

## 2.5万字详解23种设计模式—创建型模式（简单工厂、工厂方法、抽象工厂、单例-多线程安全详解、建造者、原型）的详细解读、UML类图、及代码演示

原创

王德印

已于 2023-03-26 11:10:44 修改

9494

收藏 364

版权

分类专栏：

Java

后端

文章标签：

设计模式

Java

同时被 2 个专栏收录 ▾

4 订阅

11 篇文章

订阅专栏

有兴趣的小伙伴们可以看其他文章：

[微服务 分布式 集群 负载均衡详述](#)

[springcloud五大核心组件详述](#)

[代码中如何干掉太多的if else即if else的多种替代方案以提高代码质量通过公司代码审查](#)

### 设计模式

- 一、设计模式的认识
- 二、设计模式的分类
  - (1) 根据其目的
  - (2) 根据范围
- 三、设计模式的优点
- 四、设计模式中关键点
- 五、创建型模式
  - (1) 简单（静态）工厂模式
    - 1.认识
    - 2.UML图解
    - 3.代码实现
    - 4.总结
    - 5.升级版本
    - 6.再升级（重要）
    - 7.开发常用版本：多方法工厂
    - 8.应用场景
  - (2) 工厂方法模式
    - 1.认识
    - 2.UML类图
    - 3.代码实现
    - 4.总结
    - 5.工厂方法与简单工厂的区别
    - 6.应用场景
  - (3) 抽象工厂模式
    - 1.认识
    - 2.UML类图
    - 3.代码实现
    - 4.总结
    - 5.应用场景
    - 6.个人总结
  - (4) 单例模式
    - 1.认识
    - 2.UML类图
    - 3.代码实现
      - (1) 饿汉式
      - (2) 懒汉式
      - (3) 线程安全的懒汉式
      - (4) DCL单例----高性能的懒汉式
      - (5) 静态内部类的方式
      - (6) 枚举单例模式
      - (7) 静态内部类升级版
      - (8) 容器式单例
      - (9) ThreadLocal单例
  - (5) 原型模式
    - 1.认识
    - 2.UML类图
    - 3.代码实现
    - 4.总结
    - 5.应用场景
  - (6) 建造者模式
    - 1.认识：
    - 2.传统的builder模式
      - ①.UML类图：
      - ②.代码如下：
    - 3.传统builder模式的变种
      - ①说明：
      - ②代码：
    - 4.总结
    - 5.应用场景
- 六、个人体会

关注微信公众号微信搜索[ 老板再来一杯时光]回复[ 进群]即可进入无广告交流群！【 进群】即可获取【 java基础经典面试】一份和【DDD领域驱动设计实战落地解惑】PDF一份！！

前言：经学习于多位网上大佬并结合自己的感悟和理解，本文简述了各大设计模式，并通过UML和代码详细说明了创建型模式：

①简单（静态）工厂：升级版本，再升级版本，多方法工厂共四种实现方式。

②工厂方法

③抽象工厂

④单例模式：饿汉式、懒汉式、加锁懒汉式、DCL单例----双重锁高性能、静态内部类、枚举单例、升级版静态内部类、容器式单例、ThreadLocal单例共九种实现方式。

⑤原型模式：浅拷贝和深拷贝

⑥建造者模式：传统的build模式和build模式变种

必看文章: [Mysql执行顺序写sql不再是问题\\*\\*！！\\*\\*](#)  
想详细了解[微服务与分布式的](#)可以看小编这篇博客[微服务 微服务架构 分布式 集群 负载均衡](#)和[spring cloud五大神兽](#)

### 一、设计模式的认识

设计模式(Design Pattern)是前辈们经过相当长的一段时间的试验和错误总结出来的，是软件开发过程中面临的通用问题的解决方案。这些解决方案使用设计模式是为了可重用代码、让代码更容易被他人理解、保证代码可靠性。

### 二、设计模式的分类

#### (1) 根据其目的

即模式是用来做什么的，可分为创建型(Creational)，结构型(Structural)和行为型(Behavioral)三种：

- ①创建型模式主要用于创建对象。
- ②结构型模式主要用于处理类或对象的组合。
- ③行为型模式主要用于描述对类或对象怎样交互和怎样分配职责。

#### (2) 根据范围

即模式主要是用于处理类之间关系还是处理对象之间的关系，可分为类模式和对象模式两种：

类模式处理类和子类之间的关系，这些关系通过继承建立，在编译时刻就被确定下来，是属于静态的。

对象模式处理对象间的关系，这些关系在运行时刻变化，更具动态性。

范围\目的	创建型模式	结构型模式	行为型模式
类模式	工厂方法模式	(类) 适配器模式	解释器模式 模板方法模式
对象模式	抽象工厂模式 建造者模式 原型模式 单例模式	(对象) 适配器模式 桥接模式 组合模式 装饰模式 外观模式 享元模式 代理模式	职责链模式 命令模式 迭代器模式 中介者模式 备忘录模式 观察者模式 状态模式 策略模式 访问者模式

### 三、设计模式的优点

- ①可以提高程序员的思维能力、编程能力和设计能力。
- ②使程序设计更加标准化、代码编制更加工程化，使软件开发效率大大提高，从而缩短软件的开发周期。
- ③使设计的代码可重用性高、可读性强、可靠性高、灵活性好、可维护性强。

现在这样说肯定有些懵逼，需要在实际开发中才能体会得到真正的好处。

### 四、设计模式中关键点

#### (1) 创建型模式：

**简单工厂**：一个工厂类根据传入的参量决定创建出那一种产品类的实例。

**工厂方法**：定义一个创建对象的接口，让子类决定实例化那个类。

**抽象工厂**：创建相关或依赖对象的家族，而无需明确指定具体类。

**建造者模式**：封装一个复杂对象的构建过程，并可以按步骤构造。

**单例模式**：某个类只能有一个实例，提供一个全局的访问点。

**原型模式**：通过复制现有的实例来创建新的实例。

#### (2) 结构型模式\*

**外观模式**：对外提供一个统一的方法，来访问子系统中的一群接口。

**桥接模式**：将抽象部分和它的实现部分分离，使它们都可以独立的变化。

**组合模式**：将对象组合成树形结构以表示""部分-整体""的层次结构。

**装饰模式**：动态的给对象添加新的功能。

**代理模式**：为其他对象提供一个代理以便控制这个对象的访问。

**适配器模式**：将一个类的方法接口转换成客户希望的另外一个接口。

**亨元（蝇量）模式**：通过共享技术来有效的支持大量细粒度的对象。

#### (3) 行为型模式

**模板模式**：定义一个算法结构，而将一些步骤延迟到子类实现。

**解释器模式**：给定一个语言，定义它的文法的一种表示，并定义一个解释器。

**策略模式**：定义一系列算法，把他们封装起来，并且使它们可以相互替换。

**状态模式**：允许一个对象在其对象内部状态改变时改变它的行为。

**观察者模式**：对象间的一对多的依赖关系。

**备忘录模式**：在不破坏封装的前提下，保持对象的内部状态。

**中介者模式**：用一个中介对象来封装一系列的对象交互。

**命令模式**：将命令请求封装为一个对象，使得可以用不同的请求来进行参数化。

**访问者模式**：在不改变数据结构的前提下，增加作用于一组对象元素的新功能。

**责任链模式**：将请求的发送者和接收者解耦，使的多个对象都有处理这个请求的机会。

**迭代器模式**：一种遍历访问聚合对象中各个元素的方法，不暴露该对象的内部结构。

### 五、创建型模式

#### (1) 简单（静态）工厂模式

##### 1.认识

- ①一句话来说就是，一个工厂类根据传入的参量决定创建出那一种产品类的实例。因为逻辑实现简单，所以称为**简单工厂模式**，也因为工厂中的方法一般设置为静态，所以也称为**静态工厂**，它不属于23种模式。
- ②简单工厂模式专门定义一个工厂类来负责创建其他类的实例，被创建的实例通常都具有共同的父类，在工厂类中，可以根据参数的不同返回不同类的实例。升级版简单工厂模式，通过反射根据类的全路径名生成对象。
- ③简单工厂模式就是将这部分创建对象语句分离出来，由工厂类来封装实例化对象的行为，修改时只需要修改类中的操作代码，使用时调用该类不需要考虑实例化对象的行为，使得后期代码维护升级更简单方便，有利于代码的可修改性与可读性。
- ④但是如果增加新的产品的话，需要修改工厂类的判断逻辑，违背开闭原则。

