Java面试题-设计模式+设计原则+代理

一.什么是设计模式

设计模式（Design pattern）是一套被反复使用、多数人知晓的、经过分类编目的、代码设计经验的总结。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。 毫无疑问，设计模式于己于他人于系统都是多赢的，设计模式使代码编制真正工程化，设计模式是软件工程的基石，如同大厦的一块块砖石一样。项目中合理的运用设计模式可以完美的解决很多问题，每种模式在现在中都有相应的原理来与之对应，每一个模式描述了一个在我们周围不断重复发生的问题，以及该问题的核心解决方案，这也是它能被广泛应用的原因。简单说：

- 模式：在某些场景下，针对某类问题的某种通用的解决方案
- 场景：项目所在的环境
- 问题：约束条件，项目目标等
- 解决方案：通用、可复用的设计，解决约束达到目标

二.设计模式的三个分类

- 创建型模式：对象实例化的模式，创建型模式用于解耦对象的实例化过程。
- 结构型模式：把类或对象结合在一起形成一个更大的结构。
- 行为型模式：类和对象如何交互，及划分责任和算法。

三.各分类中模式的关键点

创建型模式

- 单例模式：某个类只能有一个实例，提供一个全局的访问点。
- 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。
- 原型模式：通过复制现有的实例来创建新的实例。

结构型模式

- 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。
- 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。
- 享元（蝇量）模式：通过共享技术来有效的支持大量细粒度的对象。
- 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。

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。

四.设计模式的性能，例如单例模式哪种性能更好

单例模式是最常用到的设计模式之一，熟悉设计模式的朋友对单例模式都不会陌生。一般介绍单例模式的书籍都会提到 **饿汉式** 和 **懒汉式** 这两种实现方式。但是

除了这两种方式，本文还会介绍其他几种实现单例的方式，让我们来一起看看吧。

1.单例模式简介

单例模式是一种常用的软件设计模式，其定义是单例对象的类只能允许一个实例存在。

许多时候整个系统只需要拥有一个的全局对象，这样有利于我们协调系统整体的行为。比如在某个服务器程序中，该服务器的配置信息存放在一个文件中，这些配置数据由一个单例对象统一读取，然后服务进程中的其他对象再通过这个单例对象获取这些配置信息。这种方式简化了在复杂环境下的配置管理。

2.基本的实现思路

单例模式要求类能够有返回对象一个引用(永远是同一个)和一个获得该实例的方法（必须是静态方法，通常使用 `getInstance` 这个名称）。

单例的实现主要是通过以下两个步骤：

1. 将该类的构造方法定义为私有方法，这样其他处的代码就无法通过调用该类的构造方法来实例化该类的对象，只有通过该类提供的静态方法来得到该类的唯一实例；
2. 在该类内提供一个静态方法，当我们调用这个方法时，如果类持有的引用不为空就返回这个引用，如果类保持的引用为空就创建该类的实例并将实例的引用赋予该类保持的引用。

3.注意事项

单例模式在多线程的应用场合下必须小心使用。如果当唯一实例尚未创建时，有两个线程同时调用创建方法，那么它们同时没有检测到唯一实例的存在，从而同时各自创建了一个实例，这样就有两个实例被构造出来，从而违反了单例模式中实例唯一的原则。 解决这个问题的办法是为指示类是否已经实例化的变量提供一个互斥锁(虽然这样会降低效率)。

4.单例模式的八种写法

1、饿汉式（静态常量）[可用]

```
public class Singleton {

    private final static Singleton INSTANCE = new Singleton();

    private Singleton(){}

    public static Singleton getInstance(){
        return INSTANCE;
    }
}
```

- 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。

2、饿汉式（静态代码块）[可用]

```
public class Singleton {

    private static Singleton instance;

    static {
        instance = new Singleton();
    }

    private Singleton() {}

    public Singleton getInstance() {
        return instance;
    }
}
```

这种方式和上面的方式其实类似，只不过将类实例化的过程放在了静态代码块中，也是在类装载的时候，就执行静态代码块中的代码，初始化类的实例。优缺点和上面是一样的。

3、懒汉式(线程不安全)[不可用]

```
public class Singleton {

    private static Singleton singleton;
```

```
    private Singleton() {}

    public static Singleton getInstance() {
        if (singleton == null) {
            singleton = new Singleton();
        }
        return singleton;
    }
}
```

