**目录**

[About面向对象：](#_Toc15197_WPSOffice_Level1) [2](#_Toc15197_WPSOffice_Level1)

[抽象类与接口的使用区别：](#_Toc48_WPSOffice_Level1) [3](#_Toc48_WPSOffice_Level1)

[一．简单工厂模式：](#_Toc23122_WPSOffice_Level1) [4](#_Toc23122_WPSOffice_Level1)

[二．策略模式：](#_Toc8518_WPSOffice_Level1) [5](#_Toc8518_WPSOffice_Level1)

[三．单一职责原则](#_Toc12048_WPSOffice_Level1) [7](#_Toc12048_WPSOffice_Level1)

[四．开放-封闭原则](#_Toc30129_WPSOffice_Level1) [8](#_Toc30129_WPSOffice_Level1)

[五．依赖倒转原则](#_Toc18942_WPSOffice_Level1) [9](#_Toc18942_WPSOffice_Level1)

[六．里氏代换原则](#_Toc11257_WPSOffice_Level1) [9](#_Toc11257_WPSOffice_Level1)

[七．装饰模式](#_Toc8976_WPSOffice_Level1) [11](#_Toc8976_WPSOffice_Level1)

[八．代理模式](#_Toc31141_WPSOffice_Level1) [13](#_Toc31141_WPSOffice_Level1)

[九．工厂方法模式](#_Toc18828_WPSOffice_Level1) [14](#_Toc18828_WPSOffice_Level1)

[十．原型模式](#_Toc2384_WPSOffice_Level1) [15](#_Toc2384_WPSOffice_Level1)

[十一．模板方法模式](#_Toc31819_WPSOffice_Level1) [16](#_Toc31819_WPSOffice_Level1)

[十二．迪米特法则](#_Toc2679_WPSOffice_Level1) [17](#_Toc2679_WPSOffice_Level1)

[十三．外观模式](#_Toc28879_WPSOffice_Level1) [18](#_Toc28879_WPSOffice_Level1)

[十四． 建造者模式](#_Toc30879_WPSOffice_Level1) [20](#_Toc30879_WPSOffice_Level1)

About面向对象：

面向对象的三大基本特征：

封装

继承

多态

面向对象的程序需要考虑的几方面：

维护性

复用性

扩展性

灵活性

面向对象的四大好处：

可维护

可扩展

可复用

灵活性好

## 抽象类与接口的使用区别：

抽象类是自底向上抽象出来的：

（通过重构改善既有代码的设计）

抽象类是从子类中发现了公共的东西，泛化出父类，然后子类继承父类。所以抽象类往往都是通过重构得来的。当然如果能够事先意识到多种分类的可能，预先设计出抽象类完全可以。

接口是自顶向下设计出来的：

（接口不知晓子类的存在，方法如何实现不确认，预先定义）

预先知晓将要实现的功能，即可设计构造相应接口，而不需要了解将要实现这种功能的实例化对象是什么类的，只要相应的类实现了该接口即可。到时候就能通过该接口对象就能使用相应类实现的该接口方法功能。

## 一．简单工厂模式：

思路简要：

封装一系列继承自同一父方法的子方法类（每个方法都不相同），在工厂类里根据传入的条件经过switch选择指定方法类，创建返回该方法类对象。然后通过多态，即创建父类方法对象，然后将具体子类方法赋值给父类对象来调用。

适用场景：

**同一地方可能需要使用不同的处理方式**（即创建不同的对象），容易变化的地方，考虑创建一个单独的类来实现这个创建实例的过程，这就是工厂。

优越性：

采用多态，避免了简单代码的重复，提高了代码的复用性。（多态性）

使用工厂，便于扩展，需要扩展处理方式（新对象），只需要继承方法类（虚父类），然后在工厂类中添加逻辑判断，创建返回相应新方法对象即可。（可扩展性）

封装了子方法类，修改某一子方法类，不影响其他字方法类。（封装性、可维护性）

## 二．策略模式：

思路简要：

封装一系列继承自同一父算法的子算法类（每个方法都相同（统一命名接口方法），只是实现方式不同），通过创建一个Context类，在其中创建父算法，然后初始化时传递进入具体的子算法对象。在Context类预留的接口中直接调用相应子算法对象的实际策略方法（统一的命名接口方法）。使用时，通过直接创建Context类对象，传入相应策略方法，然后使用Context类对象的接口方法使用相应策略方法。