##### 2.UML图解

简单介绍一下UML：

**泛化**：继承

带三角箭头的实线，箭头指向类

**实现**：实现

带三角箭头的虚线，箭头指向接口

**依赖**：new A的对象当作方法参数传递进来作为B类的局部变量

带箭头的虚线，指向被使用者

**关联**：一个类作为另一个类的成员变量

带普通箭头的实心线，指向被拥有者

**聚合：**new A的对象当作方法参数传递进来作为B类的成部变量

带空心菱形的实心线，菱形指向整体

**组合：**

new A的对象当作构造方法参数传递进来作为B类的成部变量或者A类作为B类成 员变量并已经new A （A类和B类具有相同的生命周期）

带实心菱形的实线，菱形指向整体

总结：

各种关系的强弱顺序：

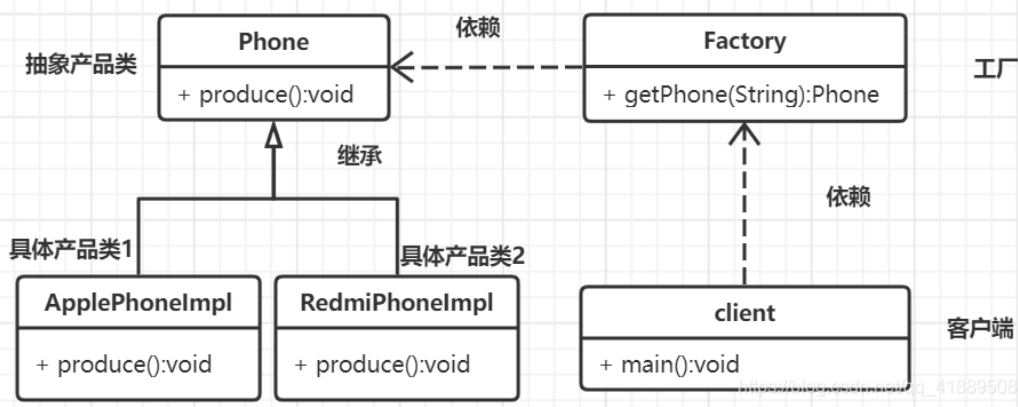
泛化 = 实现 > 组合 > 聚合 > 关联 > 依赖

**区分：**

①如果B类作为了A类的成员变量（has的关系），则一般是A类与B类是关联(A类与B类平级)、聚合（A类是整体，B类是部分）、组合的关系（A类是整体，B类是部分，且A类B类有相同的生命周期，）根据上下文语意区分：**聚合**（B类即便不在A类中也可以单独存在），**组合**（B类不在A类中就无法单独存在）。

②如果B类作为了A类的局部变量（use的关系），方法的形参，或者对静态方法的调用一般是依赖关系。

**UML类图如下：**



**UML说明：**

苹果手机和小米手机继承了手机这个抽象类，工厂类里根据客户端传入的参数生成相应的对象，如，客户说要小米，工厂给客户一个小米手机，客户说要苹果，工厂给客户一个苹果手机。

**简单工厂有三个对象：**

①抽象产品类：提供抽象方法供具体产品类实现

②具体产品类：提供具体的产品

③工厂：根据内部逻辑返回相应的产品

### 3.代码实现

（1）抽象产品类Phone

这里可以是类，也可以是接口或者抽象类，千万不要思维定式。我比较喜欢面向接口编程，所以我这里用了接口。

```
1 public interface Phone {
2     void produce();
3 }
```

（2）具体产品类

```
1 在这里插入代码片public class ApplePhoneImpl implements Phone{
2     @Override
3     public void produce() {
4         System.out.println("生产苹果手机");
5     }
6 }
```

```
1 public class RedmiPhoneImpl implements Phone{
2     @Override
3     public void produce() {
4         System.out.println("生产了小米手机");
5     }
6 }
```

（3）工厂类

```
1 public class Factory {
2
3     public Phone getPhone(String type){
4         Phone phone = null;
5         if("小米".equals(type)){
6             phone = new RedmiPhoneImpl();
7         }else if("苹果".equals(type)){
8             phone = new ApplePhoneImpl();
9         }//.....
10        return phone;
11    }
12 }
```

（4）客户端使用

```
1 @Test
2     public void test1(){
3         Factory factory = new Factory();
4
5         Phone redmiPhone = factory.getPhone("小米");
6         System.out.println(redmiPhone);
7         redmiPhone.produce();
8
9         Phone applePhone = factory.getPhone("苹果");
10        System.out.println(applePhone);
11        applePhone.produce();
12    }
```

运行结果如下：

```
"C:\Program Files (x86)\WDYin\JDK\jdk8_64\bin\java.exe" ...

com.wander.design.simplefactory.product.RedmiPhoneImpl@c46bcd4
生产了红米手机
com.wander.design.simplefactory.product.ApplePhoneImpl@3234e239
生产苹果手机

Process finished with exit code 0
https://blog.csdn.net/qq_41889508
```

4.总结

优点：

只需要传入一个正确的参数，就可以获取你所需要的对象而无需知道其创建对象的细节

缺点：

扩展性差，当增加新的产品需要修改工厂类的判断逻辑，违背开闭原则，如我想要买一个华为手机的话，除了**新增**华为手机这个产品类，还需要**修改**工厂中的逻辑

5.升级版本

通过反射创建对象，以改进了之前提到的缺点（增加新的产品需要修改工厂类的判断逻辑），现在**增加**新的具体产品的时候**不需要修改**工厂中的代码。满足了开闭原则。

(1) 工厂类代码如下：

```
1 | public class FactoryPlus {
2 |     public Phone getPhone(Class clazz) throws Exception {
3 |         return (Phone) Class.forName(clazz.getName()).newInstance();
4 |     }
5 | }
```

(2) 客户端代码如下：

```
1 | @Test
2 |     public void test2() throws Exception {
3 |         FactoryPlus factory = new FactoryPlus();
4 |
5 |         Phone redmiPhone = factory.getPhone(RedmiPhoneImpl.class);
6 |         System.out.println(redmiPhone);
7 |         redmiPhone.produce();
8 |
9 |         Phone applePhone = factory.getPhone(ApplePhoneImpl.class);
10 |        System.out.println(applePhone);
11 |        applePhone.produce();
12 |    }
```

运行结果如下：

```
"C:\Program Files (x86)\WDYin\JDK\jdk8_64\bin\java.exe" ...

com.wander.design.simplefactory.product.RedmiPhoneImpl@c46bcd4
生产了红米手机
com.wander.design.simplefactory.product.ApplePhoneImpl@3234e239
生产苹果手机

Process finished with exit code 0
https://blog.csdn.net/qq_41889508
```

(3) 总结

**优点：**工厂类中的方法逻辑，是利用反射机制生成对象返回，好处是增加一种产品时，不需要修改工厂类中的代码。满足了开闭原则。

**缺点：**这种写法粗看牛逼，细想之下，不谈reflection的效率还有以下问题：个人觉得不好，因为Class.forName(clz.getName()).newInstance()调用的是**无参**构造函数生成对象，它和new Object()是一样的性质，而工厂方法应该用于复杂对象的初始化，当需要调用**有参**的构造函数时便无能为力了，这样像为了工厂而工厂，没有实际意义。  
2 不同的产品需要不同额外参数的时候 不支持。

6.再升级（重要）

(1) 工厂类：

```
1 | public class FactoryPlusPlus {
2 |
3 |     /**<bean id="applePhone" class="com.wander.design.simplefactory.product.ApplePhoneImpl"/>
4 |      * 熟悉吧！！spring ioc 就是通过将下面的这句话配置在配置文件中，再利用反射创建对象，
5 |      * 这就是spring ioc的原理：工厂+配置文件+反射！！以达到彻底解耦的目的*/
6 |     private static String className="com.wander.design.simplefactory.product.ApplePhoneImpl";
7 |
8 |     public static Phone getPhone() throws Exception {
9 |         return (Phone) Class.forName(className).newInstance();
10 |    }
11 | }
```

(2)客户端：

```
1 | @Test
2 |     public void test3() throws Exception {
3 |         Phone phone = FactoryPlusPlus.getPhone();
4 |         phone.produce();
5 |    }
```

(3) 说明：spring ioc容器的原理就是这种方式：**工厂+配置文件+反射**，spring通过读取配置文件（<bean id="applePhone" class="com.wander.design.simplefactory.product.ApplePhoneImpl"/>），获取到className再利用反射机制Class.forName(className).newInstance()得到对象赋值给配置文件里bean标签的id属性的值，就是工厂生成的对象名。**优点：**就是满足OCP原则，在不修改源代码的前提下切换底层的实现，达到解耦的目的！

7.开发常用版本：多方法工厂

使用以上两种方法的工厂，都有两个**缺点**：

一是不同的产品需要不同额外参数的时候不支持。

二是如果使用时传递的type、Class出错，将不能得到正确的对象，容错率不高。

而多方法的工厂模式为不同产品，提供不同的生产方法，使用时 需要哪种产品就调用该种产品的方法，使用方便、容错率高。

(1) 工厂类代码如下：

```
1 | public class FactoryMoreMethod {
2 |
```

```
3 | public static Phone getApple(){
4 |     return new ApplePhoneImpl();
5 | }
6 |
7 | public static Phone getRedmi(){
8 |     return new RedmiPhoneImpl();
9 | }
10 |
11 | /** 新增华为手机产品，只需要在工厂中增加一个静态方法即可，不需要修改原有的方法**/
12 | public static Phone getHonor(){
13 |     return new HonorPhoneImpl();
14 | }
15 | }
```

