# 设计模式的七大基本原则

1.单一职责：一个类只负责一个领域类的相应职责，也就是说一个类负责一项功能

2.开闭原则：开放扩展，关闭修改

3.里氏替换：继承的时候子类除了新增功能外，尽量不要修改父类方法的预期行为

4.依赖倒置：抽象不应依赖于实现，实现依赖于抽象，也就是说要针对接口编程，不要针对实现编程

5.隔离原则：为各个类建立它们需要的专用接口，不要去建立一个庞大的接口供所有依赖它的类去调用

6.合成复用：尽量使用对象组合，不要使用继承来达到复用的目的

7.迪米特法则：最少知识原则，就是说一个对象应对其他对象尽可能的少了解，也就是**降低类之间的耦合**，每个类尽量减少对其他类的依赖；是的系统功能模块能相互独立

相互之间尽量少存在依赖关系

# 设计模式分类

## 对象创建型模式

### 简单工厂模式

定义一个工厂类，根据传入的参数的不同返回不同类的实例，被创建的实例通常具有共同的父类

你需要什么只需要传入一个正确的参数，就可以获取你需要的对象，不需要关系对象的创建过程;

工厂类需要有必要的逻辑判断，可以决定什么时候创建那一类产品类的实例，客户端可以免除创建对象的职责，仅仅消费对象即可；**简单工厂模式实现了对象创建和对象使用的分离**

可以通过使用配置文件，在不修改工厂类代码的前提下实现更换和新增的具体产品类，能够提高代码的灵活度

缺点

工厂类集中了所有产品类的创建逻辑，职责过重，一旦不能使用，那么整个系统都会受到影响

使用简单工厂模式，当新增产品时就会增加产品类，这样就会增加系统的复杂度，也违背了开闭原则

新增产品类时，会修改工厂类中的逻辑，那么增加了工厂类的复杂度，在工厂类中有大量的if..else代码，不利于系统扩展和维护

使用场景

产品类较少的情况下适合使用，由于创建的对象较少，不会让业务逻辑太过于复杂

客户端只需要传入参数，不需要关心对象具体的创建过程

### 工厂方法模式

新增产品不影响现有的代码

定义一个创建对象的接口，让子类来决定创建那个对象

角色有

抽象工厂角色：定义创建产品对象的接口，提供创建方法，所有的工厂类都有实现这个接口

抽象产品角色：产品接口，定义对象的超类型，也就是产品类的公共父类（或者公共接口）

具体工厂角色：创建具体的产品实例

具体产品角色：

优点

工厂方法模式是对简单工厂模式的延伸，弥补了简单工厂模式的不足

1.在工厂方法模式中，工厂方法用来创建具体的实例对象，同时隐藏了哪种产品类将被实例化这一细节，用户只需要关心产品对应的工厂类，不需要关心创建的过程，甚至不需要关心具体产品类的类名

2.工厂方法模式是基于多态设计的，能让工厂自己决定创建那种产品对象，如何创建对象的细节完全封装在具体工厂内部，（创建什么样的产品对象由具体工厂类自己决定）

3.当系统新增产品时，不需要修改原来的代码，只需要增加新的具体工厂和具体产品就可以了，满足开闭原则

缺点

新增产品，那么就要增加具体产品类和具体工厂方法类，在一定程度上增加了系统的复杂度，也给jvm的编译和运行带来了额外的开销

因为考虑到了可扩展性，引入了抽象层，在客户端均使用抽象层来进行定义，这就增加了系统的抽象性和代码的理解难度

使用场景

客户端不需要知道具体产品的类名，只需要知道其对应的具体工厂类即可，具体产品对象由它对应的具体工厂类来创建对象

通过抽象工厂的子类来创建指定的实例，利用面向对象的多态性和里氏替换原则，在程序运行时子类覆盖父类，是程序更加易于扩展

### 抽象工厂模式

工厂方法模式通过引入抽象工厂角色，来解决了简单工厂模式中工厂类职责过重的问题，当时每个工厂类只能生产一个产品实例，可能会导致系统中存在大量的具体工厂类，会增加系统开销，此时就可以考虑将相关产品组成一个产品族（一个产品下面的不同类型的子产品）