适用场景:

同一地方需要完成一项指定的工作，但是需要使用不同的方法来完成这一项工作。这个时候就需要策略模式创建一系列不同实现思路的策略方法，然后使用同一种方式调用所有算法。

优越性：

策略模式以相同的方式调用所有的算法（一系列策略算法），减少了各种算法类与使用算法类之间的耦合度。（灵活性、封装性）

策略模式的Strategy类层次为Context定义了一系列的可供重用的算法或行为。继承有助于析取出这些算法中的公共功能（复用性）

策略模式简化了单元测试，因为每个算法都有自己的类，可以通过自己的接口单独测试（封装性、可维护性）

将一系列不同的行为和功能封装在一个独立的Strategy类中，可以在使用这些行为的类中消除掉条件语句。（封装性）

策略模式是用来封装算法的，但实际上是封装了变化（可以用策略算法来封装几乎任何类型的规则）

只要在分析过程中听到，**需要在不同时间应用不同的业务规则**，就可以考虑使用策略模式处理。

## 三．单一职责原则

原则简要：

就一个类而言，应该仅有一个引起它变化的原因。

原因：

如果一个类承担的职责过多，就等于把这些职责耦合在一起，一个职责的变化可能会削弱或者抑制这个类完成其他职责的能力。这种耦合会导致脆弱的设计，当变化发生时，设计会遭受到意想不到的破坏。

适用场景：

**如果能够想到多于一个的动机去改变一个类，那么这个类就具有多于一个的职责**，那么就需要做软件设计真正要做的许多内容（发现职责并把那些职责相互分离）。

优越性：

易维护、易扩展、易复用、灵活多样。

## 四．开放-封闭原则

原则简要：

软件实体（类、模块、函数等）应该可以扩展，但是不可修改。

**对于扩展是开放的，对于更改是封闭的。**

原因：

使用开放-封闭原则设计的系统，面对需求的改变可以保持相对稳定的状态，从而使得系统可以在第一个版本以后不断推出新的版本。

适用场景：

因为不可能预先知晓所有变化，

**所以最初编写代码时，假设变化不会发生。当变化发生时，就创建抽象来隔离以后发生的同类变化。**

**面对需求，对程序的改动是通过增加新代码进行的，而不是更改现有的代码。（精髓）**

**开发人员应该仅对程序中呈现出频繁变化的那些部分做出抽象。（不需要刻意对每一部分进行抽象，把握好度）**

优越性：

易维护、易扩展

## 五．依赖倒转原则

原则简要：

**高层模块不应该依赖低层模块。两个都应该依赖抽象。**

**抽象不应该依赖细节。细节应该依赖抽象。**

**实质：针对接口编程，不要对实现编程。**

**（谁也不要依靠谁，除了约定的接口，大家都可以灵活自如。）**

**就是编程过程中，使用其他类的对象时，创建使用它们的抽象父类或是接口实例化，编程时不要使用具体的细节子类。**

原因：

高层模块如果依赖于底层模块实现的话。那么如果需要修改底层模块的属性功能等细节部分，就会导致依赖于它的高层模块异常或是不可用。导致了高层模块无法复用。

## 六．里氏代换原则

原则简要：

**一个软件实体如果使用的是一个父类的话，那么一定适用于其子类，而且它察觉不出父类对象和子类对象的区别。也就是说，在软件里面，把父类都替换成它的子类，程序的行为没有变化。（子类型必须能够替换掉它们的父类型）**

子类拥有父类所有非private的行为和属性。

原因：

只有当子类可以替换掉父类，软件单位的功能不受到影响时，父类才能真正被复用，而子类也能够在父类的基础上增加新的行为。

正是由于子类型的可替换性才使得使用父类类型的模块在无需修改的情况下就可以扩展。

依赖倒转原则与里氏代换原则：

依赖倒转其实可以说是面向对象的标志，用哪种语言来编写程序不重要，如果编写时考虑的都是如何针对抽象编程而不是针对细节编程，即**程序中所有的依赖关系都是终止于抽象类或者接口，那就是面向对象的设计，反之那就是过程化的设计。**

