

创建型模式

创建型模式(Creational Pattern)关注对象的创建过程

创建型模式对类的实例化过程进行了抽象，能够将软件模块中对象的创建和对象的使用分离，对用户隐藏了类的实例的创建细节

创建型模式描述如何**将对象的创建和使用分离**，让用户在使用对象时无须关心对象的创建细节，从而降低系统的耦合度，让设计方案更易于修改和扩展

创建型模式关注点：

- 创建什么(What)
- 由谁创建(Who)
- 何时创建(When)

一、简单工厂模式

1、含义

- 简单工厂模式又叫静态方法模式（因为工厂类定义了一个静态方法）
- 现实生活中，工厂是负责生产产品的；同样在设计模式中，简单工厂模式我们可以理解为负责生产对象的一个类，称为“工厂类”。

2、解决的问题

将“类实例化的操作”与“使用对象的操作”分开，让使用者不用知道具体参数就可以实例化出所需要的“产品”类，从而避免了在客户端代码中显式指定，实现了解耦。

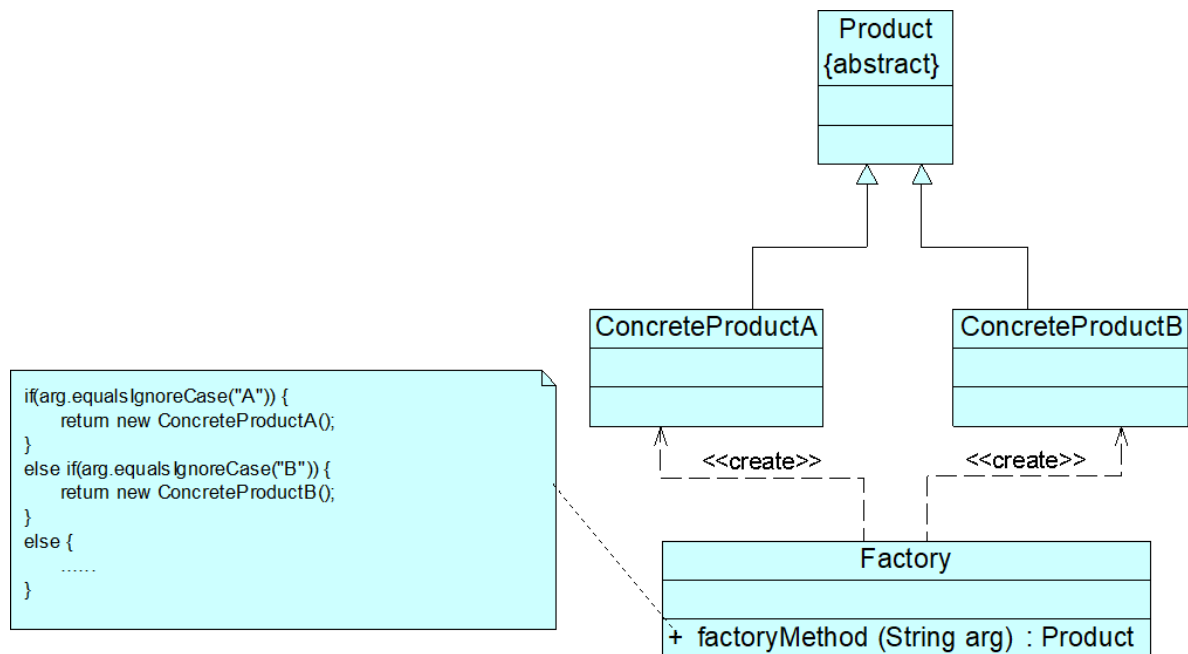
即使使用者可直接消费产品而不需要知道其生产的细节

3、模式原理

3.1 模式组成

组成（角色）	关系	作用
抽象产品（Product）	具体产品的父类	描述产品的公共接口
具体产品（Concrete Product）	抽象产品的子类；工厂类创建的目标类	描述生产的具体产品
工厂（Creator）	被外界调用	根据传入不同参数从而创建不同具体产品类的实例

3.2 UML类图



3.3 原理代码

抽象产品类代码：

```
1 public abstract class Product {
2     //所有产品类的公共业务方法
3     public void methodSame() {
4         //公共方法的实现
5     }
6
7     //声明抽象业务方法
8     public abstract void methodDiff();
9 }
```

具体产品类代码：

```
1 public class ConcreteProduct extends Product{
2     //实现业务方法
3     public void methodDiff() {
4         //业务方法的实现
5     }
6 }
```

静态工厂：

```
1 public class Factory {
2     //静态工厂方法
3     public static Product getProduct(String arg) {
4         Product product = null;
5         if (arg.equalsIgnoreCase("A")) {
6             product = new ConcreteProductA();
7             //初始化设置product
8         }
9         else if (arg.equalsIgnoreCase("B")) {
10            product = new ConcreteProductB();
11            //初始化设置product
12        }
13    }
14 }
```

```
13         return product;
14     }
15 }
```

客户端代码：

```
1 public class Client {
2     public static void main(String args[]) {
3         Product product;
4         product = Factory.getProduct("A"); //通过工厂类创建产品对象
5         product.methodSame();
6         product.methodDiff();
7     }
8 }
```

4、实例

4.1 实例概况

- 背景：小成有一个塑料生产厂，用来做塑料加工生意
- 目的：最近推出了3个产品，小成希望使用**简单工厂模式**实现3款产品的生产

4.2 使用步骤

实现代码如下：

步骤1. 创建抽象产品类，定义具体产品的公共接口

```
1 abstract class Product{
2     public abstract void Show();
3 }
```

步骤2. 创建具体产品类（继承抽象产品类），定义生产的具体产品

```
1 //具体产品类A
2 class ProductA extends Product{
3
4     @Override
5     public void Show() {
6         System.out.println("生产出了产品A");
7     }
8 }
9
10 //具体产品类B
11 class ProductB extends Product{
12
13     @Override
14     public void Show() {
15         System.out.println("生产出了产品C");
16     }
17 }
18
19 //具体产品类C
20 class ProductC extends Product{
21
22     @Override
23     public void Show() {
```

```

24     System.out.println("生产出了产品C");
25     }
26 }

```

步骤3. 创建工厂类，通过创建静态方法从而根据传入不同参数创建不同具体产品类的实例

```

1  class Factory {
2      public static Product Manufacture(String ProductName){
3          //工厂类里用switch语句控制生产哪种商品；
4          //使用者只需要调用工厂类的静态方法就可以实现产品类的实例化。
5          switch (ProductName){
6              case "A":
7                  return new ProductA();
8
9              case "B":
10                 return new ProductB();
11
12             case "C":
13                 return new ProductC();
14
15             default:
16                 return null;
17
18         }
19     }
20 }

```

步骤4. 外界通过调用工厂类的静态方法，传入不同参数从而创建不同具体产品类的实例

```

1  //工厂产品生产流程
2  public class SimpleFactoryPattern {
3      public static void main(String[] args){
4          Factory mFactory = new Factory();
5
6          //客户要产品A
7          try {
8              //调用工厂类的静态方法 & 传入不同参数从而创建产品实例
9              mFactory.Manufacture("A").Show();
10         } catch (NullPointerException e){
11             System.out.println("没有这一类产品");
12         }
13
14         //客户要产品B
15         try {
16             mFactory.Manufacture("B").Show();
17         } catch (NullPointerException e){
18             System.out.println("没有这一类产品");
19         }
20
21         //客户要产品C
22         try {
23             mFactory.Manufacture("C").Show();
24         } catch (NullPointerException e){
25             System.out.println("没有这一类产品");
26         }
27     }

```

```
28         //客户要产品D
29         try {
30             mFactory.Manufacture("D").Show();
31         } catch (NullPointerException e){
32             System.out.println("没有这一类产品");
33         }
34     }
35 }
```

结果输出：

```
1  生产出了产品A
2  生产出了产品C
3  生产出了产品C
4  没有这一类产品
```

5、简单工厂模式的优缺点与适用环境

模式优点：

1. 实现了对象创建和使用的分离
2. 客户端无须知道所创建的具体产品类的类名，只需要知道具体产品类所对应的参数即可
3. 通过引入配置文件，可以在不修改任何客户端代码的情况下更换和增加新的具体产品类，在一定程度上提高了系统的灵活性

模式缺点：

1. 工厂类集中了所有产品的创建逻辑，职责过重，一旦不能正常工作，整个系统都要受到影响
2. 增加系统中类的个数（引入了新的工厂类），增加了系统的复杂度和理解难度
3. 系统扩展困难，一旦添加新产品不得不修改工厂逻辑
4. 由于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，工厂类不能得到很好地扩展

模式适用环境：

1. 工厂类负责创建的对象比较少，由于创建的对象较少，不会造成工厂方法中的业务逻辑太过复杂
2. 客户端只知道传入工厂类的参数，对于如何创建对象并不关心

二、工厂方法模式

简单工厂模式存在一系列问题：

- 工厂类集中了所有实例（产品）的创建逻辑，一旦这个工厂不能正常工作，整个系统都会受到影响；
- 违背“开放 - 关闭原则”，一旦添加新产品就不得不修改工厂类的逻辑，这样就会造成工厂逻辑过于复杂。
- 简单工厂模式由于使用了静态工厂方法，静态方法不能被继承和重写，会造成工厂角色无法形成基于继承的等级结构。