产品等级结构：产品继承结构，

产品族：一个工厂生产的不同产品等级结构的产品

抽象工厂：定义一组创建产品族的方法，每个方法对应一种产品

抽象产品

具体工厂

具体产品

开闭原则的倾向性：

抽象工厂模式的最大缺点：增加产品族很方便，但是增加产品等级结构很麻烦，抽象工厂模式的这种性质称为开闭原则的倾向性，

增加产品族，对产品族的增加，能够很好的满足开闭原则，只需要增加具体产品并增加对应的工厂类，对现有的代码无效做任何的修改

增加产品等级结构：对应新增产品等级结构，需要修改对应的工厂类，增加创建对应产品的方法，违背了开闭原则

优点：

隔离了具体类实例的创建，使得客户不用知道什么被创建，由于这种隔离让跟换一个具体工厂变得相对容易

当产品族中多个对象被设计成一起工作时，它能够保证客户始终只使用一个产品族中的对象

增加产品族很方便，不需要修改原来的代码

缺点

对应新增产品等级结构，需要修改对应的工厂类，增加创建对应产品的方法，违背了开闭原则

使用场景

用户不需要关系对象的创建过程，将对象的创建和使用分开

属于一个产品族的产品将在一起使用，产品族中的产品可以没有任何的关联关系，但是它们都具有统一的约束，比如在同一操作系统下的按钮和文本框，它们之间没有关联，但是它们属于同一个操作系统，这就是它们共同的约束条件

系统中有多个产品族，每次只使用其中的一个产品族，新增产品族很方便

### 单例模式

确保一个类只有一个实例，而且自行实例化然后向整个系统提供这个实例，这个类必须是单例的，它提供全局访问的方法；

要点某个类只能有一个实例，自行创建这个实例，提供全局访问的方法

优点

提供了对唯一实例的访问控制，单例模式封装了它的唯一实例，可以控制使用者怎样以及何时对其进行访问

在系统内存中只有一个实例对象，所有节约了系统内存资源，

缺点

单例类没有抽象层，因此单例类的扩展有很大的难度

单例类职责过重，在一定程度上违背了单一职责原则，因为单例类是工厂角色，创建产品对象，有充当了产品对象，包含了一些业务，将工厂和产品融合在一起结构有点复杂

使用场景

系统只需要一个实例，比如系统要求提供一个唯一的序列号或者同一的资源管理器；或者考虑到创建对象会占用大量的内存资源

访问单例实例只能有一个公共访问点，不能通过其他路径访问

饿汉模式：在类加载的时候就创建，急切初始化

/\*\*

\* 饿汉模式

\*/

**public** **class** Singleton1

{

**private** Singleton1()

{

}

**public** **static** Singleton1 *singleton* = **new** Singleton1();

**public** **static** Singleton1 getInstance()

{

**return** *singleton*;

}

}

优点：线程安全，在类加载的时候就创建对象，调用速度快

缺点：资源利用率不高，这个对象如果一直不用，那么就浪费了内存资源

懒汉模式：单例实例在第一次使用的时候创建，延迟初始化

/\*\*

\* 懒汉模式

\*/

**public** **class** Singleton1

{

**private** Singleton1()

{

}

**public** **static** Singleton1 *singleton*;

**public** **static** Singleton1 getInstance()

{

**if** (*singleton* == **null**)

{

*singleton* = **new** Singleton1();

}

**return** *singleton*;

}

}

优点：只有使用的时候才会去创建单例对象，提高了内存资源的利用率，如果一直不去调用创建单例的方法，对象就不会创建

缺点：懒汉模式在单线程环境下是线程安全的，但是在对象环境下，还是可能创建多个实例的；解决办法就是加synchronized锁保证线程同步，

这样在第一次访问的时候不够快，同步也会带来额外的开销

双重校验模式

/\*\*

\* 双重校验模式

\*/

**public** **class** Singleton1

{

**private** Singleton1()

{

}

**public** **static** Singleton1 *singleton*;

**public** **static** Singleton1 getInstance()

{

**if** (*singleton* == **null**)

{

**synchronized** (Singleton1.**class**)

{

**if** (*singleton* == **null**)

{

*singleton* = **new** Singleton1();

}

}

}

**return** *singleton*;

}

}

优点：资源利用率高，在使用前不会创建对象，

缺点：第一次执行的加载不快；可能由于java内存模型的指令重排序导致创建多个对象

静态内部类

/\*\*

\* 静态内部类模式

\*/

**public** **class** Singleton1

{

**private** Singleton1()

{

}

**public** **static** Singleton1 getInstance()

{

**return** InnerInit.*singleton*;

}

**private** **static** **class** InnerInit

{

**public** **static** Singleton1 *singleton* = **new** Singleton1();

}

}

优点：能够接收内存资源，不获取单例对象是不会创建改对象的，

枚举单例

/\*\*

\* 枚举单例

\*/

**public** **enum** Singleton

{

*INSTANCE*;

**public** Singleton getInstance()

{

**return** *INSTANCE*;

}

}

优点：线程安全，效率高

私有化构造器可以通过反射初始化，枚举可以防止反射创建新实例，

序列化和反序列化后会返回一个新的实例，使用枚举就不会出现这个情况

在不使用synchronized和lock的前提下，如何实现一个线程安全的单例模式？

1.饿汉模式，在定义static变量是直接初始化（存在问题，不使用就浪费类存）；

2.使用静态内部类来实现单例，在使用单例对象时是不会创建单例对象的，使用了延迟加载，单例类被加载，但是内有没有使用静态内部类，所以就还没有创建对象，只有显示的获取内部类的定义的创建对象方式时才会实例化对象