## 七．装饰模式

思路简要：

**动态地给一个对象添加一些额外的职责，就增加功能来说，装饰模式比生成子类更为灵活。**

**（装饰模式是为已有功能动态地添加更多功能的一种方式）**

1.Component是定义一个对象接口，可以给这些对象动态地添加职责。

2.ConcreteComponent是定义了一个具体的对象，也可以给这个对象添加一些职责。

3.Decorator，装饰抽象类，继承了Component，从外类来扩展Component类的功能，但对于Component来说，是无需知道Decorator的存在的。

4.ConcreteDecorator是具体的装饰对象，起到给Component添加职责的功能。（也可以灵活的将ConcreteDecorator类和Decorator类杂糅在一起。）

在装饰类中不直接调用装饰类的功能函数，而是通过调用Decorator类对象中的SetComponent函数，将传入的装饰类对象赋值给其父类的component对象，然后一直迭代，最后调用装饰类的Operation函数，在Operation函数中调用其父类的Operation函数，实质上此时开始迭代调用不同的装饰类对象的Operation函数。

适用场景：

普通的设计思路：

当系统需要新功能的时候，向旧的类中添加新的代码。这些新加的代码通常装饰了原有类的核心职责或主要行为。（问题在于：这种做法**在主类中加入了新的字段，新的方法和新的逻辑，从而增加了主类的复杂度。而这些新加入的东西却仅仅只是为了满足一些特殊情况下特殊行为的需要。**）

装饰模式的设计思路：

**把每个需要装饰的功能放在单独的类中，并让这个类包装它所要装饰的对象（继承自同一抽象类，就可利用抽象父类的实例化对象装饰它）。**当需要执行特殊行为时，客户代码就可以在运行时根据需要有选择地、按顺序地使用装饰功能包装对象了。

优越性：

把类中的装饰功能从类中搬移去除，这样可以简化原有的类。

有效地把类的核心职责和装饰功能区分开来。而且可以去除相关类中的重复装饰逻辑。

## 八．代理模式

思路简要：

（**为其他对象提供一种代理以控制对这个对象的访问**）

定义一个抽象的类，以供代理人（类）与被代理人（类）共同继承，这样在任何可以使用被代理人（类）的地方都能用代理人做替代。在代理人（类）中定义一个被代理人（类）的引用，并且提供一个与被代理人（类）相同的接口，以便代理替代。

代理模式其实就是在访问对象时引入一定程度的间接性，因为这种间接性，可以附加多种用途。

适用场景：

①远程代理：为一个对象在不同的地址空间提供局部代表。这样就可以隐藏一个对象存在于不同地址空间的事实。

②虚拟代理：根据需要创建开销很大的对象。通过它来存放实例化需要很长时间的真实对象。

③安全代理：用来控制真实对象访问时的权限。

④智能指引：当调用真实的对象时，代理处理另外一些事。

......

优越性：

见适用场景。

## 九．工厂方法模式

思路简要：

（定义一个用于创建对象的接口，让子类决定实例化哪一个类。工厂方法使一个类的实例化延迟到其子类。）

相对于简单工厂模式，如果需要新增功能，就不需要更改原有的工厂类，而只需要增加此功能的运算类和相应的工厂类就可以了。

适用场景：

与简单工厂模式一样（**同一地方可能需要使用不同的处理方式**（即创建不同的对象），容易变化的地方，考虑创建一个单独的类来实现这个创建实例的过程，这就是工厂。）

优越性：

工厂方法克服了简单工厂违背开放-封闭原则的缺点，又保持了封装对象创建过程的优点。

## 十．原型模式

思路简要：

从一个对象再创建另外一个可定制的对象，而且不需要知道任何创建的细节。只要让想要实现原型模式的类继承ICloneable接口，实现其中唯一的方法：Clone（）就能完成原型模式了。使用MemberwiseClone（）方法复制。

