## JAVA设计模式

1. 设计模式常用七大原则
   1. 单一职责原则
      1. 基本介绍：对类来说，即一个类应该只负责一项职责，如类A负责两个不同职责：职责1，职责2。当职责1需求变更而改变A时，可能造成职责2执行错误，所以需要将类A的粒度分解为A1,A2
      2. 注意事项：
         1. 降低类的复杂度，一个类只负责一项职责
         2. 提高类的可读性，可维护性
         3. 降低变更引起的风险
         4. 通常情况下，我们应当支持单一职责原则，只有逻辑足够简单，才可以在代码级违反单一职责原则；只有类中方法数量足够少，可以在方法级别保持单一职责原则
   2. 接口隔离原则(Interface Segregation Principle)
      1. 基本介绍：客户端不应该依赖它不需要的接口，即一个类对另一个类的依赖应该建立在最小的接口上
      2. 接口隔离原则处理：将接口拆分为独立的几个接口，类A和类C分别与他们需要的接口建立依赖关系，也就是采用接口隔离原则
      3. 改进接口
         1. 类A通过接口interface1依赖B，类C通过接口interface1依赖D，如果接口interface1对于类A和类C来说不是最小接口，那么类B和类D必须去实现他们不需要的方法
         2. 将接口interface1拆分为独立的几个接口，类A和类C分别与他们需要的接口建立依赖关系，也就是采用接口隔离原则
         3. 接口interface1中出现的方法，根据实际情况拆分为三个接口
   3. 依赖倒转原则(Dependence Inversion Principle)
      1. 基本介绍
         1. 高层模块不应该依赖底层模块，二者都应该依赖其抽象
         2. 抽象不应该依赖细节，细节应该依赖抽象
         3. 依赖倒转（倒置）的中心思想是面向接口编程
         4. 依赖倒转原则是基于这样的设计理念：相对于细节的多变性，抽象的东西要稳定的多，以抽象为基础搭建的架构比以细节为基础的架构要稳定的多。在java中，抽象指的是接口或抽象类，细节就是具体的实现类
         5. 使用接口或抽象类的目的是制定好规范，而不涉及任何具体的操作，把展现细节的任务交给他们的实现类去完成
      2. 依赖关系传递的三种方式
         1. 接口传递
         2. 构造方法传递
         3. Setter方法传递
      3. 注意事项和细节
         1. 底层模块尽量都要有抽象类或接口，或者两者都有，程序稳定性更好
         2. 变量的声明类型尽量是抽象类或接口，这样我们的变量引用和实际对象间，就存在了一个缓冲层，利于程序扩展和优化
         3. 继承时遵循里氏替换原则
   4. 里氏替换原则(Liskov Subsititution Principle)
      1. 基本介绍：
         1. 如果对每个类型为T1的对象o1，都有类型为T2的对象o2，使得以T1定义的所有程序P在所有的对象o1都带换成o2时，程序P的行为没有发生改变，那么类型T2是类型T1的子类型。换句话说，所有引用基类的地方必须能透明的使用其子类的对象
         2. 在使用继承时，遵循里式替换原则，在子类中尽量不要重写父类的方法
         3. 里氏替换原则告诉我们，继承实际上让两个类耦合性增强了，在适当的情况下，可以通过聚合，组合，依赖来解决问题。
   5. 开闭原则(Open Closed Principle)
      1. 基本介绍
         1. 一个软件实体如类，模块和函数应该对扩展开放，对修改关闭，用抽象构建框架，用实现扩展细节
         2. 当软件需要变化时，尽量通过扩展软件的实体的行为来实现变化，而不是修改已有的代码来实现变化
         3. 编程中遵循其他原则，以及设计模式的目的就是遵循开闭原则
   6. 迪米特法则(Demeter Principle)
      1. 一个对象应该对其他对象保持最少的了解
      2. 类与类关系越密切，耦合度越大
      3. 又叫最少知道原则，即一个类对自己依赖的类知道的越少越好。也就是说，对于被依赖的类不管多么复杂，都尽量将逻辑封装在类的内部。对外除了提供public方法，不对外泄露任何信息
      4. 更简单的定义：只与直接的朋友通信