(2) 客户端代码：

```
1 | @Test
2 | public void test3() throws Exception {
3 |     Phone apple = FactoryMoreMethod.getApple();
4 |     apple.produce();
5 |
6 |     Phone redmi = FactoryMoreMethod.getRedmi();
7 |     redmi.produce();
8 |
9 |     Phone honor = FactoryMoreMethod.getHonor();
10 |    honor.produce();
11 | }
```

(3) 应用场景：

查看java源码：java.util.concurrent.Executors类便是一个生成Executor 的工厂，其采用的便是 多方法静态工厂模式：  
例如ThreadPoolExecutor类构造方法有5个参数，其中三个参数写法固定，前两个参数可配置，如下写。

```
1 | public static ExecutorService newFixedThreadPool(int nThreads) {
2 |     return new ThreadPoolExecutor(nThreads, nThreads,
3 |                                   0L, TimeUnit.MILLISECONDS,
4 |                                   new LinkedBlockingQueue<Runnable>());
5 | }
```

又如JDK想增加创建ForkJoinPool类的方法了，只想配置parallelism参数，便在类里增加一个如下的方法：

```
1 | public static ExecutorService newWorkStealingPool(int parallelism) {
2 |     return new ForkJoinPool
3 |         (parallelism,
4 |          ForkJoinPool.defaultForkJoinWorkerThreadFactory,
5 |          null, true);
6 | }
```

(4) 总结：多方法工厂的优势，方便创建同种类型的复杂参数对象。

## 8.应用场景

- (1) 在任何需要生成**复杂对象**的地方，都可以使用工厂方法模式。直接用new可以完成的**不需要用工厂模式**个人理解，重点就是这个复杂 （构造函数有很多参数）和 是否可以 直接用new。
- (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。（对于升级后的简单工厂模式只知道类名即可）
- (3) 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。（对于升级后的简单工厂模式已解决这个问题，符合开闭原则）
- (4) 简单工厂在源码中的使用–Calendar：

```
1 | Calendar cal = Calendar.getInstance(zone.toTimeZone(), locale);
2 | public static Calendar getInstance(TimeZone zone, Locale aLocale) {
3 |     return createCalendar(zone, aLocale);
4 | }
5 |
6 | private static Calendar createCalendar(TimeZone zone, Locale aLocale) {
7 |     '部分删减'
8 |     Calendar cal = null;
9 |
10 |    if (aLocale.hasExtensions()) {
11 |        String caltype = aLocale.getUnicodeCalendarType("ca");
```

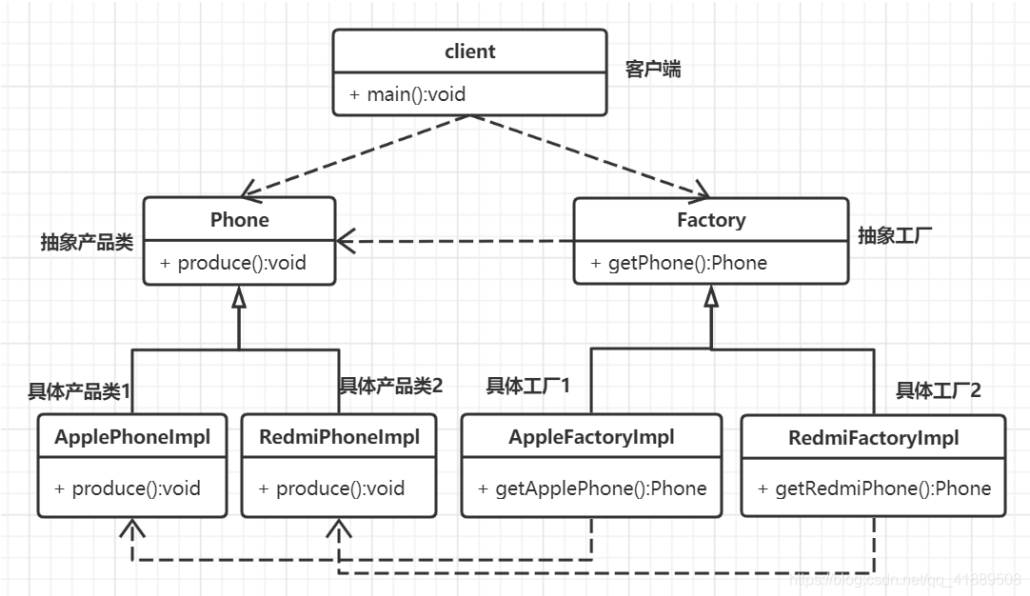


## (2) 工厂方法模式

### 1.认识

- ①一句话来说就是，定义一个创建对象的接口，让子类决定实例化那个类。因为当需要增加一个新的产品时，我们需要增加一个具体的产品类和与之对应的具体子工厂，然后在具体子工厂方法中进行对象实例化，所以称为工厂方法模式。
- ②具体来说就是定义一个用于创建对象的工厂接口，但让实现这个工厂接口的子类来决定实例化哪个具体产品类，工厂方法让类的实例化推迟到子类中进行。
- ③工厂方法模式非常符合“开闭原则”，当需要增加一个新的产品时，我们只需要增加一个具体的产品类和与之对应的具体工厂即可，无须修改原有系统。同时在工厂方法模式中用户只需要知道生产产品的具体工厂即可，无须关系产品的创建过程，甚至连具体的产品类名称都不需要知道。
- ④虽然他很好的符合了“开闭原则”，但是由于每新增一个新产品时就需要增加两个类，这样势必会导致系统的复杂度增加。

### 2.UML类图



#### UML说明：

苹果手机和红米手机实现了手机这个抽象类，苹果工厂和红米工厂实现了抽象工厂，苹果工厂当然要生产（依赖）苹果手机，红米工厂当然要生产（依赖）红米。客户要买苹果手机要去问苹果工厂要苹果手机，客户要买红米手机当然要去问红米工厂要红米手机。

#### 工厂方法有四个对象：

抽象产品类：提供抽象方法供具体产品类实现

具体产品类：提供具体的产品

抽象工厂：提供抽象方法供具体工厂实现

具体工厂：提供具体的工厂

#### 3.代码实现

- (1) 抽象产品类和简单工厂的抽象产品类一样
- (2) 具体产品类和简单工厂的具体产品类一样
- (3) 抽象工厂

```
1 public interface Factory {
2     Phone getPhone();
3 }
```

- (4) 具体工厂

```
1 public class AppleFactoryImpl implements Factory{
2
3     @Override
4     public Phone getPhone() {
5         return new ApplePhoneImpl();
6     }
7 }
```

```
1 public class RedmiFactoryImpl implements Factory{
2     @Override
3     public Phone getPhone() {
4         return new RedmiPhoneImpl();
5     }
6 }
7 }
```

- (5) 客户端

```
1 @Test
2 public void test1(){
3
4     Factory applePhoneFactory = new AppleFactoryImpl();
5     Factory redmiPhoneFactory = new RedmiFactoryImpl();
6
7     Phone applePhone = applePhoneFactory.getPhone();
8     Phone redmiPhone = redmiPhoneFactory.getPhone();
9
10    System.out.println(applePhone);
11    System.out.println(redmiPhone);
12
13    applePhone.produce();
14    redmiPhone.produce();
15 }
```

执行结果如下：

```
"C:\Program Files (x86)\WDYin\JDK\jdk8_64\bin\java.exe" ...

com.wander.design.factorymethod.product.ApplePhoneImpl@c46bcd4
com.wander.design.factorymethod.product.RedmiPhoneImpl@3234e239
生产苹果手机
生产了红米手机

Process finished with exit code 0
```

#### 4.总结

##### 优点：

- ①用户只需要关心所需产品的对应工厂，无需关心细节
- ②完全支持开闭原则，提高可扩展性。所谓的开闭原则就是对扩展开放，对修改关闭，再说白点就是实现工厂方法以后要进行扩展时不需要修改原有代码，只需要增加一个工厂实现类和产品实现类就可以。这样的好处可以降低因为修改代码引进错误的风险。

##### 缺点：

- ①每加入一种产品，会创建一个具体工厂类和具体产品类，因此，类的个数容易过多，增加复杂度。
- ②抽象工厂和抽象产品增加了系统的抽象性和理解难度

#### 5.工厂方法与简单工厂的区别

- ①可以看出，工厂方法模式特点：不仅仅做出来的产品要抽象，工厂也应该需要抽象。
- ②工厂方法使一个产品类的实例化延迟到其具体工厂子类。



- ③工厂方法的好处就是更拥抱变化。当需求变化，只需要增删相应的类，不需要修改已有的类。
- ④而简单工厂需要**修改**工厂类的方法，多方法静态工厂模式需要**增加**一个静态方法。

**缺点：**

引入抽象工厂层后，每次新增一个具体产品类，也要同时**新增**一个具体**工厂类**，所以我更青睐多方法静态工厂，每次新增一个具体产品类，工厂只需要**新增**一个**静态方法**。

6.应用场景

- (1) 客户端不知道它所需要的对象的类。（需要知道所需的对象的类使用升级版简单工厂模式，需要知道所需的参数的类使用简单工厂模式）。
- (2) 抽象工厂类通过其子类来指定创建哪个对象。
- (3) 简单工厂在源码中的使用–Collection：