1. 介绍

1.1 定义

工厂方法模式，又称工厂模式、多态工厂模式和虚拟构造器模式，通过定义工厂父类负责定义创建对象的公共接口，而子类则负责生成具体的对象。

1.2 主要作用

将类的实例化（具体产品的创建）延迟到工厂类的子类（具体工厂）中完成，即由子类来决定应该实例化（创建）哪一个类。

1.3 解决的问题

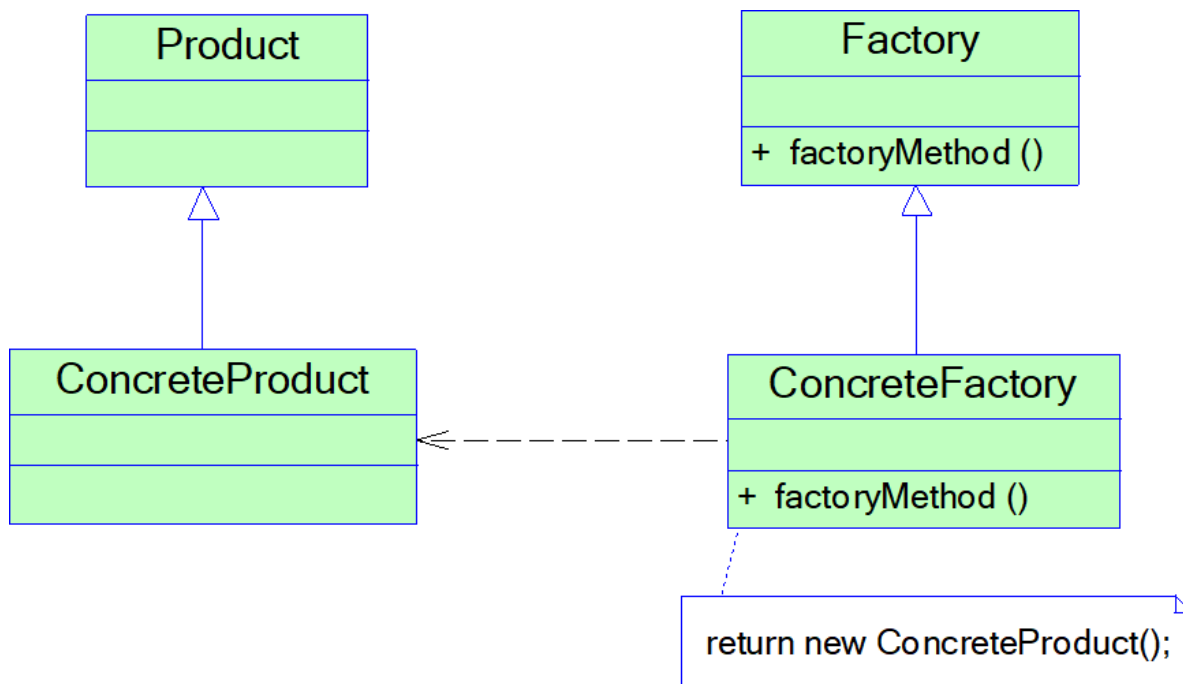
工厂一旦需要生产新产品就需要修改工厂类的方法逻辑，违背了“开放 - 关闭原则

1、即简单工厂模式的缺点

2、之所以可以解决简单工厂的问题，是因为工厂方法模式把具体产品的创建推迟到工厂类的子类（具体工厂）中，此时工厂类不再负责所有产品的创建，而只是给出具体工厂必须实现的接口，这样工厂方法模式在添加新产品的时候就不修改工厂类逻辑而是添加新的工厂子类，符合开放封闭原则，克服了简单工厂模式中缺点

2. 模式原理

2.1 UML类图



2.2 模式组成

组成（角色）	关系	作用
抽象产品 (Product)	具体产品的父类	描述具体产品的公共接口
具体产品 (Concrete Product)	抽象产品的子类；工厂类创建的目标类	描述生产的具体产品
抽象工厂 (Creator)	具体工厂的父类	描述具体工厂的公共接口
具体工厂 (Concrete Creator)	抽象工厂的子类；被外界调用	描述具体工厂；实现FactoryMethod工厂方法创建产品的实例

2.3 代码

抽象工厂类

```
1 public interface Factory {
2     public Product factoryMethod();
3 }
```

具体工厂类

```
1 public class ConcreteFactory implements Factory {
2     public Product factoryMethod() {
3         return new ConcreteProduct();
4     }
5 }
```

3. 实例讲解

接下来我用一个实例来对工厂方法模式进行更深一步的介绍。

3.1 实例概况

- 背景：小成有一间塑料加工厂（仅生产A类产品）；随着客户需求的变化，客户需要生产B类产品；
- 冲突：改变原有塑料加工厂的配置和变化非常困难，假设下一次客户需要再发生变化，再次改变将增大非常大的成本；
- 解决方案：小成决定置办**塑料分厂B**来生产B类产品；

即工厂方法模式

3.2 使用步骤

步骤1：创建**抽象工厂类**，定义具体工厂的公共接口

```
1 abstract class Factory{
2     public abstract Product Manufacture();
3 }
```

步骤2：创建**抽象产品类**，定义具体产品的公共接口；

```
1 abstract class Product{
2     public abstract void Show();
3 }
```

步骤3：创建**具体产品类**（继承抽象产品类），定义生产的具体产品；

```
1 //具体产品A类
2 class ProductA extends Product{
3     @Override
4     public void Show() {
5         System.out.println("生产出了产品A");
6     }
7 }
8
9 //具体产品B类
10 class ProductB extends Product{
11
12     @Override
13     public void Show() {
14         System.out.println("生产出了产品B");
15     }
16 }
```

```
15     }
16 }
```

步骤4：创建具体工厂类（继承抽象工厂类），定义创建对应具体产品实例的方法；

```
1  //工厂A类 - 生产A类产品
2  class FactoryA extends Factory{
3      @Override
4      public Product Manufacture() {
5          return new ProductA();
6      }
7  }
8
9  //工厂B类 - 生产B类产品
10 class FactoryB extends Factory{
11     @Override
12     public Product Manufacture() {
13         return new ProductB();
14     }
15 }
```

步骤5：外界通过调用具体工厂类的方法，从而创建不同**具体产品类的实例**

```
1  //生产工作流程
2  public class FactoryPattern {
3      public static void main(String[] args){
4          //客户要产品A
5          FactoryA mFactoryA = new FactoryA();
6          mFactoryA.Manufacture().Show();
7
8          //客户要产品B
9          FactoryB mFactoryB = new FactoryB();
10         mFactoryB.Manufacture().Show();
11     }
12 }
13
```

结果：

```
1  生产出了产品A
2  生产出了产品B
```

4. 工厂方法模式的优缺点与适用环境

模式优点

1. 工厂方法用来创建客户所需要的产品，同时还向客户隐藏了哪种具体产品类将被实例化这一细节
2. 能够让工厂自主确定创建何种产品对象，而如何创建这个对象的细节则完全封装在具体工厂内部
3. 在系统中加入新产品时，完全符合开闭原则

模式缺点

1. 系统中类的个数将成对增加，在一定程度上增加了系统的复杂度，会给系统带来一些额外的开销
2. 增加了系统的抽象性和理解难度

模式适用环境

1. 客户端不知道它所需要的对象的类（客户端不需要知道具体产品类的类名，只需要知道所对应的工厂即可，具体产品对象由具体工厂类创建）
2. 抽象工厂类通过其子类来指定创建哪个对象

5. 应用场景

1. 当一个类不知道它所需要的对象的类时，在工厂方法模式中，客户端不需要知道具体产品类的类名，只需要知道所对应的工厂即可；当一个类希望通过其子类来指定创建对象时
2. 在工厂方法模式中，对于抽象工厂类只需要提供一个创建产品的接口，而由其子类来确定具体要创建的对象，利用面向对象的多态性和里氏代换原则，在程序运行时，子类对象将覆盖父类对象，从而使得系统更容易扩展。
3. 将创建对象的任务委托给多个工厂子类中的某一个，客户端在使用时可以无须关心是哪一个工厂子类创建产品子类，需要时再动态指定，可将具体工厂类的类名存储在配置文件或数据库中。

三、抽象工厂模式

- 发现工厂方法模式存在一个严重的问题：一个具体工厂只能创建一类产品；
- 而在实际过程中，一个工厂往往需要生产多类产品；
- 为了解决上述的问题，我们又使用了一种新的设计模式：抽象工厂模式。

1. 介绍

1.1 定义

抽象工厂模式，即Abstract Factory Pattern，提供一个创建一系列相关或相互依赖对象的接口，而无须指定它们具体的类；具体的工厂负责实现具体的产品实例。