         1. 直接的朋友：每个对象都会与其他对象有耦合关系，只要两个对象之间有耦合关系，我们就说这两个对象之间是朋友关系。耦合的方式有很多，依赖，关联，组合，聚合等。其中，我们称出现成员变量，方法参数，方法返回值中的类为直接的朋友，而出现在局部变量中得类，不是直接的朋友。也就是说，陌生的类最好不要以局部变量的方式出现在类的内部
      5. 注意事项和细节
         1. 迪米特法则的核心是降低类之间的耦合性
         2. 但是注意：由于每个类都减少了不必要的依赖，因此迪米特法则只是要求降低类间（对象间）耦合关系，并不是要求完全没有依赖关系
   7. 合成复用原则(Composite Reuse Principle)
      1. 基本介紹：原则是尽量使用合成/聚合的方式，而不是使用继承
   8. 设计原则的核心思想：
      1. 找出应用中可能需要变化之处，把他们独立出来，不要和哪些不需要变化的代码混在一起
      2. 针对接口编程，而不是针对实现编程
      3. 为了交互对象之间的松耦合设计而努力
2. UML：类图
   1. 依赖关系(Dependence)
      1. 只要是在类中用到了对方，那么他们之间就存在依赖关系，如果没有对方连编译都不会通过
      2. 类中用到了对方
      3. 如果是类的成员属性
      4. 如果是方法的返回类型
      5. 是方法接收的参数类型
      6. 方法中使用到
   2. 泛化(Generalization)
      1. 实际上就是继承关系，它是依赖关系的特例
   3. 实现关系(Implementation)
      1. 实现关系其实就是A类实现了B接口，他是依赖关系的特例
   4. 关联关系(Association)
      1. 实际上就是类与类之间的联系，它是依赖关系的特例
      2. 关联具有导航性：即双向关系或单向关系
      3. 关系具有多重性
   5. 聚合关系(Aggregation)
      1. 表示的是整体和部分的关系，整体与部分可以分开，聚合关系是关联关系的特例，所以它具有关联的导航性与多重性
   6. 组合关系(Composition)
      1. 也是整体与部分的关系，但是整体与部分不可以分开。
      2. 共生共死就是组合关系
3. 单例模式的八种写法
   1. 饿汉式两种
   2. 懒汉式三种
   3. 双重检查
   4. 静态内部类
   5. 枚举
4. 设计模式的目的
   1. 代码重用性：相同功能的代码不用多次编写
   2. 可读性：编程规范性，便于其他程序员的阅读和理解
   3. 可扩展性：当需要添加新的功能是，非常方便，称为可维护性
   4. 可靠性：当我们添加新的功能后，对原来的功能没有影响
   5. 使程序呈现高内聚低耦合的特性
5. 单例设计模式
   1. 介绍：就是采取一定的方法保证在整个软件系统中，对某个类只能存在一个对象实例，并且该类只提供一个取得其对象实例的方法（静态方法）
   2. 八种方式
      1. 饿汉式（静态常量）
      2. 饿汉式（静态代码快）
      3. 懒汉式（线程不安全）
      4. 懒汉式（线程安全，同步方法）
      5. 懒汉式（线程安全，同步代码块）
      6. 双重检查
      7. 静态内部类
      8. 枚举
6. 简单工厂模式
   1. 基本介绍
      1. 简单工厂模式是属于创建型模型，是工厂模式的一种，简单工厂模式是由一个工厂对象决定创建出哪一种产品类的实例。简单工厂模式是工厂模式家族中最简单实用的模式
      2. 简单工厂模式：定义了一个创建对象的类，由这个类来封装实例化对象的行为（代码）
      3. 当软件开发中，当我们会用到大量的创建某种，莫类或者某批对象时，就会实用到工厂模式
7. 工厂方法模式
   1. 基本介绍
      1. 定义了一个创建对象的抽象方法，由子类决定要实例化的类。工厂方法模式讲对象的实例化推迟到子类
8. 抽象工厂模式
   1. 基本介绍
      1. 抽象工厂模式：定义了一个interface用于创建相关或有依赖关系的对象簇，而无须指明具体的类
      2. 抽象工厂模式可以将简单工厂模式和工厂方法模式进行整合
      3. 从设计层面看，抽象工厂模式就是对简单工厂模式的改进（或者称为进一步的抽象）
      4. 将工厂抽象成两层，AbsFactory(抽象工厂)和具体实现的工厂子类。程序员可以根据创建对象类型使用对应的工厂子类。这样将单个的简单工厂类变成了工厂簇，更利于代码的维护和扩展。
