一、面向对象设计原则

**（一）单一职责原则**

定义：一个对象应该只包含单一的职责，并且该职责被完整的封装在一个类中。（就一个类而言，应该仅有一个引起它变化的原因）

解释：在软件系统中，一个类（大到模块，小到方法）承担的职责越多，它被复用的可能性就越小，而且一个类承担的职责过多，相当于将这些职责耦合在一起，当其中一个职责变化时可能会影响其它职责的运作，因此要将这些职责进行分离，将不同的职责封装在不同的类中，即不同的变化原因封装在不同的类中，如果多个职责总是同时发生改变则可以将它们封装在同一个类中。单一职责原则是实现高内聚，低耦合的指导方针。

**（二）开闭原则**

定义：软件实体应当对外扩展开放，对修改关闭

解释：在开闭原则的定义中，软件实体可以指一个软件模块，一个由多个类组成的局部结构或一个独立的类。开闭原则就是指软件实体应尽量在不修改原有代码的情况下进行扩展。为了满足开闭原则，需要对系统进行抽象化设计，抽象化是开闭原则的关键。

**（三）里氏代换原则**

定义：所有引用基类的地方必须能透明地使用其子类的对象

解释：里氏替换原则表明，在软件中将一个基类对象替换成它的子类对象，程序将不会产生任何错误和异常，反过来则不成立，如果一个软件实体使用的是一个子类对象，那么它不一定能使用基类对象。

举例：我喜欢动物，那我一定喜欢狗，因为狗是动物的子类；但是我喜欢狗，不能据此断定我喜欢所有动物（基类推子类成立，子类推基类不一定成立）

里氏替换原则是实现开闭原则的重要方式之一，由于在使用基类对象的地方都可以使用子类对象，因此在程序中尽量使用基类类型来对对象进行定义，而在运行时在确定其子类类型，用子类对象来替换父类对象。在运用里氏代换原则时应该将父类设计为抽象类或接口，让子类继承父类或实现父接口，并实现在父类中声明的方法，在运行时子类实例替换父类实例，可以很方便地扩展系统的功能，无需修改原有子类的代码，增加新的功能可以增加一个新的子类来实现

**（四）依赖倒转原则**

定义：高层模块不应该依赖低层模块，它们都应该依赖抽象。抽象不应该依赖细节，细节应该依赖于抽象

解释：简单来说，依赖倒转原则要求针对接口编程，不要针对实现编程。依赖倒转原则要求在程序代码中传递参数时或在关联关系中尽量引用层次高的抽象层类，即使用接口和抽象类进行变量类型声明、参数类型声明、方法返回类型声明，以及数据类型的转换等，而不要用具体类来做这些事情。为了确保该原则的应用，一个具体类应该只实现接口或抽象类中声明过的方法，而不要给出多余的方法，否则将无法调用到在子类中增加新的方法。

在实现依赖倒转原则时需要针对抽象层编程，而将具体类的对象通过依赖注入的方式注入到其他对象中，依赖注入是指当一个对象要与其他对象发生依赖关系时采用抽象的形式来注入所依赖的对象。通常的注入方式有3中：

1. 构造注入：通过构造函数来传入具体类的对象

2. 设值注入（setter注入）：通过setter方法来传入具体类的对象

3.接口注入：通过在接口中声明业务方法来传入具体类的对象

这些方法在定义时使用的是抽象类型，在运行时在传入具体类型的对象，由子类对象来覆盖父类对象

**（五）接口隔离原则**

定义：客户端不应该依赖那些它不需要的接口

解释：根据接口隔离原则，当一个接口太大时需要将它分割成一些更细小的接口，使用该接口的客户端仅需知道与之相关的方法即可。这里的“接口”往往有两种不同的含义：

1. 一个类型所具有的方法特征的集合，仅仅是一种逻辑上的抽象。这时候可以把接口理解成角色，一个接口只能代表一个角色，每个角色都有它特定的一个接口，此时这个原则可以叫做“角色隔离原则”

2. 某种语言具体的“接口”，有严格的定义和结构，比如Java和C#中的interface。接口仅仅提供客户端需要的行为，客户端不需要的行为则隐藏起来，应该为客户端提供尽可能小的单独的接口，而不要提供大的总接口。

**（六）合成复用原则**

定义：优先使用对象组合，而不是通过继承来达到复用的目的

解释：合成复用原则就是在一个新的对象里通过关联关系（包括组合关系和聚合关系）来使用一些已有的对象，使之成为新对象的一部分，新对象通过委派调用已有对象的方法达到复用功能的目的。简而言之，在复用时要尽量使用组合/聚合关系（关联关系），少用继承