抽象工厂模式与工厂方法模式最大的区别：抽象工厂中每个工厂可以创建多种类的产品；而工厂方法每个工厂只能创建一类

1.2 主要作用

允许使用抽象的接口来创建一组相关产品，而不需要知道或关心实际生产出的具体产品是什么，这样就可以从具体产品中被解耦。

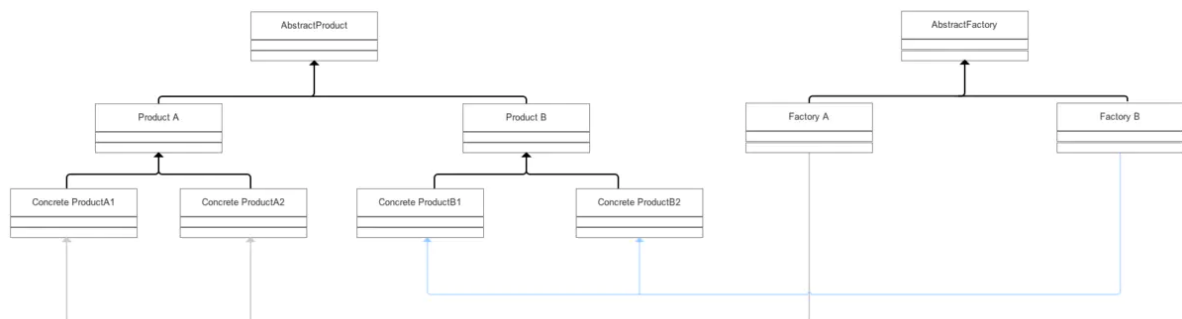
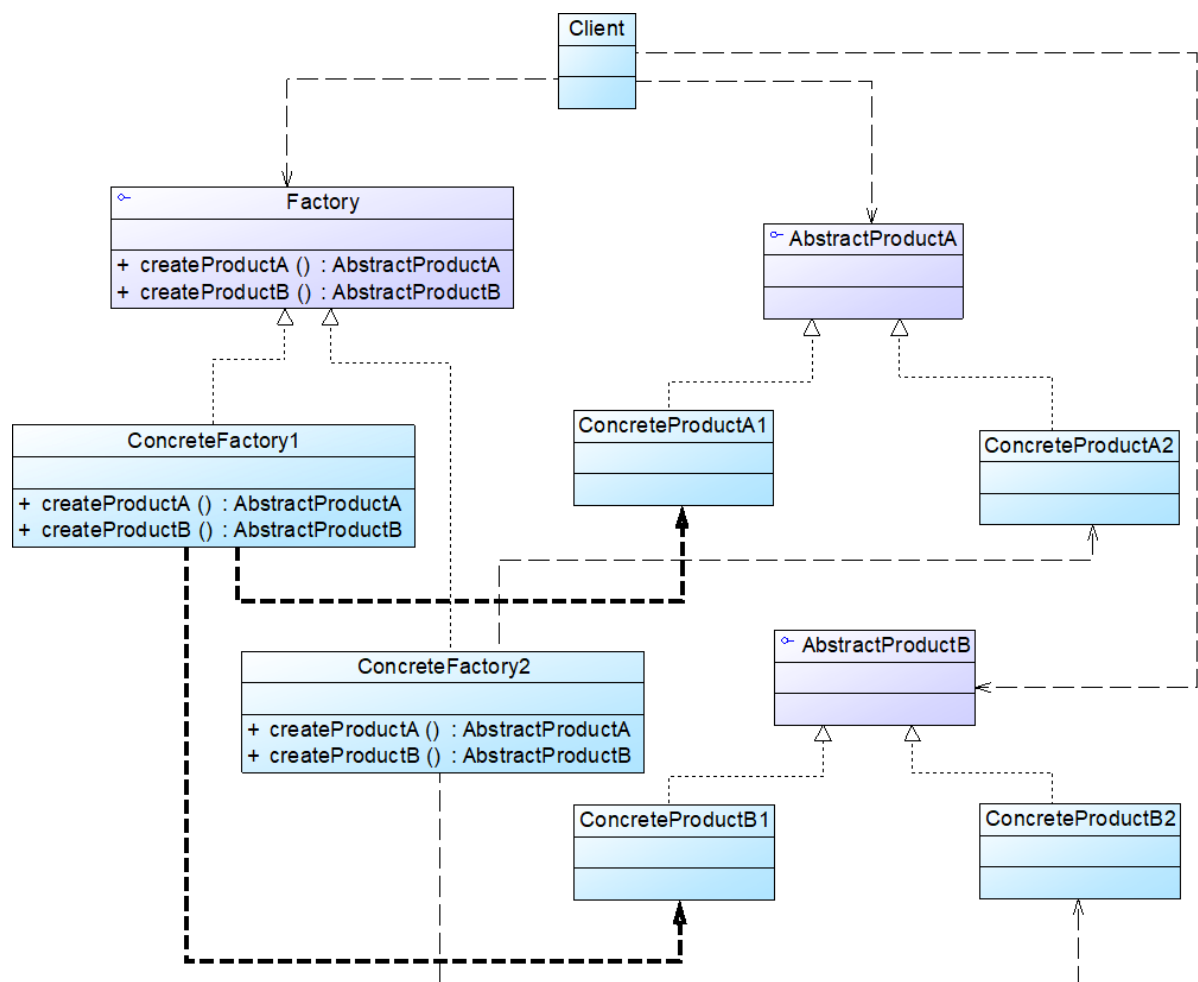
1.3 解决的问题

每个工厂只能创建一类产品

即工厂方法模式的缺点

2. 模式原理

2.1 UML类图



2.2 模式组成

组成（角色）	关系	作用
抽象产品族 (AbstractProduct)	抽象产品的父类	描述抽象产品的公共接口
抽象产品 (Product)	具体产品的父类	描述具体产品的公共接口
具体产品 (Concrete Product)	抽象产品的子类; 工厂类创建的目标类	描述生产的具体产品
抽象工厂 (Creator)	具体工厂的父类	描述具体工厂的公共接口
具体工厂 (Concrete Creator)	抽象工厂的子类; 被外界调用	描述具体工厂; 实现FactoryMethod工厂方法创建产品的实例

2.3 代码

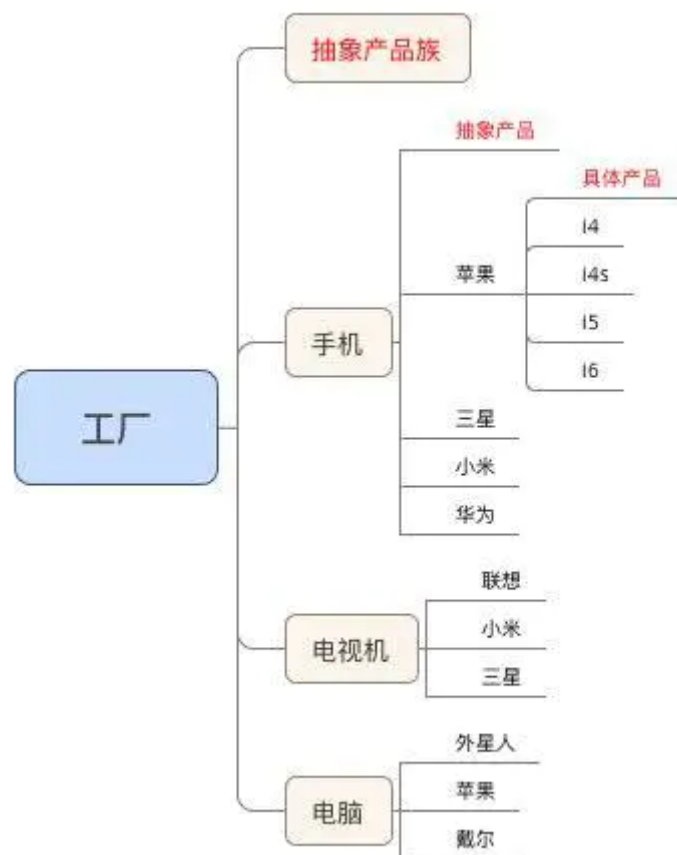
抽象工厂类

```
1 public interface AbstractFactory {  
2     public AbstractProductA createProductA(); //工厂方法一  
3     public AbstractProductB createProductB(); //工厂方法二  
4     .....  
5 }
```

具体工厂类

```
1 public class ConcreteFactory1 implements AbstractFactory {  
2     //工厂方法一  
3     public AbstractProductA createProductA() {  
4         return new ConcreteProductA1();  
5     }  
6  
7     //工厂方法二  
8     public AbstractProductB createProductB() {  
9         return new ConcreteProductB1();  
10    }
```

如何理解抽象产品族、抽象产品和具体产品的区别呢？请看下图



3. 实例讲解

接下来我用一个实例来对抽象工厂模式进行更深一步的介绍。

3.1 实例概况

- 背景：小成有两间塑料加工厂（A厂仅生产容器类产品；B厂仅生产模具类产品）；随着客户需求的变化，A厂所在地的客户需要也模具类产品，B厂所在地的客户也需要容器类产品；
- 冲突：没有资源（资金+租位）在当地分别开设多一家注塑分厂
- 解决方案：在原有的两家塑料厂里增设生产需求的功能，即A厂能生产容器+模具产品；B厂间能生产模具+容器产品。

即抽象工厂模式

3.2 使用步骤

步骤1： 创建**抽象工厂类**，定义具体工厂的公共接口

```
1 abstract class Factory{
2     public abstract Product ManufactureContainer();
3     public abstract Product ManufactureMould();
4 }
```

步骤2： 创建**抽象产品族类**，定义具体产品的公共接口；

```
1 abstract class AbstractProduct{
2     public abstract void Show();
3 }
```

