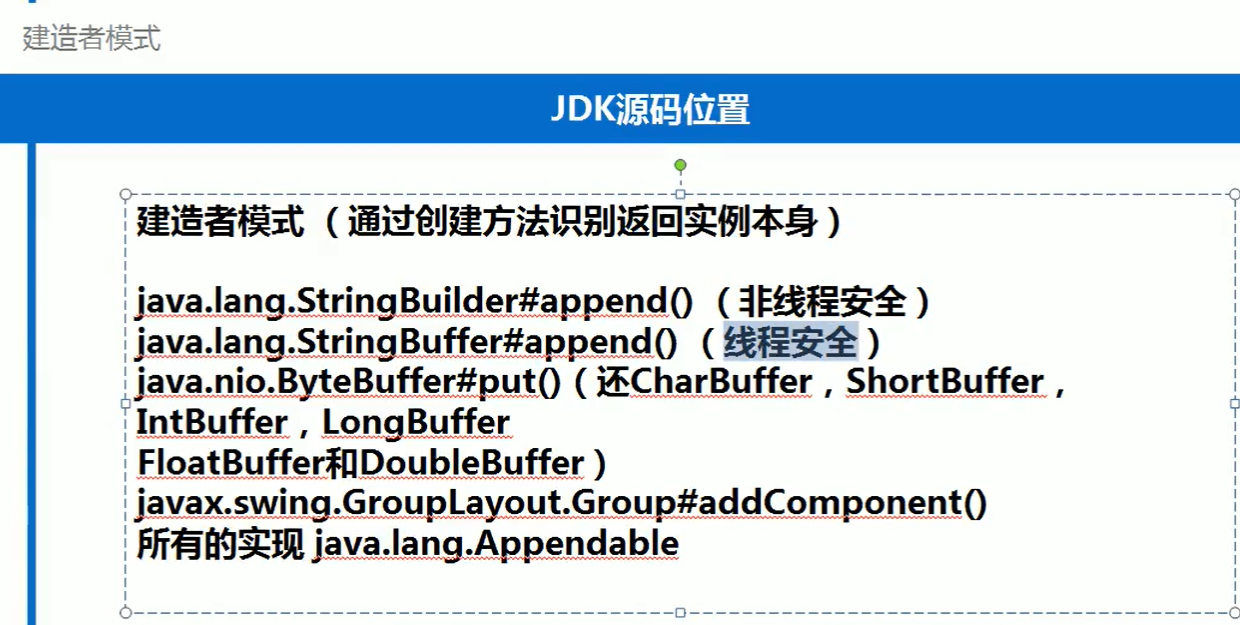
**5）测试**

public class Test {  
 public static void main(String[] args) {  
 Worker worker = new Worker();  
 //具体的构造类传给指挥者去控制具体产品的创建。需要时直接掉用。用户不比知道对象的具体创建  
 RobotBuilder builder = new SmartRobotBuilderImpl();  
 Robot robot = worker.builderRobot(builder);  
 System.*out*.println("产品的head:"+robot.getHead());  
 }  
}

**结果：**

**产品的head:HeadSmart**

**JDK源码位置：**



**框架源码位置：**

Mybatis中的SqlSessionFactoryBuilder.

springMVC通过UriComponentsBuilder构建UriComponents类.