**（七）迪米特法则**

定义：每一个软件单位对其他单位只有最少的知识，而且局限于那些与本单位密切相关的软件单位

解释：迪米特法则要求一个软件实体应当尽可能少地与其他实体发生相互作用。如果一个系统符合迪米特法则，那么当其中一个模块发生修改时就会尽量少地影响其他模块，扩展会相对容易，这是对软件实体之间通信的现在，迪米特法则要求限制软件实体之间通信的宽度和深度。应用迪米特法则可以降低系统的耦合度，使类与类之间保持松散的耦合关系

迪米特法则还有几种定义形式，包括不要和“陌生人”说话、只与你的直接朋友通信等。在迪米特法则中，对于一个对象，其朋友包括以下几类：

1. 当前对象本身（this）

2. 以参数形式传入到当前对象方法中的对象

3. 当前对象的成员对象

4．如果当前对象的成员对象是一个集合，那么集合中的元素也都是朋友

5. 当前对象所创建的对象

简而言之，就是通过引入一个合理的第三者来降低现有对象之间的耦合度

**创建型模式、结构型模式和行为型模式 3 种。**

**创建型模式：**

用于描述“怎样创建对象”，它的主要特点是“将对象的创建与使用分离”。

单例、原型、工厂方法、抽象工厂、建造者等 5 种创建型模式。

**结构型模式**：

用于描述如何将类或对象按某种布局组成更大的结构。

代理、适配器、桥接、装饰、外观、享元、组合等 7 种结构型模式。

行为型模式：用于描述类或对象之间怎样相互协作共同完成单个对象都无法单独完成的任务，以及怎样分配职责。

模板方法、策略、命令、职责链、状态、观察者、中介者、迭代器、访问者、备忘录、解释器等 11 种行为型模式。

**二、设计模式**

**（一）单例模式（Singleton）**

应用场景：

优点：

1.在单例模式中，活动的单例只有一个实例，对单例类的所有实例化得到的都是相同的一个实例。这样就 防止其它对象对自己的实例化，确保所有的对象都访问一个实例

2.单例模式具有一定的伸缩性，类自己来控制实例化进程，类就在改变实例化进程上有相应的伸缩性。

3.提供了对唯一实例的受控访问。

4.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，当 需要频繁创建和销毁的对象时单例模式无疑可以提高系统的性能。

5.允许可变数目的实例。

6.避免对共享资源的多重占用。

缺点：

1.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。

2.由于单利模式中没有抽象层，因此单例类的扩展有很大的困难。

3.单例类的职责过重，在一定程度上违背了“单一职责原则”。

4.滥用单例将带来一些负面问题，如为了节省资源将数据库连接池对象设计为的单例类，可能会导致共享连接池对象的程序过多而出现连接池溢出；如果实例化的对象长时间不被利用，系统会认为是垃圾而被回收，这将导致对象状态的丢失。

使用注意事项：

1.使用时不能用反射模式创建单例，否则会实例化一个新的对象

2.使用懒单例模式时注意线程安全问题

3.单例模式和懒单例模式构造方法都是私有的，因而是不能被继承的，有些单例模式可以被继承（如登记式模式）

适用场景：

单例模式只允许创建一个对象，因此节省内存，加快对象访问速度，因此对象需要被公用的场合适合使用，如多个模块使用同一个数据源连接对象等等。如：

1.需要频繁实例化然后销毁的对象。

2.创建对象时耗时过多或者耗资源过多，但又经常用到的对象。

3.有状态的工具类对象。

4.频繁访问数据库或文件的对象。

以下都是单例模式的经典使用场景：

1.资源共享的情况下，避免由于资源操作时导致的性能或损耗等。如上述中的日志文件，应用配置。

2.控制资源的情况下，方便资源之间的互相通信。如线程池等。

应用场景举例：

1.外部资源：每台计算机有若干个打印机，但只能有一个PrinterSpooler，以避免两个打印作业同时输出到打印机。内部资源：大多数软件都有一个（或多个）属性文件存放系统配置，这样的系统应该有一个对象管理这些属性文件