步骤3： 创建**抽象产品类**，定义具体产品的公共接口；

```
1 //容器产品抽象类
2 abstract class ContainerProduct extends AbstractProduct{
3     @Override
4     public abstract void Show();
5 }
6
7 //模具产品抽象类
8 abstract class MouldProduct extends AbstractProduct{
9     @Override
10    public abstract void Show();
11 }
```

步骤4： 创建**具体产品类**（继承抽象产品类），定义生产的具体产品；

```
1 //容器产品A类
2 class ContainerProductA extends ContainerProduct{
3     @Override
4     public void Show() {
5         System.out.println("生产出了容器产品A");
6     }
7 }
8
9 //容器产品B类
10 class ContainerProductB extends ContainerProduct{
11     @Override
```

```

12     public void Show() {
13         System.out.println("生产出了容器产品B");
14     }
15 }
16
17 //模具产品A类
18 class MouldProductA extends MouldProduct{
19
20     @Override
21     public void Show() {
22         System.out.println("生产出了模具产品A");
23     }
24 }
25
26 //模具产品B类
27 class MouldProductB extends MouldProduct{
28
29     @Override
30     public void Show() {
31         System.out.println("生产出了模具产品B");
32     }
33 }

```

步骤5: 创建具体工厂类（继承抽象工厂类），定义创建对应具体产品实例的方法；

```

1 //A厂 - 生产模具+容器产品
2 class FactoryA extends Factory{
3
4     @Override
5     public Product ManufactureContainer() {
6         return new ContainerProductA();
7     }
8
9     @Override
10    public Product ManufactureMould() {
11        return new MouldProductA();
12    }
13 }
14
15 //B厂 - 生产模具+容器产品
16 class FactoryB extends Factory{
17
18     @Override
19     public Product ManufactureContainer() {
20        return new ContainerProductB();
21    }
22
23     @Override
24     public Product ManufactureMould() {
25        return new MouldProductB();
26    }
27 }

```

步骤6: 客户端通过实例化具体的工厂类，并调用其创建不同目标产品的方法创建不同具体产品类的实例

```

1 //生产工作流程

```

```

2 public class AbstractFactoryPattern {
3     public static void main(String[] args){
4         FactoryA mFactoryA = new FactoryA();
5         FactoryB mFactoryB = new FactoryB();
6         //A厂当地客户需要容器产品A
7         mFactoryA.ManufactureContainer().Show();
8         //A厂当地客户需要模具产品A
9         mFactoryA.ManufactureMould().Show();
10
11        //B厂当地客户需要容器产品B
12        mFactoryB.ManufactureContainer().Show();
13        //B厂当地客户需要模具产品B
14        mFactoryB.ManufactureMould().Show();
15
16    }
17 }

```

结果:

```

1 生产出了容器产品A
2 生产出了容器产品B
3 生产出了模具产品A
4 生产出了模具产品B

```

4. 抽象工厂模式的优缺点与适用环境

模式优点:

1. 隔离了具体类的生成, 使得客户端并不需要知道什么被创建
2. 当一个产品族中的多个对象被设计成一起工作时, 它能够保证客户端始终只使用同一个产品族中的对象
3. 增加新的产品族很方便, 无须修改已有系统, 符合开闭原则

模式缺点:

增加新的产品等级结构麻烦, 需要对原有系统进行较大的修改, 甚至需要修改抽象层代码, 这显然会带来较大的不便, 违背了开闭原则

模式适用环境:

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
2. 系统中有多于一个的产品族, 但每次只使用其中某一产品族
3. 属于同一个产品族的产品将在一起使用, 这一约束必须在系统的设计中体现出来
4. 产品等级结构稳定, 在设计完成之后不会向系统中增加新的产品等级结构或者删除已有的产品等级结构

四、建造者模式

1. 简介

1.1 模式说明

隐藏创建对象的建造过程 & 细节, 使得用户在不知对象的建造过程 & 细节的情况下, 就可直接创建复杂的对象

用户只需要给出指定复杂对象的类型和内容; 建造者模式负责按顺序创建复杂对象 (把内部的建造过程和细节隐藏起来)

1.2 作用（解决的问题）

1. 降低创建复杂对象的复杂度
2. 隔离了创建对象的构建过程 & 表示

从而：

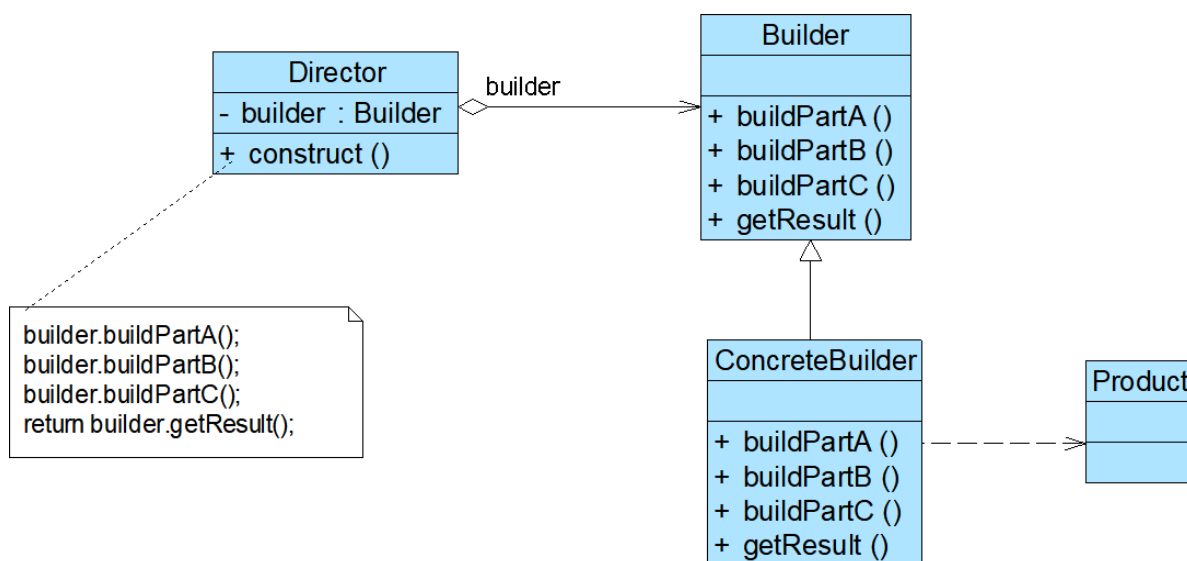
- 方便用户创建复杂的对象（不需要知道实现过程）
- 代码复用性 & 封装性（将对象构建过程和细节进行封装 & 复用）

例子：造汽车 & 买汽车。

1. 工厂（建造者模式）：负责制造汽车（组装过程和细节在工厂内）
2. 汽车购买者（用户）：你只需要说出你需要的型号（对象的类型和内容），然后直接购买就可以使用了（不需要知道汽车是怎么组装的（车轮、车门、发动机、方向盘等等））

2. 模式原理

2.1 UML类图



2.2 模式讲解

1. 指挥者（Director）直接和客户（Client）进行需求沟通；
2. 沟通后指挥者将客户创建产品的需求划分为各个部件的建造请求（Builder）；
3. 将各个部件的建造请求委派到具体的建造者（ConcreteBuilder）；
4. 各个具体建造者负责进行产品部件的构建；
5. 最终构建成具体产品（Product）。

2.2 代码

复杂对象类

```
1 public class Product {
2     private String partA; //定义部件，部件可以是任意类型，包括值类型和引用类型
3     private String partB;
4     private String partC;
5
6     //partA的Getter方法和Setter方法省略
7     //partB的Getter方法和Setter方法省略
8     //partC的Getter方法和Setter方法省略
9 }
```

抽象建造者类

```

1 public abstract class Builder {
2     //创建产品对象
3     protected Product product=new Product();
4     public abstract void buildPartA();
5     public abstract void buildPartB();
6     public abstract void buildPartC();
7
8     //返回产品对象
9     public Product getResult() {
10         return product;
11     }
12 }

```

具体建造者类

```

1 public class ConcreteBuilder1 extends Builder{
2     public void buildPartA() {
3         product.setPartA("A1");
4     }
5
6     public void buildPartB() {
7         product.setPartB("B1");
8     }
9
10    public void buildPartC() {
11        product.setPartC("C1");
12    }
13 }

```

指挥者类

```

1 public class Director {
2     private Builder builder;
3
4     public Director(Builder builder) {
5         this.builder=builder;
6     }
7
8     public void setBuilder(Builder builder) {
9         this.builder=builder;
10    }
11
12    //产品构建与组装方法
13    public Product construct() {
14        builder.buildPartA();
15        builder.buildPartB();
16        builder.buildPartC();
17        return builder.getResult();
18    }
19 }

```

客户类


```
1 | Builder builder = new ConcreteBuilder1(); //可通过配置文件实现
2 | Director director = new Director(builder);
3 | Product product = director.construct();
```

3. 实例讲解

接下来我用一个实例来对建造者模式进行更深一步的介绍。

3.1 实例概况

- 背景 小成希望去电脑城买一台组装的台式主机
- 过程
 1. 电脑城老板 (Diretor) 和小成 (Client) 进行需求沟通 (买来打游戏? 学习? 看片?)
 2. 了解需求后, 电脑城老板将小成需要的主机划分为各个部件 (Builder) 的建造请求 (CPU、主板 blabla)
 3. 指挥装机人员 (ConcreteBuilder) 去构建组件;
 4. 将组件组装起来成小成需要的电脑 (Product)