3.使用枚举方式，不仅能够避免多线程同步问题，还能防止反序列化重新创建新的对象

以上几种方式说明：这几种方式都是利用了类加载器的时候初始化单例，即使用了classloader的线程安全机制

Classloader的线程安全机制就是classloader使用loadclass方法在类加载的时候使用了synchronized关键字，除非被重写，这个方法默认在整个装载过程中都是同步的，也就是保证了线程安全，这几种方法虽然没有显示的使用synchronized方法，但是底层实现原理还是用到了synchronized关键字

使用CAS来实现（:Compare and Swap,）

CAS（比较替换）：是一种无锁算法，CAS有3个操作数，内存值v,旧的预期值A，要修改的新值B,当且仅当预期值和内存中的值相等时，将内存中的值修改为新值，否则什么都不做

**public** **class** Singleton

{

// 创建原子引用类

**private** AtomicReference<Singleton> instance = **new** AtomicReference<>();

**public** Singleton getInstanc()

{

// 无限循环，直到获取值

**for** (;;)

{

// 获取当前的value

Singleton s = instance.get();

**if** (s != **null**)

{

// 不为空就直接返回

**return** s;

}

s = **new** Singleton();

// 创建对象s作为新增和期望值null进行比较替换，如果替换成功，那么当前创建的对象就是单例对象，不成功就进行下一次循环判断

**if** (instance.compareAndSet(**null**, s))

**return** s;

}

}

}

使用CAS的好处就是不使用锁机制就能保证线程安全，CAS是基于忙等算法，依赖底层硬件的实现，相对于锁，他没有线程切换和阻塞的额外消耗，，可以支持较大的并发

CAS有一个重要的缺点：如果忙等一直执行不成功（一直死循环），会对cup造成较大的执行开销

如果有非常多的线程同时执行到创建对象这一步，那么就是在内存中会出现大量对象，非消耗许多内存，可能会引起内存溢出

### 原型模式

通过一个原型对象克隆出一个一模一样的对象，克隆的对象是一个全新的对象，在内存中有新的地址，对克隆对象的修改不会影响到原对象

浅克隆：复制对象只会复制对象本身和它的值类型，不会复制引用类型；也就是实现cloneable接口的类对象调用clone方法时，生成的新对象中的引用类型还是和原对象一样指向同一对象

Class A implement Cloneable

A a=new A();

A a1=a.clone()

深克隆：无论原型对象的值类型还是引用类型，都会复制一份给克隆对象

深克隆使用序列化和反序列化来实现

在原对象中定义一个clone方法

Public Object clone(){

ByteArrayoutputstream bas=new ByteArrayOutPutStream();

ObjectOutPutStream oos = new ObjectOutPutStream(bas);

Oos.writeObject(this);

ByteArrayInputStream bai=new ByteArrayInputStream(oos);

ObjectInputStream ooi =new ObjectInputStream(bai);

Return Ooi.readObject();

}

优点

当创建新的对象实例比较复杂时，使用原型模式可以简化对象实例化的过程，通过复制对象可以提升新实例的创建效率

原型模式提供了简化的创建结构，工厂模式通常需要一个和产品类同等级结构的工厂类来实现对象的创建，原型模式不需要这样，原型模式的复制是通过封装在原型类的克隆方法来实现的