2. Windows的TaskManager（任务管理器）就是很典型的单例模式（这个很熟悉吧），想想看，是不是呢，你能打开两个windows task manager吗？ 不信你自己试试看哦~

3. windows的Recycle Bin（回收站）也是典型的单例应用。在整个系统运行过程中，回收站一直维护着仅有的一个实例。

4. 网站的计数器，一般也是采用单例模式实现，否则难以同步。

5. 应用程序的日志应用，一般都何用单例模式实现，这一般是由于共享的日志文件一直处于打开状态，因为只能有一个实例去操作，否则内容不好追加。

6. Web应用的配置对象的读取，一般也应用单例模式，这个是由于配置文件是共享的资源。

7. 数据库连接池的设计一般也是采用单例模式，因为数据库连接是一种数据库资源。数据库软件系统中使用数据库连接池，主要是节省打开或者关闭数据库连接所引起的效率损耗，这种效率上的损耗还是非常昂贵的，因为何用单例模式来维护，就可以大大降低这种损耗。

8. 多线程的线程池的设计一般也是采用单例模式，这是由于线程池要方便对池中的线程进行控制。

9. 操作系统的文件系统，也是大的单例模式实现的具体例子，一个操作系统只能有一个文件系统。

10. HttpApplication 也是单位例的典型应用。熟悉ASP.Net(IIS)的整个请求生命周期的人应该知道HttpApplication也是单例模式，所有的HttpModule都共享一个HttpApplication实例.

**饿汉模式（安全）**

优点：这种写法比较简单，就是在类装载的时候就完成实例化。避免了线程同步问题。

缺点：在类装载的时候就完成实例化，没有达到Lazy Loading的效果。如果从始至终从未使用过这个实例，则会造成内存的浪费。

public class SingleInstance1 {

private static SingleInstance1 single=new SingleInstance1();

private SingleInstance1(){

}

public static SingleInstance1 getInstance(){

return single;

}

}

**静态代码块** **（安全）**

类加载时就创建实例对象 和模式一的优缺点差不多

public class SingleInstance1 {

private static SingleInstance1 single=null;

static{

single=new SingleInstance1();

}

private SingleInstance1(){

}

public static SingleInstance1 getInstance(){

return single;

}

}

**懒汉模式**

多线程不安全 会创建多个对象

这种写法起到了Lazy Loading的效果，但是只能在单线程下使用。

如果在多线程下，一个线程进入了if (singleton == null)判断语句块，还未来得及往下执行，另一个线程也通过了这个判断语句，这时便会产生多个实例。所以在多线程环境下不可使用这种方式。

public class SingleInstance {

private static SingleInstance2 single =null;

private SingleInstance2(){

}

public static SingleInstance2 getInstance(){

if(single==null){

single=new SingleInstance2();

}

return single;

}

}

**双重检查[推荐用]**

Double-Check概念对于多线程开发者来说不会陌生，如代码中所示，我们进行了两次if (singleton == null)检查，这样就可以保证线程安全了。

这样，实例化代码只用执行一次，后面再次访问时，判断if (singleton == null)，直接return实例化对象。

优点：线程安全；延迟加载；效率较高。

public class SingleInstance5 {

private static SingleInstance5 single =null;

private SingleInstance5(){

}

public static SingleInstance5 getInstance(){

if(single==null){

synchronized(SingleInstance5.class){

if(single==null){

single=new SingleInstance5();

}

}

}

return single;

}

}

**枚举**

借助JDK1.5中添加的枚举来实现单例模式。

不仅能避免多线程同步问题，而且还能防止反序列化重新创建新的对象。可能是因为枚举在JDK1.5中才添加，所以在实际项目开发中，很少见人这么写过。

优点

系统内存中该类只存在一个对象，节省了系统资源，对于一些需要频繁创建销毁的对象，使用单例模式可以提高系统性能。

缺点

当想实例化一个单例类的时候，必须要记住使用相应的获取对象的方法，而不是使用new，可能会给其他开发人员造成困扰，特别是看不到源码的时候。

public enum SingleInstance {

//创建对象时 直接SingleInstance7.INSTANCE; 对象在枚举中是一个常量,直接引用即可

INSTANCE;

public void whateverMethod() {

System.out.println("Something");

}

//如果不写 会自动提供一个默认的私有无参构造器

private SingleInstance7(){

}

}

**原型模式（Prototype）**

原型（Prototype）模式的定义如下：用一个已经创建的实例作为原型，通过复制该原型对象来创建一个和原型相同或相似的新对象。在这里，原型实例指定了要创建的对象的种类。用这种方式创建对象非常高效，根本无须知道对象创建的细节。例如，Windows 操作系统的安装通常较耗时，如果复制就快了很多。在生活中复制的例子非常多，这里不一一列举了。