3.2 使用步骤

步骤1: 定义组装的过程 (Builder) : 组装电脑的过程

```
1 | public abstract class Builder {
2 |
3 |     //第一步: 装CPU
4 |     //声明为抽象方法, 具体由子类实现
5 |     public abstract void BuildCPU();
6 |
7 |     //第二步: 装主板
8 |     //声明为抽象方法, 具体由子类实现
9 |     public abstract void BuildMainboard ();
10 |
11 |     //第三步: 装硬盘
12 |     //声明为抽象方法, 具体由子类实现
13 |     public abstract void BuildHD ();
14 |
15 |     //返回产品的方法: 获得组装好的电脑
16 |     public abstract Computer GetComputer ();
17 | }
```

步骤2: 电脑城老板委派任务给装机人员 (Director)

```
1 | public class Director{
2 |     //指挥装机人员组装电脑
3 |     public void Construct(Builder builder){
4 |
5 |         builder. BuildCPU();
6 |         builder.BuildMainboard ();
7 |         builder. BuildHD ();
8 |     }
9 | }
```

步骤3: 创建具体的建造者 (ConcreteBuilder) :装机人员

```
1 | //装机人员1
```

```

2   public class ConcreteBuilder extend Builder{
3       //创建产品实例
4       Computer computer = new Computer();
5
6       //组装产品
7       @Override
8       public void BuildCPU(){
9           computer.Add("组装CPU")
10      }
11
12      @Override
13      public void BuildMainboard () {
14          computer.Add("组装主板")
15      }
16
17      @Override
18      public void BuildHD () {
19          computer.Add("组装主板")
20      }
21
22      //返回组装成功的电脑
23      @Override
24      public Computer GetComputer () {
25          return computer
26      }
27  }

```

步骤4：定义具体产品类 (Product) ：电脑

```

1   public class Computer{
2
3       //电脑组件的集合
4       private List<String> parts = new ArrayList<String>();
5
6       //用于将组件组装到电脑里
7       public void Add(String part){
8           parts.add(part);
9       }
10
11      public void Show(){
12          for (int i = 0;i<parts.size();i++){
13              System.out.println("组件"+parts.get(i)+"装好了");
14          }
15          System.out.println("电脑组装完成，请验收");
16      }
17  }
18  }

```

步骤5：客户端调用-小成到电脑城找老板买电脑

```

1   public class Builder Pattern{
2       public static void main(String[] args){
3
4       //逛了很久终于发现一家合适的电脑店
5       //找到该店的老板和装机人员
6       Director director = new Director();

```

```
7     Builder builder = new ConcreteBuilder();
8
9     //沟通需求后，老板叫装机人员去装电脑
10    director.Construct(builder);
11
12    //装完后，组装人员搬来组装好的电脑
13    Computer computer = builder.GetComputer();
14    //组装人员展示电脑给小成看
15    computer.Show();
16
17    }
18
19 }
```

结果输出：

```
1  组件CUP装好了
2  组件主板装好了
3  组件硬盘装好了
4  电脑组装完成，请验收
```

4. 特点

在全面解析完后，分析下其优缺点：

4.1 优点

- 易于解耦 将产品本身与产品创建过程进行解耦，可以使用相同的创建过程来得到不同的产品。也就是说细节依赖抽象。
- 易于精确控制对象的创建 将复杂产品的创建步骤分解在不同的方法中，使得创建过程更加清晰
- 易于拓展 增加新的具体建造者无需修改原有类库的代码，易于拓展，符合“开闭原则”。

每一个具体建造者都相对独立，而与其他的具体建造者无关，因此可以很方便地替换具体建造者或增加新的具体建造者，用户使用不同的具体建造者即可得到不同的产品对象。

4.2 缺点

- 建造者模式所创建的产品一般具有较多的共同点，其组成部分相似；如果产品之间的差异性很大，则不适合使用建造者模式，因此其使用范围受到一定的限制。
- 如果产品的内部变化复杂，可能会导致需要定义很多具体建造者类来实现这种变化，导致系统变得很庞大。

5. 应用场景

- 需要生成的产品对象有复杂的内部结构，这些产品对象具备共性；
- 隔离复杂对象的创建和使用，并使得相同的创建过程可以创建不同的产品。

五、原型模式

原型模式是一种比较简单的模式，也非常容易理解，实现一个接口，重写一个方法即完成了原型模式。在实际应用中，原型模式很少单独出现。经常与其他模式混用，他的原型类Prototype也常用抽象类来替代。

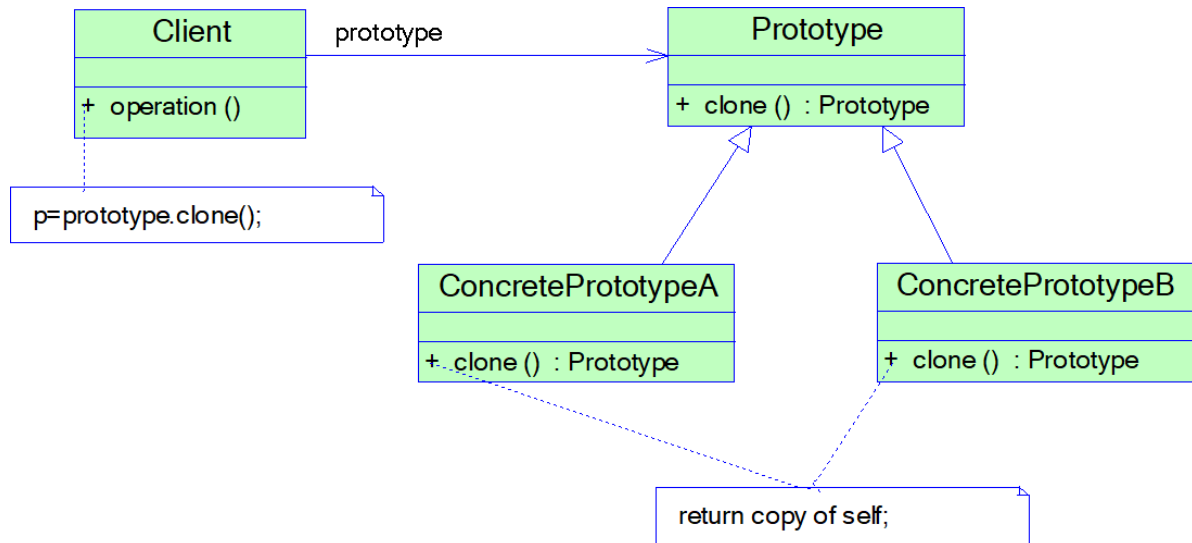
通过复制一个原型对象得到多个与原型对象一模一样的新对象

原型模式包含以下3个角色：

- Prototype (抽象原型类)
- ConcretePrototype (具体原型类)
- Client (客户类)

原型模式又可分为浅拷贝和深拷贝，区别在于对引用数据类型的成员变量的拷贝

1、UML



问题：

现在有一只羊（包含属性：名字Dolly、年龄2），需要克隆10只属性完全相同的羊。

2、浅拷贝

1.在原先Sheep类基础上实现Cloneable接口，重写clone方法。

```

1 public class Sheep implements Cloneable{
2     private String name;
3     private int age;
4
5     @Override
6     protected Object clone() { //克隆该实例，使用默认的clone方法来完成
7         Sheep sheep = null;
8         try {
9             sheep = (Sheep)super.clone();
10        } catch (Exception e) {
11            System.out.println(e.getMessage());
12        }
13        return sheep;
14    }
15
16    public Sheep(String name, int age) {
17        this.name = name;
18        this.age = age;
19    }
20
21    @Override
22    public String toString() {
23        return "Sheep{" +
24            "name='" + name + '\'' +
25            ", age=" + age +
26            '}';
27    }
28 }

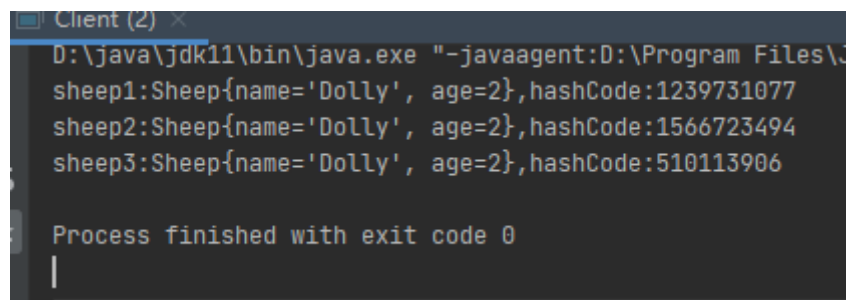
```

```
27     }
28 }
```

2.客户端调用

```
1  public class Client {
2      public static void main(String[] args) {
3          Sheep sheepDolly=new Sheep("Dolly",2);
4
5          Sheep sheep1 = (Sheep)sheepDolly.clone();
6          Sheep sheep2 = (Sheep)sheepDolly.clone();
7          Sheep sheep3 = (Sheep)sheepDolly.clone();
8          //....
9
10         System.out.println("sheep1:"+sheep1+",hashCode:" +
sheep1.hashCode());
11         System.out.println("sheep2:"+sheep2+",hashCode:" +
sheep2.hashCode());
12         System.out.println("sheep3:"+sheep3+",hashCode:" +
sheep3.hashCode());
13         //...
14     }
15 }
```