适用场景：

初始化操作时很低效的，一般在初始化信息不发生变化的情况下，克隆是最好的方法。复制一份，修改不同的部分。不用重新初始化对象，而是动态地获得对象运行时的状态。

优越性：

大量执行初始化操作会导致性能低下，使用原型模式，既隐藏了对象创建的细节，又对性能是大大提高。

坑：

需要注意**浅复制**与**深复制**的区别：

MemberwiseClone（）方法：

字段是值类型的，则对该字段执行逐位赋值。

字段是引用类型，则复制引用但不复制引用的对象。（由此导致的结果就是原始对象和其副本引用指向同一对象！）

## 十一．模板方法模式

思路简要：

提取子类中所有的公共代码。

适用场景：

（

定义一个操作中的算法的骨架，而将一些步骤延迟到子类中。模板方法使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤。

）

当我们要完成在某一细节层次一致的一个过程或一系列步骤，但其个别步骤在更详细的层次上的实现可能不同时，通常考虑用模板方法模式来处理。

当不变的和可变的行为在方法的子类实现中混合在一起的时候，不变的行为就会在子类中重复出现。这样我们通过模板方法模式把这些行为搬移到单一的地方，这样就能帮助子类摆脱重复的不变行为的纠缠。

优越性：

模板方法模式通过把不变行为搬移到超类，去除子类中的重复代码来提现它的优势，提供了一个很好的代码复用平台。

## 十二．迪米特法则

（最少知识原则）

原则简要：

在类的结构设计上，每一个类都应当尽量降低成员的访问权限。

迪米特法则，强调了类之间的松耦合。

如果两个类不必彼此直接通信，那么这两个类就不应当发生直接的相互作用。如果其中一个类需要调用另一个类的某一个方法话，可以通过第三者转发这种调用。

原因：

类之间的耦合越弱，越有利于复用，一个处在弱耦合的类被修改，不会对有关系的类造成波及。（信息的隐藏促进了软件的复用）

优越性：

可维护性、可复用性。

## 十三．外观模式

思路简要：

为子系统中的一组接口提供一个一致的界面，此模式定义了一个高层接口，这个接口使得这一子系统更加容易使用。

在一个主类的构造函数中，将要使用的实用类全部实例化，然后通过该主类的函数使用这些实用类。（相当于将实用类群的使用过程组合封装隐藏，使用者无需了解细节）

（Facade类中实现方法组）

适用场景与优越性：

①设计初期阶段：

应该要有意识的将不同的几个层（例如MVC架构：数据访问层、业务逻辑层、表示层）分离。层与层之间建立外观Facade，可以为复杂的子系统提供一个简单的接口，使得耦合性大大降低。

②开发阶段：

子系统往往因为不断的重构演化而变得越来越复杂，大多数的模式使用时都会产生很多很小的类，这本是好事，但也给外部调用它们的用户程序带来了使用上的困难，增加外观Facade可以提供一个简单的接口，减少它们之间的依赖。

③维护阶段：

在维护一个遗留的大型系统时，可能这个系统已经非常难以维护和扩展了，但因为它包含非常重要的功能，新的需求开发必须要依赖于它。此时用外观模式Facade是非常合适的。可以为新系统开发一个外观Facade类，来提供设计粗糙或高度复杂的遗留代码的比较清晰简单的接口，让新系统与Facade对象交互，Facade与遗留代码交互所有复杂的工作。