## 2.6 原型模式

原型模式就是进行对象克隆。

**浅克隆**：克隆对象的引用地址。需要实现java,lang.Cloneable.

**深克隆**：复制类的成员变量。需要实现java.io.Serializable.基于序列化与反序列化实现。

适用场景：1对象创建成本较大。2要保持对象状态，对象变化小的时候。3克隆比构造方法使用方便时。

**优缺点：**

**优点：**

1. 创建复杂对象时，提高效率。
2. 深克隆保持对象状态。
3. 原型模式提供了能序列化的价格。

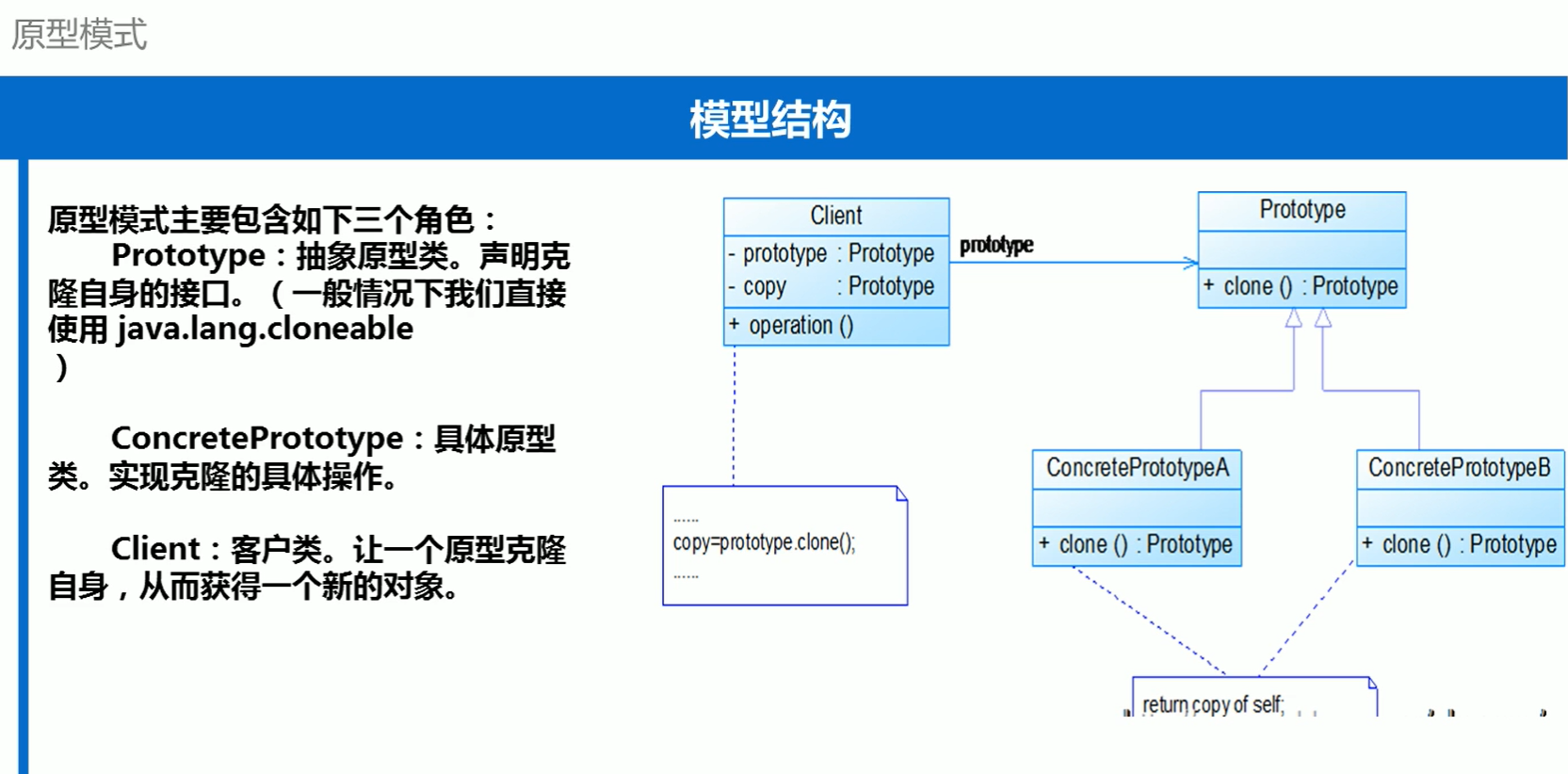
**缺点：**

(1)深度克隆代码可能较复杂。

(2)每个类都要有克隆方法，对原有类就行改造时，一定程度违反了“开闭原则”

**注：**

1. 克隆对象和原对象不是同一个，x.clone()!=x;
2. 克隆对象类型和原来对象类型一样，x.clone().getClass() = x.getClass().