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。

4、懒汉式(线程安全，同步方法)[不推荐用]

```
public class Singleton {

    private static Singleton singleton;

    private Singleton() {}

    public static synchronized Singleton getInstance() {
        if (singleton == null) {
            singleton = new Singleton();
        }
        return singleton;
    }
}
```

解决上面第三种实现方式的线程不安全问题，做个线程同步就可以了，于是就对 `getInstance()` 方法进行了线程同步。

- 缺点：效率太低了，每个线程在想获得类的实例时候，执行 `getInstance()` 方法都要进行同步。而其实这个方法只执行一次实例化代码就够了，后面的想获得该类实例，直接 return 就行了。方法进行同步效率太低要改进。

5、懒汉式(线程安全，同步代码块)[不可用]

```
public class Singleton {
```

```
    private static Singleton singleton;

    private Singleton() {}

    public static Singleton getInstance() {
        if (singleton == null) {
            synchronized (Singleton.class) {
                singleton = new Singleton();
            }
        }
        return singleton;
    }
}
```

由于第四种实现方式同步效率太低，所以摒弃同步方法，改为同步产生实例化的的代码块。但是这种同步并不能起到线程同步的作用。跟第 3 种实现方式遇到的情形一致，假如一个线程进入了 `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。

6、双重检查[推荐用]

```
public class Singleton {

    private static volatile Singleton singleton;

    private Singleton() {}

    public static Singleton getInstance() {
        if (singleton == null) {
            synchronized (Singleton.class) {
                if (singleton == null) {
                    singleton = new Singleton();
                }
            }
        }
        return singleton;
    }
}
```

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。

7、静态内部类[推荐用]

```
public class Singleton {

    private Singleton() {}

    private static class SingletonInstance {
        private static final Singleton INSTANCE = new Singleton();
    }

    public static Singleton getInstance() {
        return SingletonInstance.INSTANCE;
    }
}
```

这种方式跟饿汉式方式采用的机制类似，但又有不同。两者都是采用了类装载的机制来保证初始化实例时只有一个线程。不同的地方在饿汉式方式是只要 `Singleton` 类被装载就会实例化，没有 `Lazy-Loading` 的作用，而静态内部类方式在 `Singleton` 类被装载时并不会立即实例化，而是在需要实例化时，调用 `getInstance` 方法，才会装载 `SingletonInstance` 类，从而完成 `Singleton` 的实例化。

类的静态属性只会在第一次加载类的时候初始化，所以在这里，JVM 帮助我们保证了线程的安全性，在类进行初始化时，别的线程是无法进入的。

- 优点：避免了线程不安全，延迟加载，效率高。

8、枚举[推荐用]

```
public enum Singleton {
    INSTANCE;
    public void whateverMethod() {

    }
}
```