（开发Facade与老系统的交互，新系统开发人员只需要了解Facade的接口，直接调用这些接口即可。）

## 建造者模式

（生成器模式）

思路简要：

先定义一个抽象的建造类，把建造过程给稳住。

然后创建一个很重要的类（指挥者类Director），用它来控制建造过程，也用它来隔离用户与建造过程的关联。

适用场景：

将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示。

将一个产品的内部表象与产品的生成过程分割开来，从而可以使一个建造过程生成具有不同的内部表象的产品对象。

如果我们使用了建造者模式，那么用户就只需指定需要建造的类型就可以得到它们，而具体建造的过程和细节就不需知道了。

主要是用于创建一些复杂的对象，这些对象内部构建间的建造顺序通常是稳定的，但对象内部的构建通常面临着复杂的变化。

建造者模式是在当创建复杂对象的算法应该独立于该对象的组成部分以及它们的装配方式时适用的模式。

优越性：

使得建造代码与表示代码分离，由于建造者隐藏了该产品的组装过程，所以若是需要改变一个产品的内部表示，只需要再定义一个具体的建造者就可以了。

## 观察者模式

思路简要：

观察者模式定义了一种一对多的依赖关系，让多个观察者对象同时监听某一个主题对象。这个主题对象在状态发生时，会通知所有观察者对象，使它们能够自动更新自己。

适用场景：

当一个对象的改变需要同时改变其他对象的时候。而且它不知道具体有多少对象有待改变时，应该考虑适用观察者模式。

优越性：

观察者模式所做的工作实质上是解除耦合，将一个抽象模型的两个方面（其中一方面依赖于另一方面）封装在独立的对象中使它们各自独立地改变和复用。让耦合的双方都依赖于抽象，而不是依赖于具体。从而使得各自的变化都不会影响另一边的变化。（依赖倒转原则）

.Net中通过事件委托实现了观察者模式。

## 抽象工厂模式

思路简要：

抽象工厂模式，提供一个创建一系列相关或相互依赖对象的接口，而无需指定它们具体的类。

通过反射+配置文件能够克服抽象工厂模式需要大量改动代码的缺陷。

适用场景：

在同一地方有大量变化，通过工厂类来封装变化。

优越性：

易于交换产品系列，由于具体工厂类在一个应用中只需要在初始化的时候出现一次，这就使得改变一个应用的具体工厂变得非常容易，它只需要改变具体工厂即可使用不同的产品配置。

它让具体的创建实例过程与客户端分离，客户端是通过它们的抽象接口操纵实例，产品的具体类名也被具体工厂的实现分离，不会出现在客户代码中。

## 状态模式

思路简要：

当一个对象的内在状态改变时允许改变其行为，这个对象看起来像是改变了其类。

适用场景：

主要解决的是当控制一个对象状态转换的条件表达式过于复杂时的情况。把状态的判断逻辑转移到表示不同状态的一系列类当中，可以把复杂的判断逻辑简化。

（当一个对象的行为取决于它的状态，并且它必须在运行时刻根据状态改变它的行为时，就可以考虑用状态模式）

优越性：

将与特定状态相关的行为局部化，并且将不同状态的行为分割开来。

（将特定的状态相关行为都放入一个对象中，由于所有与状态相关的代码都存在于某个ConcreteState中，所以通过定义新的子类可以很容易地增加新的状态和转换）

状态模式通过把各种状态转移逻辑分布到State的子类之间，来减少相互间的依赖。

## 适配器模式（Adapter）

思路简要：

将一个类的接口转换成客户希望的另外一个接口。Adapter模式使得原本由于接口不兼容而不能一起工作的那些类可以一起工作。

（分为类适配器模式和对象适配器模式两类）

适用场景：

系统的数据和行为都正确，但接口不符时，考虑适用，目的是使控制范围之外的一个原有对象与某个接口匹配。（主要应用于希望复用一些现存的类，但是接口又与复用环境要求不一致的情况。）

只有当使用双方都不太容易修改的时候再考虑使用适配器模式适配。（最好是开始就重构统一接口。）

优越性：

能够复用一些现存的类。是的原本由于接口不兼容而不能一起工作的那些类可以一起工作。

## 备忘录模式

思路简要：

在不破坏封装性的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态。这样以后就可将该对象恢复到原先保存的状态。

适用场景：

当对象角色的状态改变时候，有可能这个状态无效，这时候就可以使用暂时存储起来的备忘录将状态复原。

局限性：

角色状态需要完整的存储到备忘录对象中时，如果状态数据很大很多，那么在资源消耗上，备忘录对象会非常耗内存。