#### 2.6.1 浅克隆

浅克隆的对象必须实现克隆的对象必须实现java.lang.Cloneable接口。重写clone()方法。

浅克隆实际克隆的是对象的引用地址。克隆修改时可能会对原克隆对象带来修改（引用类型如果不是new以后进行Set()赋值）。

School类

public class School{  
 private String name;  
  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
}

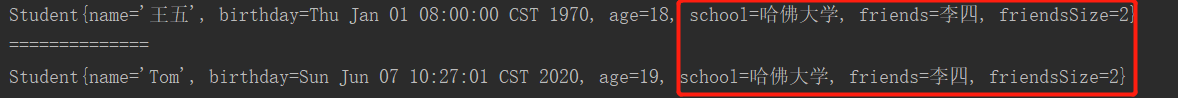
Student类

*/\*\*  
 \* 浅克隆使用.clone()方法进行克隆。克隆的对象必须实现java.lang.Cloneable接口。重写clone()方法。  
 \*/*public class Student implements Cloneable {  
 private String name;  
 private Date birthday;  
 private Integer age;  
  
 //以下两个属性需要特别关注  
 private School school;  
 private ArrayList<String> friends;  
  
 */\*\*  
 \*  
 \** ***@return*** *\** ***@throws*** *CloneNotSupportedException  
 \*/* @Override  
 protected Student clone() throws CloneNotSupportedException {  
 return (Student)super.clone();  
 }  
  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
  
 public Date getBirthday() {  
 return birthday;  
 }  
  
 public void setBirthday(Date birthday) {  
 this.birthday = birthday;  
 }  
  
 public Integer getAge() {  
 return age;  
 }  
  
 public void setAge(Integer age) {  
 this.age = age;  
 }  
  
 public School getSchool() {  
 return school;  
 }  
  
 public void setSchool(School school) {  
 this.school = school;  
 }  
  
 public ArrayList<String> getFriends() {  
 return friends;  
 }  
  
 public void setFriends(ArrayList<String> friends) {  
 this.friends = friends;  
 }  
  
 @Override  
 public String toString() {  
 //只是为了测试  
 return "Student{" +  
 "name='" + name + '\'' +  
 ", birthday=" + birthday +  
 ", age=" + age +  
 ", school=" + school.getName() +  
 ", friends=" + friends.get(0) +  
 ", friendsSize=" + friends.size() +  
 '}';  
 }  
  
}

测试类

public class Test {  
 public static void main(String[] args) throws CloneNotSupportedException {  
 Student stu1 = new Student();  
 stu1.setName("王五");  
 stu1.setBirthday(new Date(0));  
 stu1.setAge(18);  
 School school = new School();  
 school.setName("伦敦大学");  
 stu1.setSchool(school);  
 ArrayList<String> array = new ArrayList<>();  
 array.add("李四");  
 stu1.setFriends(array);  
  
 //\_\_\_\_\_\_\_\_\_\_\_\_\_\_浅克隆（克隆的是地址引用）,修改对象属性需要不改变原来对象实现克隆，需要重新new 对象。不建议这么做\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
 Student stu2 = stu1.clone();  
 stu2.setName("Tom");//String是fianl类型，这个地方相当于new String。stu1不变,stu2改变成功。  
 stu2.setBirthday(new Date());//new Date。stu1不变,stu2改变成功。  
 stu2.setAge(19);//final修饰，效果和基本类型相似，stu1不变,stu2改变成功。。  
 stu2.getSchool().setName("哈佛大学");//对象引用，没有new对象。导致stu1改变，stu2改变成功。造成原对象出问题。  
 stu2.getFriends().add("李六");//对象引用，没有new list.导致直接在原来list上进行了add。  
  
 System.*out*.println(stu1.toString());  
 System.*out*.println("==============");  
 System.*out*.println(stu2.toString());  
 }  
}