Collection(抽象工厂)：

```
1 public interface Collection<E> extends Iterable<E> {
2     Iterator<E> iterator();
3 }
```

ArrayList(具体工厂)：

```
1 public class ArrayList<E>{
2     public Iterator<E> iterator() {
3         return new Itr();
4     }
5 }
```

Iterator(抽象产品)：

```
1 public interface Iterator<E> {
2     boolean hasNext();
3 }
```

Itr（具体产品）：

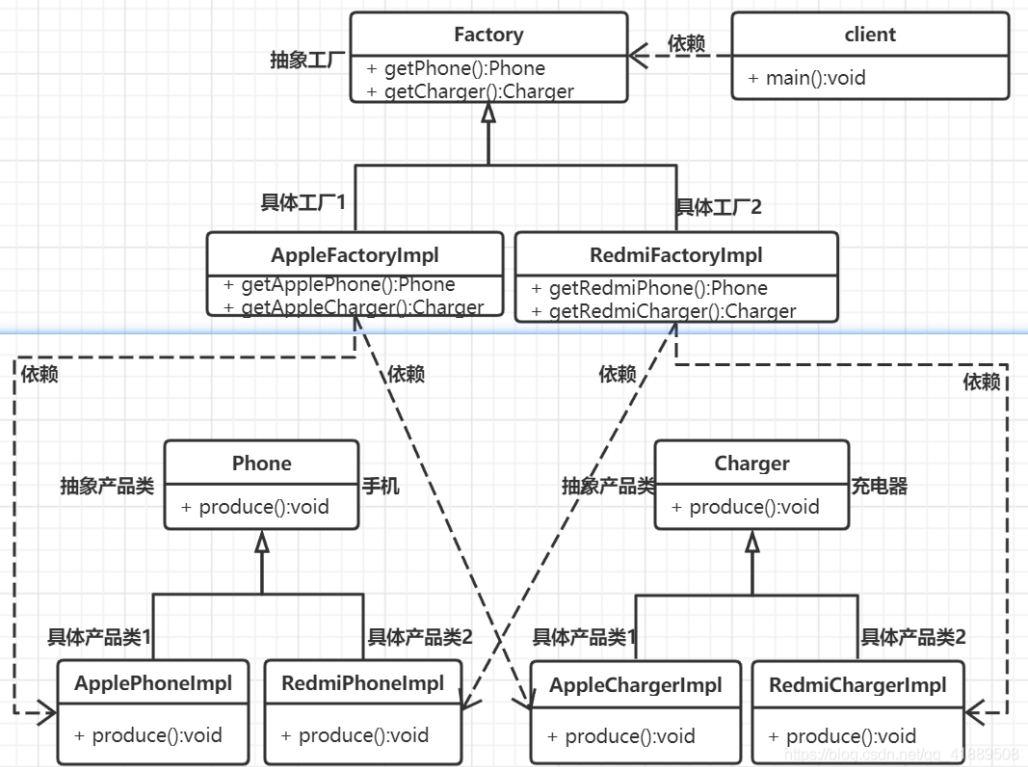
```
1 private class Itr implements Iterator<E> {
2     int cursor;          // index of next element to return
3     int lastRet = -1;    // index of last element returned; -1 if no such
4     int expectedModCount = modCount;
5
6     public boolean hasNext() {
7         return cursor != size;
8     }
9     '省略代码...'
10 }
```

(3) 抽象工厂模式

1.认识

- ①一句话来说就是，创建相关或依赖对象的家族，而无需明确指定具体类。因为我们可以定义具体产品类实现不止一个抽象工厂接口，一个工厂也可以生成不止一个产品类，是三个模式中较为抽象，并具一般性的模式。我们在使用中要注意使用抽象工厂模式的条件。
- ②所谓抽象工厂模式就是提供一个接口，用于创建相关或者依赖对象的家族，而不需要明确指定具体类。他允许客户端使用抽象的接口来创建一组相关的产品，而不需要关心实际产出的具体产品是什么。这样一来，客户就可以从具体的产品中被解耦。它的优点是隔离了具体类的生成，使得客户端不需要知道什么被创建了，而缺点就在于新增新的行为会比较麻烦，因为当添加一个新的产品对象时，需要更改接口及其下所有子类。

2.UML类图



**UML说明：**

具体的苹果手机产品和具体的红米手机产品实现了手机产品抽象类，具体的苹果充电器产品和具体的红米充电器产品实现了充电器产品抽象类。具体的苹果工厂和具体的红米工厂实现了手机抽象工厂，然后苹果工厂生产苹果手机和苹果充电器，红米工厂生成红米手机和红米充电器。客户想要苹果手机和苹果充电器就要向苹果工厂要产品（对象），客户想要红米手机和红米充电器就要向红米工厂要产品（对象）。

**工厂方法有四个对象：**

- ①**抽象产品类：**为每种具体产品声明接口，如图中Phone手机抽象类和Charger充电器抽象类
- ②**具体产品类：**定义了工厂生产的具体产品对象，实现抽象产品接口声明的业务方法，如图中ApplePhoneImpl、RedmiPhoneImpl，AppleChargerImpl,RedmiChargerImpl
- ③**抽象工厂：**它声明了一组用于创建一种产品的方法，每一个方法对应一种产品，如上述类图中的Factory就定义了两个方法，分别创建Phone和Charger
- ④**具体工厂：**它实现了在抽象工厂中定义的创建产品的方法，生产一组具体产品，这组产品构件成了一个产品种类，每一个产品都位于某个产品等级结构中，如上述类图中的AppleFactoryImpl和RedmiFactoryImpl

3.代码实现

- (1) 抽象的产品

```
1 public interface Phone {
2     void produce();
3 }
```

```
1 public interface Charger {
2     void produce();
3 }
```

(2) 具体的产品

①苹果具体的产品

```
1 public class AppleChargerImpl implements Charger{
2
3     @Override
4     public void produce() {
5         System.out.println("生产苹果充电器");
6     }
7 }
```

```
1 public class ApplePhoneImpl implements Phone {
2     @Override
3     public void produce() {
4         System.out.println("生产苹果手机");
5     }
6 }
```

②红米具体的产品

```
1 public class RedmiChargerImpl implements Charger{
2     @Override
3     public void produce() {
4         System.out.println("生产红米充电器");
5     }
6 }
7 }
```

```
1 public class RedmiPhoneImpl implements Phone {
2     @Override
3     public void produce() {
4         System.out.println("生产了红米手机");
5     }
6 }
```

(3) 抽象工厂

```
1 public interface Factory {
2     Phone getPhone();
3
4     Charger getCharger();
5 }
```

(4) 具体的工厂

```
1 public class AppleFactoryImpl implements Factory {
2
3     @Override
4     public Phone getPhone() {
5         return new ApplePhoneImpl();
6     }
7
8     @Override
9     public Charger getCharger() {
10        return new AppleChargerImpl();
11    }
12 }
```

```
1 public class RedmiFactoryImpl implements Factory {
2
3     @Override
4     public Phone getPhone() {
5         return new RedmiPhoneImpl();
6     }
7
8     @Override
9     public Charger getCharger() {
10        return new RedmiChargerImpl();
11    }
12 }
```

(5)客户端

```
1 @Test
2     public void test1(){
3         Factory appleFactory = new AppleFactoryImpl();
4         Phone applePhone = appleFactory.getPhone();
5         Charger appleCharger = appleFactory.getCharger();
6         System.out.println(appleFactory);
7         applePhone.produce();
8         appleCharger.produce();
9
10        Factory redmiFactory = new RedmiFactoryImpl();
11        Phone redmiPhone = redmiFactory.getPhone();
12        Charger redmiCharger = redmiFactory.getCharger();
13        System.out.println(redmiFactory);
14        redmiPhone.produce();
15        redmiCharger.produce();
16
17    }
```



(5) 执行结果

```
"C:\Program Files (x86)\WDYin\JDK\jdk8_64\bin\java.exe" ...

com.wander.design.abstractfactory.factory.AppleFactoryImpl@c46bcd4
生产苹果手机
生产苹果充电器
com.wander.design.abstractfactory.factory.RedmiFactoryImpl@3234e239
生产了红米手机
生产红米充电器

Process finished with exit code 0
```

[https://blog.csdn.net/qq\\_41889508](https://blog.csdn.net/qq_41889508)

4.总结

优点：

- ①具体产品在应用层代码隔离，无须关系创建细节
- ②将一个系列的产品统一到一起创建
- ③对于增加新的产品族（一个具体工厂就是一个产品族），抽象工厂模式很好地支持了“开闭原则”，只需要增加具体产品并对应增加一个新的具体工厂，对已有代码无须做任何修改。

缺点：

- ①规定了所有可能被创建的产品集合，产品族扩展新的产品（工厂中添加新的方法）困难。如果产品族扩展新的产品，需要修改所有的工厂角色，包括抽象工厂类，在所有的工厂类中都需要增加生产新产品的方法，违背了“开闭原则”。
- ②增加了系统的抽象性和理解难度