3.运行结果



```
Client (2) x
D:\java\jdk11\bin\java.exe "-javaagent:D:\Program Files\
sheep1:Sheep{name='Dolly', age=2},hashCode:1239731077
sheep2:Sheep{name='Dolly', age=2},hashCode:1566723494
sheep3:Sheep{name='Dolly', age=2},hashCode:510113906

Process finished with exit code 0
```

现在小羊有了一个朋友小牛，Sheep类添加了一个引用属性Cow，我们同样再克隆一遍。

Sheep类

```
1  public class Sheep implements Cloneable{
2      private String name;
3      private int age;
4      public Cow friend;//新朋友Cow对象，其余不变
5
6      @Override
7      protected Object clone() {
8          Sheep sheep = null;
9          try {
10             sheep = (Sheep)super.clone();
11         } catch (Exception e) {
12             System.out.println(e.getMessage());
13         }
14         return sheep;
15     }
16
17     public Sheep(String name, int age) {
18         this.name = name;
```

```

19         this.age = age;
20     }
21
22     @Override
23     public String toString() {
24         return "Sheep{" +
25             "name='" + name + '\'' +
26             ", age=" + age +
27             '}';
28     }
29 }

```

新添的Cow类

```

1  public class Cow {
2      private String name;
3      private int age;
4
5      public Cow(String name, int age) {
6          this.name = name;
7          this.age = age;
8      }
9
10     @Override
11     public String toString() {
12         return "Cow{" +
13             "name='" + name + '\'' +
14             ", age=" + age +
15             '}';
16     }
17 }

```

客户端调用克隆

```

1  public class Client {
2      public static void main(String[] args) {
3          Sheep sheepDolly=new Sheep("Dolly",2);
4          sheepDolly.friend=new Cow("Tom",1); //并实例化朋友
5
6          Sheep sheep1 = (Sheep)sheepDolly.clone();
7          Sheep sheep2 = (Sheep)sheepDolly.clone();
8          Sheep sheep3 = (Sheep)sheepDolly.clone();
9          //....
10
11         System.out.println("sheep1:"+sheep1+",hashCode:" +
sheep1.hashCode());
12         System.out.println("sheep1.friend:"+sheep1.friend+",hashCode:" +
sheep1.friend.hashCode()+"\n");
13
14         System.out.println("sheep2:"+sheep2+",hashCode:" +
sheep2.hashCode());
15         System.out.println("sheep2.friend:"+sheep2.friend+",hashCode:" +
sheep2.friend.hashCode()+"\n");
16
17         System.out.println("sheep3:"+sheep3+",hashCode:" +
sheep3.hashCode());

```

```

18         System.out.println("sheep3.friend:"+sheep3.friend+",hashCode:" +
sheep3.friend.hashCode()+"\n");
19         //...
20     }
21 }

```

运行结果

```

Client (2) x
D:\java\jdk11\bin\java.exe "-javaagent:D:\Program Files\JetBrains\I
sheep1:Sheep{name='Dolly', age=2},hashCode:557041912
sheep1.friend:Cow{name='Tom', age=1},hashCode:788117692

sheep2:Sheep{name='Dolly', age=2},hashCode:1566723494
sheep2.friend:Cow{name='Tom', age=1},hashCode:788117692

sheep3:Sheep{name='Dolly', age=2},hashCode:510113906
sheep3.friend:Cow{name='Tom', age=1},hashCode:788117692

Process finished with exit code 0

https://blog.csdn.net/qq\_45034708

```

通过运行结果发现，浅拷贝通过Object的clone()成功克隆实例化了三个新对象，**但是并没有克隆实例化对象中的引用属性，也就是没有克隆friend对象**，三个新克隆对象的friend还是指向原克隆前的friend，即同一个对象。

这样的话，他们四个的friend是引用同一个，若一个对象修改了friend属性，势必会影响其他三个对象的该成员变量值。

小结：

浅拷贝是使用默认的 clone()方法来实现 基本数据类型的成员变量，浅拷贝会直接进行值传递（复制属性值给新对象）。引用数据类型的成员变量，浅拷贝会进行引用传递（复制引用值(内存地址)给新对象）。

3、深拷贝

方法一：机灵的人儿看出，再clone一遍cow不就好了，但是手动递归下去不推荐。

1. Cow类也实现Cloneable接口

```

1 public class Cow implements Cloneable{
2     private String name;
3     private int age;
4
5     public Cow(String name, int age) {
6         this.name = name;
7         this.age = age;
8     }
9     //无引用类型，直接clone即可
10    @Override
11    protected Object clone() throws CloneNotSupportedException {
12        return super.clone(); //直接抛出了，没用try-catch
13    }
14    @Override

```

```

15     public String toString() {
16         return "Cow{" +
17             "name='" + name + '\'' +
18             ", age=" + age +
19             '}';
20     }
21 }

```

2. Sheep类的clone再添加调用cow的clone

```

1  public class Sheep implements Cloneable{
2      private String name;
3      private int age;
4      public Cow friend; //新朋友Cow对象，其余不变
5
6      @Override
7      protected Object clone() throws CloneNotSupportedException {
8          Object deep = null;
9          //完成对基本数据类型(属性)和String的克隆
10         deep = super.clone();
11         //对引用类型的属性，进行再次clone
12         Sheep sheep = (Sheep)deep;
13         sheep.friend = (Cow)friend.clone();
14
15         return sheep;
16     }
17
18     public Sheep(String name, int age) {
19         this.name = name;
20         this.age = age;
21     }
22
23     @Override
24     public String toString() {
25         return "Sheep{" +
26             "name='" + name + '\'' +
27             ", age=" + age +
28             '}';
29     }
30 }

```

3. 客户端调用

```

1  public class Client {
2      public static void main(String[] args) throws CloneNotSupportedException
3      {
4          Sheep sheepDolly=new Sheep("Dolly",2);
5          sheepDolly.friend=new Cow("Tom",1); //并实例化朋友
6
7          Sheep sheep1 = (Sheep)sheepDolly.clone();
8          Sheep sheep2 = (Sheep)sheepDolly.clone();
9          Sheep sheep3 = (Sheep)sheepDolly.clone();
10         //....
11
12         System.out.println("sheep1:"+sheep1+",hashCode:" +
13             sheep1.hashCode());
14     }
15 }

```



```

12         System.out.println("sheep1.friend:"+sheep1.friend+",hashCode:" +
sheep1.friend.hashCode()+"\n");
13
14         System.out.println("sheep2:"+sheep2+",hashCode:" +
sheep2.hashCode());
15         System.out.println("sheep2.friend:"+sheep2.friend+",hashCode:" +
sheep2.friend.hashCode()+"\n");
16
17         System.out.println("sheep3:"+sheep3+",hashCode:" +
sheep3.hashCode());
18         System.out.println("sheep3.friend:"+sheep3.friend+",hashCode:" +
sheep3.friend.hashCode()+"\n");
19         //...
20     }
21 }

```

4.运行结果

```

D:\java\jdk11\bin\java.exe "-javaagent:D:\Program Files\J
sheep1:Sheep{name='Dolly', age=2},hashCode:557041912
sheep1.friend:Cow{name='Tom', age=1},hashCode:510113906

sheep2:Sheep{name='Dolly', age=2},hashCode:1622006612
sheep2.friend:Cow{name='Tom', age=1},hashCode:66233253

sheep3:Sheep{name='Dolly', age=2},hashCode:1286783232
sheep3.friend:Cow{name='Tom', age=1},hashCode:1874154700

Process finished with exit code 0
https://blog.csdn.net/qq\_45034708

```

方法二：通过对象序列化实现深拷贝(推荐)

1. Cow类实现序列化接口，不必实现Cloneable接口了

```

1 public class Cow implements Serializable {
2     private String name;
3     private int age;
4
5     public Cow(String name, int age) {
6         this.name = name;
7         this.age = age;
8     }
9
10    @Override
11    public String toString() {
12        return "Cow{" +
13            "name='" + name + '\'' +
14            ", age=" + age +
15            '}';
16    }
17 }

```

2.在Sheep类实现序列化接口

```

1 public class Sheep implements Serializable { //实现序列化接口
2     private String name;
3     private int age;
4     public Cow friend;
5
6     public Sheep(String name, int age) {
7         this.name = name;
8         this.age = age;
9     }
10
11     @Override
12     public String toString() {
13         return "Sheep{" +
14             "name='" + name + '\'' +
15             ", age=" + age +
16             '}';
17     }
18     public Object deepClone() { //深拷贝
19         //创建流对象
20         ByteArrayOutputStream bos = null;
21         ObjectOutputStream oos = null;
22         ByteArrayInputStream bis = null;
23         ObjectInputStream ois = null;
24
25         try {
26             //序列化
27             bos = new ByteArrayOutputStream();
28             oos = new ObjectOutputStream(bos);
29             oos.writeObject(this); //当前这个对象以对象流的方式输出
30
31             //反序列化
32             bis = new ByteArrayInputStream(bos.toByteArray());
33             ois = new ObjectInputStream(bis);
34             Sheep sheep = (Sheep) ois.readObject();
35
36             return sheep;
37         } catch (Exception e) {
38             e.printStackTrace();
39             return null;
40         } finally {
41             //关闭流
42             try {
43                 bos.close();
44                 oos.close();
45                 bis.close();
46                 ois.close();
47             } catch (Exception e2) {
48                 System.out.println(e2.getMessage());
49             }
50         }
51     }
52 }