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。

5.单例模式的优点

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。

6.单例模式的缺点

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。

7.单例模式的适用场合

- 需要频繁的进行创建和销毁的对象；
- 创建对象时耗时过多或耗费资源过多，但又经常用到的对象；
- 工具类对象；
- 频繁访问数据库或文件的对象。

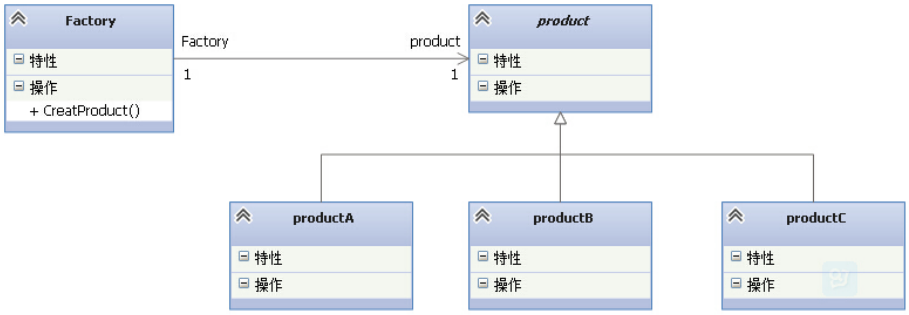
五.策略模式 + 简单工厂模式的实践

1.前言

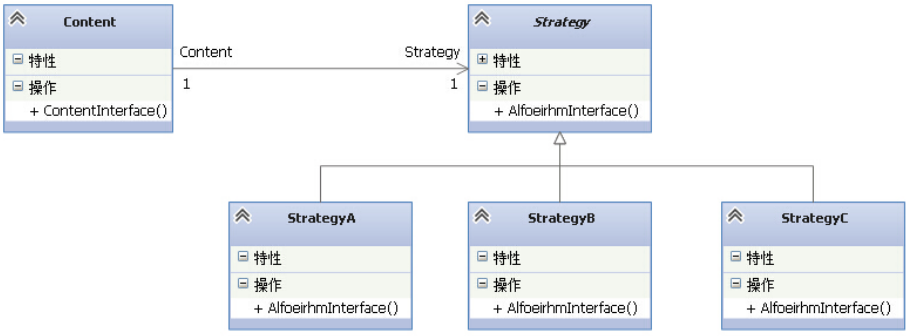
简单工厂模式和策略模式是大部分程序员，在学习设计模式时接触得最早，或在工作实践中也是用得相对比较多的两个设计模式。

一个是创建型，另一个是行为型，然而两种不同类型的模式，在某些地方也有一丝的相似之处，同时在某种场景下结合使用，能起到特别好的效果。

简单工厂模式



策略模式



2.问题

简单工厂模式和策略模式很相似。怎么相似？都是三个业务子类继承抽象父类，通过传入参数到容器类（工厂模式的 factory 类，策略模式的 Content 类），选择对应的类进行行为操作。

其实，UML 图的确从外形上看没多大区别，但是，本质却是大大不同。

3.简单工厂模式

上面提到过，简单工厂模式是创建型模式，创建型模式顾名思义，也就是说在创建对象的时候，遇到了瓶颈才会选择的设计模式。那么该什么情况使用呢。

简单工厂模式的实质是由一个工厂类根据传入的参数，**动态决定**应该**创建并且返回**哪一个产品类（这些产品类继承自一个父类或接口）的实例。

那么也就是说：

- 有已知的产品类
- 你无法准确的知道编译哪个产品类

- 需要在运行时决定创建哪个产品类
- 产品类不多

很明显看出，在创建对象上的灵活性高，但是工厂类只能创建可能会使用到的产品类，假如新添了产品类就得修改工厂类，这样就会违反开闭原则。

4.策略模式

策略模式是行为型模式，它定义了一系列的算法，并将每一个算法封装起来，而且使它们还可以相互替换。策略模式让算法独立于使用它的客户而独立变化。

在一段代码里，使用了逻辑控制（if-else, swich-case）来决定算法，算法有相似的方法和函数，就可以选择策略模式。

那么也就是说：

- 某方法里有多个条件语句，条件语句代码块里有许多行为过程。
- 其算法能封装到策略类
- 算法随意切换
- 算法与客户端隔离

这样一来，通过选择对应的策略类，作为参数传到 Content 类里，在**运行时**配置对应的算法。

5.区别总结

从上面的描述总结出，在运行时，两者都是通过传入参数进行配置，简单工厂模式则是选择创建出需要的对象，而策略模式则是配置出需要的行为算法。一个是对象创建，另一个是行为算法的替换。

六.策略模式与状态模式的区别

Java 开发者，要想恰当的使用状态模式和策略模式，必须清楚的理解它们之间的区别。虽然状态模式和策略模式拥有相似的结构，虽然它们都基于 SOLID 设计原则中的 O（开闭原则），但是，它们的意图是完全不同的。

策略模式通过封装一组相关算法，为 `Client` 提供运行时的灵活性。`Client` 可以在运行时，选择任一算法，而不改变使用算法的 `Context` 。一些流行的策略模式的例子是写那些使用算法的代码，例如加密算法、压缩算法、排序算法。另一

方面，状态模式允许对象，在不同的状态拥有不同的行为。因为现实世界中的对象通常都是有状态的，所以它们在不同状态，行为也不一样。例如，VM（自动售货机）只在 `hasCoin` 状态才给你吐商品；你不投币，它是不会吐的。现在你可以清楚的看出它们的不同之处了：它们的意图是不同的。状态模式帮助对象管理状态，而策略模式允许 `Client` 选择不同的行为。