5.应用场景

抽象工厂在实际的开发中运用并不多，主要是在开发工程中很少会出现多个产品种类的情况，大部分情况使用以上两种工厂模式即可解决

6.个人总结

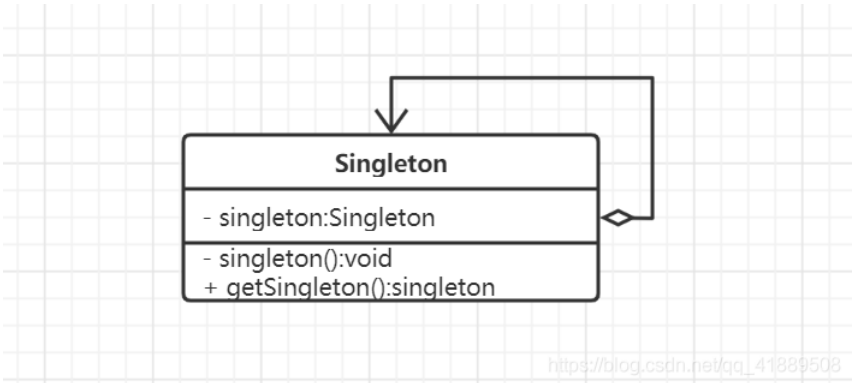
一句话总结工厂模式：**方便创建 同种产品类型的 复杂参数 对象**  
工厂模式重点就是适用于 **构建同产品类型**（同一个接口 基类）的不同对象时，这些对象new很复杂，**需要很多的参数**，而这些参数中大部分都是**固定**的，so，懒惰的程序员使用工厂模式封装之。  
（如果构建某个对象很复杂，需要很多参数，但这些参数大部分都是“**不固定**”的，应该使用建造者Builder模式）

（4）单例模式

1.认识

- ①一句话来说就是，某个类只能有一个实例，提供一个全局的访问点。
- ②单例模式的要点有三个：一是某个类只能有一个实例；二是它必须自行创建这个实例；三是它必须自行向整个系统提供这个实例。
- ③使用Singleton的好处还在于可以节省内存，因为它限制了实例的个数，有利于Java垃圾回收（garbage collection），而且确保所有对象都访问唯一实例。但是不适用于变化的对象，如果同一类型的对象总是要在不同的用例场景发生变化，单例就会引起数据的错误，不能保存彼此的状态。用单例模式，就是在适用其优点的状态下使用

2.UML类图



UML说明：

- 1.构造方法私有化：可以使得该类不被实例化即不能被new
- 2.在类本身里创建自己的对象
- 3.提供一个公共的方法供其他对象访问

3.代码实现

（1）饿汉式

①第一种：

```
1 public class Singleton {
2
3     /**
4      * static:
5      * ①表示共享变量，语意符合
6      * ②使得该变量能在getInstance()静态方法中使用
7      * final:
8      * ①final修饰的变量值不会改变即常量，语意也符合，当然不加final也是可以的
9      * ②保证修饰的变量必须在类加载完成时就已经进行赋值。
10     * final修饰的变量，前面一般加static
11     */
12     private static final Singleton singleton = new Singleton();
13
14     /**
15      * 私有化构造方法，使外部无法通过构造方法构造除singleton外的类实例
16      * 从而达到单例模式控制类实例数目的目的
17      */
18     private Singleton(){}
19
20     /**
21      * 类实例的全局访问方法
22      * 因为构造方法以及被私有化，外部不可能通过new对象来调用其中的方法
23      * 加上static关键词使得外部可以通过类名直接调用该方法获取类实例
24      * @return
25      */
26     public static Singleton getSingleton() {
27         return singleton;
28     }
29 }
```

②第二种

```
1 public class SingletonStatic {
2
3     private static final SingletonStatic singletonStatic;
4
5     /**
6      * 和第一种没有什么区别，这种看起来高大上面试装逼使用
7      */
8     static {
9         singletonStatic = new SingletonStatic();
10    }
11
12    private SingletonStatic() {}
13
14    public static SingletonStatic getSingletonStatic(){
15        return singletonStatic;
16    }
17 }
```

说明：

①优点：一般使用static和final修饰变量（具体作用已经在代码里描述了），只在类加载时才会初始化，以后都不会，线程绝对安全，无锁，效率高。

②缺点：类加载的时候就初始化，不管用不用，都占用空间，会消耗一定的性能(当然很小很小，几乎可以忽略不计，所以这种模式在很多场合十分常用而且十分简单)

注：这里有两个小知识点：

a.如果是final非static成员，必须在构造器、代码块、或者直接定义赋值

b.如果是final static 成员变量，必须直接赋值 或者在静态代码块中赋值

## (2) 懒汉式

```
1 public class Singleton {
2
3     private static Singleton singleton = null;
4
5     private Singleton(){}
6
7     public static Singleton getSingleton() {
8         if(singleton == null){
9             singleton = new Singleton();
10        }
11        return singleton;
12    }
13 }
```

说明：

①优点：在外部需要使用的时候才进行实例化，不使用的时候不会占用空间。

②缺点：线程不安全。看上去，这段代码没什么明显问题，但它不是线程安全的。假设当前有N个线程同时调用

getInstance（）方法，由于当前还没有对象生成，所以一部分同时都进入if语句new Singleton(),那么就会由多个线程创建多个多个user对象。

## (3) 线程安全的懒汉式

```
1 public class Singleton {
2
3     private static Singleton singleton;
4
5     private Singleton(){};
6
7     private static synchronized Singleton getSingleton(){
8         if(singleton == null){
9             singleton = new Singleton();
10        }
11        return singleton;
12    }
13 }
```

说明：

①优点：解决了懒汉式线程不安全的问题

②缺点：线程阻塞，影响性能。

## (4) DCL单例----高性能的懒汉式

```
1 public class Singleton {
2     /*volatile在这里发挥的作用是：禁止指令重排序（编译器和处理器为了优化程序性能
3     * 而对指令序列进行排序的一种手段。）
4     * singleton = new Singleton();这句代码是非原子性操作可分为三行伪代码
5     * a:memory = allocate() //分配内存，在jvm堆中分配一段区域
6     * b:ctorInstanc(memory) //初始化对象，在jvm堆中的内存中实例化对象
7     * c:instance = memory //赋值，设置instance指向刚分配的内存地址
8     * 上面的代码在编译运行时，可能会出现重排序从a-b-c排序为a-c-b。
9     * 重排序是为了优化性能，但是不管怎么重排序，在单线程下程序的执行结果不能被改变
10    * 保证最终一致性。而在多线程环境下，可能发生重排序，会影响结果。
11    * ①若A线程执行到代码singleton = new Singleton()时;
12    * ②同时若B线程进来执行到代码到第一层检查if (singleton == null)
13    * ③当cpu切换到A线程执行代码singleton = new Singleton();时发生了指令重排序，
14    * 执行了a-b，没有执行c,此时的singleton对象只有地址，没有内容。然后cpu又切换到了B线程，
15    * 这时singleton == null为false (==比较的是内存地址)，
16    * 则代码会直接执行到了return，返回一个未初始化的对象（只有地址，没有内容）。
17    */
18    private volatile static Singleton singleton;
19
20    private Singleton() {
21    }
22
23    public static Singleton getSingleton() {
24        /* 第一层检查，检查是否有引用指向对象，高并发情况下会有多个线程同时进入
25        * ①当多个线程第一次进入，所有线程都进入if语句
26        * ②当多个线程第二次进入，因为singleton已经不为null，因此所有线程都不会进入if语句，
27        * 即不会执行锁，从而也就不会因为锁而阻塞，避免锁竞争*/
28        if (singleton == null) {
29            /* 第一层锁，保证只有一个线程进入，
30            * ①多个线程第一次进入的时候，只有一个线程会进入，其他线程处于阻塞状态
31            * 当进入的线程创建完对象出去之后，其他线程又会进入创建对象，所以有了第二次if检查
32            * ②多个线程第二次是进入不到这里的，因为已被第一次if检查拦截*/
33            synchronized (Singleton.class) {
34                /* 第二层检查，防止除了进入的第一个线程的其他线程重复创建对象*/
35                if (singleton == null) {
36                    singleton = new Singleton();
37                }
38            }
39        }
```

```
40 |         return singleton;
41 |     }
42 | }
```

说明:

代码注释已详细讲解volatile在该单例模式的作用，已经双重锁的作用。

①优点：解决了线程阻塞的问题

②缺点：多个线程第一次进入的时候会造成大量的线程阻塞，代码不够优雅。

(5) 静态内部类的方式

```
1 | public class Singleton {
2 |
3 |     private Singleton(){}
4 |
5 |     private static class LayzInner{
6 |         private static Singleton singleton = new Singleton();
7 |     }
8 |
9 |     public static Singleton getSingleton(){
10 |         return LayzInner.singleton;
11 |     }
12 | }
```

说明:

①优点：第一次类创建的时候加载，避免了内存浪费，不存在阻塞问题，线程安全，唯一性

②缺点：序列化-漏洞：反射，会破坏内部类单例模式

(6) 枚举单例模式

```
1 | public enum EnumSingleton {
2 |     INSTANCE;
3 |     private Singleton singleton;
4 |     EnumSingleton(){
5 |         singleton = new Singleton();
6 |     }
7 |     public Singleton getSingleton(){
8 |         return singleton;
9 |     }
10 | }
```

说明:

单元素的枚举类型已经成为实现Singleton的最佳方法，无法反射创建对象，但是特殊的饿汉式。

(7) 静态内部类升级版

借鉴枚举单例的内部实现的方式

```
1 | public class Singleton {
2 |     private Singleton(){
3 |         if(LayzInner.singleton != null){
4 |             throw new RuntimeException("不能够进行反射! ");
5 |         }
6 |     }
7 |
8 |     private static class LayzInner{
9 |         private static Singleton singleton = new Singleton();
10 |     }
11 |
12 |     public static Singleton getSingleton (){
13 |         return LayzInner.singleton;
14 |     }
15 | }
```

说明:

①优点：第一次类创建的时候加载，避免了内存浪费，不存在阻塞问题，线程安全，唯一性，解决了反射会破坏内部类单例模式的问题

②缺点：不是官方的

(8) 容器式单例

```
1 | public class Singleton {
2 |     private Singleton() {
3 |     }
4 |
5 |     private static Map<String, Object> ioc = new ConcurrentHashMap<>();
6 |
7 |     public static Object getBean(String className) {
8 |         synchronized (ioc) {
9 |             if (ioc.containsKey(className)) {
10 |                 Object o = null;
11 |                 try {
12 |                     o = Class.forName(className).newInstance();
13 |                 } catch (Exception e) {
14 |                     e.printStackTrace();
15 |                 }
16 |                 return o;
17 |             } else {
18 |                 return ioc.get(className);
19 |             }
20 |         }
21 |     }
22 | }
```

说明: Spring ioc 单例 是懒汉式 枚举上的升级

(9) ThreadLocal单例

```
public class Singleton {
    private Singleton(){}
    private static final ThreadLocal<Singleton> threadLocal =
        new ThreadLocal<Singleton>(){
            @Override
            protected Singleton initialValue(){
                return new Singleton();
            }
        };
    public static Singleton getInstance(){
        return threadLocal.get();
    }
}
```

在Spring的第三方包baomidou的多数  
据源中，有用到这种写法

说明：局部单例模式：某一个线程里唯一

①ThreadLocal的作用呢，是提供线程内的局部变量，在多线程环境访问时，能保证各个线程内的ThreadLocal变量各自独立。也就是说每个线程的ThreadLocal变量是自己专用的，其他线程是访问不到的。

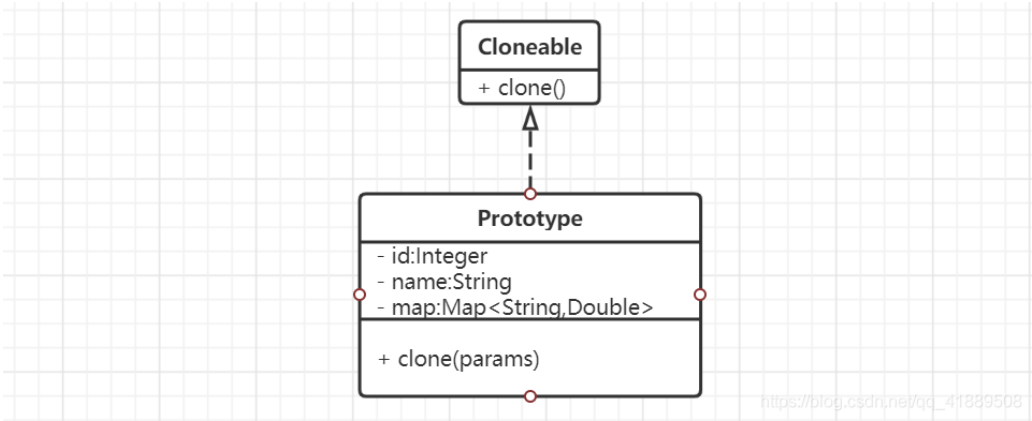
②ThreadLocal最常用于在多线程环境下存在对非线性安全对象的并发访问，而且该对象不需要在线程内共享，如果对该对象加锁，会造成大量线程阻塞影响程序性能，这时候就可以使用ThreadLocal来使每个线程都持有该对象的副本，这是典型的空间换取时间从而提高执行效率的方式。例如项目里经常使用的SimpleDateFormat日期格式化对象，该对象是线程不安全的，而且不需要在线程内共享，因此可以使用ThreadLocal保证其线程安全。

(5) 原型模式

1.认识

- ①一句话来说就是，通过复制现有的实例来创建新的实例。因为是同过原有的对象创建新的对象，所以称为原型模式。
- ②原型模式是用原型实例指定创建对象的种类，并且通过复制这些原型创建新的对象。原型模式允许一个对象再创建另外一个可定制的对象，无须知道任何创建的细节。
- ③用于创建重复的对象，同时又能保证性能。
- (1) 浅拷贝：我们只拷贝对象中的基本数据类型（8种），对于数组、容器、引用对象等都不会拷贝，只会拷贝对这些对象的引用。
- (2) 深拷贝：不仅能拷贝基本数据类型，还能拷贝那些数组、容器、引用对象（不仅拷贝对这些对象的引用，而且拷贝对象本身）。

2.UML类图



UML说明：  
实体类实现Cloneable接口，重写clone方法

3.代码实现

(1) Prototype类：

```
1 public class Prototype implements Cloneable {
2
3     private Integer id;
4     private String name;
5     private Map<String, Double> map;
6
7     @Override
8     protected Prototype clone() throws CloneNotSupportedException {
9         // 浅拷贝方式
10        Prototype prototype = (Prototype) super.clone();
11        // 深拷贝方式：对每一个复杂类型分别进行克隆
12        // 测试浅拷贝的时候注释下面代码
13        prototype.map = (Map<String, Double>) ((HashMap)this.map).clone();
14        return prototype;
15    }
16
17    public Prototype(Integer id, String name, Map<String, Double> map) {
18        this.id = id;
19        this.name = name;
20        this.map = map;
21    }
22    /**省略get、set方法和toString方法*/
23 }
```

(2) 客户端：

```
1 public class Client {
2     @Test
3     public void test() throws CloneNotSupportedException {
4         Map<String, Double> map = new HashMap<>();
5         map.put("数学",100D);
6         Prototype prototype = new Prototype(1,"小明",map);
7
8         Prototype prototype1 = prototype.clone();
9         Map<String, Double> map1 = prototype1.getMap();
10        map1.put("数学",99d);
11
12        System.out.println(prototype);
13        System.out.println(prototype1);
14    }
15 }
```

(3) 执行结果：

浅拷贝：

```
"C:\Program Files (x86)\WDYin\JDK\jdk8_64\bin\java.exe" ...

Prototype{id=1, name='小明', map={数学=99.0}}
Prototype{id=1, name='小明', map={数学=99.0}}

Process finished with exit code 0
```

改变其中一个对象map的值，两个对象的map内容都发生了变化

深拷贝：

```
"C:\Program Files (x86)\WDYin\JDK\jdk8_64\bin\java.exe" ...

Prototype{id=1, name='小明', map={数学=100.0}}
Prototype{id=1, name='小明', map={数学=99.0}}

Process finished with exit code 0
```

改变其中一个对象map的值，该对象的map内容发生了变化，另一个对象map的内容没有发生变化

#### 4.总结

优点：

- ①提高了性能，在需要短时间创建大量的对象和创建对象很耗时的情况下，原型模式比通过new对象大大提高了时间效率。
- ② 逃避构造函数的约束。

缺点：

- 1、配备克隆方法需要对类的功能进行通盘考虑，这对于全新的类不是很难，但对于已有的类不一定很容易，特别当一个类引用不支持串行化的间接对象，或者引用含有循环结构的时候。
- 2、实现原型模式每个派生类都必须实现 Clone接口。

#### 5.应用场景

- 1.通过new产生一个对象需要非常繁琐的数据准备或访问权限，则可以使用原型模式。比如，向数据库表插入多条测试数据，可以用到。
- 2.在实际项目中，原型模式很少单独出现，一般是和工厂方法模式一起出现，通过clone的方法创建一个对象，然后由工厂方法提供给调用者。原型模式已经与Java融为浑然一体，大家可以随手拿来使用。