9. 原型模式(Prototype)
   1. Java中Object类是所有类的根类，Object类提供了一个clone()方法，该方法可以将一个java对象复制一份，但是需要实现clone()的java类必须要实现一个接口Colneable，该接口表示该类能够复制且具有复制的能力 ==》原型模式
   2. 基本介绍
      1. 是指用原型实例指定创建对象的种类，并且通过拷贝这些原型，创建新的对象
      2. 原型模式是一种创建型设计模式，允许一个对象再创建另外一个可定制的对象，无需知道如何创建的细节
      3. 工作原理是：通过讲一个原型对象传给内个要发动创建的对象，这个要发动创建的对象通过请求原型对象拷贝它们自己来实施创建，即对象.clone()
   3. 浅拷贝基本介绍
      1. 对于数据类型是基本数据类型的变量，浅拷贝会直接进行值传递，也就是将该属性值复制一份给新的对象。
      2. 对于数据类型是引用数据类型的成员变量，比如说成员变量是某个数组，某个类的对象等，那么浅拷贝会进行引用传递，也就是只是将该成员变量的引用值（内存地址）复制一份给新的对象，因为实际上两个对象的该成员变量都指向同一个实例。在这种情况下，在一个对象中修改该成员变量会影响到另一个对象的该成员变量值
      3. 浅拷贝使用默认的clone()方法来实现
   4. 深拷贝基本介绍
      1. 复制对象的所有基本数据类型的成员变量值
      2. 为所有引用数据类型的成员变量申请存储空间，并赋值每个引用数据类型成员变量多引用的对象，直到该对象可达的所有对象。也就是说，对象进行深拷贝要对整个对象进行拷贝
      3. 深拷贝实现方式1：重写clone()方法来实现深拷贝
      4. 深拷贝实现方式2：通过对象序列化来实现深拷贝
10. 建造者模式(Builder Pattern)
    1. 基本介绍
       1. 又叫生成器模式，是一种对象构建模式。它可以将复杂对象的建造过程抽象出来（抽象类别），使这个抽象过程的不同实现方法可以构造出不同表现（属性）的对象
       2. 建造者模式是一步一步创建一个复杂的对象，它允许用户只通过指定复杂对象的类型和内容就可以构建它们，用户不需要知道内部的具体构建细节
11. 适配器模式(Adapter Pattern)
    1. 基本介绍
       1. 将某个类的接口转换成客户端期望的另一个接口表示，主的目的是兼容性，让原本因接口不匹配不能一起工作的两个类可以协同工作。其别名为包装类(Wrapper)
       2. 适配器属于结构型模式
       3. 主要分为三类：类适配器模式，对象适配器模式，接口适配器模式
    2. 类适配器模式
       1. 介绍：Adapter类，通过继承src类，实现dst类接口，完成src->dst的适配
    3. 对象适配器模式
       1. 模式介绍
          1. 基本思路和类的适配器思路相同，只是将Adapter类做修改，不是继承src类，而是持有src类的实例，以解决兼容性的问题。即： 持有src类，实现dst接口，完成src->dst的适配
          2. 根据“合成复用原则”，在系统中尽量使用关联关系来替代继承关系
          3. 对象适配器是适配器模式中常用的一种
    4. 接口适配器模式/缺省适配器模式
       1. 介绍：当不需要全部实现接口提供的方法时，可先设计一个抽象类实现接口，并为该接口中每个方法提供一个默认实现（空方法），那么该抽象类的子类可有选择的覆盖父类的某些方法来实现需求
       2. 适用于一个接口不想使用其所有的方法的情况
12. 桥接模式(Bridge)
    1. 基本介绍
       1. 指的是：将实现与抽象放在两个不同的类层次中，使两个层次可以独立改变
       2. 是一种结构型设计模式
       3. Bridge模式基于类的最小设计原则，通过使用封装，聚合及继承等行为让不同的类承担不同的职责，它的主要特点是把抽象（abstract）与行为实现（implementation）分离开来，从而可以保持各部分的独立性以及应对它们的功能扩展
13. 装饰者模式(Decorator)
    1. 定义
       1. 装饰者模式：动态的将新功能附加在对象上。在对象功能扩展方面，它比继承更有弹性，装饰者模式也体现了开闭原则（ocp）
14. 组合模式(Composite Pattern)