测试结果：



#### 2.6.2 普通深克隆

浅克隆的对象必须实现克隆的对象必须实现java.lang.Cloneable接口。重写clone()方法。

对非java的引用对象（如list,array等）。可以实现java.lang.Cloneable接口。重写clone()方法。

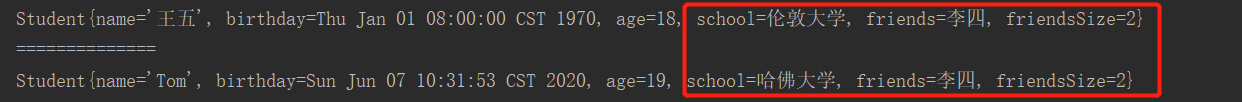
public class School implements Cloneable{  
 private String name;  
  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
  
 @Override  
 protected School clone() throws CloneNotSupportedException {  
 return (School)super.clone();  
 }  
}

*/\*\*  
 \* 浅克隆使用.clone()方法进行克隆。克隆的对象必须实现java.lang.Cloneable接口。重写clone()方法。  
 \*/*public class Student implements Cloneable {  
 private String name;  
 private Date birthday;  
 private Integer age;  
  
 //以下两个属性需要特别关注  
 private School school;  
 private ArrayList<String> friends;  
  
 */\*\*  
 \*  
 \** ***@return*** *\** ***@throws*** *CloneNotSupportedException  
 \*/* @Override  
 protected Student clone() throws CloneNotSupportedException {  
 //方式一 浅克隆  
 //return (Student)super.clone();  
  
 //方式二 普通深克隆 对非java的引用对象（如list,array等）。可以实现java.lang.Cloneable接口。重写clone()方法。  
 // 通过以下实现克隆，但是修改较大。  
 Student student = (Student) super.clone();  
 student.school= school.clone();  
 school.clone();  
 return student;  
 }  
  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
  
 public Date getBirthday() {  
 return birthday;  
 }  
  
 public void setBirthday(Date birthday) {  
 this.birthday = birthday;  
 }  
  
 public Integer getAge() {  
 return age;  
 }  
  
 public void setAge(Integer age) {  
 this.age = age;  
 }  
  
 public School getSchool() {  
 return school;  
 }  
  
 public void setSchool(School school) {  
 this.school = school;  
 }  
  
 public ArrayList<String> getFriends() {  
 return friends;  
 }  
  
 public void setFriends(ArrayList<String> friends) {  
 this.friends = friends;  
 }  
  
 @Override  
 public String toString() {  
 //只是为了测试  
 return "Student{" +  
 "name='" + name + '\'' +  
 ", birthday=" + birthday +  
 ", age=" + age +  
 ", school=" + school.getName() +  
 ", friends=" + friends.get(0) +  
 ", friendsSize=" + friends.size() +  
 '}';  
 }  
  
}

public class Test {  
 public static void main(String[] args) throws CloneNotSupportedException {  
 Student stu1 = new Student();  
 stu1.setName("王五");  
 stu1.setBirthday(new Date(0));  
 stu1.setAge(18);  
 School school = new School();  
 school.setName("伦敦大学");  
 stu1.setSchool(school);  
 ArrayList<String> array = new ArrayList<>();  
 array.add("李四");  
 stu1.setFriends(array);  
  
 //\_\_\_\_\_\_\_\_\_\_\_\_\_\_普通深度克隆

Student stu2 = stu1.clone();  
 stu2.setName("Tom");//String是fianl类型，这个地方相当于new String。stu1不变,stu2改变成功。  
 stu2.setBirthday(new Date());//new Date。stu1不变,stu2改变成功。  
 stu2.setAge(19);//final修饰，效果和基本类型相似，stu1不变,stu2改变成功。。  
 stu2.getSchool().setName("哈佛大学");//对象引用，做了克隆处理。stu1不变，stu2改变成功。原对象没有出问题。  
 stu2.getFriends().add("李六");//对象引用，没有new list.导致直接在原来list上进行了add。  
  