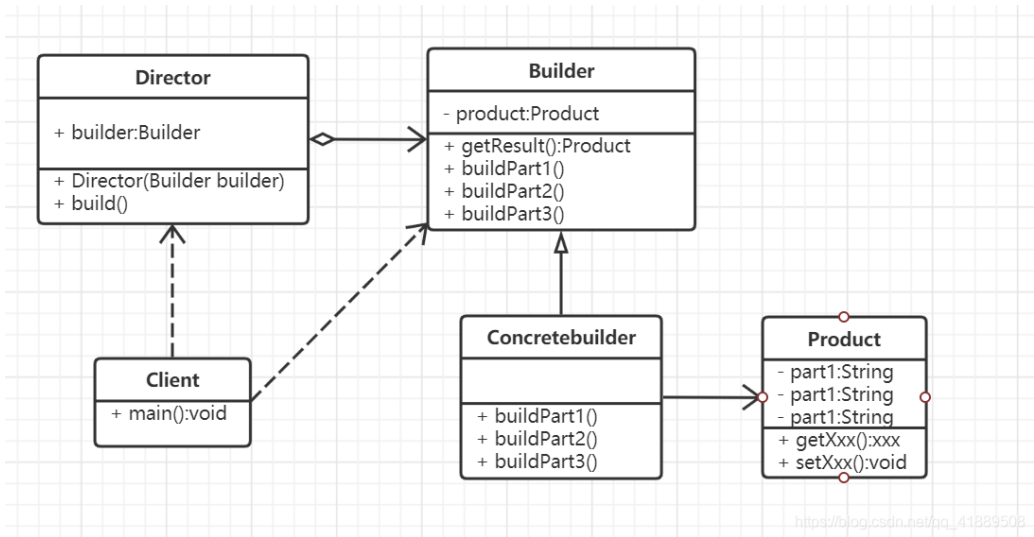
### (6) 建造者模式

1.认识：

- ①一句话来说：封装一个复杂对象的构建过程，并可以按步骤构造。因为需要对对象一步步建造起来，所以称为建造者模式。
- ②将复杂产品的构建过程封装分解在不同的方法中，使得创建过程非常清晰，能够让我们更加精确的控制复杂产品对象的创建过程，同时它隔离了复杂产品对象的创建和使用，使得相同的创建过程能够创建不同的产品。但是若内部变化复杂，会有很多的建造类。

#### 2.传统的builder模式

①.UML类图：



UML说明：

- Product（产品角色）： 一个具体的产品对象。
- Builder（抽象建造者）： 创建一个Product对象的各个部件指定的抽象接口。
- ConcreteBuilder（具体建造者）： 实现抽象接口，构建和装配各个部件。
- Director（指挥者）： 构建一个使用Builder接口的对象。它主要是用于创建一个复杂的对象。它主要有两个作用，一是：隔离了客户与对象的生产过程，二是：负责控制产品对象的生产过程。

②.代码如下：

1.产品类：

```
1 public class Product {
2     private String part1;//可以是任意类型
3     private String part2;
4     private String part3;
5     /**set get 方法省略
6 }
7
```

2.抽象建造者

```
1 public abstract class Builder{
2
3     Product product = new Product();
4     public abstract void buildPart1();
5     public abstract void buildPart2();
6     public abstract void buildPart3();
7     public Product getResult(){
8         return product;
9     };
10 }
```

3.具体建造者

```
1 public class ConcreteBuilder extends Builder {
2
3     @Override
4     public void buildPart1() {
5         System.out.println("建造part1");
6     }
7
8     @Override
9     public void buildPart2() {
10        System.out.println("建造part2");
11    }
12
13    @Override
14    public void buildPart3() {
15        System.out.println("建造part3");
16    }
17 }
```

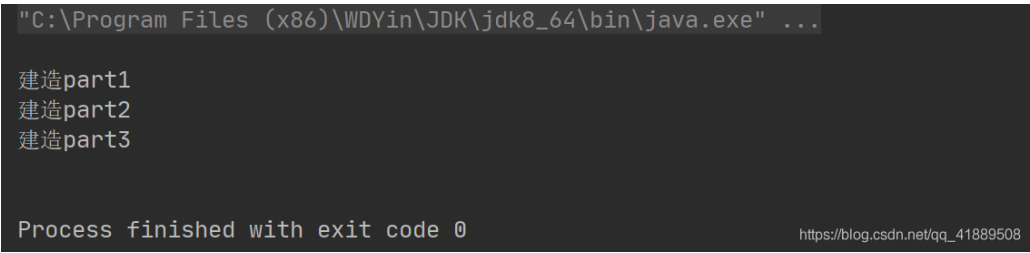
4.指挥者：

```
1 public class Director {
2
3     private Builder builder;
4
5     public Director(Builder builder) {
6         this.builder = builder;
7     }
8
9     public Product build(){
10        builder.buildPart1();
11        builder.buildPart2();
12        builder.buildPart3();
13        return builder.getResult();
14    }
15 }
```

5.客户端

```
1 public class Client {
2
3     @Test
4     public void test() {
5         Builder builder = new ConcreteBuilder();
6         Director director = new Director(builder);
7         director.build();
8     }
9
10 }
```

6.执行结果



3.传统builder模式的变种

①说明：

一个场景：当一个类的构造函数参数个数超过3个，而且这些参数有些是可选的参数，考虑使用构造者模式。

传统builder模式的变种， 首先省略了director 这个角色，将对象的构建交给了client端，然后将builder 写到了要构建的产品类里面，最后采用了链式调用。

②代码：

第一种：构造函数实现

```
1 public class product {
2     //主键
3     private Long id;
4     //产品名称
5     private String name;
6     //产品类型
7     private String type;
8     //产品价格
9     private String price;
10
11     public product(Long id) {
12         this.id = id;
13     }
14
15     public product(Long id, String name) {
16         this.id = id;
17         this.name = name;
18     }
19
20     public product(Long id, String name, String type) {
21         this.id = id;
22         this.name = name;
23         this.type = type;
24     }
25
26     public product(Long id, String name, String type, String price) {
27         this.id = id;
28         this.name = name;
29         this.type = type;
30         this.price = price;
31     }
32 }
```

第二种：JavaBean的实现方式



```
1 public class product {
2
3     //主键
4     private Long id;
5     //产品名称
6     private String name;
7     //产品类型
8     private String type;
9     //产品价格
10    private String price;
11
12    public Long getId() {
13        return id;
14    }
15
16    public String getName() {
17        return name;
18    }
19
20    public String getType() {
21        return type;
22    }
23
24    public String getPrice() {
25        return price;
26    }
27 }
```

以上两种方式存在的弊端：

第一种主要是使用及阅读不方便。你可以想象一下，当你要调用一个类的构造函数时，你首先要决定使用哪一个，然后里面又是一堆参数，如果这些参数的类型很多又都一样，你还要搞清楚这些参数的含义，很容易就传混了。。。那酸爽谁用谁知道。

第二种方式在构建过程中对象的状态容易发生变化，造成错误。因为那个类中的属性是分步设置的，所以就容易出错。

为了解决这两个痛点，builder模式就横空出世了。

第三种：

1.实现步骤：

- ①在Product中创建一个静态内部类 Builder，然后将Product 中的参数都复制到Builder类中。
- ②在Product中创建一个private的构造函数，参数为Builder类型
- ③在Builder中创建一个public的构造函数
- ④在Builder中创建设置函数，对Product中那些可选参数进行赋值，返回值为Builder类型的实例
- ⑤在Builder中创建一个build()方法，在其中构建Product的实例并返回

2.代码：

product产品：

```
1 public class product {
2     //主键
3     private Long id;
4     //产品名称
5     private String name;
6     //产品类型
7     private String type;
8     //产品价格
9     private String price;
10
11    public product(Builder builder) {
12        this.id = builder.id;
13        this.name = builder.name;
14        this.type = builder.type;
15        this.price = builder.price;
16    }
17
18    public static class Builder {
19        //主键
20        private Long id;
21        //产品名称
22        private String name;
23        //产品类型
24        private String type;
25        //产品价格
26        private String price;
27
28        public Builder() {
29
30        }
31
32        public Builder setId(Long id) {
33            this.id = id;
34            return this;
35        }
36
37        public Builder setName(String name) {
38            this.name = name;
39            return this;
40        }
41
42        public Builder setType(String type) {
43            this.type = type;
44            return this;
45        }
46
47        public Builder setPrice(String price) {
48            this.price = price;
49            return this;
50        }
51
52        public product build() {
53            return new product(this);
54        }
55    }
56 }
```

client客户端：

```
1 public class Client {
2
```

```
3 public static void main(String[] args) {
4
5     Product product = new Product.Builder()
6         .setId(1L)
7         .setName("产品")
8         .setPrice(2d)
9         .build();
10 }
11 }
```

#### 4.总结

优点：

- 1、建造者独立，易扩展。将复杂产品的构建过程封装分解在不同的方法中，使得创建过程非常清晰，能够让我们更加精确的控制复杂产品对象的创建过程。
- 2、便于控制细节风险。它隔离了复杂产品对象的创建和使用，使得相同的创建过程能够创建不同的产品。

缺点：

- 1、产品必须有共同点，范围有限制。
- 2、如内部变化复杂，会有很多的建造类，导致系统庞大。

应用场景

- 1、需要生成的对象具有复杂的内部结构。
- 2、需要生成的对象内部属性本身相互依赖。

#### 5.应用场景

JAVA 中的 StringBuilder和Lombok中的@Build注解

### 六、个人体会

设计模式是一种解决问题的思维和方式，不要生搬硬套，为了设计模式而模式~~

想要继续学习结构性模式和行为型模式的童鞋可以关注小编哦，稍候更新~~

**必看文章：** [Mysql执行顺序写sql不再是问题\\*\\*！](#)！\*\*

想详细了解[微服务与分布式的](#)可以看小编这篇博客

[微服务 分布式 集群 负载均衡详述](#)

[spring boot springcloud dubbo概述](#)

对SpringCloud有兴趣的，关注一下小编的博客，随后更新

如果看到这里，说明你喜欢这篇文章，请转发，点赞。关注微信公众号微信搜索[[老板再来一杯时光](#)]回复[[进群](#)]或者扫描下方二维码即可进入无广告交流群！[【进群】](#)即可获取[【java基础经典面试】](#)一份和获取[【DDD领域驱动设计实战落地解惑】](#)PDF一份

限时免费中

2.5万字详解23种设计模式—创建型模式（简单工厂、工厂方法、抽象工厂、单例-多线程安全详解、建造者、原型）的详细解读、UML类图



向“C知道”追问



PHP中常用的三种[设计模式详解](#)【[单例模式](#)、[工厂模式](#)、[观察者模式](#)】

01-02

本文实例讲述了PHP中常用的三种[设计模式](#)。分享给大家供大家参考，具体如下： PHP中常用的三种[设计模式](#)：[单例模式](#)、[工厂模式](#)、观...