    1. 基本介绍
       1. 又叫部分整体模式，它创建了对象组的树形结构，讲对象组合成树形结构以表示“整体-部分”的层次关系
       2. 组合模式依据树形结构来组合对象，用来表示部分以及整体层次
       3. 属于结构性设计模式
       4. 组合模式使得用户对单个对象和组合对象的访问具有一致性，即：组合能让客户以一致的方式处理个别对象以及组合对象
15. 外观模式(Facade)
    1. 基本介绍
       1. 外观模式，也叫过程模式：外观模式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
       2. 外观模式通过定义一个一致的接口，用以屏蔽内部子系统的细节，使得调用端只需要跟这个接口发生调用，而无需关注这个子系统的内部细节
16. 享元模式(Flyweight Pattern)
17. 代理模式(Proxy)
    1. 基本介绍
       1. 代理模式：为一个对象提供一个替身，以控制这个对象的访问，即通过代理对象访问目标对象，这样做的好处是：可以在目标对象实现的基础上，增强额外的功能操作，即扩展目标对象的功能
       2. 被代理的对象可以是远程对象，创建开销大的对象或需要安全控制的对象
       3. 代理模式有不同的形式，主要有三种，静态代理，动态代理(JDK代理，接口代理)和Cglib代理 (可以在内存动态的创建对象，而不需要实现接口，它是属于动态代理的范畴)
18. 模板方法(Template Method Pattern)
    1. 基本介绍
       1. 在一个抽象类公开定义了执行它的方法的模板。它的子类可以按需要重写方法实现，但调用会以抽象类中定义的方式运行
       2. 简单说，模板方法模式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，使得子类可以不改变一个算法的结构，就可以重定义该算法的某些特定步骤
       3. 这种类型的设计模式是行为型设计模式。
19. 命令模式(Command Pattern)
20. 访问者模式(Visitor Pattern)
21. 迭代器模式(Iterator Pattern)
22. 观察者模式(Observer)
23. 中介者模式(Mediator Pattern)
    1. 基本介绍
       1. 用一个中介对象来封装一系列的对象交互。中介者使各个对象不需要显示的相互引用，从而使其耦合松散，而且可以独立地改变它们之间的交互
       2. 中介者模式属于行为型模式，使代码易于维护
       3. 比如MVC模式，C（controller控制器）是M（Model模型）和V（View视图）的中介者，在前后端交互时起到中间人的作用
24. 备忘录模式(Memento Pattern)
    1. 基本介绍
       1. 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。这样就可以将该对象恢复到原先保存的状态
       2. 直接体现就是可以进行存档，一些软件的ctrl+z的后退功能
25. 解释器模式(Interpreter Pattern)
26. 状态模式(State Pattern)
    1. 基本介绍
       1. 主要用来解决对象在多种状态转变时，需要对外输出不同的行为的问题。状态和行为是一一对应的，状态之间可以相互转变
       2. 当一个对象的内在状态转变时，允许改变其行为，这个对象看起来像是改变了其类
27. 策略模式(Strategy Pattern)
    1. 基本介绍
       1. 策略模式中，定义算法族，分别封装起来，让他们之前可以相互转换，此模式让算法的变化独立于使用算法的客户
       2. 这算法体现了几个设计原则，第一，把变化的代码从不变的代码中分离出来；第二，针对接口编程而不是具体类（定义了策略接口）；第三，多用组合聚合，少用继承（客户通过组合方式使用继承）
28. 职责链模式(Chain of Responsibility Pattern)
    1. 基本介绍
       1. 又叫责任链模式，为请求者创建了一个接受者对象的链，这种模式对请求的发送者和接受者进行解耦。
       2. 职责链模式通常每个接受者都包含对另一个接受者的引用，如果一个对象不能处理该请求，那么它会相同的请求传给下一个接受者，依次类推
       3. 这种模式属于行为型模式