```

3.客户端调用

```

1 public class Client {


```

```

2      public static void main(String[] args) throws CloneNotSupportedException
3      {
4          Sheep sheepDolly=new Sheep("Dolly",2);
5          sheepDolly.friend=new Cow("Tom",1); //并实例化朋友
6
7          Sheep sheep1 = (Sheep)sheepDolly.clone();
8          Sheep sheep2 = (Sheep)sheepDolly.clone();
9          Sheep sheep3 = (Sheep)sheepDolly.clone();
10         //....
11
12         System.out.println("sheep1:"+sheep1+",hashCode:" +
13         sheep1.hashCode());
14         System.out.println("sheep1.friend:"+sheep1.friend+",hashCode:" +
15         sheep1.friend.hashCode()+"\n");
16
17         System.out.println("sheep2:"+sheep2+",hashCode:" +
18         sheep2.hashCode());
19         System.out.println("sheep2.friend:"+sheep2.friend+",hashCode:" +
20         sheep2.friend.hashCode()+"\n");
21
22         System.out.println("sheep3:"+sheep3+",hashCode:" +
23         sheep3.hashCode());
24         System.out.println("sheep3.friend:"+sheep3.friend+",hashCode:" +
25         sheep3.friend.hashCode()+"\n");
26         //...
27     }
28 }

```

4.运行结果



```

Client (2) x
D:\java\jdk11\bin\java.exe "-javaagent:D:\Program Files\JetBrains\Intel
sheep1:Sheep{name='Dolly', age=2},hashCode:1355316001
sheep1.friend:Cow{name='Tom', age=1},hashCode:670035812

sheep2:Sheep{name='Dolly', age=2},hashCode:1870647526
sheep2.friend:Cow{name='Tom', age=1},hashCode:1204167249

sheep3:Sheep{name='Dolly', age=2},hashCode:1047503754
sheep3.friend:Cow{name='Tom', age=1},hashCode:1722023916

Process finished with exit code 0 https://blog.csdn.net/qq_45034708

```

4、原型模式总结:

- 创建新的对象比较复杂时，可以利用原型模式简化对象的创建过程，同时也能够提高效率
- 可以不用重新初始化对象，动态地获得对象运行时的状态。
- 如果原始对象发生变化(增加或者减少属性)，其它克隆对象的也会发生相应的变化，无需修改代码
- 若成员变量无引用类型，浅拷贝clone即可；若引用类型的成员变量很少，可考虑递归实现clone，否则推荐序列化。

六、单例模式

1. 实例引入

- 背景：小成有一个塑料生产厂，但里面只有一个仓库。
- 目的：想用代码来实现仓库的管理
- 现有做法：建立仓库类和工人类

其中，仓库类里的quantity=商品数量；工人类里有搬运方法MoveIn(int i)和MoveOut(int i)。

- 出现的问题：通过测试发现，每次工人搬运操作都会新建一个仓库，就是货物都不是放在同一仓库，这是怎么回事呢？（看下面代码）

```
1 package scut.designmodel.SingletonPattern;
2
3 //仓库类
4 class StoreHouse {
5     private int quantity = 100;
6
7     public void setQuantity(int quantity) {
8         this.quantity = quantity;
9     }
10
11     public int getQuantity() {
12         return quantity;
13     }
14 }
15
16 //搬货工人类
17 class Carrier{
18     public StoreHouse mStoreHouse;
19     public Carrier(StoreHouse storeHouse){
20         mStoreHouse = storeHouse;
21     }
22     //搬货进仓库
23     public void MoveIn(int i){
24         mStoreHouse.setQuantity(mStoreHouse.getQuantity()+i);
25     }
26     //搬货出仓库
27     public void MoveOut(int i){
28         mStoreHouse.setQuantity(mStoreHouse.getQuantity()-i);
29     }
30 }
31
32 //工人搬运测试
33 public class SinglePattern {
34     public static void main(String[] args){
35         StoreHouse mStoreHouse1 = new StoreHouse();
36         StoreHouse mStoreHouse2 = new StoreHouse();
37         Carrier Carrier1 = new Carrier(mStoreHouse1);
38         Carrier Carrier2 = new Carrier(mStoreHouse2);
39
40         System.out.println("两个是不是同一个? ");
41
42         if(mStoreHouse1.equals(mStoreHouse2)){//这里用equals而不是用 == 符号，因
43             为 == 符号只是比较两个对象的地址
44             System.out.println("是同一个");
45         }
```

```

44         }else {
45             System.out.println("不是同一个");
46         }
47         //搬运工搬完货物之后出来汇报仓库商品数量
48         Carrier1.MoveIn(30);
49         System.out.println("仓库商品余
量: "+Carrier1.mStoreHouse.getQuantity());
50         Carrier2.MoveOut(50);
51         System.out.println("仓库商品余
量: "+Carrier2.mStoreHouse.getQuantity());
52     }
53 }

```

结果:

```

1  两个是不是同一个?
2  不是同一个
3  仓库商品余量: 130
4  仓库商品余量: 50

```

2. 单例模式介绍

2.1 模式说明

实现1个类只有1个实例化对象 & 提供一个全局访问点

2.2 作用 (解决的问题)

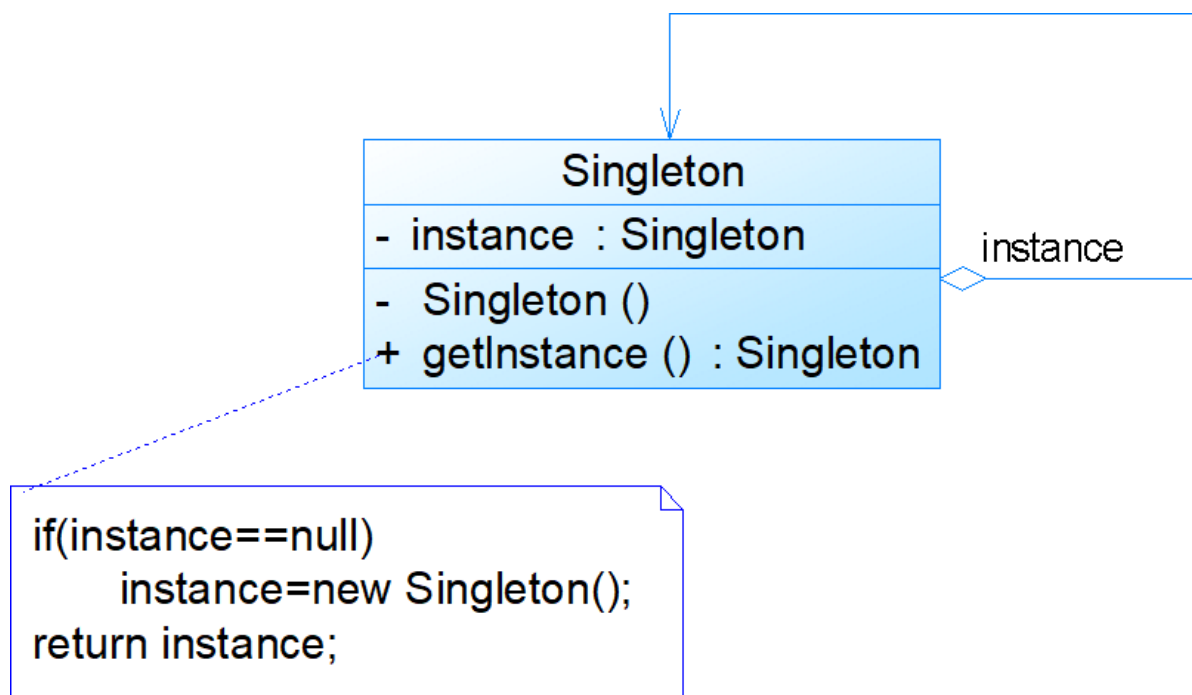
保证1个类只有1个对象, 降低对象之间的耦合度

从上面可看出: 工人类操作的明显不是同一个仓库实例, 而全部工人希望操作的是同一个仓库实例, 即只有1个实例

2.3 工作原理

在Java中, 我们通过使用对象 (类实例化后) 来操作这些类, 类实例化是通过它的构造方法进行的, 要是想实现一个类只有一个实例化对象, 就要对类的构造方法下功夫:

2.4 UML



2.5 代码

```
1 public class Singleton {
2     private static Singleton instance=null; //静态私有成员变量
3
4     //私有构造函数
5     private Singleton() {
6     }
7
8     //静态公有工厂方法，返回唯一实例
9     public static Singleton getInstance() {
10         if(instance==null)
11             instance=new Singleton();
12         return instance;
13     }
14 }
```