创建者模式

weixin\_55086330的博客

614

创建者模式 这类模式提供创建对象的机制，能够提升已有[代码](#)的灵活性和可复用性。创建者模式包括：工厂方法、抽象工厂、生成器、...

日志解析LogParser类的[工厂](#)注册[单例](#)模式实现（[线程安全](#)）

碣石观海的博客

171

一、LogParserFactory的实现 1.为每个调用线程分配一个唯一的LogParser（日志解析）对象，以使在线程内部重复使用；以键值对<Thread,...

Java[设计模式](#)创建者模式（1）--- [单例设计模式](#)

shuai\_h的博客

35

[单例](#)模式（Singleton Pattern）是 Java 中最简单的[设计模式](#)之一。这种类型的[设计模式](#)属于[创建型模式](#)，它提供了一种创建对象的最佳方...

Java[设计模式](#)----创建者模式

m0\_57098080的博客

571

Java中[设计模式](#)概述以及五种创建者模式

[2.5万字详解23种设计模式—创建型模式（简单工厂、工厂方法、抽象工厂、单例-多线程安全详解、建造者、原型）的详细解读、UML类图](#)

qc

294

25000 字[详解](#) 23 种[设计模式](#)（多图 + [代码](#)） [最新发布](#)

weixin\_45727359的博客

30

.目录[创建型模式](#)结构型模式行为型模式总结前言一直想写一篇介绍[设计模式](#)的文章，让读者可以很快看完，而且一看就懂，看懂就会用...

java语言程序设计与数据结构基础篇，[2万字](#)20个项目实例

EDGNB123的博客

974

一、前言 聊的是八股的文，干的是搬砖的活！面我的题开发都用不到，你为什么要闻？可能这是大部分程序员求职时的经历，甚至也是...

Java 中的面向对象编程（高级篇，[2万字详解](#)）总

林二月的博客

565

一、类变量/静态变量 1. 类变量引入 提出一个问题： 有一群小孩在玩堆雪人,不时有新的小孩加入,请问如何知道现在共有多少人在玩?, 编...

[2.5 万字详解：23 种设计模式](#)

公众号：Java后端

276

本文简述了各大[设计模式](#)，并通过UML和[代码](#)详细说明。本文大约共 2.5W 字，建议收藏。下方是本文的目录：一、[设计模式](#)的认识二、...

2018体验营销科技指南：全球营销云[解读](#)白皮书

08-13

[解读](#)个性化营销云的应用；个性化营销云在客服领域的应用；个性化营销云应用在跨平台用户行为追踪；个性化营销云对广告精准投放与...

PHP[设计模式](#)（一）[工厂](#)模式Factory实例[详解](#)【[创建型](#)】

12-18

本文实例讲述了PHP[设计模式](#)（一）[工厂](#)模式Factory。分享给大家供大家参考，具体如下： 在面向对象编程中, 最通常的[方法](#)是一个new...

黑马Spring的4天教程--第一天：[工厂](#)模式、[单例](#)对象的[线程安全问题](#)、[单例工厂](#)模式、Spring负... qq\_26882339的博客

324

在ui的包中，调用service包中的类，进而调用dao包中的类。 实体层domain包中，是User类的属性和set、get、和toString等[方法](#)，有些书上...

手写一个[线程安全](#)的[单例工厂](#)模式

Monday\_\_Friday的博客

1198

简单介绍一下两个模式 [单例](#)模式：一个类只会被产生一个静态的对象。 [工厂](#)模式：构造[方法](#)不对外提供。提供一个[方法](#)，包括产生对象...

Java[设计模式](#)之创建者模式

不当初

590

[Java[设计模式](#)之[创建型模式](#)1 简介 在最近看的一篇文章中，提到了关于新年目标制定的方式的实践，觉得不错，附上图。尤其是目标...

Java中23种[设计模式](#)详解

HERO\_1990的博客

63

一、[设计模式](#)的分类 总体来说[设计模式](#)分为三大类：[创建型模式](#)，共 5 种：[工厂方法](#)模式、[抽象工厂](#)模式、[单例](#)模式、[建造者](#)...

[建造者](#)模式

weixin\_47382783的博客

1287

一、[建造者](#)模式介绍 1、定义 [建造者](#)模式又叫生成器模式，是一种对象构建模式。它是将一个复杂的对象的构建与它的表示分离，使得同...

[抽象工厂](#)模式与[建造者](#)模式——用一个实例来解释

虚幻的元亨利贞的博客

1025

最近在学习[设计模式](#)，在[创建型模式](#)中比较难理解的主要是[工厂](#)模式与[建造者](#)模式，二者有许多相似之处，但是本质上还是不同的。 1.抽...

Java 23[设计模式](#)代码详解


“相关推荐”对你有帮助么？

 非常没帮助

 没帮助

 一般

 有帮助

 非常有帮助

关于我们 招贤纳士 商务合作 寻求报道 400-660-0108 kefu@csdn.net 在线客服 工作时间 8:30-22:00

公安备案号11010502030143 京ICP备19004658号 京网文〔2020〕1039-165号 经营性网站备案信息 北京互联网违法和不良信息举报中心 家长监护 网络110报警服务 中国互联网举报中心 Chrome商店下载 账号管理规范 版权与免责声明 版权申诉 出版物许可证 营业执照

©1999-2023北京创新乐知网络技术有限公司



王德印

码龄5年

暂未认证

18	12万+	4万+	6万+	
原创	周排名	总排名	访问	等级
841	338	232	78	1284
积分	粉丝	获赞	评论	收藏





私信


关注

搜博文主文章

🔍

## 热门文章

2.5万字详解23种设计模式—创建型模式  
(简单工厂、工厂方法、抽象工厂、单例-多线程安全详解、建造者、原型)的详细解读、UML类图、及代码演示 9494

微服务springcloud环境下基于Netty搭建  
websocket集群实现服务器消息推送---netty  
是yyds  8570

2.5万字讲解DDD领域驱动设计，从理论到实践掌握DDD分层架构设计，赶紧收藏起来吧  7097

详解单体架构 微服务 微服务架构 微服务各个组件 分布式 集群 负载均衡 5039

代码中如何干掉太多的if else即if else的多种替代方案以提高代码质量通过公司代码审查

4591

## 分类专栏

	大数据	1 篇
	kafka	1 篇
	DDD	1 篇
	领域驱动设计	1 篇
	设计模式	1 篇
	ajax	1 篇

### 最新评论

springboot kafka 实现延时队列  
王德印: 加群领取完整代码

springboot kafka 实现延时队列  
回眸一笑 我就那么我看你: 是不是少给了一个KafkaDelayConfig文件

springboot kafka 实现延时队列

qq\_41506474: 对呀，想问kafka延时队列如何实现低资源消耗呢？

微服务springcloud环境下基于Netty搭建we...  
王德印: 关注微信公众号加群领取源码

微服务springcloud环境下基于Netty搭建we...  
user\_1688: 不是所有的微服务都需要gateway，我的理解是这样的

您愿意向朋友推荐“博客详情页”吗?



强烈不推荐 不推荐 一般般 推荐 强烈推荐

## 最新文章

## springboot kafka 实现延时队列

springboot下使用最常用的【策略设计模式】  
优雅干掉if else!!!

2.5万字讲解DDD领域驱动设计，从理论到实践掌握DDD分层架构设计，赶紧收藏起来吧

2023年 1篇	2022年 4篇
2021年 2篇	2020年 4篇
2019年 7篇	

## 目录

## 设计模式

## 一、设计模式的认识

## 二、设计模式的分类

- (1) 根据其目的
- (2) 根据范围

### 三、设计模式的优点

#### 四、设计模式中关键点

## 五、创建型模式

- (1) 简单（静态）工厂模式
- (2) 工厂方法模式
- (3) 抽象工厂模式
- (4) 单例模式