原型模式的结构与实现

由于 Java 提供了对象的 clone() 方法，所以用 Java 实现原型模式很简单。

1. 模式的结构

原型模式包含以下主要角色。

抽象原型类：规定了具体原型对象必须实现的接口。

具体原型类：实现抽象原型类的 clone() 方法，它是可被复制的对象。

访问类：使用具体原型类中的 clone() 方法来复制新的对象。

**工厂模式**

工厂方法（FactoryMethod）模式的定义：定义一个创建产品对象的工厂接口，将产品对象的实际创建工作推迟到具体子工厂类当中。这满足创建型模式中所要求的“创建与使用相分离”的特点。

模式的结构与实现

工厂方法模式由抽象工厂、具体工厂、抽象产品和具体产品等4个要素构成。本节来分析其基本结构和实现方法。

工厂方法模式的主要优点有：

用户只需要知道具体工厂的名称就可得到所要的产品，无须知道产品的具体创建过程；

在系统增加新的产品时只需要添加具体产品类和对应的具体工厂类，无须对原工厂进行任何修改，满足开闭原则；

其缺点是：每增加一个产品就要增加一个具体产品类和一个对应的具体工厂类，这增加了系统的复杂度。

1. 模式的结构

工厂方法模式的主要角色如下。

抽象工厂（Abstract Factory）：提供了创建产品的接口，调用者通过它访问具体工厂的工厂方法 newProduct() 来创建产品。

具体工厂（ConcreteFactory）：主要是实现抽象工厂中的抽象方法，完成具体产品的创建。

抽象产品（Product）：定义了产品的规范，描述了产品的主要特性和功能。

具体产品（ConcreteProduct）：实现了抽象产品角色所定义的接口，由具体工厂来创建，它同具体工厂之间一一对应。

**代理模式**

**代理模式的定义与特点**

代理模式的定义：由于某些原因需要给某对象提供一个代理以控制对该对象的访问。这时，访问对象不适合或者不能直接引用目标对象，代理对象作为访问对象和目标对象之间的中介。

代理模式的主要优点有：

代理模式在客户端与目标对象之间起到一个中介作用和保护目标对象的作用；

代理对象可以扩展目标对象的功能；

代理模式能将客户端与目标对象分离，在一定程度上降低了系统的耦合度；

其主要缺点是：

在客户端和目标对象之间增加一个代理对象，会造成请求处理速度变慢；

增加了系统的复杂度；

代理模式的结构与实现

代理模式的结构比较简单，主要是通过定义一个继承抽象主题的代理来包含真实主题，从而实现对真实主题的访问，下面来分析其基本结构和实现方法。

**模式的结构**

代理模式的主要角色如下。

抽象主题（Subject）类：通过接口或抽象类声明真实主题和代理对象实现的业务方法。

真实主题（Real Subject）类：实现了抽象主题中的具体业务，是代理对象所代表的真实对象，是最终要引用的对象。

代理（Proxy）类：提供了与真实主题相同的接口，其内部含有对真实主题的引用，它可以访问、控制或扩展真实主题的功能。

**建造者模式（Bulider）**

模式的定义与特点

建造者（Builder）模式的定义：指将一个复杂对象的构造与它的表示分离，使同样的构建过程可以创建不同的表示，这样的设计模式被称为建造者模式。它是将一个复杂的对象分解为多个简单的对象，然后一步一步构建而成。它将变与不变相分离，即产品的组成部分是不变的，但每一部分是可以灵活选择的。

该模式的主要优点如下：

各个具体的建造者相互独立，有利于系统的扩展。

客户端不必知道产品内部组成的细节，便于控制细节风险。

其缺点如下：

产品的组成部分必须相同，这限制了其使用范围。

如果产品的内部变化复杂，该模式会增加很多的建造者类。

建造者（Builder）模式和工厂模式的关注点不同：建造者模式注重零部件的组装过程，而工厂方法模式更注重零部件的创建过程，但两者可以结合使用。

模式的结构与实现

建造者（Builder）模式由产品、抽象建造者、具体建造者、指挥者等 4 个要素构成，现在我们来分析其基本结构和实现方法。

模式的结构

建造者（Builder）模式的主要角色如下。

产品角色（Product）：它是包含多个组成部件的复杂对象，由具体建造者来创建其各个滅部件。

抽象建造者（Builder）：它是一个包含创建产品各个子部件的抽象方法的接口，通常还包含一个返回复杂产品的方法 getResult()。

具体建造者(Concrete Builder）：实现 Builder 接口，完成复杂产品的各个部件的具体创建方法。

指挥者（Director）：它调用建造者对象中的部件构造与装配方法完成复杂对象的创建，在指挥者中不涉及具体产品的信息。