可以使用深克隆的方式来保存对象的状态，使用原型模式将原对象复制一份保存起来，以便在之后做类似回退操作时可以使用

缺点

克隆方式的实现是在类的内部，要修改克隆方法就要修改原来的代码，违背了开闭原则

实现深克隆时的代码比较复杂，需要克隆所有的引用类型，当对象之间存在多处引用嵌套时，为了实现深克隆，每一层对象都必须支持克隆，实现起来会比较麻烦

使用场景

创建新对象成本较大，使用原型模式来复制一个新对象，减少创建的开销；如果对象相似，那么就只需要稍微的修改成员变量即可

系统要保存对象状态，而对象的变化比较小，或者对象本身占据了少量的内存，可以使用原型模式配置备忘录模式来实现

使用原型模式复制对象比使用new创建一个新的对象方便；使用克隆方法，jvm会直接操作内存拷贝原始数据流，简单直接，速度比实例化快；

### 建造者模式

建造者模式返回的不仅是一个简单的产品，返回的是由一个或者多个部件组成的复杂产品

将复杂对象的创建过程和它的表示分离，使用同一的构件过程可以得到不同的表示

（创建产品，创建产品的部件，组装这些部件，也就是对对象内部值类型和引用类型的赋值）

角色

Builder（抽象建造者）：提供了创建产品的方法，可以是抽象类也可以是接口；定义的方法分为2中类型，一种是创建复制对象的方法，一种是返回复制对象的方法

concreteBuilder（具体建造者）：实现builder接口，实现各个部件的创建和装配，明确定义创建的对象；提供返回创建对象的方法

product（产品）：被构件的复杂对象，内部包含了多个部件构成

Director（指挥者）：复杂安排对象创建的次序，

Abstract class Builder{

Product p=new Product();//创建产品

Abstract void setPart1();组装部件1

Abstarct void setPart2();组装部件2

}

建造者模式和工厂模式的区别

建造者模式返回的是一个完整的复杂对象，而工厂方法模式返回的是一系列相关的产品

工厂模式只是将产品创建出来就可以了，不用关系产品类的组成部分；建造者模式不仅要创建产品，而且还要关系这个产品的组成细节，组成过程

优点

客户端不需要知道对象的内部组成细节，将产品本身和产品的创建过程解耦，使得相同的创建过程可以创建不同的产品

每个具体的建造者是独立的，与其他的建造者无关，那么就很方便的添加新的建造者，复合开闭原则

可以根据精准的控制对象的创建细节和过程

缺点

建造者创建的产品一般具有较多的而相同点，其组成部分相似，如果产品之间差异很大，或者说组成部分都不相同，那么就不适合用建造者模式，因此它的使用范围受到一定的限制；

（当产组成部分差异很大时不适合用建造者模式）

产品内部的变化复杂可能需要定义很多的建造者类来实现这种变化，导致系统很庞大

使用场景

需要生成的产品对象有复杂的内部结构，这些产品通常包含许多的成员属性

对象的创建过程独立于创建该对象的类，将创建过程封装在指挥者类中

隔离复杂对象的创建和使用

## 结构型模式

### 适配器模式

做兼容用的，将一个接口转换为客户希望的另一个接口，使接口不兼容的那些类能够一起工作，

新增一个适配器类来解决原来接口不兼容的问题，使得原来没有关系的类可以在一起工作，根据适配器类和适配者类的不同，可以

将适配器模式分为对象适配器和类适配器

对象适配器是适配器与适配者之间的关联关系，类适配器是适配器与适配者之间的继承关系；它们之间的区别就是适配者和适配器的关系不同

在java中大多使用对象适配器，因为java不支持多继承，对象适配器的使用会受到限制，如果适配器类是final，那么就不能被继承

适配器角色

Target（目标方法）：目标抽象定义客户需要的接口（方法），可以是一个抽象类或者接口，也可以是具体类

Adapter（适配器类）：调用另一个接口，做转换器

Adaptee(适配者类)：被适配的角色，已经存在一个接口，这个接口需要适配

缺省适配器

不需要实现一个接口提供的所有方法，可以先设计一个抽象类实现该接口，并为接口中每个方法提供一个默认的实现，那么子类实现这个抽象类后就可以选择性的修改某些方法来实现需求，适用于不想使用接口中的全部方法，在jdk的hashmap就是使用的这种方式(HashMap和父类abstractMap)

优点

将目标类和适配者类解耦，通过引入适配器类来重用现有的适配者类，无效修改原来的代码

增加了类的可用性和复用性，一个适配者类可以在不同的模块中复用

缺点

类适配器

就是一个适配器只能适配一个适配者类，不能同时适配多个适配者

适配者不能为final类，那么就不能使用类适配

Java不支持多继承，类适配器的使用会受到限制

对象适配器

与类适配器相比，要置换适配者类之间的某些方法比较麻烦

与类适配器模式相比，要在适配器中置换适配者类的某些方法比较麻烦。如果一定要置换掉适配者类的一个或多个方法，可以先做一个适配者类的子类，将适配者类的方法置换掉，然后再把适配者类的子类当做真正的适配者进行适配，实现过程较为复杂

使用场景

系统需要使用一些现有的类，而这些类的接口（如方法名）不符合系统的需要，甚至没有这些类的源代码。

想创建一个可以重复使用的类，用于与一些彼此之间没有太大关联的一些类，包括一些可能在将来引进的类一起工作。

代理模式和适配器模式的区别

适配器模式是因为新旧接口不一致导致出现客户端无法满足的需求，旧的接口是不能被重构的，也就是任然在使用的，可以在新接口和就接口之间实现转换；实现转换的这个类就是转换器

代理虽然也是同样增加了一层，代理提供的接口和原来的接口是一样的，代理模式的作用是不直接将实现暴露给客户端，而是通过代理层，代理做一些处理；

也即是说目标对象不会直接被调用，适配者对象可以直接使用

桥接模式

组合模式

装饰模式

外观模式

享元模式

代理模式

## 行为型模式