另一个不那么容易能看出来的区别是：是谁促使了行为的改变。策略模式中，是 `Client` 提供了不同的策略给 `Context`；状态模式中，状态转移由 `Context` 或 `State` 自己管理。另外，如果你在 `State` 中管理状态转移，那么它必须持有 `Context` 的引用。例如，在 VM 的例子中， `State` 对象需要调用 VM 的 `setState()` 方法去改变它的状态。另一方面， `Strategy` 从不持有 `Context` 的引用，是 `Client` 把所选择的 `Strategy` 传递给 `Context` 。由于状态模式和策略模式的区别，是流行的 Java 设计原则类面试题之一，我们将会在本文探讨在 Java 中，状态模式和策略模式的异同，这可以加深你对它们的理解。

1.相似之处

1. 添加新的状态或策略都很容易，而且不需要修改使用它们的 Context 对象。
2. 它们都让你的代码符合 OCP 原则。在状态模式和策略模式中，Context 对象对修改是关闭的，添加新的状态或策略，都不需要修改 Context。
3. 正如状态模式中的 Context 会有初始状态一样，策略模式同样有默认策略。
4. 状态模式以不同的状态封装不同的行为，而策略模式以不同的策略封装不同的行为。
5. 它们都依赖子类去实现相关行为。

2.不同之处

状态模式和策略模式的结构是相似的，但它们的意图不同

1. 策略模式封装了一组相关算法，它允许 Client 在运行时使用可互换的行为；状态模式帮助一个类在不同的状态显示不同的行为。
2. 状态模式封装了对象的状态，而策略模式封装算法或策略。因为状态是跟对象密切相关的，它不能被重用；而通过从 Context 中分离出策略或算法，我们可以重用它们。
3. 在状态模式中，每个状态通过持有 Context 的引用，来实现状态转移；但是每个策略都不持有 Context 的引用，它们只是被 Context 使用。
4. 策略实现可以作为参数传递给使用它的对象，例如 `Collections.sort()`，它的参数包含一个 `Comparator` 策略。另一方面，状态是 Context 对象自己

的一部分，随着时间的推移，Context 对象从一个状态转移到另一个状态。

5. 虽然它们都符合 `OCP` 原则，策略模式也符合 `SRP` 原则（单一职责原则），因为每个策略都封装自己的算法，且不依赖其他策略。一个策略的改变，并不会导致其他策略的变化。
6. 另一个理论上的不同：策略模式定义了对象“怎么做”的部分。例如，排序对象怎么对数据排序。状态模式定义了对象“是什么”和“什么时候做”的部分。例如，对象处于什么状态，什么时候处在某个特定的状态。
7. 状态模式中很好的定义了状态转移的次序；而策略模式并无此需要：Client 可以自由的选择任何策略。
8. 一些常见的策略模式的例子是封装算法，例如排序算法，加密算法或者压缩算法。如果你看到你的代码需要使用不同类型的相关算法，那么考虑使用策略模式吧。而识别何时使用状态模式是很简单的：如果你需要管理状态和状态转移，但不想使用大量嵌套的条件语句，那么就是它了。
9. 最后但最重要的一个不同之处是，策略的改变由 Client 完成；而状态的改变，由 Context 或状态自己。

七.说说反模式设计

简单的来说，反模式是指在对经常面对的问题使用的低效，不良，或者有待优化的设计模式/方法。甚至，反模式也可以是一种错误的开发思想/理念。在这里我举一个最简单的例子：在面向对象设计/编程中，有一条很重要的原则， 单一责任原则(Single responsibility principle)。其中心思想就是对于一个模块，或者一个类来说，这个模块或者这个类应该只对系统/软件的一个功能负责，而且该责任应该被该类完全封装起来。当开发人员需要修改系统的某个功能，这个模块/类是最主要的修改地方。相对应的一个反模式就是上帝类(God Class)，通常来说，这个类里面控制了很多其他的类，同时也依赖其他很多类。整个类不光负责自己的主要单一功能，而且还负责了其他很多功能，包括一些辅助功能。很多维护老程序的开发人员们可能都遇过这种类，一个类里有几千行的代码，有很多功能，但是责任不明确单一。单元测试程序也变复杂无比。维护/修改这个类的时间要远远超出其他类的时间。很多时候，形成这种情况并不是开发人员故意的。很多情况下主要是由于随着系统的年限，需求的变化，项目的资源压力，项目组人员流动，系统结构的变化而导致某些原先小型的，符合单一原则类慢慢的变的臃肿起来。最后当这个类变成了维护的噩梦(特别是原先熟悉的开发人员离职后)，重构该类就变成了一个不容易的工程。