 System.*out*.println(stu1.toString());  
 System.*out*.println("==============");  
 System.*out*.println(stu2.toString());  
 }  
}

测试结果：



#### 2.6.3深克隆

深克隆克隆是复制成员变量。深克隆的对象不需要实现java.lang.Cloneable接口。但是需要实现序列化Serializable 接口。提供克隆方法把序列化的对象反序列化后复制给对象。

School类

public class School implements Serializable {  
 private String name;  
  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
}

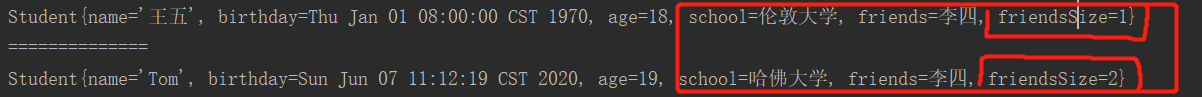
Student类

*/\*\*  
 \* 深克隆直接复制成员变量,需要实现序列化Cloneable。使用序列化与反序列化读IO流  
 \*/*public class Student implements Serializable {  
 private String name;  
 private Date birthday;  
 private Integer age;  
  
 //以下两个属性需要特别关注  
 private School school;  
 private ArrayList<String> friends;  
  
 */\*\*  
 \*  
 \** ***@return*** *\** ***@throws*** *Exception  
 \*/* public Student deepClone() {  
 //方式三 深度克隆  
 Student student = null;  
 try {  
 //序列化将对象写道IO流  
 ByteArrayOutputStream out = new ByteArrayOutputStream();  
 ObjectOutputStream oo = new ObjectOutputStream(out);  
 oo.writeObject(this);  
  
 //从IO流将对象反序列化出来  
 ByteArrayInputStream in = new ByteArrayInputStream(out.toByteArray());  
 ObjectInputStream oi = new ObjectInputStream(in);  
 student = (Student)oi.readObject();  
 } catch (IOException e) {  
 e.printStackTrace();  
 } catch (ClassNotFoundException e) {  
 e.printStackTrace();  
 }  
  
 return student ;  
 }  
  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
  
 public Date getBirthday() {  
 return birthday;  
 }  
  
 public void setBirthday(Date birthday) {  
 this.birthday = birthday;  
 }  
  
 public Integer getAge() {  
 return age;  
 }  
  
 public void setAge(Integer age) {  
 this.age = age;  
 }  
  
 public School getSchool() {  
 return school;  
 }  
  
 public void setSchool(School school) {  
 this.school = school;  
 }  
  
 public ArrayList<String> getFriends() {  
 return friends;  
 }  
  
 public void setFriends(ArrayList<String> friends) {  
 this.friends = friends;  
 }  
  
 @Override  
 public String toString() {  
 //只是为了测试  
 return "Student{" +  
 "name='" + name + '\'' +  
 ", birthday=" + birthday +  
 ", age=" + age +  
 ", school=" + school.getName() +  
 ", friends=" + friends.get(0) +  
 ", friendsSize=" + friends.size() +  
 '}';  
 }  
  
}

测试类

public class Test {  
 public static void main(String[] args) throws CloneNotSupportedException {  
 Student stu1 = new Student();  
 stu1.setName("王五");  
 stu1.setBirthday(new Date(0));  
 stu1.setAge(18);  
 School school = new School();  
 school.setName("伦敦大学");  
 stu1.setSchool(school);  
 ArrayList<String> array = new ArrayList<>();  
 array.add("李四");  
 stu1.setFriends(array);  
  
 //深度克隆，复制前后对象互不影响  
 Student stu2 = stu1.deepClone();  
 stu2.setName("Tom");  
 stu2.setBirthday(new Date());  
 stu2.setAge(19);  
 stu2.getSchool().setName("哈佛大学");  
 stu2.getFriends().add("李六");  
  
 System.*out*.println(stu1.toString());  
 System.*out*.println("==============");  
 System.*out*.println(stu2.toString());  
 }  
}

测试结果：



# 3 结构型模式