3. 实例讲解

```
1 package scut.designmodel.SingletonPattern;
2
3 import java.util.concurrent.locks.Lock;
4 import java.util.concurrent.locks.ReentrantLock;
5
6 //单例仓库类
7 class StoreHouse {
8
9     //仓库商品数量
10    private int quantity = 100;
11    //自己在内部实例化
12    private static StoreHouse ourInstance = new StoreHouse();
13    //让外部通过调用getInstance()方法来返回唯一的实例。
14    public static StoreHouse getInstance() {
15        return ourInstance;
16    }
17
18    //封闭构造函数
19    private StoreHouse() {
20    }
21
22    public void setQuantity(int quantity) {
23        this.quantity = quantity;
24    }
25
26    public int getQuantity() {
27        return quantity;
28    }
29 }
30
31 //搬货工人类
32 class Carrier{
33     public StoreHouse mStoreHouse;
34     public Carrier(StoreHouse storeHouse){
35         mStoreHouse = storeHouse;
36     }
37     //搬货进仓库
```

```

38     public void MoveIn(int i){
39         mStoreHouse.setQuantity(mStoreHouse.getQuantity()+i);
40     }
41     //搬货出仓库
42     public void MoveOut(int i){
43         mStoreHouse.setQuantity(mStoreHouse.getQuantity()-i);
44     }
45 }
46
47 //工人搬运测试
48 public class SinglePattern {
49     public static void main(String[] args){
50         StoreHouse mStoreHouse1 = StoreHouse.getInstance();
51         StoreHouse mStoreHouse2 = StoreHouse.getInstance();
52         Carrier carrier1 = new Carrier(mStoreHouse1);
53         Carrier carrier2 = new Carrier(mStoreHouse2);
54
55         System.out.println("两个是不是同一个? ");
56
57         if(mStoreHouse1.equals(mStoreHouse2)){
58             System.out.println("是同一个");
59         }else {
60             System.out.println("不是同一个");
61         }
62         //搬运工搬完货物之后出来汇报仓库商品数量
63         carrier1.MoveIn(30);
64         System.out.println("仓库商品余
量: "+carrier1.mStoreHouse.getQuantity());
65         carrier2.MoveOut(50);
66         System.out.println("仓库商品余
量: "+carrier2.mStoreHouse.getQuantity());
67     }
68 }

```

结果:

```

1 | 两个是不是同一个?
2 | 是同一个
3 | 仓库商品余量: 130
4 | 仓库商品余量: 80

```

4. 特点

4.1 优点

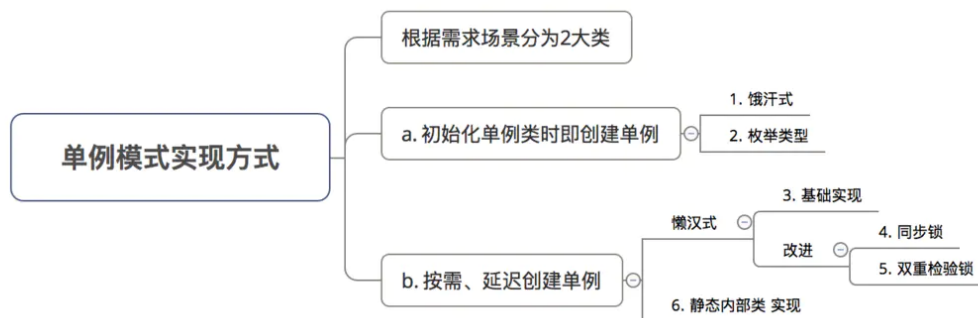
- 提供了对唯一实例的受控访问;
- 由于在系统内存中只存在一个对象, 因此可以节约系统资源, 对于一些需要频繁创建和销毁的对象单例模式无疑可以提高系统的性能;
- 可以根据实际情况需要, 在单例模式的基础上扩展做出双例模式, 多例模式;

4.2 缺点

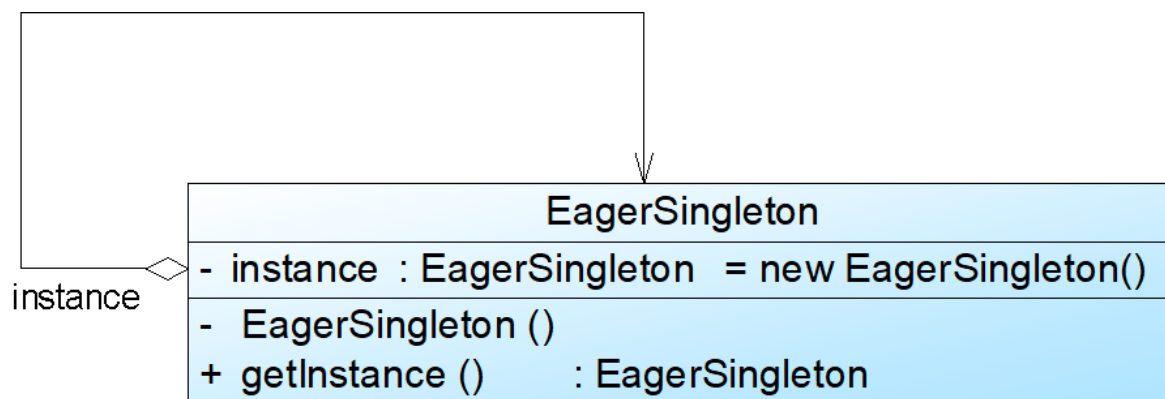
1. 单例类的职责过重, 里面的代码可能会过于复杂, 在一定程度上违背了“单一职责原则”。
2. 如果实例化的对象长时间不被利用, 会被系统认为是垃圾而被回收, 这将导致对象状态的丢失。

5. 单例模式的实现方式

单例模式的实现方式有多种，根据需求场景，可分为2大类、6种实现方式。具体如下：

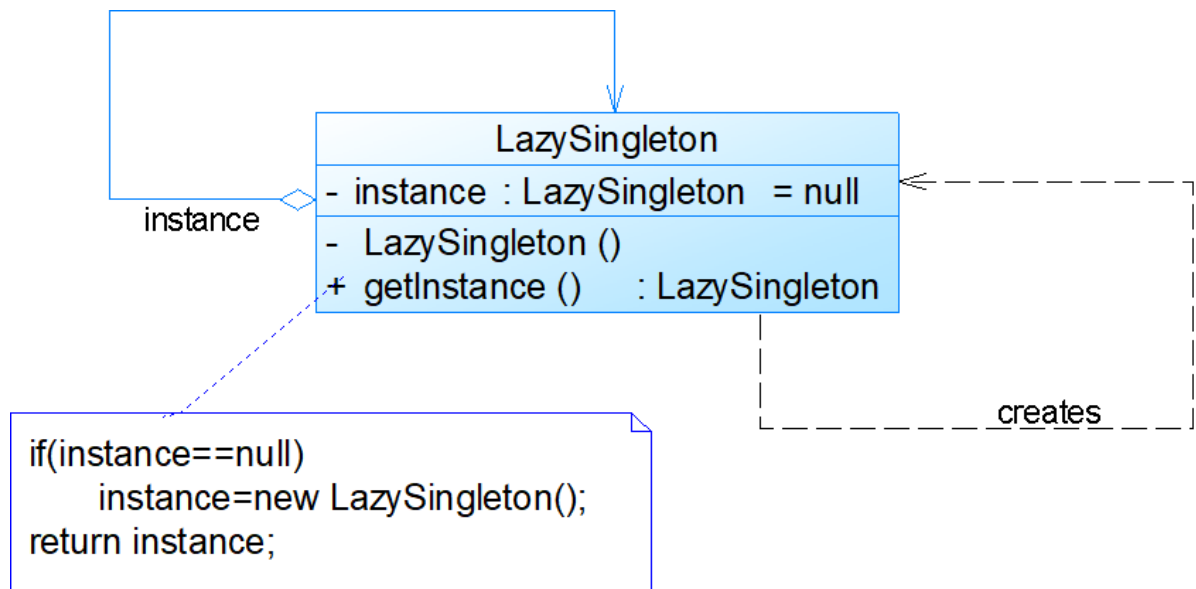


饿汉式



```
1 class Singleton {
2
3     // 1. 加载该类时，单例就会自动被创建
4     private static Singleton ourInstance = new Singleton();
5
6     // 2. 构造函数 设置为 私有权限
7     // 原因：禁止他人创建实例
8     private Singleton() {
9     }
10
11    // 3. 通过调用静态方法获得创建的单例
12    public static Singleton newInstance() {
13        return ourInstance;
14    }
15 }
```

懒汉式



```

1  class Singleton {
2      // 1. 类加载时，先不自动创建单例
3      // 即，将单例的引用先赋值为 Null
4      private static Singleton ourInstance = null;
5
6      // 2. 构造函数 设置为 私有权限
7      // 原因：禁止他人创建实例
8      private Singleton() {
9      }
10
11     // 3. 需要时才手动调用 newInstance () 创建 单例
12     public static Singleton newInstance() {
13         // 先判断单例是否为空，以避免重复创建
14         if( ourInstance == null){
15             ourInstance = new Singleton();
16         }
17         return ourInstance;
18     }
19 }
  
```

饿汉式单例类与懒汉式单例类的比较：

- 饿汉式单例类：无须考虑多个线程同时访问的问题；调用速度和反应时间优于懒汉式单例；资源利用效率不及懒汉式单例；系统加载时间可能会比较长
- 懒汉式单例类：实现了延迟加载；必须处理好多个线程同时访问的问题；需通过双重检查锁定等机制进行控制，将导致系统性能受到一定影响