八.说说你对设计原则的理解

1.口诀

为了便于记忆，我们可以使用一个口诀来记忆面向对象设计原则：**开口合里最单依**

- 开：开闭原则
- 口：接口隔离原则
- 合：组合/聚合原则
- 里：里式替换原则
- 最：最少知识原则（迪米特法则）
- 单：单一职责原则
- 依：依赖倒置原则

开闭原则(Open-Closed Principle, OCP)

一个软件实体应当对扩展开发,对修改关闭.说的是,再设计一个模块的时候,应当使这个模块可以在不被修改的前提下被扩展.换言之,应当可以在不必修改源代码的情况下改变这个模块的行为，在保持系统一定稳定性的基础上，对系统进行扩展。这是面向对象设计（OOD）的基石，也是最重要的原则。

接口隔离原则(Interface Segregation Principle, ISP)

- 一个类对另外一个类的依赖是建立在最小的接口上。
- 使用多个专门的接口比使用单一的总接口要好.根据客户需要的不同,而为不同的客户端提供不同的服务是一种应当得到鼓励的做法.就像"看人下菜碟"一样,要看客人是谁,再提供不同档次的饭菜.
- 胖接口会导致他们的客户程序之间产生不正常的并且有害的耦合关系.当一个客户程序要求该胖接口进行一个改动时,会影响到所有其他的客户程序.因此客户程序应该仅仅依赖他们实际需要调用的方法.

组合/聚合复用原则(Composite/Aggregate Reuse Principle，CARP)

在一个新的对象里面使用一些已有的对象,使之成为新对象的一部分;新的对象通过这些向对象的委派达到复用已有功能的目的.这个设计原则有另一个简短的表述:要尽量使用合成/聚合,尽量不要使用继承.

里氏代换原则(Liskov Substitution Principle，LSP)

由 Barbar Liskov (芭芭拉.里氏) 提出，是继承复用的基石。

所有引用基类的地方必须透明的使用其子类的对象。只要父类能出现的地方子类也可以出现，而且替换为子类不会产生任何错误或异常，但是反过来就不行，有子类出现的地方，父类未必就能适应。

最少知识原则(Least Knowledge Principle，LKP)

一个对象应当对其他对象有尽可能少的了解.

没有任何一个其他的 OO 设计原则象迪米特法则这样有如此之多的表述方式,如下几种：

- 只与你直接的朋友们通信(Only talk to your immediate friends)
- 不要跟"陌生人"说话(Don't talk to strangers)
- 每一个软件单位对其他的单位都只有最少的知识,而且局限于那些本单位密切相关的软件单位

就是说,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,如果其中的一个类需要调用另一个类的某一个方法的话,可以通过第三者转发这个调用。

单一职责原则(Simple responsibility pinciple，SRP)

就一个类而言,应该仅有一个引起它变化的原因,如果你能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.应该把多于的指责分离出去,分别再创建一些类来完成每一个职责.

依赖倒置原则(Dependence Inversion Principle)

要求客户端依赖于抽象耦合.

- 模块间的依赖通过抽象发生，实现类之间不发生直接的依赖关系，其依赖关系是通过接口或抽象类产生的。
- 接口或抽象类不依赖实现类
- 实现类依赖接口或抽象类

采用依赖倒置原则可以减少类间的耦合性，提高系统的稳定，降低并行开发引起的风险，提高代码的可读性和可维护性。

九.静态代理/动态代理（CGLIB 与 JDK）

JDK 动态代理类和委托类需要都实现同一个接口。也就是说只有实现了某个接口的类可以使用 Java 动态代理机制。但是，事实上使用中并不是遇到的所有类都会给你实现一个接口。因此，对于没有实现接口的类，就不能使用该机制。而 CGLIB 则可以实现对类的动态代理。